Perl

Co je Perl?

Perl je jednoduchý a praktický interpretační programovací jazyk. V praxi je využívaný zejména na WWW serverech a při zpracování textu. V oblasti zpracovávání souborů patří bez nadsázky vůbec k těm nejlepším nástrojům, které existují. Ostatně právě kvůli tomu byl Perl navržen.

Co je cílem seriálu?

Odpověď je jednoduchá - cílem je pomoci čtenářům naučit se Perl.

Co je programování?

Programování můžeme pro naše potřeby chápat jako vytváření programu. **Program**je seznam činností, které byly nařízeny počítači k vykonání. Programem je například tato posloupnost příkazů:

- požádej uživatele o dvě čísla
- načti je od něj
- sečti je
- pošli jejich součet na výstup

Obecněnji lze říci, že programy nejsou jen pro počítače. Stačí nahlédnot do kuchařky. S trochou nadsázky se dá říct, že každý recept je programem. Program musí být napsán v jazyce, kterému rozumí jak autor, tak ten, pro koho je určen - v našem případě počítač.

Programování tedy znamená přesvědčit počítač, aby udělal to, co po něm chceme.

Něco z historie

Název Perl vznikl z akronymu Practical Extraction and Report Language. Autorem je Kanaďan Larry Wall (narozen 10. 3. 1949, osobní stránka). Napsal jej pro svou potřebu, protože mu chybělo něco jednoduššího než Céčko, co zároveň zvládne i větší požadavky. Potom v roce 1987 uvolnil Perl verze 1.0 pro veřejnost a sám se divil zájmu, který tím vyvolal. Proto se začal ještě dále vyvíjet. Z nástroje pro zpracování textu se povýšil na programovací jazyk s mnoha doplňky. Dnes se o vývoj Perlu stará skupina lidí v čele s Larrym Wallem.

Poslední stabilní verzí Perlu je verze 5.12.3 ze dne 21. 1. 2011. Perlu 5.x již táhne na dospělost, neboť je aktuální již od roku 1994. S velkým očekáváním však komunita čeká na Perl 6. Je vyvíjen již řadu let a uvolnění se dlouho odkládá. Manažeři dnes po těchto zkušenostech tvrdí: "Perl 6 has no schedule". Zvýšit se má rychlost a efektivita a přijde také řada revolučních změn, které Perl opět posunou daleko před konkurenci. Specifikace s vlastnostmi jazyka je již hotová - v ní se nyní dělají pouze drobné úpravy. Čeká se na implementaci interpretu - na něj si ještě nějakou dobu musíme počkat.

Na popularitě získal Perl i tím, že se osvědčil jako výborný nástroj pro **CGI skriptování**. CGI je externí program, který je spouštěn na žádost WWW serveru. CGI skriptům se budeme věnovat později v samostatných dílech. Dnes již nemá CGI takovou popularitu, protože ho z jeho pozice vytlačují jazyky typu PHP (PHP mimochodem z Perlu vychází). Podotkněme, že pro webové programování však Perl díky mod_perlu můžeme používat pohodlně dále.

Stručně o vlastnostech

Původně byl Perl napsán pro unixovou platformu, ale dnes běží i na mnoha dalších. Mezi nimi se dá relativně dobře přenášet. Samozřejmě, odlišnosti jsou, avšak často do kódu nemusíme zpravidla vůbec zasáhnout.

Hybnou silou lidstva je lenost. A pro líné lidi budeme těžko hledat tak jednoduchý jazyk, ve kterém napíšeme tolik jako v Perlu. Perl se řídí filozofií "za málo peněz hodně muziky". Množství kódu, které je třeba napsat, je oproti většině jiným jazykům velice malé. Psát programy lze neobyčejně rychle. Stejně tak se Perl díky těmto vlastnostem dají základy relativně rychle naučit. Ovšem časem je třeba si uvědomit, že psaní programů zde není přepisování konstrukcí jiných jazyků do syntaxe Perlu. Perl má svoji vlastní filozofii a největší efektivity dosáhneme, když ji porozumíme. Stejně jako u cizích řečí je třeba se naučit v Perlu myslet, abychom využili všeho, co nám skýtá. Ponořit se do hlubin Perlu bude vyžadovat velké úsilí. Cesta k tomu je dlouhá a plná nástrah, ale stojí za to.

Perl je pod GNU GPL Actistic licencí. Můžete ho používat zdarma a to i v komerčních projektech. K programování v Perlu potřebujeme jen interpret Perlu a textový editor. Není nutný ani kompilátor, který ale existuje také a Umožňuje nám poskytovat program ostatním, aniž by viděli zdrojový kód.

V Perlu se dá programovat strukturovaně i objektově.

Jednou z hlavních výhod Perlu je existence serveru CPAN (Comprehensive Perl Archive Network). Nalezneme zde interpret pro různé platformy, dokumentaci a především obrovskou databázi knihoven řešících rozličné problémy, která nemá pro žádný jiný jazyk obdoby. Díky ní se můžeme spolehnout na to, že Perl jen tak ze zemského povrchu nezmizí, ale naopak tu bude ještě hodně dlouho. Archivu se budeme věnovat ve speciálním dílu a knihovny z něj budeme následně často využívat. Existují i české mirrory, například ftp.linux.cz/pub/perl.

Jaké má Perl nevýhody? V Perlu nelze deklarovat proměnné a je zde velmi benevolentní typová kontrola (v Perlu 6 to bude řešeno). Syntaxe Perlu je velmi volná - třeba jen obyčejná podmínka se dá napsat nesčetně způsoby. Je vhodné se držet nějakých pravidel, protože Perl člověku dovolí někdy až příliš mnoho. Taková je ale filozofie jazyka.

Perl se běžně nepoužívá pro mnohasetstránková díla, ale spíše pro kratší a efektivní programy. Není to ale pravidlo. V Perlu existují i velké projekty, na kterých pracují celé týmy vývojářů.

Říká se, že Perl není příliš vhodným jazykem pro začínající programátory. Umožňuje používat nesrozumitelné konstrukce a člověk si může navyknout na některé zlozvyky. Pravdou je, že, pokud člověk nepustí Perl z řetězu, není to tak zlé. Většinou stačí snažit se psát přehledně a používat mód strict, později se lze seznámit s jinými mechanizmy, které člověku pomáhají psát programy čitelně.

Instalace a příprava

S kvalitními nástroji je práce vždy příjemnější. Předtím, než začneme programovat, je tedy nutné si tyto nástroje představit. Co bude tedy potřeba? Nějaký operační systém (v seriálu předpokládáme nějaký Linux, avšak lze použít i jiný systém), interpret jazyka Perl, textový editor a později, až se budeme věnovat databázím, tak Postgres nebo MySQL a také k vyvíjení CGI skriptů WWW server.

Operační systém

Perl je multiplatformní jazyk a spustíme ho prakticky na všech linuxových distribucích i jiných operačních systémech.

Interpret Perlu

S největší pravděpodobností ho již v systému máte. Přesvědčit se o tom můžete tak, že v konzoli zadáte příkaz:

```
$ perl -v
This is perl, v5.10.0 built for i586-linux-thread-multi

Copyright 1987-2007, Larry Wall
```

```
Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using "man perl" or "perldoc perl".  If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.
$
```

Pokud vidíme podobný výstup, máme interpret instalován a o nic se dál starat nemusíme. První řádek výstupu obsahuje číslo verze. Je výhodné mít vždy tu nejnovější stabilní verzi.

Co dělat když Perl instalovaný nemáme? Nejjednodušší je ho nainstalovat z médií, ze kterých byla instalována naše distribuce. Pokud tento způsob nemůžeme z nějakého důvodu praktikovat, je nutné Perl nějak získat - nejlépe stáhnout aktuální verzi z internetu. Ta je vždy k dispozici na www.perl.org/get.html. Soubor se obvykle jmenuje stable.tar.gz. Stačí ho jen rozbalit, přejít do získaného adresáře, zkompilovat a nainstalovat:

```
$ ./configure
$ make
# make install
```

V případě úspěchu nyní Perl máme nainstalován, o čemž se ostatně můžeme přesvědčit:

```
$ perl -v
```

Programy pak budeme spouštět takto:

```
$ perl program.pl
```

Textový editor

Lze použít kterýkoliv, jehož výstupem je neformátovaný text. Měl by umět zvýrazňovat syntaxi Perlu a odsazovat. Pohodlně lze pracovat například v emacsu s cperl-mode.

Nedáte-li dopustit na vim, lze ho samozřejmě používat taktéž. K aktivaci zvýrazňování syntaxe je třeba příkaz :syntax on. Není třeba se starat o volbu jazyka - vim ho rozpozná podle přípony..

Existuje také řada propracovaných grafických vývojových prostředí. Volba je na každém.

Databáze

Nainstalujme si Postgres nebo MySQL nebo nějakou vaší oblíbenou. Jsou-li přímo v linuxové distribuci, máme vystaráno, jinak je najdeme na zmíněných odkazech. V seriálu se naučíme používat obě výše uvedené, avšak práce s ostatními je analogická. Perl podporuje širokou paletu databází, takže není třeba se bát absence příslušných ovladačů.

WWW server

Budeme předpokládat nějakou aktuální verzi Apache.

Překladač

Překladač je k dispozici v základní distribuci Perlu a je užitečný v případech, kdy chceme poskytnout program, ale zdrojový kód si chceme ponechat pro sebe. To se hodí při psaní komerčních aplikací. Překladač může nahradit interpret, neboť ho ke své činnosti nepotřebuje. Zdrojový kód však musí obsahovat všechny potřebné knihovny, a je tak o mnoho větší než obyčejný kód. Použitím překladače vznikne binární soubor. Překladač použijeme takto:

```
$ perlcc program.pl
```

Tak získáme spustitelný soubor ./a.out. Volbou -o můžeme určit získanému souboru jméno.

```
$ perlcc -o program program.pl
```

Tím byl vytvořen soubor ./program.

Ukázkový program

Podívejme se na úvod na program vypisující text. Ten zapíšeme v Céčku takto:

```
#include <stdio.h>
int main(void){
printf ("Program v Cecku\n");
}
```

V Perlu lze týž efekt docílit následovně:

```
print "Program v Perlu"\n;
```

Uložíme-li tento kód do souboru program.pl, lze ho interpretovat následovně:

```
$ perl program.pl
Program v Perlu
$
```

Perl (2) - Úvod do syntaxe

Dnes si napíšeme několik nejjednodušších programů a ukážeme si pár dalších tipů.

Spouštění programu

Do prvního řádku každého perlovského programu se píše řetězec #!/usr/bin/perl. To proto, aby systém věděl, čím má program spouštět. Jakmile má shell zpracovat skript, pozná z prvního řádku, pro který interpret je tento skript určen. Po znacích #! následuje právě cesta k interpretu, který má být použit ke spuštění vámi napsaného programu.

Máme-li interpret jinde, tak je třeba změnit cestu za #! (ta se dá se zjistit příkazem whereis nebo env, případně i jinak - záleží na distribuci). Vhodnější bývá uvést #!/usr/bin/env perl, neboť takto je cesta k interpretu nalezena i v případě, kdy to je něco jiného než /usr/bin/perl

Takto zapsaný úvodní řádek tedy nepoužívá jen Perl, ale interpretované jazyky obecně. Píšeme-li například v Pythonu, budeme do prvního řádku psát #!/usr/bin/env python apod.

Je třeba podotknout, že takto napsaný první řádek program obsahovat nemusí. Jsou však důvody, proč je to vítané.

Normálně bychom spouštěli program takto:

```
$ perl program.pl
```

Ale, použijeme-li onen magický řádek, stačí, když napíšeme jen toto:

```
$ ./program.pl
```

Ještě předtím však musíme označit soubor jako spustitelný:

```
$ chmod 700 program.pl
```

Uživatele tak nemusí zajímat, v jakém jazyce je program napsaný. To je hlavní výhodou.

Důležité je to i proto, že když program s #!/usr/bin/env perl nahrajeme do adresáře /usr/bin (resp. libovolného adresáře z proměnné prostředí PATH) a potom z *kteréhokoliv* místa zadáme následující příkaz, náš program se spustí.

```
$ program.pl
```

Příklad na úvod

Nyní jsme připraveni na první příklad. Vytvořme nový soubor s názvem program.pla v textovém editoru a do něj uložme tento kód:

#!/usr/bin/perl

```
print "Náš 1. program v Perlu!\n";
```

Už ze zápisu je celkem zřejmé, co bude program dělat. Jak jsme se v minulém dílu dozvěděli, je program posloupnost nějakých nařízení počítači. Přesně tak to platí i zde. Řádek print "Náš 1. program v Perlu!\n"; přeložený do naší řeči, znamená Vytiskni na výstup text *Náš 1. program v Perlu!* a nový řádek. print je **příkaz**, kterým dáváme počítači na vědomí, že chceme vypsat text, který napíšeme za print do uvozovek. Středník dává interpretu na vědomí, že zde příkaz končí a začíná nový. Středník musí být mezi každými dvěma příkazy. Výstup našeho prvního programu bude vypadat takto:

```
$ ./program.pl
Náš 1. program v Perlu!
$
```

Speciální znaky

Není to zatím nic jiného, než výpis textu. Zarazit může asi jediná věc - co znamená ta dvojice znaků \n. Je to označení ukončení řádku. Kdybychom tuto sekvenci znaků vynechali, dostali bychom opticky nepříliš hezký výstup:

```
$ ./program.pl
Náš 1. program v Perlu!$
```

\n je jednou z tzv. escape sekvencí. **Escape sekvence** (v tabulce je jejich částečný výčet) jednoduše řečeno nahrazují znaky, které nemůžeme v Perlu za daných okolností napsat (z různých důvodů: například mají nějaký jiný význam nebo pro ně nemáme symbol na klávesnici).

Symbol	Význam
\n	nový řádek
\t	tabulátor
\a	pípnutí systémového zvonku
\u*a*	Znak a - ať je jakýkoliv - bude psán velkým písmenem
\l*a*	Znak a - ať je jakýkoliv - bude psán malým písmenem
\U...\E	Vše mezi escape sekvencemi \U a \E bude psané velkými písmeny
\L...\E	Vše mezi escape sekvencemi \L a \E bude psané malými písmeny
\\	Zpětné lomítko
\"	Uvozovka (používá se v řetězcích označených uvozovkami)
\'	Apostrof (používá se v řetězcích označených apostrofy)

Proč je například takový znak pro nový řádek potřeba? Pokud ve zdrojovém kódu místo escape sekvenece \n jen odentrujeme, neuspějeme. Perl se k znaku nového řádku chovátak, že by ho interpretoval jako mezeru.

Poznamenejme, že Perl mimo data nedělá rozdíly mezi bílými znaky (mezera, tabulátor, znak nového řádku). Jakkoliv rozsáhlý program bychom v Perlu teoreticky mohli napsat na jediný řádek. Stačilo by konce řádků nahradit mezerami.

Zkusme si zaexperimentovat a v našem příkladu si do uvozovek za print přidejme nějakou další escape sekvenci - například \a a sledujme (v tomto případě poslouchejme - tedy pokud nemáme nastaven vizuální zvonek), co se stane.

Přípona perlovského programu

Programům dáváme vždy příponu .pl. Není povinná (systém typ programu stejně vždy určuje podle řádku #!*/cesta/k/interpretu*), ale pro přehlednost je dobrým zvykem ji používat. Později budeme používat také příponu .pm pro moduly. Pro programy v Perlu 6 se používá například .pl6, .p6, .pm6 atd.

Komentáře

Komentáře jsou úseky programu, které interpret ignoruje. Na výsledný program nemají žádný efekt. K čemu jsou tedy dobré? Zvyšují přehlednost zdrojového kódu a jsou velmi užitečné v případě, kdy chceme kód po delší době editovat. Obzvlášť, jedná-li se o složitější program. Uvozují se křížkem - od něj do konce řádku interpret vše ignoruje. Uveďme příklad:

#!/usr/bin/env perl

```
#########################################
# Autor:   Jiří Václavík                #
# Verze:   20050204                     #
# Název:   program.pl                   #
# Činnost: Vypisuje text                #
#########################################

print "Náš už druhý program v Perlu. S komentáři!\n";#Další komentář

#A ještě jeden
```

Jak používat komentáře? Měli by usnadňovat pochopení programu při čtení jeho zdrojového kódu. Nesprávně napsaný komentář je na obtíž (Typickou ukázkou nevhodných komentářů je výše uvedený program - zatím však nemáme znalosti pro program, kde by komentáře byly potřeba. Dostaneme se k tomu.)

Další možností, jak použít komentář je informace o názvu, verzi, autorovi a činnosti popř. něčeho dalšího na začátku rozsáhlejšího programu. Komentáře píšeme pro to, aby usnadňovaly pochopení. Je třeba psát proč je ta a ta část programu napsaná takto a ne jinak, jaký význam zde má tato proměnná, co tu bude potřeba dodělat nebo něco, co zkrátka bude při pozdějším čtení kódu k užitku. Komentář typu #vypisuje text Ahoj za příkazem print "Ahoj"; nám asi moc nepomůže. Ovšem setkat se s ním lze poměrně často, zejména v případech, kdy autor zkrátka komentář napsat chce. Nejlépší metoda, která člověka naučí psát komentáře, je vlastního čtení.

Perl nepodporuje víceřádkové komentáře. Existuje sice speciální syntaxe, která by k tomu teoreticky mohla sloužit, ale není vhodné ji používat, protože je určena k jiným účelům. Řetězec =komentar označuje začátek takového bloku a jeho konec označíme =cut.

#!/usr/bin/perl

```
print "Text před\n";

=komentar
Víceřádkový
"komentář"
Na toto se podívejte a zase na to zapomeňte
=cut
print "Text za\n";
```

Perl (3) - Proměnné

Věnovat se budeme základnímu kamenu programů: proměnným. Jak do proměnné přiřadit data a jak je z ní opětovně dostat?

Proměnná

V každém programu, který má mít vstup nebo výstup, potřebujeme pracovat s nějakými hodnotami. Neobejdeme se tedy bez proměnných. **Proměnnou** si můžeme představit jako označenou oblast v paměti, do níž jsou uložena data.

Přístup k proměnným se liší jazyk od jazyka. V Perlu to funguje tak, že proměnnou můžeme použít kdekoliv v programu, aniž bychom se o ní zmiňovali dříve (deklarovali ji). Perl si vše potřebné obstará sám. Přesněji řečeno, platí to, pokud nepoužíváme režim strict.

Proměnné můžeme kategorizovat podle různých kritérií. My toto dělení zatím ponecháme stranou a budeme se věnovat tzv. skalárním proměnným, které uchovávají jedinou hodnotu. Taková proměnná se v Perlu uvozuje dolarem.

Ukažme si příklad:

```
#!/usr/bin/perl
$promenna = "obsah proměnné\n";
print $promenna;
```

Uložme si tento kód do souboru, nastavme práva pro spouštění a spusťme ho. Příkaz $promenna = "obsah promenne\n"; uloží do proměnné s názvem $promennařetězec v uvozovkách. Tuto operaci můžeme nazvat **přiřazení** (je to vlastně uložení nějaké hodnoty do proměnné). Co se děje dál už víme z minulého dílu. Příkaz printtiskne na výstup. V tomto případě vytiskne obsah proměnné $promenna, který se místo názvu proměnné automaticky dosadí.

Název proměnné

Každá proměnná, jak jsme si již ukázali v příkladu, začíná znakem dolaru. Název se může skládat s písmen anglické abecedy, číslic a podtržítka. Přitom platí, že první znak jména proměnné nesmí být číslice. Délka proměnných je libovolná od 1 znaku až po 255. Délce 255 se v praxi ani zdaleka nepřiblížíme (můžete to vyzkoušet, interpret zahlásí Identifier too long).

Ani zde se nevyhneme všudypřítomné diskuzi o vhodném pojmenování proměnných. Obecně platí, že bychom měli proměnné pojmenovávat tak, aby z názvu bylo jasné, co je v ní uloženo. Ideální jsou krátké a výstižné názvy. Pokud neexistuje krátký a výstižný název, volíme dlouhý a výstižný název. Zároveň bychom neměli používat názvy typu $pocetobyvatelceskerepubliky, protože s absencí mezer člověk musí docela dlouho přemýšlet, co je tam vlastně napsané. K čitelnosti pomůže přidání podtržítek: $pocet_obyvatel_ceske_republiky už je lepší. Delší proměnné mají tu nevýhodu, že jsou náchylnější k překlepům a navíc je psaní programů o něco pomalejší. Avšak lepší je zvolit proměnnou s delším než s nejasným názvem.

Název proměnné tedy budeme volit podle toho, co hodnota v ní vyjadřuje. Podívejme se na jeden odstrašující příklad za všechny:

```
$promenna1 = $promenna2 * $promenna3 * $promenna4;
```

To je takřka nicneříkající. Zato následující kód je již hezky čitelný:

```
$obsah_kvadru = $a * $b * $c;
```

Perl rozlišuje velikost písmen identifikátorů. Proměnná $promenna není totožná s proměnnou $PROMENNA. Existuje ale nepsané pravidlo, že jména proměnných se zapisují malými písmeny. Je to z důvodu přehlednosti. Jiné druhy identifikátorů (zatím jsme si o nich neříkali) se píší zas jiným způsobem - podívejme se na tabulku.

Typ identifikátoru	Způsob zápisu dle zvyku	Příklad
Proměnné, podprogramy	malými písmeny	$promenna, &podprogram()
Balíky	První písmeno slova velké, ostatní malá	MojeTrida
Návěští, konstanty, zdroje dat	velkými písmeny	NAVESTI:, KONSTANTA, <STDIN>

V jednom programu se nemůže vyskytovat více než jedna proměnná stejného názvu a stejného typu ve smyslu datového skalár, pole, hash atd. To je logické, neboť bychom je nemohli rozlišit. Později zjistíme, že existuje něco jako rozsah platnosti proměnné, což nám pojem "stejný název proměnné" představí ještě v trochu jiném světle. Není totiž vůbec tak zřejmé, kdy proměnná je a kdy není dostupná. Tomuto se ale budeme věnovat později.

Práce s proměnnými

Do proměnné můžeme přiřazovat řetězce, čísla (ať už celá nebo desetinná), seznamy hodnot, ovladače nebo odkazy. Během provádění programu se může proměnná mezi číslem a řetězcem automaticky konvertovat. Perl je v tomto směru velmi benevolentní. Pokud na začátku programu použijeme proměnnou jako číslo, může být v jeho průběhu měněna na řetězec a obráceně. Dokonce nemusíme měnit její obsah. Jakýkoliv řetězec si Perl sám převede na číslo a naopak. V dalším dílu uvedeme pravidla, kterými se tato konverze řídí.

Délka informace uvnitř proměnné je omezena jen velikostí paměti.

Co je to přiřazení již víme. Uveďme si ještě několik příkladů:

```
$promenna = "normální řetězec";

#řetězec je uprostřed 2× zlomen. Do proměnné se uloží i escape znaky.
$promenna = "řetězec\nna 3\nřádcích";

#$promenna obsahuje číslo
$rok = 2005;

#desetinné číslo
$e = 2.71828;
```

```
#nespravné názvy proměnných. Interpret hlásí chybu.
$x&y = 2005;
$44 = 2006;
```
Operátor = není jediným přiřazovacím příkazem. Další poznáme v příštím dílu.

Uvození řetězců

Všechny hodnoty v uvozovkách jsou Perlem brány jako řetězec. Mimo uvozovek lze použít i apostrofy. Existují ještě jiné způsoby zápisu, ale těmi se nyní nemá smysl zatěžovat. Apostrofy se liší od uvozovek tím, že ignorují escape znaky a proměnné v řetězci. Podívejme se na rozdíl.

```
$promenna = 2005;
print 'retezec s escape sekvencemi \n\a a promennymi $promenna';
```
Uložme a spusťme program. Dostaneme kupodivu toto:

```
$./program.pl
retezec s escape sekvencemi \n\a a promennymi $promenna\n
$
```
Existují dvě výjimky, které řetězec uvozený apostrofy zobrazí jinak, než jak je napsaný. Těmi výjimkami jsou \' a \\, umožňující do řetězce vložit apostrof a zpětné lomítko. Nebýt nich, tak apostrof nevytiskneme.

Stejný příklad, jen zaměníme apostrofy za uvozovky:

```
$promenna = 2005;
print "retezec s escape sekvencemi \n\a a promennymi $promenna";
```
Po spuštění dostaneme:

```
$./program.pl
retezec s escape sekvencemi
a promennymi 2005
$
```
(Pokud nejde spustit, pravděpodobně chybí na začátku souboru #!/usr/bin/perlnebo nemáte správně nastavená práva.)

Kopie proměnných

Je možné mít na obou stranách operátoru přiřazení proměnné. Vyhodnotí se proměnná napravo (Perl si zjistí její hodnotu) a tato hodnota se přiřadí do proměnné nalevo.

```
$pi = 3.142;
$pi_kopie = $pi;
print $pi_kopie;
```
V takovém případě se vytvoří kopie původní proměnné.

Na jednom řádku můžeme přiřadit stejnou hodnotu i více proměnným. To asi příliš často nevyužijeme, ale je vhodné o tom vědět. Ve skutečnosti existuje něco jako návratová hodnota po operaci přiřazení.

```
$a = $b = 2005;
```
Nyní jak hodnota $a tak i $b obsahují hodnotu 2005.

Přepsání hodnoty proměnné

Když provedeme přiřazení do stejné proměnné dvakrát po sobě, původní obsah se ztratí a nahradí se novým.

```
$p = 1;
print $p; #1

$p = 2;
print $p; #2
```
Deklarace proměnných

Dosud jsme proměnné nikdy nedeklarovali. V Perlu je to možné. Je dobré toho využívat, protože v programu budeme mít větší pořádek. Bude potom na pohled jasnější, kde která proměnná existuje a kde už ne. Nebudeme se věnovat deklaracím do hloubky, to ponecháme na jindy. Uveďme jen, že k deklaraci slouží deklarátor my. Jeho parametrem je jméno proměnné, která bude platná v daném bloku programu.

Příkaz use strict;, uvedený na začátku programu spouští speciální mód, který nás hlídá abychom (mimo jiné) všechny proměnné deklarovali. Pokud zapomeneme nějakou proměnnou deklarovat, program zahlásí chybu.

Další užitečná věc je přepínač -w. Pokud spustíme Perl příkazem perl -w, budou se vypisovat varování, která mohou napomoci odstranit případné chyby. Přepínač lze připsat do prvního řádku programu.

```
#!/usr/bin/perl -w
use strict;

my $p;

$p = 3;
print $p;
```
Častější však bývá uvedení řádku use warnings;, který dělá podobnou věc.

```
#!/usr/bin/perl
use strict;
use warnings;

my $p;

$p = 3;
print $p;
```
V režimu strict nám Perl nedovolí použít proměnnou, aniž bychom ji deklarovali. Odstraníme-li řádek my $p; program vypíše chybové hlášení a skončí.

Režim strict se ruší příkazem no strict;. Ten budeme používat jen výjimečně. Pokračujme ale přesto v kódu.

```
no strict;

$q = 3;
print $q;
```

Proměnnou $q jsme teď definovat nemuseli.
Vícee o use strict;, deklaracích proměnných a případně jiných direktivách bude napsáno v dílech o modulech.

Perl (4) - Čísla a řetězce

Řetězce a čísla jsou v Perlu dva druhy vzájemně zaměnitelných hodnot. Obsah každé skalární proměnné můžeme interpretovat jako číslo i jako řetězec. Mezi řetězci a čísly totiž existuje automatická konverze. Budeme se též věnovat základním operátorům.

Perl datové typy příliš nerozlišuje. V zásadě je ale můžeme rozdělit na tři typy:

- skalární hodnoty
- pole skalárů
- hashe skalárů

Přitom poslední dva typy jsou pouze seznamy skalárních hodnot. Skalární hodnoty mohou být řetězci, čísly nebo odkazy. Již několikkrát zaznělo, že Perl se chová k datovým typům velmi benevolentně. Perl dokáže vyjádřit číslo jako řetězec a naopak, takže toto rozdělení nehraje tak velkou roli jako u striktnějších jazyků. V zásadě každou proměnnou můžeme interpretovat jako číslo nebo jako řetězec (když to je potřeba). Dokonce i odkaz, pole nebo hash. Těmi se ale budeme zabývat později.

Zápis čísel

Perl nerozlišuje mezi celými a racionálními čísly. Bere automaticky všechna čísla jako racionální - tedy jako jeden datový typ. Dělíme-li vzájemně dvě celá čísla, která jsou nedělitelná, řešením je racionální číslo (10 / 3 = 3.33333333333333). To je chování, které programátor zpravidla chce. Kdybychom to samé dělali v Céčku, výsledkem po dělení celých čísel musí být také celé číslo (10 / 3 = 3). Ve skutečnosti se dá Perl také přepnout na celočíselný režim pomocí příkazu use integer; a do racionálního režimu se vrátíme příkazem no integer. Potom Perl vrací vždy celé číslo.

```
#!/usr/bin/perl

print 10 / 3;#3.33333333333333

use integer;#odteď celočíselný režim
print 10 / 3;#3

no integer;#a zase racionální
print 10 / 3;#3.33333333333333
```

Charakteristickým znakem Perlu je možnost použít na cokoliv rozmanité způsoby zápisu. Zápis čísel není výjimkou. Celé číslo zapíšeme jednoduše jako posloupnost číslic. Zapisujeme-li desetinné číslo, desetinná část se odděluje tečkou, nikoliv čárkou. (Konvence zápisu se dle zvyklosti stát od státu liší, Perl vychází ze syntaxe v USA). Možný je zápis v semilogaritmickém tvaru (například číslo 9,4 . 10^3 v Perlu napíšeme jako 9.4e3).
V zápisu čísel existuje speciání syntaxe pro jiné než desítkovou soustavu:

Soustava	Zápis	Použitelné znaky
desítková	klasicky	0-9
osmičková	před číslem je 0	0-7
šestnáctková	před číslem je 0x	0-9, a-f, A-F
dvojková	Před číslem je 0b	0, 1

Jestliže použijeme jiné znaky než ty z uvedených intervalů, obdržíme většinou chybovou hlášku.
V zápisu lze používat takzvané unární operátory + a -. Je to znaménko + nebo - před číslem. Příkladem může být -18 nebo +95. Přívlastek unární se udává proto, že má takový operátor pouze jediný operand. Operandem je myšlena hodnota vcházející do operace. Mimo unární operátory existují v Perlu ještě binární a ternární.
Delší čísla mohou obsahovat pro přehlednost podtržítka. Při vyhodnocení jsou odstraněny.
Jako příklad si uveďme, jakými různými způsoby se dá napsat číslo 2345:

- 2345
- 2345.0
- 2345.
- 2.345e3
- 2.345E3
- 2.34_5e3
- 2_345
- 23____4_5
- +2345
- 04451
- 0x929
- 0b100100101001
- 0b100_100_101_001

Matematické operace

Proměnné můžeme sčítat, odčítat, násobit, dělit nebo s nimi jinak operovat. Zde je tabulka standardních matematických operátorů:

Operátor	Operace	Příklad	Výsledek
+	sčítání	9 + 15	21
-	odčítání	6 - 4	2
*	násobení	3 * 5	15
/	dělení	9 / 4	2.25

**	umocňování	3 ** 4	81
%	zbytek po celočíselném dělení	9 % 4	1

U umocňování je levý operand základem a pravý mocnitelem (exponentem). Takže zápis 3 ** 4 v Perlu je ekvivalentní matematickému zápisu 3^4. Pozor na to, že 3 ^ 4 znamená něco úplně jiného.
Zbytek po celočíselném dělení (**modulus**) se dá vysvětlit takto: levý operand vydělíme pravým a výsledné číslo zaokrouhlíme celočíselně dolů (to je výsledek celočíselného dělení). Zaokrohlené číslo vynásobíme pravým operandem a tento nový výsledek odečteme od levého operandu. Získaná hodnota je modulus.
Zde je několik triviálních příkladů:

```
$a = 1;
$b = 2;
$c = 4;

$soucet = $a + $b;    # výsledkem je 1 + 2 = 3
$rozdil = $c / $a;    # 4 / 1 = 4
$v1 = $b + $b ** $c;  # 2 + 2 ^ 4 = 18
$v2 = ($b + $b) ** $c;# (2 + 2) ^ 4 = 256

$v3 = 5 + 9 * 10;     # 95
$v4 = (5 + 9) * 10;   # 140
```

Stejně jako v matematice lze závorkovat tam, kde potřebujeme změnit prioritu operací.
Další operátory přiřazování
Přiřazování je základní operací a věnovali jsme mu již část minulého dílu. Vysvětlili jsme si, jak funguje operátor =. Ten přiřadí do proměnné určenou hodnotu. Dnes se podíváme na jiné přiřazovací operátory.
Tabulka přiřazovacích operátorů pro čísla s přepisem na standardní operátor =:

Operátor	Příklad	Přepis	Nová hodnota $p pro původní hodnotu $p = 10
=	$p = 3	$p = 3	3
+=	$p += 3	$p = $p + 3	13
-=	$p -= 3	$p = $p - 3	7
*=	$p *= 3	$p = $p * 3	30
/=	$p /= 3	$p = $p / 3	3.33333333333333
**=	$p **= 3	$p = $p ** 3	1000
%=	$p %= 3	$p = $p % 3	1
++	$p++	$p = $p + 1	11
--	$p--	$p = $p - 1	9

Pro názornost se podívejme, jak bychom mohli tyto operátory zapsat v programu. V proměnné $p je hodnota 10. Zdvojnásobme ji, přičtěme 12, umocněme dvěma a vydělme 4. Vzniklou hodnotu dělme 15 a zbytek po celočíselném dělení tiskněme na výstup. Napíšeme si trochu neohrabaný program, který to spočítá.

```
#!/usr/bin/perl

$p = 10; # V proměnné $p je hodnota 10
$p *= 2; # Zdvojnásobme ji,
$p += 12;# přičtěme 12,
$p **= 2;# umocněme na 2.,
$p /= 4; # a vydělme 4.
$p %= 15;# Vzniklou hodnotu dělme 15 a zbytek po celočíselném dělení

print "Řešením úlohy je číslo $p.\n";# tiskněme na výstup
```

Inkrementace, dekrementace
inkrementace je zvýšení hodnoty v proměnné o 1. Dekrementace je naopak snížení hodnoty v proměnné o 1. Jak funguje inkrementace?

```
$p = 10;
$p++;   # Hodnota $p se zvýší o 1
print $p# Vytiskne hodnotu 11 - tedy hodnotu o 1 větší než byla původní hodnota $p
```

Dekrementace je totéž obráceně (1 se odečítá) a zapisuje se operátorem --.

```
$p = 10;
$p--;
print $p # tiskne 10 - 1 = 9
```

Tyto zápisy jsou tedy ekvivalentní:

```
$p = $p + 1;
$p += 1;
$p++;
```

Výhodou operátoru inkrementace je zejména přehlednost.
Postfixová, prefixová a infixová notace operátorů
Existuje několik druhů zápisu operátorů podle toho, v jakém pořadí se uvádějí operátory a operandy. I na pořadí zápisu samozřejmě výsledek závisí.
Způsoby notace si ukážeme na operátorech ++ a --. Ty lze totiž napsat i před proměnnou. To má význam jen v případě, kdy nepoužijeme zápis osamoceně, ale například při přiřazení. Nejen, že inkrementace ovlivní hodnotu proměnné, ale celá operace (stejně jako každá jiná operace) má takzvanou návratovou hodnotu, kterou můžeme také někam přiřadit. Použijeme-li zápis před proměnnou (prefixový zápis), bude hodnota nejdříve inkrementována, až poté přiřazena (tj. až poté se vyhodnotí návratová hodnota). Při zápisu za proměnnou (postfixový zápis), bude nejprve přiřazena a až poté inkrementována.

Podívejme se na příklad pro postfixový zápis:

```
$a = 3;
$b = $a++;#do $b se přiřadí hodnota proměnné $a a až potom se zvýší $a o 1. $b tedy bude mít hodnotu 3

print $a; #4
print $b; #3
```

u Prefixového zápisu je výsledek jiný:

```
$a = 3;
$b = ++$a;#tady je to naopak. Nejdříve se $a zvýší o 1 a až potom se hodnota $a přiřadí do $b

print $a; #4
print $b; #4
```

Takto je to u inkrementace a dekrementace, ale každý operátor funguje svým způsobem. Nemá smysl se tím nyní příliš zabývat, protože zápis bývá většinou intuitivní.

Infixová notace znamená, že je operátor mezi operandy. Například 5 + 9. U unárních operátorů nemá význam.

Řetězce

Proměnné mohou uchovávat také řetězce.

Dostupnými operacemi s textem v Perlu, které si dnes představíme, je zřetězení a opakování. Zřetězení používá operátor tečky a jeho výsledkem je spojení řetězců z operandů na levé a pravé straně.

```
$a = "larry" . "wall";
print $a; # vytiskne larrywall

$jmeno = "larry";
$prijmeni = "wall";

$cele_jmeno = $jmeno . " " . $prijmeni;

print $cele_jmeno; # vytiskne larry wall
```

Operátor opakování se se bude hodit studentům. Chceme-li 100x vypsat nějaký řetězec, použijeme zápis:

```
print "Budu nosit domácí úkoly.\n" x 100;
```

Jde vlastně o autozřetězení, kdy několikrát postupně zřetězujeme s původním řetězcem. Zkusme spustit program:

```
$./program.pl
Budu nosit domácí úkoly.
Budu nosit domácí úkoly.
Budu nosit domácí úkoly.
Budu nosit domácí úkoly.
Budu nosit domácí úkoly.
..... (ještě 94×)
Budu nosit domácí úkoly.
$
```

Stejně jako u řetězců můžeme využít několik přiřazovacích operátorů.

Operátor	Příklad	Přepis	Nová hodnota $p pro původní hodnotu $p = "abc"
=	$p = "xxx"	$p = "xxx"	"xxx"
.=	$p .= "def"	$p = $p . "def"	"abcdef"
x=	$p x= 3	$p = $p x 3	"abcabcabc"

Konverze mezi čísly a řetězci

Co se stane, když provádíme numerické výpočty s řetězci a naopak - řetězcové operace s čísly? Již jsme naznačili, že Perl automaticky převede řetězec na číslo nebo číslo na řetězec.

Pravidla konverze jsou prostá. Jestliže na začátku řetězce není číslo, převede Perl řetězec na 0. V opačném případě je řetězec převeden na nejdelší číslo, kterým řetězec začíná a ostatní znaky jsou vypuštěny. Přitom se nebere ohled na bílé znaky na začátku.

```
print 10 + "265pps";
# vytiskne 275

print 10 + "pps";
# vytiskne 10

print "2k5" + "123px";
# vytiskne 125

print "2e5" + "123px";
# tady je zrada! Vytiskne 200123, protože "2e5" je konvertováno na 2e5 = 2 * 10^5, nikoliv na 2

print " 2k5" + "123px";
# bílý znak na začátku (mezera) jako by nebyl, Perl ho ignoruje a řetězec konvertuje na 2.
#Výsledkem tedy je 125
```

Konverze z čísla na řetězec je intuitivní. Získaný řetězec bude prostě obsahovat číslo.

```
print 12 . 6;
#vytiskne 126

print "retezec" . 6;
#vytiskne retezec6
```

Seznam operátorů a jejich priorita

V matematice bývá zvykem, že některé operace se vykonávají přednostně. Například umocňování má přednost před násobením a násobení před sčítáním. Pokud výraz obsahuje operátory stejné priority, zpracováváme ho zleva doprava.

Pravidla pro pořadí vykonávání jsou nezbytná, neboť výsledek nemusí být při různém vyhodnocení jednoznačný. K tomu, která operace se kdy vykoná, slouží v Perlu tabulka priorit. Ty operace, které již známe nebo poznáme brzy (horizont dvou dílů), jsou zvýrazněny zeleně:

Operátor	Operace	Asociativita	Arita
print, sort atd. (nejvyšší priorita)	Operátory pro seznamy (levé)	zleva	
->	Dereference	zleva	2
++, --	Inkrementace, dekrementace	není	1
**	Umocňování	zprava	2
+, -, !, \	Unární plus, minus, not (logické), odkaz	zprava	1
=~, !~	Operace pro práci s regulárními výrazy	zleva	2
*, /, %, x	Násobení, dělení, modulus, opakování	zleva	2
+, -, .	Plus, minus, spojení řetězců	zleva	2
>> <<	Bitový posun	zleva	2
rand atd.	Pojmenované unární operátory	není	1
<, <=, >=, >, lt, le, ge, gt	Porovnávání	není	2
==, !=, <=>, eq, ne, cmp	Porovnávání	není	2
&	and (bitové)	zleva	2
|, ^	or, xor (bitové)	zleva	2
&&	and (logické)	zleva	2
||	or (logické)	zleva	2
.., ...	Operátor rozsahu	není	2
?:	Operátor podmínky	zprava	3
=, +=, -=, *=, /=, .=, x=, **=, %=	Přiřazovací operátory	zprava	2
=>, ,	Čárka	zleva	2
open atd.	Operátory pro seznamy (pravé)	není	
not	not (logické)	zprava	1
and	and (logické)	zleva	2
or, xor (nejnižší priorita)	or (logické)	zleva	2

Čím výše je operátor, tím má vyšší prioritu - a tedy vykoná se přednostně oproti operacím s operátorem pod ním.

Směr vykonávání (asociativita) určuje, jak se bude vykonávat operace se stejnou prioritu. Jsou 3 možnosti:

- výraz se bude vyhodnocovat zleva
- výraz se bude vyhodnocovat zprava
- asociativita není pro některé operace důležitá nebo není možná

Přirozenost požadavku asociativity si lze uvědomit na výrazu 8 / 4. Výsledkem je 2, nikoliv 0.5 (8 dělíme čtyřmi, ne opačně, protože výraz se vykonává zleva).

Vlastnosti operátorů jsou tedy obecně priorita, asociativita, počet operandů (arita), případně i typ a kontext operandů. Chceme-li změnit prioritu, použijeme kulaté závorky. Lze je používat i tam, kde nejsou nutné (třeba když si nejsme prioritou jistí a nechce se nám hledat, ale je to vhodné i tam, kde by si někdo, kdo bude kód v budoucnu číst, taky nemusel být jistý).

Perl (5) - Podmínky

Ukážeme si jeden z prostředků, který je nezbytný v každém menším programu. Díky podmínkám se lze v jistém bodě vykonávání programu rozhodnout mezi různými větvemi, kudy se dál budeme ubírat.

Podmínka rozhoduje, kudy se má skript vykonávat. Díky podmínkám nemusejí příkazy probíhat od prvního k poslednímu, ale lze části vynechávat. Podmínka tedy určuje, zda má být určitý příkaz nebo blok kódu (dále v článku) vykonán. Spojením několika podmínek pak získáváme různé programové větve, z nichž se provede pouze jedna.

Uvedeme si na začátek příklad podmínky v našem jazyce, která by mohla názorněji osvětlit jejich smysl.

```
#!/usr/bin/nase_rec

..... (nějaký kód) .....

jestliže (bylo hlasováno){
Tiskni "Již jste hlasoval!";
}

jestliže (nebylo hlasováno){
Tiskni "Ok, můžete hlasovat:";
..... (formulář na hlasování) .....
}
..... (nějaký další kód) .....
```

Zde je pochopení intuitivní. Cílem programu je zabezpečit, aby nehlasoval jeden člověk vícekrát. Proto si program nejdříve musí zkonrolovat, zda už stejný člověk nehlasoval. Pokud zjistí že ano, vypíše program Již jste hlasoval!. V opačném případě zobrazí hlasovací lístek.

Tak funguje podmínka i v Perlu. Stačí se jen naučit, jak ji přeložit do jeho syntaxe.

Blok příkazů

Ještě předtím ale vysvětleme pojem **blok příkazů**. Jeho význam tkví v seskupení několika příkazů tak, abychom s nimi mohli najednou pohodlně nakládat. Označuje se složenými závorkami a struktura vypadá takto:

```
{
příkaz1;
příkaz2;
příkaz3;
...
příkazn;
}
```

Za posledním příkazem bloku je středník nepovinný (středník odděluje, nikoliv zakončuje příkazy), ale je dobré ho psát pro případ, že bychom ho někdy chtěli editovat. Za složenou koncovou závorku se středník nepíše.

Blok je nyní pro nás skupina příkazů, které budou podmínkou (později poznáme jiné strukturované příkazy, pro které se blok také používá) vykonány. Bezprostředně za podmínkou (testem) jestliže (bylo hlasováno) v příkladu můžeme právě takový blok nalézt. Vše, co je mezi { a } bude vykonáno, byl-li test úspěšný. V opačném případě bude blok přeskočen a skript pokračuje za ním.

Pravda (true) a nepravda (false)

Z každého výrazu, to jest z každé hodnoty po průchodu libovolnou operací, získáme nějakou hodnotu novou. Tato výsledná hodnota může být dvou typů. Buď true nebo false. Podle typu hodnoty se pak vyhodnocuje například právě test podmínky. V našem příkladu jsme za klíčové slovo jestliže napsali do jednoduchých závorek test. Co to ale test je a jak je vyhodnocován? Již bylo naznačeno, jak tyto dvě informace souvisí. Je-li výsledek testu pravdivý (true), bude následující blok příkazů vykonán. Bude-li výsledkem false, bude naopak přeskočen.

Jaká jsou pravidla pro určování, kterého typu je výraz (nebo-li hodnota nebo test)? Podmínka je false v případě, že je test (ještě jednou připomeňme, že to je prostě nějaký výraz) vyhodnocen jako:

- prázdný řetězec ("")
- nula (ať 0 nebo "0"), což je díky konverzi totéž
- nedefinovaná hodnota (když je testem proměnná, do které jsme buď nic nepřiřazovali nebo přiřazovali nedefinovanou hodnotu)

Ve všech ostatních případech je test vyhodnocen jako true.

Podmínka if

Už víme, kdy je test vyhodnocován jako true a kdy jako false. Poslední informací, kterou potřebujeme je, že podmínku zapisujeme slovem if. Nyní se můžeme podívat na několik příkladů podmínek.

```
if (0){
print "1. blok nebude vykonán: 0 je vyhodnocena jako false\n";
}

if (""){
print "2. blok nebude vykonán: prázdný řetězec je vyhodnocen jako false\n";
}

if (1){
print "3. blok bude vykonán: 1 je vyhodnocena jako true\n";
}

if ("nejaky text"){
print "4. blok bude vykonán: neprázdný (nenulový) řetězec je vyhodnocen jako true\n";
}

if (1 + 2){
print "5. blok bude vykonán: 1 + 2 = 3 je vyhodnoceno jako true\n";
}

if ($promenna_do_ktere_jsme_nic_nepriradili){
print "6. blok nebude vykonán: proměnná, o které jsme se nikde nezmínili má nedefinovanou hodnotu\n";
}

$promenna = 2005;

if ($promenna){
print "7. blok bude vykonán: v proměnné je uložena hodnota 2005, která je true\n";
}
```

Uložme si tento kód do souboru a zkusme spustit. Tam, kde jsou testy pravdivé, budou vytišteny příslušné zprávy.

Podmínka s alternativou if...else

Konstrukce if...else se používá k vytváření podmínek s dvěma větvemi - if větev a else větev. Pokud je pomínka vyhodnocena jako false, automaticky se provede blok kódu, uvedený za else. Pokud je test true, vykoná se blok kódu za if a druhý blok se již nevykoná.

```
$promenna = 2005;

if ($promenna){
print '1. blok bude vykonán, pokud je $promenna vyhodnocena jako true';
}
```

```
else{
print '2. blok bude vykonán, nebude-li vykonán blok 1. (to nastane nebude-li
$promenna vyhodnocena jako true, ale jako false)';
}
```

Tento kód bychom mohli ekvivalentně přepsat jen pomocí jednoduchých podmínek. Poznamenejme, že uvedení vykřičníku obrací typ hodnoty.

```
$promenna = 2005;

if ($promenna){
print '1. blok bude vykonán, pokud je $promenna vyhodnocena jako true';
}
if (!$promenna){
print '2. blok bude vykonán, nebude-li vykonán blok 1. (to nastane nebude-li
$promenna vyhodnocena jako true, ale jako false)';
}
```

Z tohoto přepisu je vidět, co se stane, zkusíme-li změnit hodnotu testované proměnné na false.

Podmínka if...*n**elsif...(else)

Fakticky ale můžeme do podmínky přidat libovolný počet větví. V takovém případě se prostřední větve uvádějí za slovo elsif. Jak probíhá vyhodnocování? Nejprve se testuje hodnota v testu za if. Není-li true (blok se nevykoná), vyhodnotí Perl test za prvním slovem elsif. Není-li ani ten úspěšný, vyhodnocuje Perl další větev elsif a takto pokračuje, dokud není některý test vyhodnocen jako true. Stane-li se tak, je následující blok vykonán a ostatní větve přeskočeny. Větví elsif můžete použít libovolné množství. A jestliže testy ve všech větvích dopadnou jako false, vykoná se blok za větví else (pokud je uvedená).

Podívejme se opět na úsek kódu.

```
$definovana_promenna = 2005;

if ($nejaka_promenna){
print "1. blok nebude vykonán: proměnná není definována\n";
}
elsif($dalsi_promenna){
print "2. blok nebude vykonán: ani tato proměnná není definavána\n";
}
elsif($jeste_dalsi_nedefinovana_promenna){
print "3. blok nebude vykonán: co dodat\n";
}
elsif($definovana_promenna){
print "4. blok bude vykonán: test byl vyhodnocen jako true\n";
}
elsif(0){
print "5. blok nebude vykonán: všechny větve za úspěšně vykonaným
4. blokem budou předskočeny, navíc 0 je false\n";
}
elsif(3.1416){
print "6. blok nebude vykonán: i přesto, že test by byl true. Je
totiž také přeskočen\n";
}
```

Pro úplnost ještě uveďme příklad s přítomnou větví else:

```
$definovana_promenna = 2005;

if ($nejaka_promenna){
print "1. blok nebude vykonán: proměnná není definována\n";
}
elsif($dalsi_promenna){
print "2. blok nebude vykonán: ani tato proměnná není definavána\n";
}
elsif(""){
print "3. blok nebude vykonán: ještě jednou false\n";
}
else{
print "4. blok bude vykonán: to proto, že doteď žádný test nebyl
vyhodnocen jako true\n";
}
```

Podmínkový operátor ?...:

Podmínkový operátor dělá totéž jako podmínková konstrukce. Hodí se na spíše u konstruování výrazů (než příkazy) a zejména u krátkých podmínek se tento zápis velmi často používá. Je to často kvůli přehlednosti výhodnější než podmínková konstrukce a lze pomocí něj simulovat i celou konstrukci if...*n**elsif...(else). Obecný zápis vypadá takto:

test ? *příkaz_v_případě_true* : *příkaz_v_případě_false*;

test ? *výraz_v_případě_true* : *výraz_v_případě_false*;

Pro srovnání, if funguje takto:

```
if (test){
příkaz_v_případě_true;
}else{
příkaz_v_případě_false;
}
```

Lze konstruovat i více podmínkových větví, což se nezřídka používá.

test1 ? *výraz_v_případě_test1* :
test2 ? *výraz_v_případě_test2* :

```
test3 ? výraz_v_případě_test3 :
výraz_jinak
;
```

Uveďme dva konkrétní příklady, které dělají totéž. První není příliš vhodně napsaný, protože pro podmiňování příkazů působí podmínkový operátor dost neohrabaně. Zato v tom druhém tiskneme návratovou hodnotu operátoru, což se obvykle očekává.

```
1 ? print "TRUE" :  print "FALSE";
print  1 ? "TRUE" :  "FALSE";
```

Protože 1 je true, vytiskne se TRUE. Teď ještě ukažme něco méně samoúčelného. Sice zatím neznáme klíčové slovo return, avšak často se tento operátor využívá právě tam.

```
return $jmenovatel ? $citatel / $jmenovatel
: undef
;
```

Podmínka unless...(*n**elsif)...(else)

Mimo výše zmiňovaných existuje podmínka s opačným významem než má if a příkazy s podmínkou na konci. Na začátku řekněme, že většinou nebývá dobrým nápadem tyto konstrukce používat, protože často zbytečně znečitelňují program. Zbytek článku klidně přeskočte. Obzvlášť pro začínající je možná lepší, když tyto konstrukce znát nebudou.

Podmínka unless je negované if. Česky se dá if vyjádřit slovem jestliže, unless slovy jestliže ne. Kód ve větvi unless se provede, vyhodnotí-li se test jako false. unless vzniklo jako zjednodušení pro if, kdy testujeme nějaký negovaný výraz.

V těle unless lze použít jak elsif, tak else. Použití elsif může být poněkud matoucí. unless totiž provede blok, je-li test false, a elsif ho provede, je-li test true.

Podívejme se na příklad použití unless, který se ovšem dá jednoduše převést na konstrukci if...else):

```
$definovana_promenna = 2005;

unless ($definovana_promenna){
print "1. blok nebude vykonán: test vrátí true (proměnná je
definována) a vykonává se větev else\n";
}
else{
print "2. blok bude vykonán\n";
}
```

Na závěr ještě jednou varujme před používáním unless. V naprosté většině případů je použití matoucí a je lepší dát přednost klasickému if.

Podmínka za příkazem

Syntakticky je správně též prohození bloku příkazů a podmínky. Následující kód bude fungovat.

```
$bylo_hlasovano = 1;

print "Díky za hlas!" if ($bylo_hlasovano);
print "Nehlasoval jste!" unless ($bylo_hlasovano);
```

I na tomto místě je třeba důrazně upozornit, že nebývá vhodné umístit podmínku za příkaz. Tento typ podmínky se používá výlučně pro příkazy typu last, next, redo a občas return. V jakémkoliv jiném případě je mnohem lepší použít klasické podmínky.

Podmínka za blokem

Stejné jako podmínka za příkazem až na to, že se místo jednoho příkazu vykoná celý blok. Před blok musíme uvést klíčové slovo do. Lze použít jak konstrukci do...if, tak i do...unless.

```
do{
$x = 2 ** 10;
print $x;
}if (1);
```

Za testem musí být středník. To proto, že za ním už není blok, ale následují další příkazy.

I tento odstavec je pouze informativní a z důvodu přehlednosti bychom neměli takové konstrukce používat, i když je to možné.

Podmínka case a switch

Jednou z mála běžných konstrukcí, které Perl standardně nenabízí je case, která testuje týž výraz na různé hodnoty. Lze ji ale napsat řadou způsobů pomocí operátorů. Operátorové konstrukce se dají často použít ve speciálních případech místo konstrukce if...*mnoho**elsif...(else), která nebývá příliš šťastným řešením.

```
$p = 2;

CASE:{
$p == 1 and do{print '$p je jedna!'; last CASE;};
$p == 2 and do{print '$p je dva!';   last CASE;};
$p == 3 and do{print '$p je tři!';   last CASE;};
$p == 4 and do{print '$p je čtyři!'; last CASE;};
do{print '$p není 1, 2, 3 ani 4!';}; #Tento do blok bude
proveden v případě, že nevyhověla žádná předešlá podmínka
}
```

Platí-li pravdivostní výraz na začátku řádku, vykoná se blok za klíčovým slovem do a dál program pokračuje za blokem označeným CASE. Využíváme zde triku s operátorem and, který nevyhodnocuje druhý operand, pokud již po vyhodnocení prvního je jasné, že výsledkem bude false. Značení bloků a klíčové slovo last probereme až v některém z dalších dílů.

Ani předchozí postup však není zrovna nejpohodlnější. Dalším a nejjednodušším způsobem zavedení této funkcionality je proto použití pragmy feature.

```
use feature qw(switch);
```

Po uvedení takového příkazu získáme novou konstrukci given...when. Stačí nám tak napsat následující.

```
given ($p) {
when(1){
print '$p je jedna!';
}
when(2){
print '$p je dva!';
```

```
}
when(3){
print '$p je tři!';
}
when($_ > 3){
print '$p je větší než 3!';
}
default{
print '$p je něco jiného!';
}
}
```

Perl (6) - Pravdivostní výrazy

Výrazy jsou základním kamenem programovacích jazyků. Je jimi cokoliv, co má nějakou hodnotu. Jinými slovy, výrazy se vyskytují v podstatě na každém řádku kódu.

Každý zápis, jehož vyhodnocením získáme nějakou hodnotu, je výrazem. Připomeňme, že s výrazy se setkáváme již od počátku seriálu. Například konstanty (tedy samotná čísla nebo řetězce), proměnné, příkaz print jsou všechno výrazy. Dnes si uděláme jejich systematičtější přehled a značně rozšíříme arzenál operátorů, kterými se určuje, jak se má výraz vyhodnocovat.
Trochu nepřesně lze říci, že výraz se středníkem na konci se stává příkazem. Proto, pokud jakémukoliv příkazu odejmeme středník, získáme výraz. Příkaz tedy má také nějakou hodnotu - a to i když není potřebná. Například, k čemu nám je, že příkaz print vrátí hodnotu 1? Většinou k ničemu, nicméně z principu je vyhodnocení nezbytné, neboť musí jít vyhodnotit každý výraz. Dokonce lze říci, že program je posloupností výrazů a středníků.
Pravdivostním výrazem myslíme výraz, který se vyhodnocuje na typ pravda / nepravda (například test u podmínek). Ve skutečnosti je ale každý výraz pravdivostní. Vyhodnocením libovolného výrazu je totiž možné získat pravdu nebo nepravdu. I přesto se teď zaměřme jen na výrazy tvořené relačními, logickými a pro zajímavost i bitovými operátory.

Způsoby vyhodnocování

Začněme ukázkou. Co vytisknou následující příkazy?

```
print 9 > 6;
print 9 > 18;
```

V prvním případě bylo vytisknuto 1, ve druhém případě "" (tedy prázdný řetězec). Proč? Číslo 9 je skutečně větší než 6. Výraz se tedy vyhodnotí jako pravda a tiskne se hodnota true. Protože Perl neobsahuje žádný speciální booleovský datový typ, uchovávající pouze hodnoty false a true, tiskne se místo true hodnota 1. Naopak prázdný řetězec značí false.
Obvyklou konvencí je, že úspěšně vykonaný příkaz vrátí pravdivou hodnotu a neúspěšně vykonaný nepravdivou. Zkuste na základě tohoto a předchozího odstavce určit, co se vytiskne následujícím příkazem:

```
print print "";
```

Test porovnání dvou hodnot můžeme užít v podmínce:

```
$a = 3;
$b = 2;

if ($a > $b){# $a je skutečně větší než $b (3 > 2). Výsledkem tohoto výrazu je 1 - tedy true
print '$a je větší než $b';
print "\n";
}
else{
print '$b je větší než $a';
print "\n";
}
```

Všimněme si, že jsme použili apostrofy jako označení řetězce. To proto, aby se místo $a a $b nevytiskly jejich hodnoty. (i když by to v tomto případě zas tolik nevadilo)
Zkusme ještě další příklad:

```
if ($zisk < 0){
if ($zakazky == 0){
$bankrot = 1;
}
else{
$zamestnancu *= 3/4;
}
}
elsif ($zisk > 1e7){
$plat *= 1.05;
}
```

Operátory pro porovnávání (relační operátory):

Operace	Operátor pro čísla	Operátor pro řetězce	Celý název
je rovno	==	eq	EQual
není rovno	!=	ne	Not Equal
je menší než	<	lt	Less Than
je větší než	>	gt	Greater Than
je menší nebo rovno než	<=	le	Less than or Equal
je větší nebo rovno než	>=	ge	Greater than or Equal
je menší, rovno nebo menší než	<=>	cmp	CoMParison

Výsledky porovnávání čísel jsou intuitivní - řídí se zákony matematiky.

Ale jak se porovnávají řetězce? Perl vezme první znak levého operandu a porovná jej s pravým. To samé s druhým atd. Každý znak bere jako číslo, určené jeho ASCII hodnotou. První odlišnost, kterou Perl objeví rozhodne o výsledku porovnání. Není-li žádná odlišnost, výrazy jsou si rovny.

```
$a = 4;
$b = 8;

#výraz $a == $b bude vyhodnocen jako false. 4 není rovno 8
if ($a == $b){
print "$a == $b\n";
}

#výraz $a != $b bude vyhodnocen jako true. 4 není rovno 8.
if ($a != $b){
print "$a != $b\n";
}

#výraz $a > $b bude vyhodnocen jako false. 4 není větší než 8.
if ($a > $b){
print "$a > $b\n";
}

#výraz $a != $b bude vyhodnocen jako true. 5 * 4 je větší než 2 * 8.
if (5 * $a >= 2 * $b){
print "5 * $a >= 2 * $b\n";
}

#'a' je v ASCII tabulce paradoxně až za 'A', takže je větší.
#Platí, že jakékoliv malé písmeno je vždy větší než jakékoliv velké písmeno.
if ("a" gt "A"){
print "a > A\n";
}

#u číselného porovnávání se Perl řídí matematicky správně, u řetězců to ale zdánlivě neplatí.
#"1slon" bude vyhodnocen jako 1, "slon" jako 0. 1 > 0
if ("1slon" == "slon"){
print "1slon > slon\n";
}

#slon sice není 1slon, ale je to myš. Oba výrazy budou vyhodnoceny jako 0, tudíž se rovnají.
if ("slon" == "myš"){
print "slon == myš\n";
}
```

Operátor <=> vrací hodnotu 1, je-li pravý výraz větší než levý, hodnotu -1, je-li levý výraz větší než pravý nebo hodnotu 0, jsou-li výrazy na obou stranách stejné.

```
$a = 1;
$b = 2;

print $a <=> $b; #-1
print $b <=> $a; #1
print $a <=> 1;  #0
```

Logické operátory

Obyčejné testy většinou nestačí. Občas potřebujeme podmínku typu je-li a > b a zároveň a > d. Od toho jsou logické operace.

Operace	Operátor	Céčkovský zápis
Konjunkce (logické a)	and	&&
Disjunkce (logické nebo)	or	||
Negace	not	!
Defined-or		//

Céčkovský zápis operátorů má stejný význam jako operátory, liší se jen v prioritě.

Konjunkce je splněna, jsou-li oba její operandy true. Disjunkce je splněna, je-li alespoň 1 její operand true (je-li true první operand, druhý je přeskočen a už se nevyhodnocuje - výsledek by to neovlivnilo; této vlastnosti můžeme použít při konstrukcí podmínek bez řídících konstrukcí).

a	b	a and b	a or b
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Defined-or dělá totéž jako logické nebo až na to, že levý operand je testován na definovanost (nikoliv na pravdivost). Nebudeme se jím zatím zabývat, avšak jde o užitečný nástroj pro konstrukci podmínek bez řídících konstrukcí.

Negace vrací opačnou hodnotu než její operand. Jde o unární operátor.

```
$a = 4;
```

```
#1 je true, $a také, proto je test vyhodnocen jako true
if (1 and $a){
print "1. test\n";
}

#1 je true, ale prázdný řetězec ne, proto je test vyhodnocen jako false
if (1 and ""){
print "2. test\n";
}

#1 je true, výsledek je taky true
if (1 or ""){
print "3. test\n";
}

#1 je true, výsledek je taky true
if (1 or 2){
print "4. test\n";
}

#1 je true, negace obrací výraz, takže test je false
if (!1){
print "5. test\n";
}

#1 and "MP" je true, potom je true i 0 or (1 and "MP"). Tento celý výraz je operandem
#druhého or a protože je true, je i tato operace or true. Výsledek negujeme a dostáváme false.
if (!(0 or (1 and "MP") or ($a and ""))){
print "6. test\n";
}
```

Přiřazení jako test podmínky

Jako test se hojně používá i přiřazení (nejčastěji ve spojení se čtením dat ze souboru, databáze apod.). Přiřazujeme-li hodnotu true, je i celý test true. Přiřazujeme-li hodnotu false, je test také false.

```
if ($a = 0){ #0 je false
print "TRUE\n";
}
else{
print "FALSE\n";
}
```

Připomeňme, že to není porovnávání. To bychom museli použít operátor ==.

Příklad - absolutní hodnota

Zkusme napsat program, který vypíše absolutní hodnotu čísla. Stačí vynásobit hodnotu číslem -1, pokud je záporná.

```
#!/usr/bin/env perl
use strict;

my $hodnota = -3;

if ($hodnota < 0){
$hodnota *= -1;
}

print "$hodnota\n";
```

Chceme-li, dá se podmínka pomocí podmínkového operátoru zahrnout přímo do funkce print.

```
#!/usr/bin/env perl
use strict;

my $hodnota = 3;

print $hodnota < 0 ? $hodnota * -1 : $hodnota . "\n";
```

V tomto případě se nejdříve vyhodnotí výraz $hodnota < 0 ? $hodnota * -1 : $hodnota a poté se tiskne.

Bitové operátory

Ještě se zmiňme o dalším typu operací - bitových. Pokud vás nezajímají, klidně tuto část přeskočte, protože se s nimi běžný člověk nesetkává často.

Bitové operace mají význam jen v případě celočíselných operandů (zadáme-li desetinné číslo, Perl si desetinnou část odřízne). Podívejme se na tabulku bitových operátorů.

Operátor	Operace	Počet operandů
~	bitová negace (bitové not)	1
&	bitový součin (bitové and)	2
|	bitový součet (bitové or)	2
^	exkluzivní bitový součet (bitové xor - výlučné nebo)	2
>>	bitový posun doprava	2
<<	bitový posun doleva	2

Bitové operátory fungují tak, že je operand (nebo operandy) převeden do dvojkové soustavy. U operátorů &, | a ^ Perl postupně porovnává bity na stejných pozicích.
Následující tabulka ilustruje význam bitových operátorů:

bit a	bit b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	1	0

bitové not
Každý výsledný bit je doplňkem bitu operandu do 1. Funguje tudíž na různých počítačích jinak (podle rozsahu čísla). Zkusíme negaci pro 110101110000101:

```
0b11111111111111111111111111111111
$a  0b00000000000000000110101110000101
----------------------------------
~$a 0b11111111111111111001010001111010
```

bitové and
Pokud jsou v pravém i levém operandu bity na stejných pozicích 1, výsledkem na daném bitu je 1. V ostatních případech je výsledkem 0.

```
$a = 1631;
$b = 55;
print $a & $b; #23
```

1631 a 55 vyjádříme ve dvojkové soustavě. Odtud je zřejmé, jak se k výsledku došlo.

```
$a      0b11001011111    1631
$b      0b     110111     55
          -------------    ----
$a & $b 0b00000010111      23
```

bitové or
Pokud jsou v operandech oba bity na stejných pozicích 0, výsledkem bitu na dané pozici ve výsledku je 0. Vyskytne-li se v jednom z nich 1 (tedy všechny ostatní případy), výsledkem je 1.

```
$a = 1322;
$b = 986;
print $a | $b; #2042
```

To je zřetelné odtud:

```
$a      0b10100101010    1322
$b      0b 1111011010     986
          -------------    ----
$a | $b 0b11111111010    2042
```

bitové xor
Číslice 1 a 0 nebo 0 a 1 se vyhodnotí jako 1, v ostatních případech je výsledkem 0.

```
$a = 30549;
$b = 6806;
print $a ^ $b; #28099
```

Bitově:

```
$a      0b111011101010101    30549
$b      0b001101010010110     6806
          -----------------    -----
$a ^ $b 0b110110111000011    28099
```

bitový posun
Operátor >> umazává zprava bity, operátor << (zleva) přidává nuly.

```
$a = 61;
$b = 2;
print $a >> $b; #15
print $a << $b; #244
```

Bitově:

```
$a       0b  111101    61
$b             2      2
           ----------    ---
$a >> $b  0b    1111     15
$a << $b  0b11110100    244
```


Dosud jsme psali jen programy, které s uživatelem komunikovaly jednostranně. Je čas naučit se pracovat s ovladači umožňujícími i standardní vstup.

Ovladač (manipulátor) slouží ke čtení dat z nějakého zdroje (nebo-li zpřístupňuje zdroj dat) nebo naopak k zasílání dat. Před jakoukoliv komunikací tedy je třeba vytvořit ovladač. V Perlu jsou některé ovladače vytvořené již automaticky. Mezi ně patří mimo jiné STDIN (čte data ze standardního vstupu - klávesnice) a STDOUT (tiskne na standardní výstup; tento ovladač jsme nevědomky používali, když jsme tiskli pomocí print). Na ovladačích je hezké to, že zobecňují práci se zdroji dat. Ze souboru tak čteme stejným způsobem jako ze standardního vstupu.
Čtení dat z klávesnice
Čtení z již otevřeného ovladače je přímočaré. Zkusme si spustit následující program:

```
print "Jak se jmenuješ?\n";

$jmeno = <STDIN>;
```

```
chomp $jmeno;

print "Jsi $jmeno!\n";
```

Do proměnné $jmeno přiřazujeme hodnotu získanou ze standardního vstupu - STDIN. Chceme-li data získávat, musíme dát STDIN mezi < a >. Program na nás čeká, až něco napíšeme a odentrujeme. V té chvíli se do proměnné $jmeno uloží získaná hodnota a skript normálně pokračuje dál. Stojí za to připomenout, že do proměnné $jmeno se ukládá i ENTER v podobě znaku konce řádku.

Na dalším řádku voláme funkci chomp (zatím nevíme, co to funkce je, ale můžeme ji chápat podobně jako operátor), které předáváme jako parametr proměnnou $jmeno. Funkce chomp odstraňuje z konce řetězce znak konce řádku, pokud tam je. Zkusme tento řádek smazat a rozdíl bude ihned zřejmý.

Perl umožňuje tyto dva řádky zkrátit do jednoho:

```
chomp  ($jmeno = <STDIN>);
```

Poznámka: o operátoru <> se v literatuře někdy mluví jako o diamantovém operátoru nebo operátoru dvojité šipky.

Standardní výstup

Funkci print používáme, aniž bychom se starali o to, kam posílá data. Vždy se objevily na standardním výstupu. Tak se print chová implicitně. My ale můžeme ovladač změnit. Ve skutečnosti print přijímá ještě jméno ovladače výstupu. Následující zápisy jsou ekvivalentní.

```
print "Toto se tiskne na výstup\n";
print STDOUT "Toto se tiskne na výstup\n";
```

Jak vidíme, stačí před tisknutý řetězec uvést ovladač. Od řetězce je oddělen jen bílým znakem. Protože nevrací žádná data, nepíše se mezi <>.

Chceme-li tisknout text do souboru, je postup úplně stejný.

```
print OVLADACSOUBORU "Toto se tiskne do souboru\n";
```

Jak vytvořit ovladač souboru bude vysvětleno až v díle o práci se soubory, takže se tím dnes více zabývat nebudeme.

Příklad - absolutní hodnota interaktivně

V minulém dílu jsme napsali program, který počítal absolutní hodnotu čísla. Dnes ho vylepšíme tak, aby počítal absolutní hodnotu čísla, které bude přečteno ze standardního vstupu.

```
#!/usr/bin/env perl
use strict;

my $hodnota;
print "Zadejte hodnotu:";
chomp  ($hodnota = <STDIN>);

if ($hodnota < 0){
$hodnota *= -1;
}

print "Absolutní hodnota zadaného čísla je $hodnota\n";
```

Příklad - elementární statistické charakteristiky

V další ukázce načte program ze standardního vstupu tentokrát hned tři čísla. Výsledkem bude jejich průměr, součet a největší a nejmenší ze zadaných čísel. Algoritmus není nijak složitý, takže ho není třeba podrobně popisovat.

```
#!/usr/bin/env perl
use strict;

my  ($a, $b, $c);                    #hodnoty, zadané ze vstupu
my  ($soucet, $prumer, $nejvetsi, $nejmensi); #proměnné, ve kterých budou výsledky
#proměnné lze definovat i takto. my přijímá jako argument seznam skalárních hodnot

print "Zadejte 1. hodnotu: "; chomp ($a = <STDIN>);
print "Zadejte 2. hodnotu: "; chomp ($b = <STDIN>);
print "Zadejte 3. hodnotu: ";
chomp ($c = <STDIN>);

$soucet = $a + $b + $c;
$prumer = $soucet / 3;

if ($a > $b and $a > $c){
$nejvetsi = $a;
}
elsif($b > $c){
$nejvetsi = $b;
}
else{
$nejvetsi = $c;
}

if ($a < $b and $a < $c){
$nejmensi = $a;
}
elsif($b < $c){
$nejmensi = $b;
}
else{
$nejmensi = $c;
}
```

```
print "Součet zadaných čísel je $soucet\n";
print "Průměr zadaných čísel je $prumer\n";
print "Největší ze zadaných čísel je číslo $nejvetsi\n";
print "Nejmenší ze zadaných čísel je číslo $nejmensi\n";
```

S pomocí cyklů, polí a funkcí by se dal předchozí příklad řešit lépe. Nám toto řešení zatím bude stačit.

Ovladač <>

Při čtení dat mezi < a > být ve skutečnosti žádný identifikátor nemusí. V tom případě bere Perl data ze souboru, který jsme přidali programu jako argument. Pokud jsme argument nepřidali, má <> stejný význam jako <STDIN>. Dodejme, že <> je ve skutečnosti pouze aliasem za <ARGV>. Následující příklad ilustruje použití.

Máme soubor kralove s tímto obsahem:

```
Přemysl Otakar I. 1197-1230
Václav I. 1230-1253
Přemysl Otakar II. 1253-1278
```

Dále máme vlastní program kralove.pl, který bude pouze přepisovat vstup na standardní výstup.

```
#!/usr/bin/env perl
use strict;
my $radek = <>;
print "$radek\n";
```

Spusťme program bez argumentu:

```
$ perl kralove.pl
Perl ceka na standardni vstup - neuvedli jsme parametr [ENTER]
Perl ceka na standardni vstup - neuvedli jsme parametr
$
```

A nyní udělejme totéž s názvem souboru jako argumentem:

```
$ perl kralove.pl kralove
Přemysl Otakar I. 1197-1230
$
```

V prvním případě program žádný argument nedostal, takže měl zápis <> stejný význam jako <STDIN> - zdrojem dat byl standardní vstup. V druhém případě byl argumentem soubor s prvními českými krály a Perl jako zdroj dat určil právě soubor kralove. Proto načítal z něj.

Ještě podotkněme, že Perl načte ze zdroje dat vždy jen jeden řádek. Proč, to bude blíže vysvětleno v odstavci o kontextech. Také je třeba zmínit, že pomocí speciální proměnné $/ lze načítat i jiné úseky, než řádky.

Počet čtení ze zdroje dat

Ve speciální proměnné $. je uložen počet přístupů k ovladači. Každý ovladač má své počítadlo. $. obsahuje hodnotu aktuálního (tedy většinou toho posledně použitého) ovladače. Chování této proměnné vysvětluje následující kód. Nejprve čteme několikrát ze STDIN, přičemž se $. inkrementuje. Poté čteme z jiného ovladače a $. se inkrementuje opět od 0. Nakonec zavoláme ještě STDIN, proměnná $. se nenuluje, ale pokračuje tam, kde u STDIN skončila.

```
$a = <STDIN>; print $.;#1
$b = <STDIN>; print $.;#2
$c = <STDIN>; print $.;#3
  $d = <>;      print $.;# 1
  $e = <>;      print $.;# 2
$f = <STDIN>; print $.;#4
```

I přesto je možné získat počet čtení z jiného než posledně použitého ovladače. Funkcí seek se nastavuje aktuální ovladač.

Perl (8) - Některé základní vestavěné funkce

Vestavěné funkce usnadňují základní úkony. Udělejme si nyní malý přehled, abychom poznali alespoň některé základní možnosti. Dále v průběhu seriálu budeme postupně poznávat další.

Jako snad v každém rozšířeném programovacím jazyce jsou i součástí Perlu funkce. Funkce je nějaký program uvnitř programu, kterému je obvykle předáván parametr nebo parametry (pravidlem to ale není) a na základě něj vrátí funkce určitou hodnotu, vytiskne něco na výstup nebo provede jinou činnost. Protože mají funkce návratovou hodnotu, jsou to v kontextu volání výrazy. Nejméně dvě funkce jsme již poznali: print a chomp.

Parametr funkce se zapisuje do závorek. Jestliže závorky neuvedeme, Perl většinou sám pozná, co argumentem je a co ne. Proto můžeme závorky téměř vždy vynechat. Tuto konvenci ostatně u print používáme stále a stejně tak ji budeme používat u dalších vestavěných funkcí.

Později budeme psát naše vlastní funkce. Při volání takových funkcí závorky zpravidla používat budeme. To samé platí o jinak získanýchfunkcích, které byly do našeho kódu nějak exportované.

Matematické funkce

Začněme u funkcí, které jsou jako funkce chápány v matematice.

Absolutní hodnota potřetí

Posledně jsme vylepšovali náš příklad na výpočet absolutní hodnoty tak, aby byl interaktivní. Absolutní hodnota je příklad jako kovaný pro použití funkce. Místo zdlouhavého testování, zda je vstupní číslo záporné nebo není, použijeme předdefinovanou funkci abs:

```
#!/usr/bin/perl
use strict;

my $hodnota;
print "Zadejte hodnotu:";
chomp ($hodnota = <STDIN>);

$hodnota = abs $hodnota;
print "Absolutní hodnota zadaného čísla je $hodnota\n";
```

Funkce abs přijímá jako parametr číslo a vrací jeho absolutní hodnotu. Tu lze získat přiřazením této funkce do proměnné. Pokud chceme návratovou hodnotu funkce přímo vytisknout, uveďme ji jako parametr funkce print.

Goniometrické funkce

Funkce	Popis	Příklad	Výsledek
sin(*hodnota_v_rad*)	Vrací sinus parametru	sin 3.141592	0.000...
cos(*hodnota_v_rad*)	Vrací kosinus parametru	cos 3.141592	-0.999...
atan2(*x*, *y*)	Vrací arkus tangens podílu *x* / *y*	atan2 (1, 1)	0.785...

Víme, že $\tan^{-1}$ (1) = pi / 4. Po vynásobení rovnice číslem 4 tak získáme Ludolfovo číslo. Příklad také ilustruje nepovinnost psát závorky i v případech, kdy má funkce více argumentů.

```
print 4 * atan2 1, 1; #3.14159265358979
```

Další matematické funkce

Funkce	Popis	Příklad	Výsledek
log(*hodnota*)	Přirozený logaritmus (základem je Eulerovo číslo)	log 1	0
exp(*exponent*)	Vrací Eulerovo číslo umocněné exponentem	exp 1	2.718...
sqrt(*číslo*)	druhá odmocnina čísla	sqrt 4	2
int(*číslo*)	Vrací celou část čísla (parametru)	int 4.7	4
oct(*číslo_v_8kové_soustavě*)	Převádí z 8kové do 10kové soustavy	oct "777"	511
hex(*číslo_v_16kové_soustavě*)	Převádí z 16kové do 10kové soustavy	hex "ff"	255

Funkce pro práci s řetězci

Funkce	Popis	Příklad	Výsledek
chomp(*řetězec*)	Odstraní případný znak \n./td>	$p = "abc\n";chomp $p	abc
chop(*řetězec*)	Odstraní poslední znak řetězce	$p = "abc";chop $p	ab
uc(*řetězec*)	Převede v řetězci malá písmena na velká	uc "AbCdE"	ABCDE
lc(*řetězec*)	Převede v řetězci velká písmena na malá	lc "AbCdE"	abcde
ucfirst(*řetězec*)	Převede 1. písmeno na velké	uc "abcde"	Abcde
lcfirst(*řetězec*)	Převede 1. písmeno na malé	lc "ABCDE"	aBCDE
length(*řetězec*)	Počet znaků v řetězci	length "abcde"	5
chr(*ASCII_kód*)	Vrátí znak náležící ASCII kódu	chr 100	d
ord(*znak*)	Vrátí ASCII kód	ord "d"	100

Funkce substr - ořezávání a nahrazování v řetězcích

substr je velmi obecná a často používaná funkce. Pracuje s podřetězci - vyhledává je v řetězcích, případně je nahrazuje jinými. Řadu speciálních případů volání lze nahradit voláním jednodušších funkcí, které postupně poznáme. Často také lze s řetězcem manipulovat jako s polem, což bývá často rychlejší.

Funkce substr má celkem 4 parametry. Zde je schéma volání:

```
substr(řetězec, od, kolik, náhrada);
```

První parametr určuje řetězec, se kterým bude substr pracovat. Druhým parametrem je pozice, která určuje začátek podřetězce. Další parametr definuje, kolik znaků bude obsahovat podřetězec. Posledním parametrem říkáme, čím chceme vybraný podřetězec nahradit. Povinné jsou jen první dva argumenty.

```
print substr "Pchjongjang", 3;
#Vráceno je vše od 3. pozice napravo - tedy jongjang

print substr "Pchjongjang", 3, 4;
#Vráceny jsou 4 po sobě jdoucí znaky, z nichž 1. je na pravo od 3. pozice - tedy jong

$slovo = "Pchjongjang";
substr $slovo, 3, 4, "jing";
print $slovo;
#Pchjingjang
#Jestliže chceme nahrazovat, je nutné předat funkci řetězec v proměnné - ne jen řetězec (který je vlastně neměnnou konstanou).

$slovo = "Pchjongjang";
substr $slovo, 3, 4, "";
print $slovo;
#Pchjang
```

Hodnoty *od* a *kolik* mohou být také záporné. U *od* to znamená první pozici na konci atd., u *kolik* určujeme, kolik znaků má zůstat do konce řetězce.

```
$slovo = "Pchjongjang";
print substr $slovo, -4, 3;
#jan

$slovo = "Pchjongjang";
print substr $slovo, -4, -3;
#j

$slovo = "Pchjongjang";
substr $slovo, -4, 4, "jing";
```

```
print $slovo;
#Pchjongjing
#Jsou nahrazeny 4 znaky vpravo za 4. pozicí od konce

$slovo = "Pchjongjang";
substr $slovo, -11, 9, "Beiji";
print $slovo;
#Beijing
```

substr má ještě jednu možnost zápisu. Paradoxem je, že se v něm funkce vyskytuje nalevo od operátoru přiřazení. To je napříč programovacími jazyky naprosto unikátní jev. Tyto zápisy mají na *řetězec* stejný efekt:

```
substr(retezec, od, kolik, náhrada);
substr(retezec, od, kolik) = náhrada;
```

Takhle to pak vypadá:

```
$slovo = "Pchjongjang";
substr($slovo, 3, 4) = "XXX";
print $slovo; #PchXXXjang

$slovo = "Pchjongjang";
$f = (substr($slovo, 3, 4) = "XXX");
print $slovo;#PchXXXjang
print $f;#XXXj
```

Opět jde ale o ukázku, kterou je dobré vidět a zase na ni zapomenout, neboť bychom si zbytečně znepřehledňovali kód.

Hledání podřetězců

Obě funkce mají 3 parametry:

```
index(řetězec, hledaný_podřetězec, od);
rindex(řetězec, hledaný_podřetězec, od);
```

Funkce index a rindex vyhledávají podřetězce uvnitř řetězců. Vracejí pozici prvního znaku *hledaného podřetězce* v *řetězci*. Poslední parametr je nepovinný a specifikuje pozici, od které začíná hledání. Není-li v řetězci podřetězec nalezen, vrací funkce -1. index hledá první výskyt, rindex poslední výskyt.

```
print index  "Zimbabwe", "w";#6
print rindex "Zimbabwe", "w"; #6
print index  "Zimbabwe", "b";#3
print rindex "Zimbabwe", "b"; #5
print index  "Zimbabwe", "b", 4; #5
print rindex "Zimbabwe", "b", 3;#3
print rindex "Zimbabwe", "x"; #-1
print rindex "Zimbabwe", "Z"; #0
```

Funkce die

Uvedení funkce die znamená výjimku a konec programu. Ty příkazy, které jsou za die, se neprovádí. Jako parametr je možné uvést hlášku, která se zobrazí na výstupu.

```
if ($selhani == 1){
die "Nepovedlo se zapsat data!\n";
}
```

Pokud parametr neukončíme koncem řádku, automaticky se vypíše i místo, kde byla funkce die volána.

Zatím se této funkci nebudeme věnovat podrobněji. Brzy na ni dozajista opět narazíme.

Funkce exit

die a exit mají podobný význam. die, jak už víme, ukončí okamžitě program. exit také. Rozdíl je v tom, kdy se tyto funkce volají. Nám bude zatím stačit, že exit nevydá žádné hlášení o chybě (nemá žádný výstup).

Definovanost proměnných

Funkce defined rozlišuje nedefinovanou hodnotu od všech ostatních - včetně 0 nebo prázdného řetězce. V případě nedefinované hodnoty vrací false.

(Pseudo)náhoda

Naučme se generovat náhodné číslo z rovnoměrného rozdělení. Nejprve musíme inicializovat generátor náhodných čísel. K tomu máme příkaz srand bez parametrů.

```
srand;
```

V tuto chvíli spustil Perl stopky. Zbývá z nich zachytit nějakou hodnotu.

```
$i = rand 983;
```

Do proměnné $i bylo přiřazeno náhodné desetinné číslo z intervalu [0; 983]. Neuvedeme-li mez, vrací funkce hodnoty z intervalu [0; 1].

Chceme-li realizace z diskrétního rovnoměrného rozdělení, dosáhneme toho zaokrouhlením jednotlivých realizací. Například pro generování z dvouprvkové množiny {0, 1} (nebo-li true/false) bychom psali toto.

```
$i = int rand 2;
```

Příklad - řešení rovnice 2. stupně

Programovat se nelze naučit čtením manuálů nebo opisováním kódu z učebnic, ale psaním svých programů. Je vhodné, aby si každý, kdo se chce naučit programovat, zkusil sám něco napsat.

Zde je příležitost. Zadání úlohy zní: Napište program, který na základě zadání koeficientů ze standartního vstupu spočítá kořeny kvadratické rovnice v množině reálných čísel a vypíše je na výstup (nezapomeňte na všechny podmínky).

Řešení

Takto rozsáhlý příklad jsme zatím nedělali. Sám algoritmus není obtížný, ale je třeba si průběžně uvědomovat, co je v kterém kroku potřeba udělat a co vše může nastat.

U všech větších příkladů se v seriálu budeme snažit postupovat tak, jak bychom ho pravděpodobně řešili na základě intuice. V návodech pro začátečníky by zdrojový kód neměl být vysvětlován řádek po řádku, aniž by byly přiblíženy myšlenky, kterými se k výslednému kódu dospělo.

Tento příklad je ale jedním z těch jednodušších, které lze řešit i metodou od prvního řádku k poslednímu. Takže nejdříve načteme data ze vstupu:

```
#!/usr/bin/perl
```

```
use strict;

my ($a, $b, $c, $x1, $x2);

print "Zadejte koeficienty kvadratické rovnice ax^2 + bx + c = 0\n";
print "a = "; $a = <STDIN>;
print "b = "; $b = <STDIN>;
print "c = "; $c = <STDIN>;
```

Poté už můžeme začít s výpočtem. Vzorec x_1 = (-b + odmocnina(b^2 - 4*a*c)) / (2*a) popř. x_2 = (-b - odmocnina(b^2 - 4*a*c)) / (2*a) jistě každý zná. Je nutné si uvědomit podmínky řešitelnosti - tj. dávat si pozor jestli neodmocňujeme záporné číslo, nedělíme nulou nebo se nemění počet řešení. Nejprve ošetříme podmínkami speciální případy a poté dopíšeme obecné řešení.

Ošetřme tedy případ, kdy je diskriminant záporný (pro větší názornost používám zápis $b ** 2 - 4 * $a * $c < 0, ale samozřejmě lze nerovnici upravit a použít $b ** 2 < 4 * $a * $c):

```
if ($b ** 2 - 4 * $a * $c < 0){
die "V množině reálných čísel nemá rovnice řešení.\n";
}
```

Pokud je diskriminant roven nule, řešení je jen jedno:

```
if ($b ** 2 - 4 * $a * $c == 0 and $a != 0){
$x1 = $b / (2 * $a);
die "Řešením dané rovnice je $x1.\n";
}
```

Další speciální situace nastane, je-li koeficient a = 0. V tom případě jde o lineární rovnici. Do podmínky zahrňme i test koeficientu b, protože to je další speciální případ.

```
if ($a == 0 and $b != 0){
$x1 = -$c / $b;
die "Řešením dané rovnice je $x1.\n";
}
```

Je-li totiž i koeficient b = 0, jedná se o rovnici nultého stupně. Ta má jako řešení buď všechna reálná čísla a nebo řešení nemá.

```
if ($a == 0 and $b == 0 and $c != 0){
die "V množině reálných čísel nemá rovnice řešení.\n";
}

if ($a == 0 and $b == 0 and $c == 0){
die "Řešením rovnice jsou všechna reálná čísla.\n";
}
```

Všechny speciální případy jsou vyřešeny, tak se můžeme pustit do obecného řešení. Sem se program dostane pouze pokud nevyhověly vstupní hodnoty některému předchozímu případu.

```
$x1 = ($b + sqrt($b ** 2 - 4 * $a * $c)) / (2 * $a);
$x2 = ($b - sqrt($b ** 2 - 4 * $a * $c)) / (2 * $a);

print "Řešením dané rovnice je $x1 a $x2.\n";
```

Poznámka: Samozřejmě to není jediné řešení problému. Další možností by mohla být například podmínka if ($a == 0){ ... } else { ... }, v jejíž větvích by byly další podmínky..

Zde máme výsledek:

```
$ ./kvadraticka_rovnice.pl
Zadejte koeficienty kvadratické rovnice ax^2 + bx + c = 0
a = 5
b = 2
c = 5
V množině reálných čísel nemá rovnice řešení
$ ./kvadraticka_rovnice.pl
Zadejte koeficienty kvadratické rovnice ax^2 + bx + c = 0
a = 2
b = 5
c = 3
Řešením dané rovnice je 1.5 a 1.
$ ./kvadraticka_rovnice.pl
Zadejte koeficienty kvadratické rovnice ax^2 + bx + c = 0
a = 5
b = 10
c = 5
Řešením dané rovnice je 1.
$ ./kvadraticka_rovnice.pl
Zadejte koeficienty kvadratické rovnice ax^2 + bx + c = 0
a = sh
b = hgf
c = shgfd
Řešením rovnice jsou všechna reálná čísla.
$
```

V posledním případě byly řetězce automaticky konvertovány na čísla.

Další hojně užívanou řídící strukturou, bez které se neobejde žádný větší program, je cyklus.

Cyklus umožňuje vykonávat určitou část programu opakovaně. Funguje v podstatě podobně jako podmínka. Na základě vyhodnocení testu se provede blok kódu. Rozdíl mezi podmínkou a cyklem je v tom, že u cyklu se vyhodnocuje test opakovaně. Dokud není test vyhodnocen jako false (nebo v některých speciálních cyklech true), stále se znovu a znovu opakuje blok kódu. Existuje hned několik druhů cyklů.

Cyklus while - předem neznámý počet iterací

Dokud platí *test*, provádí se blok kódu.

```
while (test){
    příkazy
}
```

Cyklus by měl mít možnost být ukončen - tj. vepsat nějaký kód, který ho bude přibližovat ke konci, až nakonec také skončí. Pro naše účely to lze vyřešit například takto:

```
$i = 0;
while ($i < 10){
print "Probíhá cyklus $i.\n";
    $i++;
}
```

10× je proveden kód uvnitř cyklu. Je to stejné, jako kdybychom za sebe nakopírovali 10× blok.

```
$ perl while1.pl
Probíhá cyklus 0.
Probíhá cyklus 1.
Probíhá cyklus 2.
Probíhá cyklus 3.
Probíhá cyklus 4.
Probíhá cyklus 5.
Probíhá cyklus 6.
Probíhá cyklus 7.
Probíhá cyklus 8.
Probíhá cyklus 9.
$
```

Ano, je to dost netypická situace pro použití while. V praxi bychom v této situaci bez váhání sáhli po cyklu for. Avšak tento zápis se dá velmi snadno pochopit. Teď už ale přijde na řadu skutečné while. Další příklad totiž do nekonečna vypisuje to, co napíšete. V praxi má taková forma cyklu velmi časté využití.

```
while ($radek = <STDIN>){
    chomp $radek;
print "Napsal jste: $radek\n";
}
```

Po spuštění kódu získáváme výstup:

```
$ perl while2.pl
Tento cyklus lze ukončit jen násilně. [ENTER]
Tento cyklus lze ukončit jen násilně.
Sám program, pokud není přerušen, by neskončil nikdy. [ENTER]
Sám program, pokud není přerušen, by neskončil nikdy.
[čeká na další vstup]
```

Zkusme kód poupravit tak, aby se při zadání q nebo exit ukončil. Dosáhneme toho přidáním podmínky (funkce die často nebude stačit. Příští díl osvětlí lepší řešení).

```
while ($radek = <STDIN>){
    chomp $radek;
if ($radek eq "q" or $radek eq "exit"){
    die "Konec\n";
}
print "Napsal jste: $radek\n";
}
```

Cyklus for - předem známý počet iterací

Cyklus for se od while liší tím, že je předem znám počet iterací (jde sice změnit v bloku, ale potom je často lepší užít while). for a while jsou vzájemně zaměnitelné (for lze napsat jako while a naopak). Ale většinou je jeden konkrétní vhodnější a ten druhý méně.

Cyklu for se budeme podrobněji věnovat později, neboť je dobré mít zvládnutou práci se seznamy.

Podívejme se nyní pouze na základní užití. V každé iteraci přiřadíme do proměnné číslo postupně od 1 do 10.

```
for my $i (1 .. 10){
print "Probíhá cyklus $i.\n";
}
```

Céčkovské for

Ve zbytku dílu se podívejme na několik zajímavých konstrukcí, které však pravděpodobně nikdy nepoužijeme. Proto klidně zbytek článku bez obav přeskočte.

Taktéž existuje cyklus s klasickou céčkovskou notací. Pro zajímavost ho zde zmíníme, ale kvůli přehlednosti bychom na něj měli hned zase zapomenout a nikdy ho nepoužívat. Syntaxe je následující:

```
for (počáteční_hodnota_počítadla; konečná_hodnota_počítadla; krok){
příkazy
}
```

Počáteční_hodnota_počítadla spouští počítadlo. *Konečná_hodnota_počítadla* je vlastně *testem*. Dokud platí, je cyklus opakován (tj. proveden blok kódu). A *krokem* je výraz, který změní hodnotu počítadla.

```
for ($i=1; $i<10; $i++){
print "Probíhá cyklus $i.\n";
```

```
}
```
Výstup je stejný jako u prvního příkladu u while. Nejprve se přiřadí do proměnné $i hodnota 1. Je-li test true, provede se blok, proměnná se inkrementuje (krok) a pokud opět platí test, opakuje se vše znovu. Ve chvíli, kdy test neplatí, je blok přeskočen a program pokračuje za ním.
Složitost výrazů v hlavičce for není omezena jen na takto jednoduché záležitosti. Můžeme třeba přidat další počítadlo. Operátor , (čárka) odděluje se dva výrazy, přičemž první se vyhodnotí a následně zapomene. Druhý výraz se také vyhodnotí a je výsledkem celkového vyhodnocení obou výrazů.
```
for ($i=0,$j=0; $i<10,$j<10; $i++,$j+=2){
print "Probíhá cyklus $i. J: $j.\n";
}
```
Výstup:
```
$ perl for.pl
Probíhá cyklus 0. J: 0.
Probíhá cyklus 1. J: 2.
Probíhá cyklus 2. J: 4.
Probíhá cyklus 3. J: 6.
Probíhá cyklus 4. J: 8.
$
```
Je vidět, že počítadlem řídící cyklus je až druhá část výrazu - za čárkou (s proměnnou $j). Výraz s $i je sice každou iteraci zapomenut, ale hodnota je vždy změněna.
počáteční_hodnotu_počítadla, *konečnou_hodnotu_počítadla* ani *krok* není povinné uvést. Vynecháme-li *počáteční_hodnotu_počítadla*, zůstane v $ihodnota, která tam dosud byla. Pokud je nedefinovaná, v první iteraci bude stále proměnná $i nedefinovaná, ale v dalších už nemusí. Je-li totiž v *kroku* výraz $i++, v druhé iteraci bude hodnota $i = undef + 1 = 1.
Vynecháním *konečné_hodnoty_počítadla* se cyklus stává nekonečným. Nemá se co vyhodnocovat, a proto je za všech okolností true.
Neuvedení *kroku* má za následek, že se počítadlo (aspoň v hlavičce cyklu) nebude iterovat.
Tímto způsobem lze vynechat třeba i všechny tři výrazy. Zápis for (;;){...} je ekvivalentní while (1){...}.
Odtud plyne, že *počáteční_hodnota_počítadla* není nic jiného, než výraz, který se vyhodnotí před započetím cyklu, *konečná_hodnota_počítadla* je testem a *krok* výrazem, který se vyhodnocuje na konci každé iterace.
Céčkový cyklus for podle posledního tvrzení můžeme jednoduše přepsat do while (není to úplně přesné, *krok* by správně měl být až v bloku continue, který představíme příště. Toto řešení neplatí pro případy, kdy je konec bloku vynechán):
```
počáteční_hodnota_počítadla;
while (konečná_hodnota_počítadla){
příkazy
krok;
}
```
Alternativní cyklus until
Stejně jako jsou ve vztahu if a unless je ve vztahu i while a until. Cyklus je prováděn, dokud je test false. A i zde platí, že bychom takovou konstrukci neměli nadále používat.
Cyklus za příkazem
Cykly while a for se dají v Perlu napsat i za příkaz. Opět to výrazně snižuje čitelnost programu. Ukažme si alespoň pro představu zdrojový kód:
```
$i = 1;

$i++ while ($i < 10); # k $i se bude tak dlouho přičítat jednička, dokud bude $i nižší než 10
print $i;#10

$i-- until ($i < 0); # nyní se od $i bude pro změnu jednička odečítat a to tak dlouho, dokud nebude $i nižší než 0
print $i;#-1
```
Cyklus za blokem
Cykly do...while, do...until jsou dalšími variacemi cyklů. Jejich používání znovu nelze v žádném případě doporučit.
Nejprve se provede blok příkazů a až poté se vyhodnotí test (výsledný efekt je stejný, jako kdybyste blok v cyklu zkopírovali před klasický while). Je-li true (while) resp. false (until), vykonává se blok znovu.
```
$i = 0;

do{ #vykoná se i přesto, že $i má hodnotu 0. Testuje se až za blokem.
print "Vykonáno\n";
}while ($i == 1);
```
Za testem musí být stejně jako u konstrukcí do...if a do...unless středník.
I cyklus for lze napsat za příkaz nebo blok. O tom se zmíníme až budu rozebírat další cyklus foreach.
Příklad
Napišme program, který bude ze vstupu přijímat kladná čísla. Zadáním nekladného čísla skončí zadávání (toto číslo se již nepočítá) a program vytiskne počet zadaných čísel, jejich součet a průměr.
Měli bychom získat přibližně takový výstup:
```
$ perl cisla.pl
Zadávejte kladná čísla, 0 pro konec.
12
994
328
666
666
9004
0
Počet: 6
Součet: 11670
```

Průměr: 1945
$
Kostra programu bude vypadat velmi podobně jako dnes již zmíněná ukázka kódu k while. Lišit se bude tím, že v každé iteraci se bude aktualizovat součet a počet zadaných čísel a před ukončením se ještě vytisknou výsledky.

```
#!/usr/bin/perl
use strict;

my $cislo;
my ($pocet, $prumer, $soucet);

print "Zadávejte čísla, 0 pro konec.";
while (chomp($cislo = <STDIN>)){
if (int $cislo <= 0){
...vypsání statistik a konec...
}

...mezivýpočet...
}
```

V mezivýpočtu stačí navyšovat součet a počet.

```
$soucet += $cislo;
$pocet++;
```

Na závěr je třeba dokončit výpis výsledků. Spočítáme průměr, vypíšeme výsledky a ukončíme.

```
$prumer = $soucet / $pocet;
print "Počet: $pocet\n";
print "Součet: $soucet\n";
print "Průměr: $prumer\n";
die "\n"; #Pokud bychom neuvedli konec řádku, vypsala by automatická hláška (Tento řádek by šel lépe řešit pomocí funkce exit,
#která zatím nebyla probrána)
```

Když dopíšeme program, je vždy nutné otestovat jeho reakce v závislosti na podmínkách. Všechny případy musí být ošetřeny. To my nemáme, protože program hlásí chybu, pokud neuvedeme žádný parametr. Tedy na začátek bloku v podmínce připišme:

```
if ($pocet == 0){
die "Nezadal jste žádné číslo!\n";
}
```

A tím je úloha splněna. Zdrojový kód příkladu.

Perl (10) - Další řídící struktury

Podívejme se na některá klíčová slova, která mohou být používána v cyklech pro dodatečné řízení.

last, next, redo

Existuje i jiný způsob, jak řídit cykly, než pomocí testu. Ve skutečnosti můžeme napsat nekonečný cyklus a ukončit ho až za určitých podmínek v jeho bloku. Cyklus tak vůbec nemusí celý řádně proběhnout. V minulém dílu jsme k tomuto účelu použili funkci die. Ta však nevyskočí jen z cyklu, ale přímo ukončí program. Následující informace platí pro všechny cykly, které zatím známe.

Klíčové slovo last uvedené v těle cyklu cyklus okamžitě ukončuje. Provádění programu pokračuje bezprostředně za cyklem (za uzavírací složenou závorkou).

Oproti tomu next přeskočí pouze zbytek bloku a provádění pokračuje další iterací (má stejnou funkci jako céčkovské continue).

U obou těchto klíčových slov je možný parametr, kterým je jméno návěští. To použijeme, pokud se provádí cyklus uvnitř jiného cyklu. Potom vyskočíme z obou.

Teď uveďme několik ukázek kódu, které většinou používají while. V praxi bychom použili u většiny nejspíše for. Cyklus while zde používáme spíše proto, že jsme se s for ještě dostatešně neobeznámili.

```
$i = 0;
while ($i < 9){
$i++;
if ($i == 5) {
last;
}
print "Probíhá cyklus $i.\n";
}
```

Proběhne pouze 5 cyklů, protože v 5. cyklu je klíčovým slovem last cyklus ukončen a program pokračuje za blokem:

```
$ perl last.pl
Probíhá cyklus 1.
Probíhá cyklus 2.
Probíhá cyklus 3.
Probíhá cyklus 4.
$
```

Podobný program, jen s drobnou modifikací - vyměnímě last za next:

```
$i = 0;
while ($i < 9){
$i++;
if ($i == 5) {
next;
}
print "Probíhá cyklus $i.\n";
}
```

Na začátku 5. cyklu je cyklus přerušen a pokračuje dalším cyklem. Proto není u 5. cyklu zaznamenán výpis:

```
$ perl while1.pl
```

```
Probíhá cyklus 1.
Probíhá cyklus 2.
Probíhá cyklus 3.
Probíhá cyklus 4.
Probíhá cyklus 6.
Probíhá cyklus 7.
Probíhá cyklus 8.
Probíhá cyklus 9.
$
```

Příkaz redo má stejný účinek jako next, jen nedojde ke změně počítadla. Jinými slovy skočíme znovu na začátek bloku.

```
for ($i=1; $i<10; $i++){
    $i++;
    if ($i == 4) {
        redo;
    }
print "Probíhá cyklus $i.\n";
}
```

Při první iteraci je v bloku zvýšena hodnota $i na 2. V další iteraci je nejprve zvýšena v počítadle, dále v bloku a má hodnotu 4. To je důvod pro volání redo. Program přeskočí opět na začátek bloku, přičemž počítadlo hodnotu $i nemění. Ta se změní až v bloku. Aktuální hodnota je tedy 5 a od tohoto okamžiku už pokračuje skript, aniž by kdykoliv testu podmínky vyhověl.

```
$ perl while1.pl
Probíhá cyklus 2.
Probíhá cyklus 5.
Probíhá cyklus 7.
Probíhá cyklus 9.
$
```

Návěští

Další možností řídit cykly je pojmenovat blok a umístit za klíčové slovo last, nextnebo redo právě název bloku (návěští). To musí být definováno. Před cyklus, pro který má návěští platit, se uvádí jeho název následovaný dvojtečkou. Návěští se pojmenovává velkými písmeny.

```
    $i = 0;
NAVESTI:
while (){ #nekonečný cyklus
    while ($i < 10){
print "Probíhá cyklus $i.\n";
    if ($i == 3){
    last NAVESTI;
    }

    $i++;
    }
    }
```

last zde nevyskakuje z vnitřního cyklu, ale z cyklu označeného jako NAVESTI.

```
$ perl while1.pl
Probíhá cyklus 0.
Probíhá cyklus 1.
Probíhá cyklus 2.
Probíhá cyklus 3.
$
```

Dodejme, že návěští je vhodné používat z důvodu přehlednosti nejen v případě zanořených cyklů.

Blok continue

continue je nepovinným blokem příkazů na konci cyklu. Je spuštěn při každé iteraci mimo případů, kdy je cyklus přerušen klíčovým slovem redo nebo last. V následujícím příkladu je proměnná $i iterována i v případě, kdy je klíčovým slovem next přeskočeno na další iteraci. continue se proto hodí pro opakování kódu, které by muselo být vyvoláno v každé iteraci. Ovšem často místo použití continue stačí vyměnit cyklus while za for.

```
while ($i < 10){
    if ($i == 5){
        next;
    }
}continue{
$i++; #blok continue je proveden vždy. I v případě, že $i == 5.
}
```

Klíčové slovo goto

goto funguje podobně jako návěští. Parametrem goto je opět návěští, které ale nepojmenovává blok. Může být uvedeno před libovolný příkaz. Po goto se bude dále vykonávat část programu, začínající návěštím.

```
print "1\n";

goto GOTO;

print "2\n";
print "3\n";

    GOTO:

print "4\n";
print "5\n";
```

Druhé a třetí print nebude vykonáno. Když program narazí na goto, bude vše až po návěští přeskočeno:

```
$ perl goto.pl
1
4
5
$
```

Nic nám nebrání umístit návěští na goto před samotné goto.

```
print "1\n";
   $i = 0;
   GOTO:

print "2\n";
print "3\n";

   $i++;
if ($i != 2){
goto GOTO;
   }
print "4\n";
print "5\n";
```

Nastává zajímavá situace. Po druhou inkrementaci je goto vynecháno, aby cyklus nebyl nekonečný. Všimněme si, že mluvíme o cyklu, aniž bychom použili while nebo for. Slovo cyklus je skutečně na místě, protože je část kódu vykonávána opakovaně. Pokud se trochu zamyslíme, zjistíte, že jde vlastně logicky o cyklus do...while, který je jen převlečený do jiného kabátu.

```
$ perl goto.pl
1
2
3
2
3
4
5
$
```

Přes tento hezký efekt ale doporučují programátoři goto používat jen ve výjimečných situacích, kdy by bylo jiné řešení značně složité. Je to jedna z mnoha dalších konstrukcí, jejíž přispěním program ztrácí na přehlednosti.

Příklad - hádání myšleného čísla

Napíšeme si jednoduchou hru. V ní si od nás program nejprve vyžádá číslo, které bude určovat horní hranici intervalu, ve kterém leží myšlené číslo. Potom bude opakovaně vyzývat k zadání čísla a prozradí vždy, zda je myšlené číslo větší nebo menší než hádané. Přitom bude počítat pokusy. V případě uhádnutí skončí cyklus.

Na začátku programu je nutné aktivovat generátor:

```
#!/usr/bin/perl
use strict;
use warnings;

srand;
```

Dále získáme ze vstupu číslo:

```
my $max;
print "Zadej největší možné číslo: ";
chomp($max = <STDIN>);
```

Musíme podmínkou ošetřit případ, kdy je zadané číslo nekladné.

```
if (int $max <= 0){
die "Toto není správně zadané číslo.\n";
}
```

Ještě stále nemáme myšlené číslo. Vygenerujeme ho funkcí rand. Je zde ale problém a možná tušíte jaký. rand vrací desetinné číslo a my chceme celé. Desetinnou část odřízneme funkcí int. To má za vedlejší efekt posunutí intervalu o jedničku dolů. Nemáme zájem, aby mohlo být tajné číslo 0, proto přičteme jedničku.

```
my $tajne_cislo = int(rand($max)) + 1;
```

Teď na řadu přijde samotný cyklus. V něm musí být výzva k zadání čísla a vyhodnocení - zda je menší, větší nebo rovno myšlenému číslu. Pokud je rovno, vypíšeme hlášku a ukončíme cyklus pomocí klíčového slova last.

```
while (1){
print "Hádej číslo mezi 1 a $max.\n";
my $hadane_cislo;
chomp($hadane_cislo = <STDIN>);

if ($hadane_cislo == $tajne_cislo){
print "Gratuluji!\n";
last;
}elsif($hadane_cislo < $tajne_cislo){
print "To je málo. ";
}else{
print "Moc. ";
}
}
```

Na něco jsme přece jen zapomněli. Počítat pokusy. S výhodou použijeme blok continue. Před cyklem nastavíme číslo pokusu na 1

```
my $pokus = 1;
```

a bezprostředně za cyklus přidáme právě blok continue. V něm budeme přičítat pokusy.

```
continue{
$pokus++;
}
```
Protože s počítáním pokusů dříve nepočítali, musíme upravit hlášku v případě uhodnutí tajného čísla:
```
print "Gratuluji, uhodl jsi na $pokus. pokus!\n";
```
Teď spusťme program.
```
$ ./cislo.pl
Zadej největší možné číslo: 100
Hádej číslo mezi 1 a 100.
50
To je málo. Hádej číslo mezi 1 a 100.
75
Moc. Hádej číslo mezi 1 a 100.
63
To je málo. Hádej číslo mezi 1 a 100.
69
To je málo. Hádej číslo mezi 1 a 100.
72
Moc. Hádej číslo mezi 1 a 100.
70
To je málo. Hádej číslo mezi 1 a 100.
71
Gratuluji, uhodl jsi na 7. pokus!
$
```
Perl (11) - Pole - úvod

Představíme si další datovou strukturu - pole.

Skaláry a seznamy

Známe zatím dva druhy skalárních dat. Řetězce a čísla. **Skalár** (proměnná se označuje dolarem ($) před názvem) obsahuje jen jednu věc - tedy jeden řetězec , jedno číslo nebo jeden odkaz na něco. Dosud jsme nepracovali s jinými než skalárními proměnnými.

Seznamy oproti tomu obsahují více položek. V Perlu se zapisují oddělené čárkami v kulatých závorkách. Proměnnou, která uchovává seznam, je pole nebo hash. Pole se značí zavináčem (@) před názvem.

Kontext

Před samotnými poli se podívejme ještě na tzv. kontext. Perl má velmi benevolentní typovou kontrolu a může docházet k takovým věcem, jako je uvedení pole místo skaláru a naopak. Perl má nějaká pravidla, podle kterých takto uvedené pole, resp. skalár, vyhodnocuje.

Co to tedy kontext je? Je to v podstatě totéž jako v běžné řeči. Vezměme si slovo *síť*. Když zmíníme síť samostatně, nikdo nemůže vědět, kterou síť myslíme. Použijeme-li ji například v souvislosti elektřinou, pravděpodobně půjde o elektrickou síť. Podobně můžeme sítí myslet pavoučí síť, počítačovou síť nebo nějakou další. Záleží na tom, v jakém kontextu síť užijeme.

Podobně funguje kontext i v programování.

Uveďme ještě jeden a možná lépe vypovídající příklad. V matematice standardně nemůžeme násobit podtržítka s tečkami nebo umocňovat tučňáka na kružnici. Je zvykem, že operace násobení a umocňování potřebují dostat oba operandy jako číslo. Pro ostatní případy je třeba výsledek operací definovát (nejlépe nějak rozumně).

Funkce se chovají k datům podle toho, v jakém jim je předáváme kontextu - zda jde o seznam nebo skalár (podobnou otázkou je, jestli jde o číslo nebo řetězec, případně logický výraz). Stejná funkce může fungovat s oběma možnostmi. Pak se ale obvykle liší ve zpracování a výsledku. Předáme-li nějaké funkci skalár, může třeba vrátit jeho druhou mocninu. Pokud stejná funkce dostane jako parametr seznam, může vracet pole druhých mocnin původního seznamu, druhou mocninu prvního prvku původního seznamu, nebo klidně něco jiného. Právě proto je vždy dobré vědet, co děláme.

Pole

Již víme, že pole je proměnná pro uchovávání nějakého množství skalárů (čísel, řetězců, odkazů).

Jde tedy opět o oblast v paměti, do které ale, narozdíl od skalárních proměnných, můžeme uložit celou řadu hodnot. Pole je seznamem skalárních hodnot.

Skalárním proměnným v poli se říká prvky. Prvek je určen indexem a obsahuje hodnotu. Index je číslo. prvním indexem je nula, dalším 1 atd. Skalární proměnné se značí dolarem, ale pole zavináčem. Přesto se dolar používá pro označení jednotlivých prvků pole. Je to svým způsobem logické - prvek pole uchovává sám o sobě jen skalární hodnotu - například číslo - nikoliv seznam hodnot. Zavináč se používá i v případě řezu polem. Řez polem je totiž opět seznamem.

Poznámka - Podotkněme však, že v Perlu 6 budeme i samotné prvky pole označovat zavináčem.

Práce s poli je v Perlu narozdíl striktních jazyků jako Céčko velmi jednoduchá. Žádná deklarace ani alokace. Platí to, co u proměnných - prostě můžeme pole kdykoliv začít používat.

Vyvstává otázka: Máme přeci proměnné, tak proč používat pole? Oproti proměnným mají pole v jistých situacích řadu výhod. Hodí se na jiné typy úkolů a některé by bez nich byly nerealizovatelné. Představme si, že máme databázi nějakých údajů, například jmen, s kterými nahodile manipulujeme. Použijeme-li pole, nemusíme vymýšlet jména tolika proměnných, kolik lidí máme. Navíc ani nemusíme vědět, kolik jich vlastně je. Také můžeme získat počet jmen jediným příkazem nebo jednoduchým cyklem či užitím map smazat ty, jejichž příjmení začíná na S. Kdykoliv můžeme velmi jednoduše přidat další jméno. Zkrátka pole jsou skvělým pomocníkem a ani jednoduché aplikace se bez nich neobejdou.

Vytvoření pole

Pole vytvoříme, stejně jako skalární proměnnou - přiřazením. Levým operandem je identifikátor, začínající zavináčem a na pravo se uvádí v kulaté závorce hodnoty jednotlivých prvků, oddělené čárkami (tedy vlastně seznam):
```
@pole = ("hodnota1", "další", "a poslední");
```
Tím jsme vytvořili tříprvkové pole. Zde je tabulka určující jeho strukturu:

Index	Hodnota
0	"hodnota1"
1	"další"

2	"a poslední"

Zajímavá pro nás bude také funkce qw. Přesněji řečeno, nejde o funkci, ale o způsob uvození (proto ji označujeme - stejně jako uvozovky - tučně zeleně). Umožňuje položky v uvozovkách, oddělené čárkou, oddělovat jen mezerou (obecně bílým místem). Řetězec není v uvozovkách ani apostrofech. Užijeme ji nejen při delších výčtech.
Tyto příkazy mají stejný význam:

```
@pole = ("a", "b", "c");

@pole = qw(a b c);
```

Čtení z pole
Pole máme, jak z něj číst? Chceme-li získat hodnotu prvku @pole s indexem x, zápis je následující:

```
$pole[x]
```

Pozor na to, že prvek pole je skalárem (nikoliv seznamem) a označuje se v Perlu 5, jako jsme tomu byli u skalárů zvyklí, dolarem.
Takže, chceme-li získat data z našeho @pole, použijme následující:

```
print "Prvek 0: $pole[0]\n";
print "Prvek 1: $pole[1]\n";
print "Prvek 2: $pole[2]\n";
```

Výstup:

```
$ perl pole.pl
Prvek 0: hodnota1
Prvek 1: dalsi
Prvek 2: a posledni
$
```

Modifikace pole
Pole můžeme editovat i přiřazením do $pole[*index*]:

```
$pole[0] = "Perl";
$pole[4] = "Linux";
```

Za předchozí kód přidejme ještě tento:

```
print "Prvek 0: $pole[0]\n";
print "Prvek 1: $pole[1]\n";
print "Prvek 2: $pole[2]\n";
print "Prvek 3: $pole[3]\n";
print "Prvek 4: $pole[4]\n";
```

Po spuštění dostanete:

```
$ perl pole.pl
Prvek 0: Perl
Prvek 1: dalsi
Prvek 2: a posledni
Prvek 3:
Prvek 4: Linux
$
```

Výstup ukazuje hned několik skutečností. Přiřazením hodnoty do prvku pole přemažeme původní obsah (index 0). Lze přiřadit hodnotu i do prvku, který bezprostředně nenavazuje na předchozí (index 4). Prvek 3 existuje, ale zůstal nedefinovaný. To i přesto, že prvek 4 definován je.
Následující dva zápisy si jsou ekvivalentní:

```
@pole = ("a", "b", "c");

$pole[0] = "a";
$pole[1] = "b";
$pole[2] = "c";
```

Zkopírujme obsah jednoho prvku pole do jiného. Je to stejné, jako u obyčejných proměnných.

```
@pole = qw(a b c);

$pole[3] = $pole[0]; #do prvku s indexem 3 bude přiřazena hodnota, která je v prvku s indexem 0
```

Kopírovat můžeme i celá pole.

```
@pole = qw(a b c);

@pole2 = @pole; #@pole2 je kopií @pole
```

Jak @pole, tak @pole2 mají stejný obsah.
Stejně tak můžeme kopírovat část pole:

```
@pole = qw(a b c d e);

@pole2 = ($pole[2], $pole[3], $pole[0]);#@pole2 nyní obsahuje prvky 0, 1, 2 s obsahem po řadě c, d, a.
```

Chceme-li přiřadit prvek z @pole s indexem 7 do @pole2 (index 4) a zároveň prvek @pole s indexem 3 do @pole2 (index 11), lze použít zápis:

```
@pole2[7, 3] = @pole[4, 11];
```

Ekvivalentním zápisem je:

```
$pole2[7] = @pole[4];
$pole2[3] = @pole[11];
```

Pokud bude málo prvků v hranatých závorkách (ale platí to obecně, pokud přiřazujeme seznam) na pravé straně přiřazení, doplní se nedefinovanými hodnotami. Přebývají-li, jsou ignorovány prvky navíc zprava.
Podobný příklad jako předchozí:

```
@pole2 = @pole[4, 11, 25, 94]; #@pole2 nyní obsahuje prvky 0, 1, 2, 3 s obsahem stejným jako prvky s hodnotami 4, 11, 25, 94 v @poli.
```

Pole můžeme složit i z několika jiných polí.

```
@jazyky = qw(Perl Python);
@cisla = (68, 3.14);

@mix = (@jazyky, @cisla); # obsahuje prvky s hodnotami Perl, Python, 68, 3.14
```

Přidat na konec pole nějakou hodnotu se dá i následovně (později poznáme lepší způsob - funkci):

```
@jazyky = qw(Perl Python);
@jazyky = (@jazyky, "Ruby");
```

Seznam proměnných

Perl umožňuje mnoho zvláštních (přesto však nepostrádajících logiku) konstrukcí. Pro zajímavost příklad:

```
$p = ("a", "b", "c")[1];
print $p;
```

("a", "b", "c") je vlastně seznamem (polem). Z něj se do $p přiřadí prvek s indexem 1.

Stejně jako je seznamem ("a", "b", "c"), může jím být i ($p1, $p2, $p3). Seznamem je dokonce i ($p). Takto zapsané hodnoty v kulatých závorkách jsou seznamem vždy. Poslední případ je sice jednoprvkovým seznamem, ale stále seznamem. Pokud do pole přiřadíme prázdné závorky, pole bude dokonce prázdné.

V další ukázce přiřazujeme jeden seznam do jiného. Na levé straně samozřejmě musí být proměnné.

```
($a, $b) = ("a", "b");

print $a; #a
print $b; #b
```

Existuje řada variací tohoto příkladu. Pokračujme v kódu.

```
($a, $b) = ($b, $a); #prohození hodnot v proměnných

print $a; #b
print $b; #a

@mix = ($a, 1, 2, $b, "Perl"); #b, 1, 2, a, Perl
@mix2 = ($a, @mix, 4, "Linux") #b, b, 1, 2, a, Perl, 4, Linux
```

Záporné indexy

Zatím víme, že index označuje pořadí prvku. Mohou být i záporné. V tom případě je hodnota pořadím od konce pole. Pozor na to, že první prvek od konce se značí $pole[-1], nikoliv $pole[-0]. To proto, že $pole[-0] je stejné jako $pole[0].

Perl (12) - Pole - základní operace

Naučme se trochu orientovat v seznamovém kontextu a používat některé základní nástroje , mezi které patří třídění polí nebo operátor rozsahu.

Seznam ve skalárním kontextu

Podobně jako mezi řetězci a čísly existují pravidla pro konverzi mezi seznamy a skaláry.

Různé funkce a operátory se chovají ke skalárům a seznamům jinak. Pracujeme-li se seznamem jako se skalární hodnotou (to lze, za chvíli přijde vysvětlení), Perl si s tím šalamounsky poradí. Takovýto seznam ve skalárním kontextu se tváří jako proměnná, jejíž hodnotou je počet prvků seznamu. Možná je to na první pohled neintuitivní řešení, ale této vlastnosti se skutečně často využívá. Na těchto řádcích kódu si ukažme, jak přesně to funguje:

```
@pole = qw(a b c d e f);
$skalar = @pole;
print $skalar;
```

Sledujme především druhý řádek. Protože je levý operand skalár, musí do něj být uložena skalární hodnota. Pravý operand tedy bude vyhodnocován také jako skalár, ať je čímkoliv. V našem případě se do proměnné $skalar přiřadí hodnota 6, což je počet prvků @pole.

Avšak pozor na následující příklad:

```
$skalar = (10, 20, 30); # POZOR!!!
print $skalar;
```

U seznamů, které nejsou poli, je to však jinak, než by se dalo podle dosavadních tvrzení čekat. Do $skalar je přiřazena poslední hodnota seznamu, tedy 30. Přitom, kdybychom přiřazovali do pole, získali bychom tříprvkové pole. Proč tomu tak je? Závorky zde Perl chápe jako operátor pro změnu priority a čárky jako klasický operátor čárky, který levý výraz vyhodnocuje a zapomíná.

Malý test - co kdybychom neuvedli závorky?

Perl obsahuje speciální funkci, která vyjadřuje seznam ve skalárním kontextu. Funkce scalar převádí hodnotu v parametru na skalární. Je-li skalární, nic se neděje. Ale jde-li o pole, funkce vrátí počet jeho prvků.

```
@pole = qw(a b c d e f);
print scalar @pole; # vytisknuto je opět 6.
```

Teď si vezměme opačný případ. Přiřazení skalární proměnné do seznamu. Přiřazením vznikne jednoprvkové pole, jehož nultou hodnotou je hodnota na pravé straně přiřazení. Skalární hodnota se konvertuje v jednoprvkový seznam.

```
$skalar = 666;
@pole = $skalar; # index 0: hodnota 666
```

Funkce print s parametrem v seznamovém kontextu

Předáme-li funkci print pole, vytiskne hodnoty všech prvků pole.

```
@pole = qw(a b c d e f);
print @pole; # vytiskne abcdef.
```

Nemusíme předávat už vytvořené pole, ale třeba jen seznam hodnot, oddělených čárkami

```
print "a", "b", "c"; # vytiskne abc.
```

Samozřejmě i v print můžeme použít uzavření do uvozovek pomocí qw:

```
print qw(a b c); # vytiskne abc.
```

Speciální proměnné pro pole

Perl obsahuje desítky speciálních proměnných, jež mají různý význam. My si nyní představíme tři z nich, se kterými se občas setkáme při práci s poli.

Proměnná	Význam	Implicitně

$,	řetězec, který bude tištěn mezi jednotlivými prvky seznamu (ne v řetězci)	""
$"	řetězec, který bude tištěn mezi jednotlivými prvky seznamu uvedeném v řetězci označeným uvozovkami.	" "
$\	řetězec, který bude vytištěn na konci seznamu	""

Sledujte, jak se změní poslední příklad použitím proměnných $, a $\:

```
$, = ", ";
$" = "-"; #v tomto případě neovlivní výsledek
$\ = "\n";

print "a", "b", "c";
```

Vytiskne se řetězec "a, b, c\n". Mezi jednotlivými prvky pole je vždy čárka a mezera a za poslední položkou je znak nového řádku.

$" působí na tisknuté pole, vložené do řetězce:

```
$, = ", "; #v tomto případě neovlivní výsledek
$" = "-"; #zkuste zakomentovat tento řádek, implicitním obsahem je mezera

@p = ("a", "b", "c");
print "@p";
```

Vytiskne se řetězec "a-b-c". Zkusme nyní pro zvýraznění rozdílu odstranit z příkazu print uvozovky. Pole by potom nebylo v řetězci, místo $" by se mezi jednotlivé prvky tiskla hodnota v proměnné $, a výsledkem by bylo "a, b, c".

Poslední prvek pole

Zápis $#pole vrací poslední (nejvyšší) index pole. Je ekvivalentní zápisu $pole[-1].

Operátor rozsahu

Přiřaďme nyní do prvků s indexy 5-14 pole @pole2 hodnoty prvků s indexy 2-11 pole @pole:

```
@pole2[5 .. 14] = @pole[2 .. 11];
```

Výraz 5 .. 14 má stejný význam jako 5, 6, 7, 8, 9, 10, 11, 12, 13, 14. Při použití na místě indexu pole jde o tzv. **řez polem**.

Na základě této informace vytvořme pole s hodnotami 1 až 100:

```
@pole = (1 .. 100);
```

Operátor rozsahu funguje i na malá a velká písmena anglické abecedy:

```
@pole = ("c" .. "z"); # výsledkem je pole obsahující znaky c až z
```

Podívejme se na několik dalších příkladů.

```
$, = ", ";
$\ = "\n";

print 1.88 .. 2.21;                 #stejné jako 1 .. 2. Desetinná část se odřízne
print "001" .. "010";                #001, 002, 003 ... 010\n
print "abc" .. "abx";                #abc, abd, abe ... abx\n
print "bc" .. "de";                  #bc, bd, be ... bz, ca, cb ... dd, de\n
print 0 .. 9, "A" .. "F";           #nic nového, pouze se tiskne seznam složený ze dvou seznamů
print 0 .. 3, "a" .. "c", $promenna, "XXX";#prakticky totéž
print -10 .. 10;                    #záporná čísla nejsou překážkou
print 10 .. 1;                      #ale pozpátku neumí
```

Setřídění pole - funkce sort

Pro řazení polí má Perl speciální vestavěnou funkci. Funkce sort řadí (implicitně) podle ASCII tabulky. Další funkce - reverse - obrací pořadí prvků pole. Poslední prvek prohodí s prvním, předposlední s druhým atd.

```
@p = qw(a g d f e b c); #spřeházený seznam
print @p;              #agdfebc
@p = sort @p;          #setřídí podle abecedy
print @p;              #abcdefg
@p = reverse @p;       #obrátí pořadí prvků
print @p;              #gfedcba
```

Tento systém má háček. Porovnává po znaku, zleva doprava, takže 6 je větší než 44. Na čísla je předchozí zápis nepoužitelný. Řazení čísel ale zvládne sort také. Slouží k tomu speciální konstrukce {$a <=> $b}. Porovnává vždy dva prvky pole, určuje, který je větší a takto je řadí. Proměnné $a a $b jsou dvě speciální proměnné určené pro tento účel. Když prohodíme jejich pořadí, setřídí se pole sestupně.

```
@p = (6, 1, 3, 5, 4, 2);  #přeházený seznam
print @p;                 #613542
@p2 = sort {$a <=> $b} @p;#setřídí podle velikosti vzestupně
print @p2;                #123456
@p3 = sort {$b <=> $a} @p;#setřídí podle velikosti sestupně
print @p3;                #654321
```

Poznamenejme, že sort bez uvedeného bloku vlastně znamená totéž, co bychom získali uvedením {$a cmp $b}. Do bloku lze napsat v podstatě cokoliv, co vrací 0, 1 nebo -1 na základě speciálních proměnných. Například pro setřídění názvů souborů podle času jejich vzniku bychom mohli napsat toto:

```
@files = sort {(stat($a))[9] <=> (stat($b))[9]} @files;
```

Nastavení jazyka

Dalším problémem pro třídění podle abecedy je čeština. Implicitně Perl třídí písmena s háčky až na konec a písmeno ch za c. To se dá vyřešit pomocí *locales*. V systému nastavme hodnotu proměnné LANG na czech:

```
$ export LANG=czech
```

Je dobré si tento řádek připsat do nějakého konfiguračního souboru, který se spouští při zapnutí systému.

Dalším krokem je přidat do samotného programu příkaz use locale.

```
#!/usr/bin/perl
use locale;

$, = ", ";
```

```
$\ = "\n";

@p = qw(a d f č ď e c ch u); #přeházený seznam
print sort @p;
```

Pokud máme správně nastavenou proměnnou prostředí LANG, měli bychom spatřit výstup:

```
$ ./sort.pl
a, c, č, d, ď, e, f, ch, u
$
```

Teď smažme řádek use locale;:

```
$ ./sort.pl
a, c, ch, d, e, f, u, č, š
$
```

Třídění pole - funkce grep

grep je analogií stejnojmenného unixového příkazu. Vyhledává prvky v seznamu, ve kterých byl nalezen určitý podřetězec a ty vrací jako seznam. Existují dvě možnosti zápisu grepu:

```
@vyhovujici_prvky = grep {test} seznam
@vyhovujici_prvky = grep /regulární_výraz/, seznam
```

V prvním řádku grep postupně přiřazuje každý prvek seznamu do výchozí proměnné $_, ověří, zda vyhovuje výrazu ve složených závorkách, a pokud ano, prvek se ocitne i ve výsledném poli.

```
$, = ", ";
@pole = (8, 2, 4, 6, 9, 5);
print grep {$_ < 7} @pole; #vytiskne 2, 4, 6, 5 - tedy ty prvky, které jsou menší než 7
```

V druhém zápisu je namísto testovacího výrazu použit vzor. To je záležitost regulárních výrazů. grep vrací seznam prvků z původního pole, které danému vzoru vyhovují

```
$, = ", ";
@pole = ("chomp", "print", "grep", "sin", "index");
print grep /^.+in/, @pole;
```

Vypsáno bude "print, sin" - tedy řetězce, které obsahují podřetězec in, ale nezačínají jím. Regulární výrazy budeme již brzy řešit v samostatných dílech.

Příklad - výčet potřebných platidel

Podívejme se opět na trochu delší ukázku programu. Zadání úlohy je následující: Napište program, který načte ze vstupu číslo, vyjadřující peněžitý obnos. Definujte pole, ve kterém budete mít seznam hodnot českých platidel (haléřové mince neuvažujeme - tedy v poli budou prvky pro 1 Kč, 2 Kč, 5 Kč, 10 Kč ... 2000 Kč, 5000 Kč). Obnos poté rozdělte na hodnoty jednotlivých platidel tak, aby byl rozložen do co nejvyšších platidel. Přitom se snažte o obecnost - tedy například přibude-li platidlo, musí stačit jeho hodnotu přiřadit do pole platidel. Ať pracuje program. Výsledky (kolik kterých platidel) tiskněte na výstup.

Řešení

Začátek je jasný, definujeme pole platidel:

```
#!/usr/bin/perl

@platidla = (1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000);
```

Načteme částku. Tady se na chvilku zastavíme. Musíme ošetřit hned tři nestandardní případy. Uživatel vůbec nemusí zadat číslo nebo může být záporné. Oba problémy vyřešíme tak, že otestujeme, zda je částka větší než 0 *(poznámka: pokud tedy uživatel zadá "506xyz", bude částka zkonvertována na 506)*. Dalším problémem jsou desetinná čísla. Je zadáno, že haléře máme zanedbat, takže funkcí int zadanou haléřovou částku ořízneme.

```
print "Zadej částku v Kč: ";
$castka = int <STDIN>;

if ($castka < 1){
die "Toto není regulérní částka.\n";
}
```

Zamysleme se nad tím, jak budeme postupovat dál. Určitě budeme mít nějaký cyklus. V každé jeho iteraci budeme odečítat příslušný počet jednoho druhu platidla. Protože mají být platidla maximálně vysoké, bude výhodné v první iteraci odečítat nejvyšší platidlo, v druhé iteraci druhé nejvyšší atd. Proto nejprve seřaďme pole sestupně:

```
@platidla = sort {$b <=> $a} @platidla;
```

Zbývá najít nějaký vhodný test, který by rozhodoval o pokračování cyklu. Iterovat se bude tolikkrát, kolik prvků má pole @platidla. Zaveďme ještě před cyklem proměnnou $pozice, kterou nastavíme na 0. Každou iterací se inkrementuje. Až nabude hodnoty stejné jako počet prvků pole @platidla, cyklus se zastaví. A protože předem známe počet iterací, použijeme cyklus for. Nic nám nebrání začít se samotným cyklem.

```
for my $pozice (0 .. @platidla){

... n krát odečtení hodnoty platidla ...

}
```

Teď je třeba se zamyslet, jak to bude vypadat v cyklu. Zjistíme počet kusů daného platidla, které lze maximálně odečíst, aby byl výsledek kladný. Stačí od zbývající částky odečíst zbytek, který by zbyl po dělení hodnotou aktuálního platidla - výsledkem bude nejvyšší částka dělitelná hodnotou platidla, ale stále nižší než zbývající částka. Tuto vzniklou částku vydělíme aktuální hodnotou platidla:

```
$pocet_kusu = ($castka - ($castka % $platidla[$pozice])) / $platidla[$pozice];
```

Získali jsme počet kusů aktuálního platidla, který lze odečíst od zbývající částky. Tak to udělejme:

```
$castka -=  $pocet_kusu * $platidla[$pozice];
```

Poznámka: Možná jste si všimli, že pokud bychom tyto dva řádky spojili do jednoho (za $pocet_kusu v druhém dosadili hodnotu prvního), získali bychom zlomek, který obsahuje v čitateli i jmenovateli hodnotu $platidla[$pozice], a tudíž jde zkrátit. $pocet_kusu ale ještě budeme potřebovat.

To je téměř celé. Zbývá zařídit nějaký výstup. Využijme cyklu a výsledky budeme vypisovat přímo v něm. Ještě před cyklus ale vložme pro pořádek velikost původního obnosu:

```
print "K vyplacení $castka Kč bude potřeba:\n";
```
A v cyklu bude:
```
print "$pocet_kusu kusů platidla $platidla[$pozice].\n";
```
Jsme hotovi, nyní se pokochejme pohledem na chod právě vzniklého programu:
```
$ ./vycet.pl
Zadej částku v Kč: 19749
K vyplacení 19749 Kč bude potřeba:
3 kusů platidla 5000.
2 kusů platidla 2000.
0 kusů platidla 1000.
1 kusů platidla 500.
1 kusů platidla 200.
0 kusů platidla 100.
0 kusů platidla 50.
2 kusů platidla 20.
0 kusů platidla 10.
1 kusů platidla 5.
2 kusů platidla 2.
0 kusů platidla 1.
$
```
Zkusme do pole @platidla přidat ještě bankovku v hodnotě 10000. Mělo by to teoreticky také fungovat.
```
$ ./vycet.pl
Zadej částku v Kč: 19749
K vyplacení 19749 Kč bude potřeba:
1 kusů platidla 10000.
1 kusů platidla 5000.
2 kusů platidla 2000.
0 kusů platidla 1000.
1 kusů platidla 500.
1 kusů platidla 200.
0 kusů platidla 100.
0 kusů platidla 50.
2 kusů platidla 20.
0 kusů platidla 10.
1 kusů platidla 5.
2 kusů platidla 2.
0 kusů platidla 1.
$
```
Možná vám přijde, že není jednoduché se ve výstupu zorientovat okamžitě. Je tam moc řádků. Ještě program trochu pozměníme. Není-li od nějakého platidla ani kus, nebudeme řádek výbec vypisovat. Použijme konstrukci podmínka za příkazem.
Vykonání příkazu print v cyklu podmíníme nenulovým počtem kusů daného platidla:
```
print "$pocet_kusu kusů platidla $platidla[$pozice].\n" if $pocet_kusu;
```
Ještě odstraníme platidlo 10000 a program opět vyzkoušíme:
```
$ ./vycet.pl
Zadej částku v Kč: 19749
K vyplacení 19749 Kč bude potřeba:
3 kusů platidla 5000.
2 kusů platidla 2000.
1 kusů platidla 500.
1 kusů platidla 200.
2 kusů platidla 20.
1 kusů platidla 5.
2 kusů platidla 2.
$
```
Je to hned o něco lepší. Celý zdrojový kód je možné stáhnout. K dokonalosti chybí přidat deklarace proměnných a zformátovat výstup. Formátování časem budeme věnovat celý díl.
Jako bonus si můžete zkusit program upravit tak, aby správně česky skloňoval slovo kus.

Perl (13) - Hashe

Hash je další varianta seznamu, kdy indexem jednotlivých hodnot může být jakákoliv skalární hodnota.

Při práci s poli jsme byli dosud omezeni na číselné indexy, které jsou uspořádané. Lze použít i řetězce (v takovém případě spíše než o indexu hovoříme o klíči). To pak již nejde o klasické pole, ale o hash (někdy též nazývaný asociativní pole). Každý prvek hashe obsahuje dvě skalární hodnoty. Můžeme říct, že jde o seznam uspořádaných dvojic, avšak jejich postavení není symetrické, neboť klíč musí být jednoznačný. Hash si můžeme představit i jako matematickou funkci.
S hashi se pracuje podobně jako s poli. Na první pohled jsou zřejmé dvě odlišnosti. Hash se označuje procentem a klíč se píše do složených (nikoliv hranatých) závorek. Stejně jako u polí platí, že podle klíče prvku lze jednoznačně zjistit hodnotu, ale nikoliv naopak.
Hash plníme hodnotami takto:
```
%hash = (klic1, hodnota1, klic2, hodnota2, ..., klicn, hodnotan);
```
Konkrétně:
```
%vzdalenosti = ("Amsterodam", 970, "Moskva", 1900, "Kodaň", 750, "Řím", 1300, "Varšava", 630);
```
Na uvození parametrů můžeme samozřejmě použít i nám již dobře známou funkci qw. Následující zápis je ekvivalentní poslednímu uvedenému příkazu:
```
%vzdalenosti = qw(Amsterodam 970 Moskva 1900 Kodaň 750 Řím 1300 Varšava 630);
```
Další synonymní zápis získáme nahrazením každého lichého operátoru čárky operátorem =>. Operátory , a =>mají stejný význam, takže teoreticky by měly jít nahradit i sudé čárky. K tomu ale není důvod. => se užívá pro lepší přehlednost. (V našem

případě je sice jasné, co je klíčem a co hodnotou. Ale představme si, že by obojí byly na první pohled od sebe nerozeznatelné řetězce.)

```
%vzdalenosti = ("Amsterodam" => 970, "Moskva" => 1900, "Kodaň" => 750, "Řím" => 1300, "Varšava" => 630);
```

A nakonec úplně nejpřehlednější zápis získáme rozepsáním kódu do více řádků. Každý pár *klíč* => *hodnota* bude na jednom.

```
%vzdalenosti = (
"Amsterodam" =>  970,
"Moskva"     => 1900,
"Kodaň"      =>  750,
"Řím"        => 1300,
"Varšava"    =>  630
);
```

Navíc se často u klíče hashe vynechávají uvozovky.

```
%vzdalenosti = (
Amsterodam =>  970,
Moskva     => 1900,
Kodaň      =>  750,
Řím        => 1300,
Varšava    =>  630
);
```

Každý z předcházejících kódů vytvořil hash s těmito prvky:

Klíč	Hodnota
Amsterodam	970
Moskva	1900
Kodaň	750
Řím	1300
Varšava	630

Jednotlivé hodnoty získáme obdobně jako u polí. Jak daleko je to z Prahy do Moskvy?

```
print $vzdalenosti{"Moskva"};
```

Vidíme, že je to stejné jako s poli, jen se, jak už bylo zmíněno, klíč píše do složených závorek.

Konverze pole - hash

Přiřadíme-li pole do hashe, je to jako byste přiřazovali seznam (tedy hodnoty prvků pole).

```
@pole = (1, 2, 3, 4, 5, 6);
%hash = @pole;
```

Zápis má (pro konečný obsah hashe %hash) stejný význam jako tento:

```
%hash = (1, 2, 3, 4, 5, 6);
```

a proto %hash bude obsahovat prvky:

Klíč	Hodnota
1	2
3	4
5	6

Konverze hash - pole

Konverze opačným směrem je samozřejmě také možná. Hash je rozložen na hodnoty *klic1*, *hodnota1*, *klic2*, *hodnota2*... a ty jsou přiřazeny do pole:

```
@pole = %vzdalenosti;
```

@pole potom obsahuje prvky:

Klíč	Hodnota
0	Varšava
1	630
2	Amsterodam
3	970
4	Kodaň
5	750
6	Moskva
7	1900
8	Řím
9	1300

Chování funkce print

Uvedeme-li jako argument hash, nejdříve se zkonvertuje na pole a až vzniklé pole se tiskne.

```
$, = ", ";
print %vzdalenosti;
```

Výstupem tedy bude text

Varšava, 630, Amsterodam, 970, Kodaň, 750, Moskva, 1900, Řím, 1300

Funkce exists

Testuje, zda existuje klíč pole. Přičemž nezáleží na tom, zda má definovanou hodnotu. Funkce exists se často používá v podmínkách.

```
print exists($vzdalenosti{"Amsterodam"});#1
print exists($vzdalenosti{"Tokyo"}); #""
```

Funkce delete

Maže klíč a vrací jeho hodnotu.

```
print exists($vzdalenosti{"Řím"});#1
print delete($vzdalenosti{"Řím"});#130
print exists($vzdalenosti{"Řím"});#""
```

Funkce each

Čte prvek hashe.

```
while (($mesto, $vzdalenost) = each(%vzdalenosti)){
print "$mesto - $vzdalenost\n";
}
```

Příklad vypíše všechny klíče a hodnoty pole. Funkce each vrací klíč a hodnotu prvku. Přitom si pamatuje, které klíče prvky již vrátila a při opakovaném volání v cyklu vrací jiný prvek. Podotkněme, že pro tento příklad se spíše hodí cyklus for.

```
$ perl each.pl
Varšava - 630
Amsterodam - 970
Kodaň - 750
Moskva - 1900
Řím - 1300
$
```

Funkce keys

keys vrací pole prvků s hodnotami, které jsou klíči hashe.

```
print keys %vzdalenosti;#tiskne klíče všech prvků hashe %vzdalenosti
```

Vytiskněme ještě klíče v abecedním pořádku:

```
print sort keys %vzdalenosti;
```

Funkce values

Vrací hodnoty všech prvků hashe.

```
print values %vzdalenosti;#tiskne hodnoty všech prvků hashe %vzdalenosti
```

Počet prvků hashe

Zjistíme ho sečtením klíčů hashe:

```
print scalar keys %vzdalenosti;
```

Příklad - slovník

Dnešní díl zakončíme vytvořením programu, který načte ze vstupu slovo a pokud ho má v databázi pojmů, vytiskne jeho význam. Zatím budeme pojmy definovat přímo v programu, což není dobré řešení. Někdy později si předvedeme, jak k tomuto účelu použít externí soubor nebo, ještě lépe, databázi.

Řešení

Ze všeho nejdříve bude nutné definovat pár pojmů.

```
#!/usr/bin/perl
use strict;

my $slovo;#bude obsahovat hledaný pojem
my %pojmy = (
"perl" => "programovací jazyk",
"ankara" => "turecké hlavní město",
"uran" => "chemický prvek",
"Uran" => "planeta",
"klaus" => "2. prezident České republiky",
"dioda" => "elektrotechnická součástka"
);
```

Dále načteme pojem:

```
print "Zadej pojem: ";
chomp($slovo = <STDIN>);
```

Hash necháme prozkoumat cyklem foreach (probereme ho sice až v příštím díle, ale věřte, že do zadané proměnné přiřazuje postupně každý prvek zadaného pole) a pokud se hledané slovo shoduje s názvem pojmu vypíšeme informace a ukončíme. Přitom převedeme všechna písmena na malá, aby nezáleželo na velikosti písmen načteného řetězce. Z tohoto důvodu dále musíme zajistit, aby se tiskly všechny pojmy i v případě, že existuje více pojmů, lišících se jen ve velikosti (tedy konkrétněji - pokud bude dotaz znít URAN, ve výsledcích se musí objevit Uran i uran).

Poznámka - ano, je pravda, že takto nebudeme schopni definovat dva stejné pojmy, které se neliší ani ve velikosti (kupříkladu Uran - planeta a Uran - bůh). Pouze můžeme do jedné hodnoty prvku vepsat oba významy.

```
foreach my $pojem (keys %pojmy){
if (lc $pojem eq lc $slovo){
print "POJEM $pojem --- $pojmy{$pojem}\n";
}
}
```

To je téměř vše. Zbývá nám vytisknout nějakou hlášku v případě, že ve slovníku pojem odpovídající zadanému není. K tomu si definujeme další proměnnou $uspech,

```
my $uspech = 0;
```

ve které bude pravdivá hodnota, pokud byl hledaný pojem nalezen. V opačném případě zůstane hodnota nepravdivá. Pokud byl pojem nalezen, přiřadíme do ní pravdivou hodnotu. To zapíšeme do těla podmínky uvnitř cyklu.

```
$uspech = 1;
```

A v případě neúspěchu tiskneme chybové hlášení.

```
print "Zadaný pojem není ve slovníku :(\n" if !$uspech;
```
Zkusme spustit náš program (zdrojový kód):
```
$ ./slovnik.pl
Zadej pojem: uran
POJEM Uran --- planeta
POJEM uran --- chemický prvek
$ ./slovnik.pl
Zadej pojem: zinek
Zadaný pojem není ve slovníku :(
$
```
Až budeme znát regulární výrazy, budeme umět napsat slovník, který hledá i podle části názvu. Další věcí, která by našemu slovníku slušela, by bylo předání hledaného pojmu jako argumentu příkazové řádky.

Perl (14) - Další nástroje pro seznamy

Konečně se podíváme na cyklus foreach, což je jedna z nejpoužívanějších konstrukcí vůbec. Dále je v plánu několik nových funkcí a vstup v souvislosti se seznamy.

Procházení pole - cyklus foreach

foreach je cyklus, který se používá v souvislosti s poli, protože umožňuje jejich snadné procházení. Název se skládá ze slov for a each, což už mnoho napovídá o jeho významu. Standardně se foreach zkracuje na for.
```
@ptaci = ("kos", "vrabec", "papoušek", "datel"); for $klic (@ptaci){ print $klic, "\n"; }
```
Do proměnné $klic se každou iteraci cyklu přiřadí jedna hodnota pole. Tedy postupně hodnoty $ptaci[0], $ptaci[1], $ptaci[2], $ptaci[3]. Toho užijeme pro výpis dat z pole:
```
$ perl foreach.pl kos vrabec papoušek datel $
```
Chceme-li vypisovat hashe, bude se hodit funkce keys, která vrací pole klíčů hashe. Pomocí klíčů totiž lze získat i hodnoty.
```
%ptaci = (
"kos"      => "slunečnice",
"vrabec"   => "slunečnice",
"papoušek" => "proso",
"datel"    => "hmyz"
);

for $klic (keys %ptaci){
print "$ptaci{$klic} je potrava pro $klic\n";
}
```
Doufám, že jsem nějakého opeřence výběrem jídla neurazil. Každopádně odtud vidíme, že to skutečně funguje:
```
$ perl foreach.pl slunečnice je potrava pro kos slunečnice je potrava pro vrabec proso je potrava pro papoušek hmyz je potrava pro datel $
```
Cyklus by měl fungovat stejně i v případě, že přepíšeme for na foreach.

for lze napsat i za příkaz
```
print "$_, " foreach (1, 2, 3, 4, 5); print "$_, " for (1, 2, 3, 4, 5);
```
případně za blok.
```
@p = (1, 2, 3, 4, 5); do {$_ += 10; print "$_, "; } for @p;
```
Proměnná $_ zatím v tomto seriálu nebyla blíže rozebírána. Bude tak učiněno v příštím díle. Jde o implicitní proměnnou, do které se hodnoty pole postupně přiřazují.

chomp po složkách

Uvedeme-li jako parametr funkce chomp pole, je to jako byste použili funkci chomp na všechny skaláry, které pole obsahuje. Je-li parametrem hash, aplikuje se chomp pouze na hodnoty - tedy klíče zůstanou nezměněny.

Odstraňování hodnot z pole - funkce pop a shift

Jak z názvu vyplývá, pop odstraňuje prvek pole z konce seznamu (ten s nejvyšším indexem) a shift ze začátku. Obě funkce rovněž odstraněnou hodnotu vracejí.
```
@p = 1 .. 5;

print pop @p;#5
print @p;#1234

shift @p;
print @p;#234
```
Přidávání hodnot do pole - funkce push a unshift

push přidává prvek na konec seznamu, unshift na začátek. Obě funkce vrací jako návratovou hodnotou hodnotu přidaného prvku.
```
@p = 1 .. 5;

push @p, 6;
print @p;#123456

unshift @p, 7;
print @p;#7123456
```
Nahrazování hodnot v poli - funkce splice

Odstraňuje, přidává a nahrazuje prvky v poli. Má 4 parametry.

- název proměnné typu pole, na kterou se bude příkaz vztahovat
- pozice, za kterou dojde k manipulaci s polem
- počet znaků, se kterými bude manipulováno. Nezadáme-li ho, je koncová pozice automaticky nastavena na konec řetězce
- seznam prvků, které budou přidány

Mějme pole čísel:

```
$, = ", ";
@p = 1 .. 20;
```
Odstraňme prvky 8 - 11:
```
@odstranene = splice(@p, 7, 4);
```
Jak vidíme, vrací funkce jako návratovou hodnotu pole odstraněných prvků:
```
print @odstranene;#8, 9, 10, 11
```
Samotné pole @p obsahuje:
```
print @p;#1, 2, 3, 4, 5, 6, 7, 12, 13, 14, 15, 16, 17, 18, 19, 20
```
Odstraňme všechny prvky od prvku s hodnotou 13:
```
splice(@p, 8);
print @p;#1, 2, 3, 4, 5, 6, 7, 12
```
Prvky s indexy 1 a 2 nahradíme jinými. Lze přidat jiný počet prvků, než kolik jich je odebráno. Též je možné zadat jako třetí parametr 0 - v tom případě se budou prvky jen přidávat.
```
splice(@p, 1, 1, 2000, 2002, 2004, 2006);
print @p;#1, 2000, 2002, 2004, 2006, 3, 4, 5, 6, 7, 12
```
A nakonec smažme všechny prvky:
```
splice @p;
print @p;#""
```
Pole a vstup

Když jsme načítali vstup ve skalárním kontextu, vždy jsme získali jen jeden řádek ze zdroje dat. Přiřazujeme-li zdroj dat do pole, vytvoří se tolik prvků, kolik mají data řádků. Hodnotami prvků jsou právě řádky.

Modifikujme náš příklad s českými králi ze 7. dílu tak, aby se vypsali všichni, kteří jsou v souboru uloženi. Připomeňme si datový soubor kralove:
```
Přemysl Otakar I. 1197-1230
Václav I. 1230-1253
Přemysl Otakar II. 1253-1278
```
Jen zaměníme skalární proměnnou za pole:
```
#!/usr/bin/perl
use strict;

my @radky = <>;
print @radky;
```
A dostaneme všechny tři řádky.
```
$./kralove.pl kralove
Přemysl Otakar I. 1197-1230
Václav I. 1230-1253
Přemysl Otakar II. 1253-1278
$
```
Rozdělení řetězce do prvků pole - funkce split

split rozdělí řetězec na několik podřetězců a ty uloží do seznamu. Funkce hledá v řetězci výskyty oddělovače (první parametr). Oddělovačem je regulární výraz, ale zatím ho chápejme jako obyčejný řetězec, který je uvozen lomítky. Výskyty oddělovačů určují právě začátek a konec výsledných podřetězců. Oddělovač se ale v podřetězcích neobjevuje. Druhým parametrem je samotný řetězec. Lze uvést i třetí parametr, který určuje, kolik nejvíce může podřetězců být. Po vyhledání daného počtu oddělovačů je posledním podřetězcem celý text za posledním výskytem, přičemž nezáleží, zda tam nějaký výskyt ještě není.
```
$veta = "Perl letos slaví 18. narozeniny!";

($slovo1, $slovo2, $slovo3) = split(/ /, $veta);
print $slovo1;#Perl
print $slovo2;#letos
print $slovo3;#slaví
```
Oddělovačem byla mezera - tímto způsobem získáme (ve většině případů) jednotlivá slova. Ve výsledných podřetězcích již mezery nebudou.
```
@vsechna_slova = split(/ /, $veta, 4);
```
Na posledním řádku byl text rozdělen na 4 podřetězce a ty jsou následně přiřazeny do pole. Každý podřetězec bude hodnotou prvku. Pokračujme ale v kódu:
```
foreach $slovo (@vsechna_slova){
print "$slovo, ";
}
```
Každou iterací bylo vytisknuto jedno slovo věty a za ním čárka. Při páté iteraci se vytiskl zbytek textu.

Ještě jednou dnes dostanou slovo králové. Uděláme z řádků věty tak, že před období vlády přidáme text. Předpokládejme, že období je vždy za poslední mezerou v řádku.
```
while ($radek = <>){
chomp $radek;
@p = split(/ /, $radek);
$obdobi = pop @p;
print "@p vládl v letech $obdobi.\n";
}
```
Sám algoritmus funguje tak, že rozdělíme řádek mezerami, poslední prvek (období) přesuneme do jiné proměnné a můžeme tisknout.
```
$ perl kralove.pl kralove
Přemysl Otakar I. vládl v letech 1197-1230.
Václav I. vládl v letech 1230-1253.
Přemysl Otakar II. vládl v letech 1253-1278.
$
```

Spojování prvků do řetězce - funkce join

Zatímco split řetězce rozděluje, join je naopak spojuje. Jeho použití velmi podobné jako u split. Prvním parametrem je spojovač, který bude vložen do výsledného řetězce mezi každé dvě hodnoty seznamu.

```
$rada[0] = 0;
for ($i=1; $i<10; $i++){
$rada[$i] = $i + $rada[$i-1];
}

print join(" ", @rada); #0 1 3 6 10 15 21 28 36 45
print join(", ", @rada); #0, 1, 3, 6, 10, 15, 21, 28, 36, 45
print join(" *** ", @rada); #0 *** 1 *** 3 *** 6 *** 10 *** 15 *** 21 *** 28 *** 36 *** 45
print join("", @rada); #0136101521283645
```

Perl (15) - Výchozí proměnná, heredoc, symbolické odkazy

15. díl nemá jednotné téma. Půjde vesměs o věci, které jsou na celý díl krátké, dosud se nikam výrazněji nehodily a přesto je užitečné něco o nich tušit.

Ještě dříve, než se pustíme do tajemných hlubin regulárních výrazů, bychom se měli seznámit s několika tématy. Proto vznikl tento díl.

Výchozí proměnná $_

$_ je proměnná, kam se implicitně ukládají skalární hodnoty. Typicky - nezadáme-li název proměnné někde, kde je vyžadován, použije se právě $_. Význam je pak stejný, jako bychom použili název $_. Toto pravidlo neplatí všude, ale podporují ho pouze některé konstrukce nebo funkce.

```
while ($radek = <>){
if ($radek == 0){
exit;
}
print $radek;
}
```

Tento kód můžeme přepsat pomocí $_:

```
while (<>){
if ($_ == 0){
exit;
}
print $_;
}
```

Uvedeme-li jako test jen <>, bude vstup implicitně uložen do proměnné $_ (tedy stejné jako $_ = <>). Příklad můžeme ještě zkrátit. Pokud není funkci print předán žádný parametr, automaticky tiskne hodnota $_:

```
while (<>){
if ($_ == 0){
exit;
}
print;
}
```

Takovým způsobem se nechová jen print, ale řada dalších funkcí. To, jestli funkce pracuje s výchozí proměnnou zjistíme vždy v manuálové stránce perlfunc nebo perlvar.

Velmi často se výchozí proměnná také používá u cyklu for. Pokud neuvedeme v hlavičce cyklu skalární proměnnou, použije se právě $_. Díky tomu bude fungovat například následující zápis, který výpíše třetí mocniny čísel od 1 do 10:

```
print $_ ** 3, "\n" for (1..10);
```

Proměnná s názvem určeným vyhodnocením výrazu

Za proměnnou v řetězci musí být vždy mezera. Jednou z možností, jak toto "pravidlo" obejít, je uzavření názvu proměnné do složených závorek.

```
$slovo = "TEXT";
print "zhusteny${slovo}bezMEZER.";
```

Složené závorky umí ještě obecnější věc. Název proměnné lze získat vyhodnocením výrazu ve složených závorkách.

```
$p = 1;
${"p" x 3} = 2; #"p" x 3 je vyhodnoceno jako ppp, což je název proměnné
${if($ppp){$p?"a":"b"}else{$p?"c":"d"}} = 3;

print $ppp;#2
print $a;#3
print $b;#nedefinováno
print $c;#nedefinováno
print $d;#nedefinováno
```

Jak vidíme, můžeme zakomponovat i celé konstrukce jako je if. Jedinou podmínkou je, aby byl úsek kódu ve složených závorkách vyhodnocen jako řetězec, který je platným identifikátorem. Předchozí výraz je vyhodnocen jako "a". Proto bude hodnota přiřazena do proměnné $a.

Nicméně tuto vlastnost opět uvádíme spíše pro úplnost než z praktického hlediska. Mohou s ní být problémy (a to nejen ve strict režimu, který ji zakazuje) a v praxi ji tedy moc neužijeme.

Funkce undef

Funkce undef vrací nedefinovanou hodnotu. Takže, například, potřebujeme-li vytvořit pole, v němž budou mít indexy 1, 3, 4 nedefinované hodnoty, můžeme udělat toto:

```
@p = ("a", undef, "b", undef, undef, "c");
```

Speciální syntaxe zápisu řetězce

Syntaxe here-document

Potřebujete-li vytisknout nějaký delší text, jistě přivítáte tzv. here-document možnost zápisu.

```
$zaznamu = 91;
```

```
print << 'KONEC';
Vyberte z těchto voleb:
+----------------------+
| 1 ... Přidat záznam|
| 2 ... Odebrat záznam |
| 3 ... Konec        |
+----------------------+
Záznamů: $zaznamu\n
KONEC
```

Tisknuto je vše, dokud není nalezeno slovo KONEC na samostatném řádku (avšak který nesmí být zároveň posledním v souboru). Můžeme zvolit libovolné termiální slovo. Je to stejné jako např. v shellech (aneb vybavme si příkazy jako cat > soubor <<EOF).

Protože se k uvození nepoužívají ani apostrofy ani uvozovky, lze je normálně používat v tisknutém řetězci.

```
$ perl print.pl
Vyberte z těchto voleb:
+----------------------+
| 1 ... Přidat záznam  |
| 2 ... Odebrat záznam |
| 3 ... Konec          |
+----------------------+
Záznamů: $zaznamu\n
$
```

Z předchozího výpisu je zřejmé, že nebyly nahrazeny proměnné a escape znaky. To je samozřejmě možné ovlivnit. Nyní je nahradíme jejich obsahem. Změňme apostrofy kolem slova KONEC na uvozovky:

```
$zaznamu = 91;
print << "KONEC";
Vyberte z těchto voleb:
+----------------------+
| 1 ... Přidat záznam  |
| 2 ... Odebrat záznam |
| 3 ... Konec          |
+----------------------+
Záznamů: $zaznamu\n
KONEC
```

Potom se text chová jako řetězec v uvozovkách. Místo řetězce "$zaznamu" se oproti minulému příkladu vytiskla hodnota této proměnné.

```
$ perl print.pl
Vyberte z těchto voleb:
+----------------------+
| 1 ... Přidat záznam  |
| 2 ... Odebrat záznam |
| 3 ... Konec          |
+----------------------+
Záznamů: 91

$
```

Tato syntaxe není svázaná s příkazem print, jak by se doposud mohlo zdát. Lze ji použít prakticky kdekoliv místo řetězce. Tedy například při přiřazení do proměnné.

```
$x = << 'KONEC';
+----------------------+
| 1 ... Přidat záznam  |
| 2 ... Odebrat záznam |
| 3 ... Konec          |
+----------------------+
KONEC
```

Jistou nevýhodou syntaxe here-document v Perlu je, že nelze vložený řetězec odsazovat. Vzniká tím poměrně nečitelný kód. Pokud se o odsazení pokusíme, budou mezery, kterými odsadíme, součástí řetězce. Nicméně v Perlu 6 bude tato věc řešena. Pokud trváme na odsazení, existuje částečné řešení už nyní. Pomocí regulárních výrazů lze smazat všechna bílá místa na začátku řádku.

```
($x = << 'KONEC') =~ s/^\s+//gm;
radek 1
radek 2
KONEC
```

Předchozí kód je také ukázkou dalšího způsobu použití této syntaxe.

To není o here-document zdaleka vše. Koho zajímají další možnosti, tak si může najít informace v manuálové stránce perlop v části Regexp Quote-Like Operators.

Syntaxe q a qq

Další možností zápisu řetězce je použít uvození pomocí q a qq.

Zápis pomocí q napodobuje řetězec v apostrofech. Liší se tím, že apostrofy v řetězci uvozeném pomocí q můžeme normálně používat. Nahrazuje se zpětné lomítko a uvozovací znak - v našem případě lomítko.

```
$x = 99;
print q/řetězec jako v apostrofech: ' \n " $x \\ \/;
```

Vytiskne:

```
$ perl q.pl
řetězec jako v apostrofech: ' \n " $x \ /
$
```

Podobně můžeme simulovat uvozovky. Stačí zaměnit q za qq:
```
$x = 99; print qq/řetězec jako v uvozovkách: ' \n " $x \\ \/;
```
Escape znaky i proměnné se nyní normálně nahrazují hodnotami.
```
$ perl q.pl
řetězec jako v uvozovkách: '
" 99 \ /
$
```
Není nutné uvozovat lomítkem. Použijme cokoliv, co se, pokud možno, v textu minimálně objevuje. Uvozovací znak v textu předřadíme zpětným lomítkem. Použijeme-li některou ze závorek, je počátečním znakem otevření koncovým uzavření (to platí pro (), {}, [] a <>). Používáme-li křížek jako uvozovací znak, nesmí před ním být bílé místo. V tom případě bude totiž brán jako komentář.
```
print q{řetězec jako v apostrofech: ' \n " $x \\ \/}; print q#řetězec jako v apostrofech: ' \n " $x \\ \/#; print q #komentář#;
```
To samé lze dělat s qq.
```
print qq"řetězec jako v uvozovkách: ' \n \" $x \\ \/";
```
Perl (16) - Regulární výrazy - začínáme

Dneškem začíná v rámci seriálu Perl miniseriál o nesmírně mocném nástroji - regulárních výrazech.

Regulární výrazy (anglicky regular expressions, někdy zkráceně označované jako regexp, regex nebo RE) jsou reprezentací regulárních jazyků v teorii konečných automatů.
Jejich aplikace (ve smyslu zda řetězce odpovídají vzoru) se poprvé objevila v unixových nástrojích pro editaci a vyhledávání v řetězcích - tedy sed, vi, grep, awk a další (více o historii například na Wikipedii). Nejsou mezi sebou úplně kompatibilní, protože většinou bylo k původním regulárním výrazům něco přidáno (potom tedy reprezentují nadmnožinu regulárních jazyků). V praxi se to projevuje tak, že to, co funguje awku, nemusí fungovat v grepu nebo může mít jinou syntaxi apod.
Mezi nástroje, které umí regulární výrazy se samozřejmě řadí i Perl. Jen těžko byste hledali jiný jazyk, který by podporoval regulární výrazy odpovídající tak silné množině jazyků. Navíc spojením kvalitního skriptovacího jazyka s regulárními výrazy získáváme do svého arzenálu nesmírně silný nástroj. Toto je jeden z hlavních důvodů, proč se Perl těší oblibě, kterou má. Proto ani zde nebudeme regulárními výrazy šetřit a pokusíme se je přiblížit opravdu podrobně.

Co je to regulární výraz?

Regulární výraz si můžeme představit jako speciální řetězec, který je šablonou vystihující určitý jazyk (tj. množinu textových řetězců). Každý textový řetězec takové šabloně buď vyhovuje nebo ne. S touto množinou vyhovujících řetězců můžeme manipulovat - hledat její prvky v textu nebo je nahrazovat jiným řetězcem.
Vzpomeňme si, že jsme na regulární výrazy již natrefili při popisu funkcí grep a split. To jsme ale jen uvedli příklady a dále je nerozebírali.

Formální definice regulárních výrazů

Nechť X = {x_1, ..., x_n} je nějaká konečná neprázdná abeceda a l označuje prázdné slovo. Pak množinou všech regulárních výrazů RE(X) nad abecedou X je nejmenší množina slov v abecedě {x_1, ..., x_n, {}, l, *, +, ., (,)}, pro kterou platí:

- RE(X) obsahuje každý prvek abecedy X, dále pak l a {}
- Jsou-li r, s v RE(X), pak je tam i r+s a r.s
- Je-li r v RE(X), pak je tam i r*

Regulárním výrazem tedy je například (((a*((b+c).d)*)+e)*f). Některé závorky a tečky lze po dohodě vynechat a můžeme psát ((a*(b+cd)*)+e)*f.
Hodnotou regulárního výrazu r je jazyk L = [r], pro který platí:

- [{}] = {}
- [l] = {l}
- [x] = {x} pro všechna písmena z abecedy X
- [(r+s)] = [r] sjednoceno s [s]
- [(r.s)] = [r] . [s]
- [r*] = [r]*

Regulární výrazy odpovídají regulárním jazykům. Regulární výrazy v Perlu jsou nadmnožinou formální definice. Třeba proto, že obsahují i jazyk L = {0^n1^n}, který regulární není. Ačkoliv to není formálně v pořádku, budeme je nazývat regulárními výrazy i nadále.

Regulární výrazy versus žolíkové znaky

Regulární výrazy jsou někdy zaměňovány se žolíkovými znaky. Obecně, žolíkový znak může vyhovovat jednomu nebo více znakům. Oproti tomu regulární výraz je řetězec s jasně určenými podmínkami a žádný samotný znak zde nezastupuje více znaků. Například * v žolíkových znacích znamená libovolný řetězec, ale v regulárních výrazech libovolný počet opakování. Žolíkové znaky se používají téměř výhradně k hledání v názvech souborů.

Regulární výrazy v Perlu

Základní syntaxe

Regulární výraz (od teď již jen v perlové terminologii) je výraz, který má dvě možné vyhodnocení - buď true nebo false. To znamená, že následující zápis vrací 1, pokud se v řetězci vyskytuje podřetězec "vzor", v opačném případě prázdný řetězec.
```
"řetězec" =~ m/vzor/
```
Regulárnímu výrazu m/vzor/ vyhoví takové řetězce jako "vzory", "re vzor" nebo "vzor". Operátor =~ slouží k porovnávání vzorů. S přiřazováním má společný opravdu jen ten znak rovnítka, jinak jde o odlišné operace.
Následující díly seriálu se tedy budou věnovat prakticky pouze tomu, co napsat místo vzoru, abychom vytvořili požadovanou šablonu. Dodejme, že v Perlu 6 dojde ke kompletnímu přepracování regulárních výrazů a už nepůjde jen o úpravu jednoho řetězce. Budou mít podstatně intuitivnější strukturu, více integrované do jazyka a navíc ještě o něco silnější.

Další možná uvození

Uvozující m není povinné, navíc můžeme stejně jako u nám už známých q a qq lomítka nahradit jiným znakem. Negovaný operátor =~ se zapisuje jako !~.
```
print "řetězec" =~ m"vzor";
print "řetězec" =~ m!vzor!;
print "řetězec" =~ /vzor/; #nejběžnější užití
print "řetězec" !~ /vzor/;
```

Existuje několik speciálních ohraničení pro regulární výrazy. Použijí-li se apostrofy, neprobíhá vkládání obsahu proměnných. Nelze tedy psát:

```
$vzor = "[abc]{2}";
"aa" =~ m'$vzor'; #$vzor není nahrazen obsahem proměnné, ale je brán jako řetězec '$vzor'
```

Další možností je ohraničení otazníky (potom uvedení m není povinné). V takovém případě dojde k použití vzoru pouze jednou (i v testu cyklu s modifikátorem g). Teprve až je zavolána funkce reset, může se porovnání opakovat. Tuto konstrukci uvádíme pouze pro zajímavost a nelze ji doporučovat. My se budeme striktně držet používání lomítek.

Užití v testech podmínek

Porovnávání řetězců se užívá jako test při rozhodování.

```
$veta = "Existuje 10 druhů lidí - ti, kteří znají binární kód, a ti, kteří ne.";

if ($veta =~ m/10 druhů/){
print "'10 druhů' je ve větě!\n"
}

if ($veta =~ m/777/){
print "'777' je ve větě!\n"
}
```

"10 druhů" se v řetězci vyskytuje, proto byl v tomto případě výraz vyhodnocen jako true. Nezáleží na tom na jaké pozici nebo co je okolo.

Použití výchozí proměnné

Je také možné uvést jako test podmínky jen čistě regulární výraz v uvozovacích znacích. Implicitně je porovnáván s výchozí proměnnou. Příklad má stejný účinek jako předchozí, ale využívá této vlastnosti:

```
$_ = "Existuje 10 druhů lidí - ti, kteří znají binární kód, a ti, kteří ne.";

if (/existuje/){
print "'existuje' je ve větě!\n"
}

if (/Existuje/){
print "'Existuje' je ve větě!\n"
}
```

Poslední ukázka tiskne řetězec "'Existuje' je ve větě!". Na velikosti písmen tedy implicitně záleží. Způsob, kterým lze toto chování změnit, si představíme v jednom z příštích dílů.

V případě zkráceného zápisu pomocí výchozí proměnné je též možné negovat. Stačí přidat vykřičník před úvodní /.

```
if (!/2 druhy/){
print "'2 druhy' není ve větě!\n"
}
```

Metaznaky

Mezi lomítka lze napsat všechny znaky znaky kromě takzvaných metaznaků. Mají totiž nějakou jinou funkci. Patří mezi ně *, +, ., ?, ^, $, (,), {, }, [,], |, /. Lze je nahradit předřazením zpětného lomítka. Pokud si nejste jisti, zda má znak speciální význam a není alfanumerický (tedy pokud je z množiny \W), můžeme mu předřadit zpětné lomítko vždy. Sekvence zpětného lomítka a nealfanumerického znaku se chová vždy jako znak bez speciálního významu.

Proměnné v regulárních výrazech

Než se začne regulární výraz vyhodnocovat, jsou nahrazeny proměnné ve vzoru svým obsahem (což ale samozřejmě neplatí u ohraničení pomocí apostrofů).

```
$cisla = "2005";
print "0002005000" =~ /$cisla/;
```

Druhý řádek předchozího kódu bude fungovat takto:

```
print "0002005000" =~ /2005/;
```

Návratová hodnota výrazu řetězec =~ vzor

Tuto část lze zatím klidně přeskočit, protože je zde řada věcí, které jsme zatím neprobírali. Je to ale téma, které by v prvním dílu o regulárních výrazech chybět nemělo.

skalární kontext

Ve skalárním kontextu vrací v případě úspěchu 1, v případě neúspěchu nepravdivou hodnotu.

seznamový kontext

O něco složitější je to v seznamovém kontextu. Při neúspěchu bez modifikátoru g vrací prázdný seznam. V případě úspěchu vrací seznam zapamatovaných hodnot, pokud jsou, jinak vrací seznam (1). S modifikátorem g vrací výraz pole vyhovujících podřetězců.

```
$retezec = "12345";
@return1 = $retezec =~ /\d\d\d/;
@return2 = $retezec =~ /(\d)(\d)(\d)/;
@return3 = $retezec =~ /(\d)/g;
```

Pole @return1 obsahuje prvek (1). Žádná hodnota nebyla zapamatovaná. Oproti tomu v poli @return2 už takové hodnoty byly: jeho prvky (1, 2, 3) jsou právě zapamatovanými hodnotami. Poslední pole obsahuje (1, 2, 3, 4, 5), protože se vzor aplikoval celkem 5×.

Perl (17) - Regulární výrazy - kotvy

2. díl o regulárních výrazech věnujeme kotvám.

Kotvy jsou speciálními znaky, které se vyznačují nulovou délkou. Jsou totiž určeny svým okolím. Nejde tedy o znak ve svém smyslu, ale spíše o pozici. V regulárním výrazu se označuje buď speciálním metaznakem nebo escape znakem.

Určení začátku a konce řetězce

Dosud jsme vytvářeli vzory, které se mohli vyskytovat na libovolném místě řetězce. Nezřídka ale potřebujeme specifikovat, jakým podřetězcem má řetězec začínat nebo jakým končit. To je asi nejčastější užití kotev. Začátek řetězce je reprezentován znakem ^ a konec znakem $. V praxi vypadá jejich použití následovně:

```
$retezec = "text, který končí\n slovem xxx";
print $retezec =~ /xxx$/;   #true - řetězec skutečně končí na xxx
print $retezec =~ /^text/;  #true - řetězec začíná na text
print $retezec =~ /končí\n/;#true - řetězec obsahuje daný vzor
print $retezec =~ /končí$/; #false - řetězec nekončí vzorem
```

Samozřejmě nám nic nebrání určit začátek i konec řetězce zároveň. Toto jsou zápisy, které dělají totéž, ale každý jinak:

```
$retezec = "regexp";
print $retezec eq "regexp";  #tiskne 1 - $retezec má stejnou hodnotu jako "regexp"
print $retezec =~ /^regexp$/;#tiskne 1 - to samé s využitím regulárních výrazů
```

Hranice slova

Escape sekvence \b je nalezena na místech, kde končí nebo začíná slovo - tedy v místě, kde se stýká znak slova (\w) s jiným znakem (\W). Doplňkem pozice, určené znakem \b je \B. Tomu vyhovují všechny pozice mimo hranici slova.

```
print "slovo" =~ /\blov\b/; #nevyhovuje
print "slov" =~ /\blov\b/;  #nevyhovuje
print "lov," =~ /\blov\b/;  #vyhovuje; čárka není znakem slova
print "lovec" =~ /\blov\B/; #vyhovuje
```

První 3 příkazy hledají v řetězci slovo "lov" a poslední příkaz slovo, které na "lov"začíná.

Začátek řádku, začátek řetězce

Dalšími znaky se speciálním významem jsou \A a ^. ^ znamená začátek řádku a \A začátek řetězce. Abychom mohli tyto 2 znaky rozlišit, musíme ale v regulárním výrazu aktivovat režim více řádků. K tomu slouží přepínač m (regulární výraz pak bude vypadat takto: m//m). V tomto režimu bude ^ nalezeno na začátku řetězce, ale také za každým znakem nového řádku. \A se vždy vztahuje jen na začátek řetězce - je tedy v řetězci právě jednou.

O přepínačích ještě nějakou dobu mluvit nebudeme. Pro pochopení stačí vědět, že se uvádějí za koncové lomítko regulárního výrazu. Pokud k m/vzor/ *chceme přidat přepínače* x *a* y*, bude zápis vypadat takto:* m/vzor/xy.

Mějme řetězec:

```
radek 1\nradek 2\nradek 3
```

Teď do něj znázorněme znaky \A a ^. Bez použití přepínače m:

```
<\A><^>radek 1\nradek 2\nradek 3
```

A pokud je přepínač m nastaven na "zapnuto", je na začátku každého řádku ještě ^:

```
<\A><^>radek 1\n<^>radek 2\n<^>radek 3
```

Pokud nemáte představu, jak by se začátek řádku použil, zde je ještě konkrétní příklad:

```
print "MATCHED" if "radek 1\nradek 2\nradek 3" =~ /^\Aradek 1\n^/m; #vyhovuje
print "MATCHED" if "radek 1\nradek 2\nradek 3" =~ /\A^radek 1\n^/m; #vyhovuje - navíc je vidět, že nezáleží na pořadí ^ a \A
print "MATCHED" if "radek 1\nradek 2\nradek 3" =~ /^radek 1\n^/;   #nevyhovuje - není přepínač a ^ tedy funguje stejně jako začátek řetězce
```

Konec řádku, konec řetězce

Mechanizmus funguje podobně jako začátek řetězce nebo řádku. Je též nutný přepínač m.

- $ - odpovídá konci řetězci a je-li uveden přepínač m, je také před každým znakem nového řádku
- \Z - konec řetězce

S přepínačem m vypadá řetězec a neviditelné znaky v něm následovně:

```
<\A><^>radek 1<$>\n<^>radek 2<$>\n<^>radek 3<$><\Z>
```

Variantou znaku \Z je \z. Jejich význam se liší v tom, že u \z nesmí být na konci řetězce znak nového řádku.

```
$x = "...XXX\n";
print "MATCHED" if $x =~ /XXX\Z/; #vyhovuje
print "MATCHED" if $x =~ /XXX\z/; #nevyhovuje
```

Poznámka: Existuje speciální proměnná $, která umí nahradit přepínač m. Pokud $* == 1, fungují vzory jako s m, v případě, že $* == 0 ne. Tato proměnná se ale nedoporučuje používat. Dávejte raději přednost uvedení přepínače.*

Poslední nalezení vzoru

Znak \G je na pozici, kterou lze určit funkcí pos, což je místo posledního úspěšného nalezení vzoru.

Funkce pos

pos vrací pozici, na které byl nalezen poslední hledaný výskyt. Je tedy nutný přepínač g, který ale teprve budeme rozebírat. Teď napišme program, který vypíše počet výskytů určeného znaku a jeho pozice. V každé iteraci cyklu budeme hledat v daném řetězci další pozici hledaného znaku, a právě tu přidáme do pole @pozice.

```
$" = ", ";
$retezec = "matematika";
while ($retezec =~ /a/g){
 push(@pozice, pos $retezec);
}
print "Písmeno a se v řetězci vyskytuje ", scalar @pozice, "x a to na pozicích @pozice.\n";
```

Vyjádření "NEBO"

Představte si následující kód:

```
if ($volba =~ /Linux/ or $volba =~ /Perl/ or $volba =~ /C/){
 print "Správná volba!";
}
```

Zápis je dost nepohodlný a nepřehledný. Logické or lze naštěstí přesunout z podmínky přímo do regulárního výrazu. Zapisuje se znakem | (někdy se nazývá alternace). Předchozí kód tak velmi zjednodušíme:

```
if ($volba =~ /Linux|Perl|C/){
 print "Správná volba!";
}
```

Tento kód uzná každý řetězec, ve kterém se vyskytuje 1 z hledaných podřetězců. Ale takové C se může vyskytovat v leckterém řetězci. Proto bude vhodné specifikovat začátek a konec řetězce. K tomu nestačí uvést jen znaky ^ a $, ale je nutné ozávorkovat vše, co patří k OR. Závorky sdružují skupinu znaků.

Poznámka - problematice závorek se ještě budeme podrobně věnovat.

```
if ($volba =~ /^(Linux|Perl|C)$/){
print "Správná volba!";
}
```

Podobným způsobem je výhodné dělit slova. Kupříkladu

```
/Perl(5|6)/
```

bude rychlejší než

```
/Perl5|Perl6/
```

Řetězec "Perl" je totiž v 1. případě hledán pouze jednou.

Příště se konečně podíváme na třídy znaků.

Perl (18) - Regulární výrazy - množiny znaků

Ve 3. pokračování o regulárních výrazech jsou představeny třídy znaků.

Množina znaků je skupina obyčejných znaků. Pokud ji uvedeme ve vzoru, potom se taková množina chová jako 1 ze znaků, které obsahuje. Máme několik možností, jak množinu definovat.

libovolný znak

Začneme tím nejjednodušším. Znak tečka zastupuje 1 libovolný znak mimo konce řádků.

Poznámka - Pokud chcete zahrnout do množiny znaků, které tečka zastupuje i znak konce řádku, použijte přepínač /s.

Příklad vypisuje slova s názvem o délce 4:

```
@slova = qw(C++ Perl Python Ruby Java);
foreach $slovo (@slova){
print $slovo, "\n" if ($slovo =~ /^....$/); #vypíší se slova, které zabírají 4 znaky
}
```

Poznámka - Nutno ale poznamenat, že místo cyklu foreach bychom mohli jednoduše použít funkci grep:

```
$, = "\n";
print grep /^....$/, @slova;
```

Výčet znaků

Je možné specifikovat užší množinu znaků než libovolný znak. K tomu slouží hranaté závorky. Skupina znaků, které jsou v hranatých závorkách, tvoří tuto užší množinu. Znaky nejsou nijak speciálně odděleny, protože to není potřeba.

```
print "MATCHED" if "SUSE Linux 9.2" =~ /S[Uu]SE Linux 9\.[123]/;
print "MATCHED" if "SuSE Linux 9.3" =~ /S[Uu]SE Linux 9\.[123]/;
print "MATCHED" if "SUSE Linux 9.4" =~ /S[Uu]SE Linux 9\.[123]/;
```

1. a 2. příklad vrací 1, poslední příklad prázdný řetězec. Číslo 4 totiž není v dané množině povolených znaků.

Pomocí alternace můžeme vytvořit vzor, kterému vyhoví i "SUSE Linux 10.0".

```
print "MATCHED" if "SUSE Linux 10.0" =~ /S[Uu]SE Linux (9\.[123]|10\.0)/;
```

Poznámka - Zde návratová hodnota obsahuje zapamatované hodnoty. Pamatování jsme zatím neřešili.

Zkusme si spočítat, kolika různým řetězcům náš poslední vzor vyhoví. Aby to nebylo nekonečno, vzor nejdříve ukotvíme:

```
/^S[Uu]SE Linux (9\.[123]|10\.0)$/
```

Nyní pojďme znak po znaku (resp. množinu po množině), u každého zjistíme počet možností, a tyto počty mezi sebou vynásobíme. 1. znak, S, je samostatný znak - tedy množina jediného znaku. [Uu] je množina 2 znaků a jsou tedy 2 možnosti. Dále máme až k závorce vždy 1 možnost. Potom následuje 1 z řetězců 9.1, 9.2, 9.3nebo 10.0. Tedy 4 možnosti. Velikosti množin mezi sebou vynásobíme, a tím získáváme 1*2*1*1*1*1*1*1*1*1*1*4 = 8. Naší šabloně tak vyhoví 8 řetězců.

Pro výčet znaků existuje několik usnadnění. Představte si, že chcete definovat množinu, které vyhovuje libovolné anglické písmeno. Řešení může vypadat například takto: [abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]. Jistě uznáte, že to není to pravé. Je možné si ale nadefinovat proměnné, kde budou skupinu znaků obsahovat a následně je použít asi takto ve vzoru:

```
$pismeno = "[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]";
print "r" =~ /$pismeno/;
```

Znaky v množinách se speciálním významem

Stále to je příliš těžkopádné. Regulární výrazy nabízejí mnohem lepší řešení. Existuje totiž speciální operátor rozsahu. Ten se zapisuje v syntaxi regulárních výrazů pomlčkou. Předchozí problém s množinou všech písmen se tímto způsobem řeší mnohem pohodlněji. Do vzoru stačí uvést [a-zA-Z]. To samé lze dělat s čísly. [2-5] je totéž, co [2345]. Množninu znaků, které vyhoví číslice šetnáckového zápisu čísla, tak zapíšeme jako [0-9a-fA-F].

Zkuste si tipnout, co všechno vyhoví zápisu ^[x|y]$. Ač to možná intuitivně vypadá jako znak x nebo znak y, má tento zápis úplně jiný význam. Znak | nemá na tomto místě žádnou speciální funkci, proto celému regulárnímu výrazu vyhovuje znak x, ynebo |.

Jedním z těchto speciálních znaků je ^ (negace - pozor!, se začátkem řetězce má společné jen to, že používá stejný symbol). Užijete ji, pokud například chcete definovat množinu, které mají vyhovovat všechny znaky kromě velkých písmen. Píše se bezprostředně za úvodní hranatou závorku (pokud je jinde, svoji funkci ztrácí). Má ten efekt, že to, co je obvykle množinou znaků, je jejím doplňkem. Náš problém s vynecháním velkých písmen vyřešíme takto: [^A-Z] (oproti tomu v zápisu [a-c^d-f] stříška speciální význam nemá, protože již není na 1. pozici).

Stejně jako stříška ztrácí uvnitř hranatých závorek svoji funkci na jiném než 1. místě, ztrácí ji i uzavírací závorka na 1. místě. Pokud je tam uvedete, bere se jen jako znak z množiny. Je-li umístěna jinam, musíte jí předřadit zpětné lomítko. To samé platí pro pomlčku - speciální funkci má pouze, pokud dostane z obou stran nějaké smysluplné hodnoty. V opačném případě totiž buď hlásí Invalid range (je-li na jedné straně písmeno, na druhé číslo, nebo když jsou strany prohozeny), a nebo bere pomlčku jako znak bez speciálního významu (pokud je uvedena na začátku nebo na konci řetězce v hranatých závorkách).

Nejen na alfanumerické znaky se může rozsah vztahovat. Lze zadat osmičkové rozpětí znaků z ASCII tabulky. Například znaky ((osmičkový kód 050),) (051), *(052), + (053). Následující 2 zápisy jsou ekvivalentní:

```
/^[()*+]$/
/^[\050-\053]$/
```

čeština

Operátor rozsahu se na české znaky nevztahuje. Jinak je to ale s předdefinovanými množinami znaků. Použijete-li příkaz use locale; a máte-li správně nastavené proměnné prostředí, vztahují se množiny i na národní znaky.

```
use locale;
print "MATCHED" if "č" =~ /\w/;
```

Je vytisknuto MATCHED, protože znak s háčkem množině vyhovuje. Zkuste nyní odstranit use locale;. Potom toto tvrzení platit nebude.

Předdefinované množiny

Další elegantní řešení nabízejí předdefinované množiny znaků. Existují speciální escape sekvence, které je zastupují. Přitom se nemusejí uzavírat hranatými závorkami. Pro srovnání do tabulky zahrnuji i tečku.

Escape znak	Ekvivalentní zápis	Význam
\d	[0-9]	Číslice
\D	[^0-9] nebo [^\d]	Cokoliv mimo číslici
\w	[_0-9a-zA-Z]	Znaky identifikátorů
\W	[^_0-9a-zA-Z] nebo [^\w]	Cokoliv mimo znaků identifikátorů
\s	[\ \n\r\t\f]	Bílý znak
\S	[^ \n\r\t\f] nebo [^\s]	Cokoliv mimo bílého znaku
.	[^\n]	Libovolný znak mimo znaku nového řádku (chování se dá pozměnit přepínačem s)

Poznámka - Ekvivalenty \w a [_0-9a-zA-Z]) resp. \W a [^_0-9a-zA-Z] v poslední tabulce nejsou skutečně 100% ekvivalenty. Vzory [_0-9a-zA-Z] a \w se mají chovat stejně, ale v případě použití locales tomu tak není. Předveďme si důkaz:

```
use locale;
print "MATCHED" if "č" =~ /\w/;          #true
print "MATCHED" if "č" =~ /[_0-9a-zA-Z]/; #false
```

Nyní zapíšeme regulární výraz pro formát 12hodinového zápisu času.

```
/^[01]\d[:\.][0-5]\d[:\.][0-5]\d[ ]?[pPaA][mM]$/;
```

Vysvětleme si, jak se tento regulární výraz vyhodnocuje:

- ^ - začátek řetězce
- [01] - znak 0 nebo 1
- \d - znak z číslic 0 až 9
- [:\.] - Znak : nebo . (tečka). Oddělovač hodin a minut
- [0-5]\d - číslo mezi 00 a 59. Počet minut
- poslední 2 body se opakují pro vyjádření sekund
- [ ]? - nepovinná mezera
- [pPaA][mM] - určuje, zda se jedná o dopoledne nebo odpoledne (povinné).
- $ - konec řetězce

Je tu problém. Našemu výrazu vyhoví i čas 13 - 19 hodin. Musíme použít znak |. 1. operand bude hledat, zda jsou hodiny mezi 00 a 09, pokud ne, 2.operand zjistí zda nejsou ještě mezi 10 až 12.

```
/^((0\d)|(1[0-2]))[:\.][0-5]\d[:\.][0-5]\d[ ]?[pPaA][mM]$/;
```

Regulárnímu výrazu ((0\d)|(1[0-2])) vyhovují čísla 01 až 12, zbytek je stejný jako v minulém příkladu. Celý výraz ještě o něco zjednoduší přepínač /i. Zápis [pPaA][mM] pak může vypadat o něco pohodlněji.

```
/^((0\d)|(1[0-2]))[:\.][0-5]\d[:\.][0-5]\d[ ]?[pa]m$/i;
```

Sice ani toto není dokonalé řešení (vyhoví i takové věci jako "10.10:40pM"), ale pro představu nám posloužilo.

Unicode třídy

Toto asi nebudete nikdy potřebovat, ale koho to zajímá ať čte dál. Třída znaků se zadává pomocí zápisu \p{třída}. Pokud chceme doplněk třídy (tedy vše mimo toho, co do třídy patří), použijeme zápis \P{třída}.

V 1. tabulce jsou jen pro představu 2 Unicode třídy. Existuje jich ale mnohem více (man perlretut).

Třída	Význam
IsSm	matematický symbol
IsSc	znaky měny

Znakem měny je například dolar:

```
print "MATCHED" if "r" =~ /^\p{IsSc}$/;  #nevyhovuje
print "MATCHED" if "\$" =~ /^\p{IsSc}$/; #vyhovuje
```

Dále si Perl některé třídy sám definuje.

Mají 2 možnosti zápisu. Buď \p{třída} nebo [:třída:]. Zápis [:třída:] je možný jen uvnitř dalších [ ]. Samotné [:třída:] by totiž znamenalo množinu znaků :, t, ř, í, d, a. Je tedy nutný zápis [[:třída:]].

UNICODE \p{třída}	POSIX [:třída:]	Význam
IsAlpha	alpha	Anglická písmena
IsAlnum	alnum	Anglická písmena a číslice
IsASCII	ascii	Znaky ASCII 0-127
IsSpace	blank	Mezera, tabulátor
IsCntrl	cntrl	Řídící znaky
IsDigit	digit	Číslice
IsGraph	graph	Grafické
IsLower	lower	Malá anglická písmena
IsPrint	print	Tisknutelné znaky a mezera
IsPunct	punct	Interpunkce a pomocné znaky
IsSpacePerl	space	Bílý znak
IsUpper	upper	Velká anglická písmena
IsWord	word	Alfanumerický znak nebo podtržítko

IsXDigit	xdigit	Šestnáckové číslo

Použití ukazuje následující kód. Všimněte si odlišných barev hranatých závorek. Vnější definují množinu znaků, jejíž obsahem je třída xdigit.

```
print "MATCHED" if "5" =~ /^[[:xdigit:]]$/; #vyhovuje - 5 je hexadecimální
print "MATCHED" if "a" =~ /^[[:xdigit:]]$/; #vyhovuje - a je hexadecimální
print "MATCHED" if "g" =~ /^[[:xdigit:]]$/; #nevyhovuje - g není hexadecimální
```

Perl umožňuje POSIX třídy negovat. Negovaná [:třída:] je [:^třída:].

Na závěr ještě něco k definici unicode tříd. Je nutné vytvořit podprogram se stejným jménem jako třída. Jeho návratovou hodnotou jsou řádky. Je-li na řádku 1 číslo, určuje ASCII znak, určený 16kově. Jsou-li na řádku 2 čísla, oddělená mezerou, jde o rozsah. Definujeme si třídu IsZavorka, které vyhoví 1 ze znaků (,), {, }, [,], <, >- tedy závorka kulatá, složená, hranatá nebo lomená.

```
print "MATCHED" if "(" =~ /^\p{IsZavorka}$/; #vyhovuje

sub IsZavorka {
  return <<END;
    28 29
    3C
    3E
    5B
    5D
    7B
    7D
  END
}
```

Třídy lze definovat i pomocí již definovaných tříd. V takovém případě je do návratové hodnoty podprogramu třeba uvést řádek [+!-&]balík::Trida. Implicitní unicode třídy jsou v balíku utf8. + (sjednocení) funguje pro přidání znaků z příslušné třídy, ! (rozdíl) pro odebrání znaků, - (doplněk) pro všechny znaky mimo znaky z této třídy a & (průnik) vybere pouze znaky, které jsou v této třídě a třídě jiné. Pro definici třídy IsBracket, která má stejný význam jako IsZavorka stačí psát toto:

```
sub IsBracket {
  return <<END
+main::IsZavorka
END
}
```

Další možností definice unicode tříd je podle typu písma. Existují například třídy InHebrew, InMathematicalOperators. Celý seznam je na manuálové stránce perlunicode.

Zdroje

Toto téma by zabralo několik samostatných článků. Mnohem více o unicode třídách a jejich definicích na manuálových stránkách

- man perlretut
- man perlunicode.

Perl (19) - Regulární výrazy - opakování a kvantifikátory

Jedním ze základních kamenů regulárních výrazů jsou kvantifikátory, které umožňují aplikovat část regulárního výrazu vícekrát.

Z předchozích dílů o regulárních výrazech zatím nejsme schopni (opomineme alternaci, ta má v regulárních výrazech jiný účel) vytvořit šablonu pro řetězce různé délky. Právě toto mají na starosti kvantifikátory. Kvantifikátorem určujeme počet opakování části regulárního výrazu. To znamená, že určitý počet znaků za sebou můžeme porovnat s jednou množinou znaků (šablonou pro 1 znak). Dosud jsme takto porovnávali vždy pouze 1 znak.

Kvantifikátor se vždy uvádí za množinu znaků, jejíž četnost specifikuje. Množinou může opět být jak výčet znaků pomocí hranatých závorek, předdefinovaná množina tak i samotný znak.

Libovolný počet opakování

Uvedením hvězdičky dáváme najevo, že množina znaků před ní se může opakovat libovolněkrát.

Regulárnímu výrazu x* (x je množina znaků, * je počet opakování) potom vyhovuje libovolný řetězec, skládající se z písmen x nebo prázdný řetězec. V prázdném řetězci je 0 písmen x, což hvězdička též zahrnuje.

Jak bude vypadat podmínka, která určuje, zda bylo zadáno číslo?

```
print "Zadejte číslo:";
$cislo = <STDIN>;
if ($cislo !~ /^[0-9]*$/){
print "Nezadal jste číslo!\n";
}
```

Problémem je, že regulárnímu výrazu vyhoví i prázdný řetězec. Tomu se dá zamezit například přidáním další číselné množiny před výraz. Potom musí být 1. znak číslem a další, pokud jsou, také:

```
/^[0-9][0-9]*$/
```

Nejméně 1 výskyt

Asi nejlepším řešením posledního problému by bylo použití znaku +, který provádí opakování minimálně jednou, a prázdný řetězec tudíž nevyhoví. Tyto zápisy mají stejný význam:

```
/^[0-9][0-9]*$/
/^[0-9]+$/
```

Nejvíce 1 výskyt

Použijete-li jako kvantifikátor otazník, má téměř stejný význam kdyby tam žádný kvantifikátor nebyl. Tyto 2 možnosti se liší v tom, že otazníku vyhoví ještě navíc prázdný řetězec. Jedna číslice nebo prázdný řetězec vyhovuje výrazu

```
/^[0-9]?$/
```

Jiný počet opakování

Pokud se vám nehodí žádná z dosud nabízených možností, nabízejí regulární výrazy zápis počtu opakování pomocí složených závorek. Ty vymezují spodní a horní hranici počtu opakování. Existuje několik možností syntaxe.

{minimum,maximum}
{minimum,}
{počet}

Čísla vyjadřující minimum a maximum nejsou neomezené. Záleží na nastavení při kompilaci. Obvykle nelze používat větší čísla než 32766, což dokazuje hláška Perlu Quantifier in {,} bigger than 32766 in regex, která na vás vyskočí po překročení definované meze.

Vezměmě si výraz:

```
/^ab{2,4}c/
```

Vyhoví mu řetězce abbc (2 výskyty), abbbc (3 výskyty), abbbc (4 výskyty), ale už ne abc (1 výskyt) nebo abbbbbc (5 výskytů). Pokud je uveden jen 1 argument a není použita čárka, vyhoví vzoru pouze řetězce, obsahující danou množinu znaků právě tolikkrát, kolik je uvedeno v argumentu. Další možnost nabízí uvedení čárky, ale bez maxima. V takovém případě je maximum nekonečno.

Potřebujete-li zapsat do regulárního výrazu nějaký rozsah opakování s mezerou (například 2, 3 nebo 5 a více opakování, ale ne 4 opakování), musíte si vypomoci alternací.

Metaznaky +, * a ? pro počet opakování jsou tedy jen speciálními případy, které lze zapsat i pomocí složených závorek:

Metaznak	Ekvivalent
+	{1,}
*	{0,}
?	{0,1}

Ukažme si nyní program, který ověřuje bezpečnost hesla. Heslo budeme považovat za bezpečné, bude-li splňovat tyto 4 podmínky:

- obsahuje alespoň 1 číslici
- obsahuje alespoň 1 písmeno
- obsahuje alespoň 1 znak, který není znakem slova
- je delší než 5 znaků

Pro každou podmínku napíšeme regulární výraz a pokud vyhoví heslo všem těmto vzorům, máme jistotu, že je heslo bezpečné.

```
print "Zadejte heslo: ";
chomp($_ = <STDIN>);
if (/[0-9]/ and /[a-zA-Z]/ and /\W/ and /.{6,}/){
print "Heslo je bezpečné\n";
}else{
print "Heslo není bezpečné\n";
}
```

Poznámka - pokud byste chtěli, aby při zadávání hesla nezobrazovaly jeho pravé znaky, ale pouze hvězdičky (případně vůbec nic), použijte modul Term::ReadKey.

Hladovost a sytost kvantifikátorů

Metaznaky {}, *, + a ? jsou takzvaně hladové (,,greedy"). To znamená, že spolknou co možná nejdelší část řetězce. Dokazuje to tento kód:

```
$_ = "123456789";
/(\d*)/;
print $1; #tiskne hodnotu, která vyhověla vzoru - více v příštím díle
```

Je vytisknuto 123456789 - tedy celý řetězec. To i přesto, že by vyhověl třeba také prázdný řetězec, vyhodnocování regulárního výrazu by se mohlo úspěšně ukončit a program by běžel dál. Kvantifikátory jsou hladové, a tak pohlcují nejdelší vyhovující podřetězec. To samé by platilo i v případě, kdybychom jako kvantifikátor místo hvězdičky zvolili například {3,5}. Potom by předchozí kód vytiskl prvních 5 číslic a ne 3, ačkoliv by to také stačilo.

Tato vlastnost může občas činit problémy. Tak například, pokud chceme v HTML kódu vyhledat 1 obrázek a vypsat kód celé značky IMG:

```
$html = 'abc....<IMG SRC="obrazek" ALT="obrázek">....xyz';
$html =~ /(<((IMG)|(img))\ (.*)>)/;
print $1;
```

Hvězdička vlivem hladovosti spolkne znaky do konce řetězce. Ale tam v našem případě znak > není. Proto se začne z místa, po které hvězdička spolkla text (v našem případě konec řetězce) hledat. Najde ho, ukončí další vyhodnocování, protože je na konci regulárního výrazu a program vytiskne přesně to, co jsme chtěli. Takže v pořádku. Problém nastane, pokud se znak > vyskytuje ještě někde mezi místem, po které byl spolknut text a tím "správným" znakem >. To si můžeme jednoduše ilustrovat:

```
$html = '<TABLE><TR><TD><IMG SRC="obrazek" ALT="obrázek"></TD><TD>...</TD></TR></TABLE>';
$html =~ /(<((IMG)|(img))\ (.*)>)/;
print $1;
```

Začátek je stejný jako v minulém příkladu. Hledá se řetězec "<IMG ", poté hvězdička pohltí vše do konce a odtud se opačným směrem hledá znak >. Ale takový se vyskytne už v značce </TABLE>! To znamená, že získáme řetězec <IMG SRC="obrazek" ALT="obrázek"></TD></TR><TD>...</TD></TABLE>. To jsme opravdu nechtěli. A může za to právě hladovost.

Možnost, jak vzniklý problém řešit, může být přes funkce. To je zbytečně složité.

```
$html = '<TABLE><TR><TD><IMG SRC="obrazek" ALT="obrázek"></TD><TD>...</TD></TR></TABLE>';
$html =~ /(<((IMG)|(img))\ (.*)>)/;
print substr($1, 0, (index $1, ">")+1);
```

Jiné, o dost lepší řešení, lze aplikovat pomocí negované množiny znaků. Místo libovolného znaku (v našem regulárním výrazu reprezentován tečkou), specifikujeme vše mimo znaku >.

```
$html = '<TABLE><TR><TD><IMG SRC="obrazek" ALT="obrázek"></TD><TD>...</TD></TR></TABLE>';
$html =~ /(<((IMG)|(img))\ ([^>]*)>)/;
print $1;
```

Dalším řešení nabízí sytost. Existují další 4 kvantifikátory, které fungují úplně stejně jako ty nám dosud známé, až na to, že nejsou hladové. Jsou to {}?, *?, +? a ??. Zapisují se stejné jako {}, *, + a ?, jen se za ně připisuje otazník. Pohltí minimální možný počet znaků, který vyhoví vzoru. Tento vzor tedy nepohltí 5 znaků, jak by to udělal hladový kvantifikátor, ale pouze 3:

```
$_ = "123456789";
/(\d{3,5}?)/;
print $1;
```

Nakonec si ještě ukážeme řešení pro problém s vyhledáním obrázku v HTML kódu pomocí sytosti:

```
$html = '<TABLE><TR><TD><IMG SRC="obrazek" ALT="obrázek"></TD><TD>...</TD></TR></TABLE>';
$html =~ /(<((IMG)|(img))\ (.*?)>)/;
print $1;
```

Příště se podíváme na funkci kulatých závorek v regulárních výrazech.

Perl (20) - Regulární výrazy - magické závorky

Dnešní díl popisuje použití kulatých závorek pro změnu priority, seskupování a pamatování.

V regulárních výrazech mají závorky hned několik důležitých a jinak nenahraditelných významů. Mezi jejich hlavní funkce patří seskupení, priorita a paměť. Rozeberme si postupně tyto tři vlastnosti.

Priorita

Stejně jako pro operátory existuje tabulka priorit i pro metaznaky. Určuje, ke kterým znakům se váže určitý metaznak. Čím výše v tabulce, tím je metaznak prioritnější.

Metaznaky	Význam
( ), (?:)	závorky
*, ?, +, {}, *?, ??, +?, {}?	kvantifikátory
aa, \x, ^, $ apod.	posloupnost znaků, kotvy
|	alternativy

Priorita se dá měnit kulatými závorkami.

Seskupování

Kvantifikátor jsme zatím používali pouze pro znak nebo třídu znaků, která ho předcházela. Pomocí závorek můžeme docílit toho, aby se vztahoval na nějakou delší část regulárního výrazu. Srovnejme si tyto 2 vzory:

```
/(xy){3}/;
/xy{3}/;
```

V 1. případě jsou znaky xy seskupeny. To znamená, že se kvantifikátor vztahuje na celý řetězec xy v závorkách. Lze to vysvětlit také jako změnu priority - kvantifikátor má menší prioritu než závorky a proto je aplikován na celý uzávorkovaný výraz. Takovému regulárnímu výrazu vyhoví řetězec, který obsahuje xyxyxy. Další řádek je stejný až na to, že chybí závorky a znaky xy seskupeny nejsou. Kvantifikátor je prioritnější než posloupnost znaků a vztahuje se pouze na znak y. Vzoru tedy vyhoví xyyy.

Pamatování obsahu závorek

Vše, co uzávorkujete kulatými závorkami si Perl, pokud mu to nezakážete, pamatuje. To je vlastnost, se kterou lze dělat opravdu nevídané věci.

Výraz $text =~ /(...)/ vrací úsek textu, který vyhověl regulárnímu výrazu. Tímto způsobem můžeme do pole velice pohodlně získat třeba seznam slov nebo čísel. Právě seřazený výtah čísel ze zadaného textu tiskne následující ukázka.

```
$, = ", ";
$text = "333 stříbrných stříkaček stříkalo přes 33 stříbných střech";
@cisla = ($text =~ /(\d+)/g);
print sort @cisla; #tiskne 33, 333
```

Obsah závorky se ukládá okamžitě a takto uložený řetězec se dá použít ještě ve vzoru. To, co je v 1. závorce je přístupné v \1, další závorka \2 a tak dále až do \99(dále ne, protože by se takový zápis překrýval s osmičkovým zápisem znaku. \100je zavináč, \101 je znak A apod.). Tento mechanizmus nám otevírá spoustu dalších možností.

Pokusme se zapsat vzor, který vyhoví řetězci o 5 stejných číslicích. Jinými slovy vzor, kterému, pokud začíná na 1, vyhoví pouze 11111, pokud začíná na 2, vyhoví pouze 22222, atd. Pokud vzor nezačíná číslicí, nevyhoví nikdy.

Jako 1. znak hledáme číslici. Zápis množiny číslic musí být v závorkách, abychom její hodnou neztratili. Dále použijeme zapamatovanou hodnotu jako množinu znaků a otestujeme ji 4x. Nakonec označíme začátek a konec řetězce.

```
print "MATCHED" if "55555" =~ /^(\d)\1{4}$/; #vyhovuje
print "MATCHED" if "555555" =~ /^(\d)\1{4}$/;#nevyhovuje
print "MATCHED" if "11112" =~ /^(\d)\1{4}$/; #nevyhovuje
print "MATCHED" if "xxxxx" =~ /^(\d)\1{4}$/; #nevyhovuje
```

Tato vlastnost se hodí zejména při nahrazování, které bude rozebráno v příštím díle.

Podobně lze pomocí závorek získávat hodnoty i mimo samotný vzor. Obsahy jsou přístupné ve speciálních proměnných $1, $2 až $99. Nyní z řetězce opět vyseparujeme všechna čísla. Použijeme cyklus, každou iteraci bude 1 číslo zapamatováno a v těle cyklu vytištěno.

```
while ("333 stříbrných stříkaček stříkalo přes 33 stříbných střech" =~ /(\d+)/g){
print "Číslo: $1\n";
}
```

Pořadí (u zápisu \n i $n) se uděluje na základě pozice otevírací závorky. Tento systém řeší případné víceúrovňové zanoření. Nějaké podřetězce tak mohou být uloženy i vícekrát. Nic tedy nebrání například zápisu ((\d)) - ta stejná číslice bude uložena do více proměnných. Zanořování si ilustrujeme na následujícím úseku kódu. Ten vypíše první a poslední číslici nejméně trojmístného zadaného čísla a také toto celé číslo.

```
<STDIN> =~ /^((\d)\d+(\d))$/;
print "1. číslice: $2\n";
print "poslední číslice: $3\n";
print "celé číslo: $1\n";
```

Poznámka - Pokud řetězec nebude vyhovovat vzoru, proměnné zůstanou prázdné - obsah závorek se ukládá, jen pokud byl test vyhodnocen pravdivě. Problémy pak mohou nastat, pokud v proměnných $1, $2 atd. zbyly nějaké hodnoty z předchozího testu. Jde o velice nenápadnou chybu a může trvat dlouho, než je odhalena.

*Poznámka - Jaký je tedy rozdíl mezi \n i $n? \n je do paměti ukládáno okamžitě s uzavírací závorkou, takže je možné tímto způsobem zapamatovanou hodnotu použít ještě ve vzoru. Oproti tomu proměnné $n se načtou vždy až po **úspěšném** srovnání řetězce se vzorem.*

Ukládání do \n, resp. $n lze zabránit speciální syntaxí pro zápis závorek. Místo levé (otevírací) závorky použijte posloupnost znaků (?:. V takovém případě se nic ukládat nebude a proměnná $n se bude chovat jako ostatní nedefinované proměnné, což dokazuje kód:

```
"123456789" =~ /^(?:\d*)$/;
print "Hodnota uložena\n" if defined $1;
```

Vždy, když není pamatování potřeba, je lepší použít právě verzi bez zapamatování. Zvyšuje se tím rychlost vyhodnocení regulárního výrazu.

Hezky lze také využít pamatování v souvislosti s kontextem. Tato vlastnost umožňuje například snadno vyseparovat z rodného čísla den, měsíc a rok narození. Stačí jen uzavřít příslušné části regulárního výrazu do závorek a regulární výraz přiřadit do nějakého seznamu.

```
$rc = "3012013522";
($rok, $mesic, $den) = $rc =~ /^(\d\d)(\d\d)(\d\d)\d{4}$/;
print "Narozen $den.$mesic.19$rok.";
```

Poznámka - Příklad je samozřejmě zjednodušený - nebere v úvahu ženská rodná čísla a narozené jindy než mezi roky 1900 a 1999.

Nejdříve se vyhodnotil pravý operand přiřazení. Tak jsme získali seznam, který jsme přiřadili do už pojmenovaného seznamu. Problém opět nastává, pokud srovnávaný řetězec nevyhoví regulárnímu výrazu. Lze to však řešit například umístěním podmínky za příkaz.

```
print "Narozen $den.$mesic.19$rok." if ($rok, $mesic, $den) = $rc =~ /^(\d\d)(\d\d)(\d\d)\d{4}$/;
```

Speciální proměnné

V proměnné $& je část testovaného řetězce, která se shoduje se vzorem. Tímto způsobem si můžeme mimo jiné hezky ilustrovat hladovost kvantifikátorů.

```
"číslo 101 je prvočíslo" =~ /\d+.{2,10}/;
print $&;

"číslo 101 je prvočíslo" =~ /\d+.{2,10}?/;
print $&;
```

1. zápis je hladový. Spolkne 10 znaků za číslem 101: "101 je prvočí". Pokud za kvantifikátor přidáme otazník, stává se sytým a pohltí pouze nejmenší možný počet znaků, které vyhoví: "101 j".

Další proměnné $` a $' tisknou tu část testovaného řetězce, kterou nevytiskla proměnná $&. $` tiskne část před ("číslo ") a $' část za ("e prvočíslo").

```
"číslo 101 je prvočíslo" =~ /\d+.{2,10}?/;
print $`;
print $';
```

V proměnné $+ je uložen poslední závorkami zapamatovaný řetězec.

Poznámka - Je dobré mít na paměti, že proměnné $&, $`, $' a $+ značně zpomalují program.

Speciální proměnná @- si pamatuje v prvku s indexem 0 první pozici v testovaném řetězci, která vyhovuje vzoru. V dalších prvcích pak pozice začátků zapamatovaných podřetězců. Proměnná @+ dělá to samé, ale pamatuje si místo začátků konce.

Funkce split a regulární výrazy

Oddělovač, který je parametrem této funkce, lze uvést i jako regulární výraz. To je v mnoha případech opravdu užitečné. Mějme nějaký text, ze kterého načteme do pole seznam slov.

```
$, = "\n";
@pole = split (/\W+/, "Text, ve kterém je interpunkce!");
print @pole;
```

Vše mezi slovy (tedy oddělovače - mezery a interpunkce - lépe řečeno vše, co vyhovuje vzoru) je navždy ztraceno. Být tomu tak vždy nemusí. Proč, to si budeme demonstrovat na jiném příkladu. Budeme mít nějaký řetězec, ve kterém jsou promíchány čísla a právě čísla zvolíme jako oddělovač.

```
$, = " "; $\ = "\n";
print split (/\d+/, "abc01def2ghijk34567l89mnop");
print split (/(\d+)/, "abc01def2ghijk34567l89mnop");
print split (/(?:\d+)/, "abc01def2ghijk34567l89mnop");
```

V 1. případě se vytiskne seznam získaných podřetězců tak, jak jsme to dosud znali. Zajímavé to začíná být až na dalším řádku. Vzor je uveden v závorkách, které zde mají ten efekt, že se do výsledného seznamu tisknou i oddělovače! Skutečným oddělovačem je tedy (v tomto konkrétním případě) hranice mezi podřetězcem vyhovujícím vzoru a jiným znakem. Nic z původního řetězce tak není ztraceno. Posledním případem je speciální druh uzávorkování, které této vlastnosti zamezuje - oddělovače se chovají jako v 1. případě.

```
$ perl split.pl
abc def ghijk l mnop
abc 01 def 2 ghijk 34567 l 89 mnop
abc def ghijk l mnop
$
```

Je též možné ozávorkovat pouze část vzoru. V tom případě bude do výsledného seznamu přiřazena příslušná část oddělovače.

```
print split (/(\d+)45/, "***12345***12345***123***12345678***");
```

Podtržené podřetězce ukazují oddělovače. Ve výsledném seznamu ale bude jejich část (znaky 45) odmazána, protože jsou mimo závorky (částí oddělovače ale samozřejmě zůstávají).

Dále můžete také vyzkoušet závorky různě vnořovat, zdvojovat apod. V takových případech budou ve výsledném seznamu všechny uzávorkované řetězce. A to i přesto, že některé části pak budou ve výsledném seznamu vícekrát.

Perl (21) - Regulární výrazy - nahrazování

Do této chvíle jsme regulární výrazy používali pouze k vyhledávání. To je ale jen polovina toho, co skutečně dokáží.

Jak již o regulárních výrazech víme, slouží nejen pro vyhledávání vzorů, ale také pro nahrazování jejich částí. Představme si, že máme nějaký textový řetězec a potřebujeme z něj vymazat všechna čísla. Už známe několik způsobů, jak tento problém řešit. Celý řetězec by se dal rozdělit na znaky a každý získaný znak porovnat s možnými číslicemi. Další možností je rozdělit řetězec funkcí split s číselným oddělovačem. Nicméně to je všechno zbytečně pracné.

Regulární výrazy mají pro takové případy další možnou syntaxi. Chceme-li nahrazovat, použití regulárního výrazu se s prostým vyhledáváním v několika detailech liší. V zápisu se místo úvodního m používá s a jeho uvedení je povinné. Následuje vzor a řetězec, kterým bude úsek vyhovující vzoru nahrazen. Protože zde je o položku více, je třeba oddělovač navíc.

```
s/vzor/náhrada/
```

Ač je zřejmé, že tomu nemůže být jinak, podotkněme, že náhrada je vždy prostým řetězcem a nikoliv regulárním výrazem. V diskuzních fórech se dnes a denně setkáváme s reakcemi typu

```
s/nabýdku/nabídku/
```

To je právě ukázka nahrazení, jen bez využití regulárních výrazů. Autor takového příspěvku dává na vědomí, že pisatel udělal pravopisnou chybu a měla by být opravena. Tímto způsobem bychom opravu řešili v Perlu:

```
$reakce = "Takovou nabýdku nebudu komentovat...";
$reakce =~ s/nabýdku/nabídku/;
print $reakce; #tiskne opravený text
```

I kdyby bylo v řetězci více stejných chyb, opraví se jen jedna z nich. Vždy se nahrazuje pouze 1. výskyt. Po jeho nalezení totiž vyhledávání vzoru skočí úspěchem. Náš problém, kdy jsme chtěli z řetězce odstranit všechna čísla, by se dal řešit uvedením přepínače g. Ten se, podobně jako u vyhledávání, aplikuje na všechny výskyty vyhovující vzoru v řetězci. Výskyty tedy budeme nahrazovat prázdným řetězcem.

```
$retezec = "P21E251215563R413215305711L587";
$retezec =~ s/\d+//g;
print $retezec; #tiskne PERL
```

Jednoduché nahrazování - konverze znaků

Tato problematika do regulárních výrazů nepatří, ale má s nimi některé společné rysy. Z tohoto důvodu je zařazena právě zde. Činnost této konstrukce není nic jiného, než nahrazování konkrétního znaku (nikoliv regulárního výrazu) za jiný konkrétní znak. Syntaxe je podobná jako při nahrazování u regulárních výrazů:

```
$text =~ tr/abc/ABC/;
$text !~ tr/abc/ABC/;
$text =~ y/abc/ABC/;
```

y a tr jsou 2 synonymní zápisy.

Po vykonání libovolného ze zmíněných tří příkazů se obsah proměnné $text změní. Všechna a se nahradí za A, všechna b za B a všechna c za C. Obecně se tedy *n*tý znak na levé straně nahrazuje *n*tým znakem z pravé strany.

Rozdíl mezi použitím !~ nebo =~ je v návratové hodnotě. Výraz s !~ vrací true, pokud nedošlo k žádnému nahrazení. Pokud ano, vrací false. !~ se u tr příliš nepoužívá. Naproti tomu =~ vrací vždy počet nahrazení. Pokud je tento počet nenulový, návratová hodnota je tedy true.

I u tr fungují rozsahy. Není tak problémem v proměnné $text nahradit všechna malá písmena za velká, bez toho abychom je všechny vypisovali (pomiňme použití funkce uc):

```
$text =~ tr/a-z/A-Z/;
```

Teď si objasníme sporné případy použití tr:

- Pokud je na levé straně tentýž znak uvedený více než 1×, platí pouze 1. výskyt a ostatní jsou ignorovány.
- Pokud je na levé straně více znaků než na pravé, jsou znaky na pravé straně příslušně doplněny. A to tak, že poslední znak pravé strany je naklonován, kolikrát je potřeba. Například:

```
$text = "ABCDEF";
$text =~ tr/A-F/xyz/;
print $text;
```

Výsledkem bude xyzzzz. Význam tr/A-F/xyz/ je stejný jako tr/A-F/xyzzzz/, protože znak z je příslušně nakopírován.

- Pokud je na levé straně méně znaků než na pravé, je příslušný počet znaků na pravé straně zprava ignorován.

```
$text = "ABCDEF";
$text =~ tr/AB/xyz/;
print $text;
```

tr/AB/xyz/ má shodný význam se zápisem tr/AB/xy/, proto je tisknuto xyCDEF

Jako ukázku si nemohu odpustit Caesarovu šifru (v Unixu příkazy caesar, rot13). Jde o to nahradit každý znak (písmeno) znakem, který je v abecedě (ASCII tabulce, nebo zkrátka v nějaké soustavě znaků) o určitý počet znaků dále nebo blíže. Poprvé tuto šifru použili ke komunikaci Caesar a Cicero v době galské války. Písmena tehdy posouvali vždy o 3 znaky v abecedě dopředu.

Vytvoříme Caesarovu šifru s posunutím +4. Na vstupu (případně v souboru jako argumentu) bude program získávat otevřený text a následně vypisovat text šifrovaný.

K tomu potřebujeme cyklicky získávat řádky textu a zpracovávat je pomocí konstrukce tr. Posunutí +4 znamená, že a nahrazujeme za e, b za f, ..., v za z, wza a, x za b, y za c a z za d.

Takhle bude vypadat šifrovaná abeceda:

Otevřená abeceda abcdefghijklmnopqrstuvwxyz

Šifrovaná abeceda (+4) efghijklmnopqrstuvwxyzabcd

Otevřenou abecedu nahradíme šifrovanou:

```
tr/abcdefghijklmnopqrstuvwxyz/efghijklmnopqrstuvwxyzabcd/;
```

Můžeme také využít rozsahy:

```
tr/a-z/e-za-d/;
```

Vytvoříme cyklus, každou iteraci se načte řádek textu, provede se zašifrování a vytiskne se šifrovaný text:

```
while ($radek = <>){
$radek =~ tr/a-z/e-za-d/;
print $radek;
}
```

Také u tr lze pomocí výchozí proměnné vynechat operátor =~.

```
while (<>){
tr/a-z/e-za-d/;
print;
}
```

A abychom dovedli tuto ukázku téměř k dokonalosti, budeme nahrazovat i velká písmena:

```
while (<>){
tr/a-zA-Z/e-za-dE-ZA-D/;
print;
}
```

Přepínače

Pro tr existují dohromady 3 přepínače. V tabulce je jejich přehled.

Přepínač	Význam
s	Více stejných znaků za sebou, které mají být nahrazeny, jsou nahrazeny pouze 1 znakem.
c	Funguje podobně jako negace hledaných znaků. Hledá a nahrazuje se jejich doplněk.
d	Není-li dostatek znaků na pravé straně tr, nenahrazují se přebytečné znaky levé strany posledním znakem pravé strany, ale prázdným řetězcem.

Přepínač s

```
$text = "xxxxyxx";
$text =~ tr/x/X/s;
print $text;
```

Podřetězce 'xxxx' a 'xx', jsou každý nahrazeny pouze jedním znakem X. Je tak vytisknuto 'XyX'.

Přepínač c

```
$text = "xxxxyxXXxx";
 $text =~ tr/A-Z/ /c;
print $text;
```

Bez uvedení přepínače by byla nahrazena všechna velká písmena mezerou. Protože zde ale je c, nahrazuje se vše mimo velkých písmen a tiskne se ' XX '. V případné kombinaci s přepínačem s by výsledkem byl řetězec ' XX '

Přepínač d

```
$text = "ABCDEFGH";
$text =~ tr/ABC/x/d;
print $text;
```

Vytisknuto je jen 'xDEFGH'. Hledáme znaky ABC, z nich pro B a C nemáme náhradu, a protože je uveden přepínač d, budeme je nahrazovat prázdným řetězcem. V případě, že by zde přepínač d nebyl, B a C by se nahrazovaly stejně jako A znakem x.

Počet výskytů znaku v řetězci

Protože výraz s operátorem =~ vrací počet nahrazení, vrací zároveň počet původních i nových znaků. Pokud tedy nahrazujeme znak stejným znakem, žádné změny v řetězci nenastanou. Pouze získáme počet výskytů. Řádek kódu přiřazuje do proměnné $vyskytu počet výskytů znaku x v obsahu proměnné $text:

```
$vyskytu = $text =~ tr/x/x/;
```

Příště budeme pokračovat přepínači a speciálními konstrukcemi v perlových regulárních výrazech.

Perl (22) - Regulární výrazy - přepínače

Jak použít v regulárních výrazech přepínače?

Přepínače (volby) mění chování regulárního výrazu. Uvádějí se na konec regulárního výrazu (tedy za poslední lomítko, je-li uvozovacím znakem). Jejich počet není omezen. Lze je definovat globálně nebo, jak poznáme v příštím díle, lokálně.

Kompletní seznam přepínačů pro hledání a nahrazování je v tabulce:

Přepínač	Význam
e	vyhodnocuje náhradu jako výraz (jen u nahrazování)
g	pamatuje si pozici, na které skončilo poslední vyhledávání
i	nezáleží na velikosti písmen
m	metaznaky ^ resp. $ jsou na začátku resp. konci všech řádků
o	překládá vzor jen jednou
s	množina značící se tečkou zahrnuje i znak nového řádku
x	speciální syntaxe regulárních výrazů s komentáři

S některými jsme se již setkali, s některými zatím ne. Nyní jejich použití komplexně shrneme.

Velikost písma

```
print "MATCHED" if "Perl" =~ /perl/i; #vyhovuje
```

Regulární výraz vrátí true. V řetězci se sice "perl" nikde nevyskytuje, ale protože s přepínačem i se nerozlišuje velikost písmen, vyhoví i "Perl".

Vícenásobné prohledávání

Uvedením přepínače g si Perl zapamatuje, na které pozici byl nalezen výskyt a příště pokračuje od ní. To se dá využít v testu cyklu. Používá se k vyhledávání nebo nahrazování všech výskytů (implicitně je hledán jen 1. výskyt).

```
$retezec = "www.linuxsoft.cz";
while ($retezec =~ /linux/g){
$i++;
}
print "Počet výskytů slova linux v řetězci je $i\n";
```

Existuje i speciální varianta přepínače g a to gc. Jejím použitím zabráníte resetu hodnoty v případě nezdaru při porovnávání.

V seznamovém kontextu lze přepínač g užít, chceme-li získat seznam všech zapamatovaných řetězců nebo řetězců, které vyhověli vzoru.

```
@slova = ($retezec =~ /\w+/g);
```

Komentáře v regulárních výrazech

Přepínač x aktivuje speciální zápis regulárních výrazů. Čitelnost výrazu přitom rapidně roste. Jsou totiž ignorovány mezery a znaky nového řádku v regulárních výrazech. To mimo jiné umožňuje další příjemnou věc, kterou je možnost využití komentářů. Pozor si dejte jen na lomítko, popř. jiný zvolený uvozovací znak v komentáři.

Připomeňme si vzor, kterému vyhoví dvanáctihodinový čas:

```
/^((0\d)|(1[0-2]))[:\.][0-5]\d[:\.][0-5]\d[ ]?[pa]m$/i;
```

Na 1. pohled asi těžko poznáte, co by měl takový výraz vyjadřovat. S přepínačem xto bude za pár sekund jasné:

```
$cas = "12:22:11 am";
print "MATCHED\n" if $cas =~ /
#regulární výraz pro formát dvanáctihodinového času
^
((0\d)|(1[0-2]))   #HODINA - číslo mezi 00 a 12
[:\.]              #oddělovač hodin a minut
[0-5]\d            #MINUTA - číslo mezi 00 a 59
[:\.]              #oddělovač minut a sekund
[0-5]\d            #SEKUNDA - číslo mezi 00 a 59
[ ]?               #nepovinná mezera
[pa]m              #určení doby - dopoledne nebo odpoledne; přepínač i zajišťuje, že nezáleží na velikosti písmen
$
/xi;
```

Toto, jak se dozvíte příštím díle, není jediný způsob, jak vkládat do regulárních výrazů komentáře.

Jednorázová kompilace regulárního výrazu

Zejména pro urychlení programu se používá přepínač o. Regulární výraz v nějakém cyklu s přepínačem o je přeložen vždy pouze jednou a tento překlad je pak použit v každé iteraci. Tedy bez ohledu na hodnoty proměnných, které, jak víme, lze do vzorů také zakomponovat. Proměnné, uvedené v regulárním výrazu, se mohou měnit, a pak se tedy mění i samotný regulární výraz. Přepínač o tomu z výše uvedeného důvodu zamezuje. Každou iteraci je použit regulární výraz, který vznikl kompilací v první iteraci.

Ukládání regulárních výrazů

S přepínačem o souvisí jiná věc. Existuje možnost předkompilace - použití konstrukce qr//. Pokud takový regulární výraz přiřadíte do proměnné, lze ji používat místo onoho regulárního výrazu.

```
$reg = qr/\d\d\d/;
print "MATCHED" if "12" =~ $reg;   #nevyhovuje
print "MATCHED" if "123" =~ $reg;  #vyhovuje
print "MATCHED" if "77a" =~ $reg;  #nevyhovuje
print "MATCHED" if "7744" =~ $reg; #vyhovuje
```

Je též možné takový regulární výraz v proměnné zařadit do jiného regulárního výrazu.

```
print "MATCHED" if "7744" =~ /^$reg$/;   #nevyhovuje
print "MATCHED" if "7744" =~ /^\d$reg$/; #vyhovuje
```

Zkusíte-li proměnnou, ve které je regulární výraz uložen, vytisknout, bude vypadat výstup v našem případě takto: (?-xism:\d\d\d).

Náhrada jako výraz

S touto vlastností můžete plodit divy. U nahrazování jsme psali náhradu jako text. Uvedení přepínače e umožní napsat náhradu jako výraz (má to mnoho společného s eval). To by nebylo nic objevného, kdyby v ní nešlo používat zapamatované proměnné. Právě v tom tkví kouzlo.

Uvedeme si 2 příklady. Přepínač e se dobře vysvětluje pomocí funkce reverse. Převrátíme pořadí písmen ve všech slovech řetězce:

```
$retezec = "prisel jsem - videl jsem - zvitezil jsem";
$retezec =~ s/(\w+)/reverse $1/ge;
print $retezec; lesirp mesj - lediv mesj - lizetivz mesj
```

V proměnné $1 máme uložena postupně všechna (je použit také přepínač g) slova a na každé je aplikována funkce reverse. Pojďme dál a zkusme něco složitějšího. V řetězci, který budeme zpracovávat, se vyskytují ceny v amerických dolarech. Upravíme tento řetězec regulárním výrazem tak, abychom dolary nahradili českými korunami a to se vším všudy. Musíme tedy přepočítat částku v dolarech na částku v korunách a zaměnit symbol měny. Navíc je nutné ošetřit případné desetinné ceny.

```
$usd2czk = 24.133; #kurz k dolaru z 19.12.2005
$retezec = "Cena: 20.5\$";
$retezec =~ s/((\d+(.\d+)?)(\$|USD))/$2*$usd2czk.CZK/ge;
print $retezec;
```

Předpokládali jsme, že symbol měny se píše vždy za hodnotu a mezi hodnotou a symbolem není mezera. Tisknut je řetězec "Cena: 494.7265CZK". Je zde ponecháno zbytečně mnoho desetinných míst. Pomocí funkce sprintf je lze oříznout tak, aby byly vytištěny vždy právě dvě, případně doplněné nulami.

```
$retezec =~ s/((\d+(.\d+)?)(\$|USD))/sprintf("%.2f", $2*$usd2czk).CZK/ge;
```

Pokud si zkusíte nahradit vstupující text za nějaký složitější, kde se vyskytuje cena v dolarech vícekrát, jsou nahrazeny všechny. Zajímavé by také mohlo být získávání aktuálního kurzu za běhu.

Přepínač e má ještě jednu zajímavou vlastnost. Lze ho uvést pro 1 regulární výraz vícekrát. Ilustrujme si to na této ukázce.

```
$xxx = "XXX";
$_ = "***\$xxx***";
s/(\$\w+)/$1/ee;
print;
```

Počáteční stav s 2 přepínači e:

```
s/(\$\w+)/$1/ee;
```

V 1. kroku je jedno e spotřebováno na nahrazení proměnné $1 svým obsahem, tedy řetězcem '$xxx'.

```
s/(\$\w+)/$xxx/e;
```

Zbývá nám ještě poslední e, které vyhodnotí výraz $xxx - tedy nahradí proměnnou $xxx jejím obsahem.

```
s/(\$\w+)/XXX/;
```

Příště budeme v regulárních výrazech pokračovat a podíváme se na zoubek rozšířeným vzorům.

Perl (23) - Regulární výrazy - rozšířené vzory

Rozšířené vzory jsou rozšířením tradiční syntaxe regulárních výrazů o nové konstrukce.

Rozšířené vzory

Jak již bylo řečeno, rozšířená syntaxe zavádí některé speciální konstrukce. Ač možná nevědomky, již jsme se s ní setkali. Jde například o zápis závorek tak, aby jejich obsah nebyl zapamatován.

Syntaxe rozšířených vzorů vypadá obecně takto: (?*znak*). *Znak* zde zastupuje nějaký symbol nebo symboly, které blíže určují, o jaký rozšířený vzor se jedná.

*Upozornění: Některé z rozšířených vzorů jsou zařazeny pouze experimentálně a není zaručeno, že budou dostupné i ve vyšších verzích Perlu. Proto použití těchto konstrukcí konzultujte s manuálem (*man perlre*).*

Komentáře

Dalším způsobem, jak dovnitř regulárního výrazu vložit komentář je použití syntaxe (?#). Chová se stejně, jako klasické komentáře - je ignorován. Lze ho psát na libovolné místo v regulárním výrazu. Jediným omezením v komentáři je nemožnost použít uzavírací kulatou závorku. Většinou je přehlednější užít volnou syntaxi a přepínač x, nicméně rozšířený vzor tu je také.

```
print "MATCHED" if "12AFBB" =~ /^[a-fA-F0-9]+$(?#číslo v šestnáctkové soustavě)/;
print "MATCHED" if "12AFBB" =~ /^[a-fA-F0-9]+(?#číslo v šestnáctkové soustavě)$/;
```

Lokální určení přepínače

Lze nastavit, aby byl některý z přepínačů imsx aktivní jen pro určitou část vzoru. Následkem uvedení přepínače i se nerozlišují velká a malá písmena. To můžeme aplikovat na část vzoru:

```
print "MATCHED" if "abcdef" =~ /((?i)abc)def/; #vyhovuje
print "MATCHED" if "AbCdef" =~ /((?i)abc)def/; #vyhovuje - u podřetězce abc nezáleží na velikosti písmen
print "MATCHED" if "abcDef" =~ /((?i)abc)def/; #nevyhovuje - u podřetězce def záleží na velikosti písmen
```

Přepínač, kterému předřadíme znak -, je výslovně zakázán. Tímto způsobem lze globální přepínač naopak zrušit.

```
print "MATCHED" if "abcdef" =~ /((?-i)abc)def/i; #vyhovuje
print "MATCHED" if "ABCdef" =~ /((?-i)abc)def/i; #nevyhovuje - globální přepínač i, který nerozlišuje velká a malá písmena, byl přebit lokálním, který ho ruší
print "MATCHED" if "abcDef" =~ /((?-i)abc)def/i; #vyhovuje - u podřetězce def nezáleží na velikosti písmen. Přepínač i je zrušen pouze pro abc
```

Aby nebyl obsah závorek zároveň uložen, lze použít zápis (?*přepínač*:*řetězec*), což je ekvivalentní s (?:(?*přepínač*)*řetězec*)

Lokální přepínače umožňují ještě další věc. Co když v cyklu potřebujeme testovat výraz, u nějž předem nevíme, jestli bude záležet na velikosti? Určitě bychom našli nějaké okliky, jak toho docílit, ale nejlepší řešení povede právě přes lokální přepínače, které vztáhneme na celý regulární výraz.

```
foreach $vzor (qw(Praha (?i)linux)) {
print "MATCHED 1\n" if "praha" =~ /$vzor/; #nevyhovuje
print "MATCHED 2\n" if "Praha" =~ /$vzor/; #vyhovuje
print "MATCHED 3\n" if "Linux" =~ /$vzor/; #vyhovuje
print "MATCHED 4\n" if "linux" =~ /$vzor/; #vyhovuje
}
```

Pohled dopředu a dozadu

Pohled patří mezi další speciální vlastnosti. V určitém stadiu se testování řetězce může zastavit a lze porovnat část řetězce za nebo před aktuální pozicí. Pozice se přitom nemění. V konečném důsledku to znamená, že ten řetězec, který byl kontrolován z nějaké zadní nebo přední pozice, nebude součástí vyhovujícího podřetězce, přestože byl srovnáván. Změní mimo jiné věci jako návratová hodnota, hodnoty proměnných $*n* nebo proměnná $&. Syntaxe pohledu dopředu je (?=) a pohledu dozadu (?<=).

Význam těchto konstrukcí je zřejmý i z jejich přesnějších názvů - vzor (?=) se nazývá pozitivní look-ahead a (?<=) je pozitivní look-behind.

```
$r = "mandrivalinux linux linuxsoft";
@presna_shoda = ($r =~ /(?<=\W)linux(?=\W)/g);#linux
@zacinajici = ($r =~ /(?<=\W)linux\w+/g);     #linuxsoft
@koncici = ($r =~ /\w+linux(?=\W)/g);         #mandrivalinux
```

Obzvláště z 1. případu je patrné, co pohledy dopředu a dozadu dělají. Ve výsledném seznamu je jen řetězec "linux", přestože jsou ve skutečnosti kontrolovány i znaky před a po. Po odstranění pohledů bude tato kontrola zrušena.

Pokud (?=) resp. (?<=) nahradíme za (?!) (negativní look-ahead) resp. (?<!)(negativní look-behind), je význam opačný. Řetězec vyhoví pouze jestliže vzor neuvidí vepředu resp. vzadu vzor, který jsme určili.

Nakonec ještě poznamenejme, že ze zřejmých důvodů nelze v look-behind použít kvantifikátory, jež nevyjadřují konkrétní délku.

Provedení kódu Perlu uvnitř regulárního výrazu

Konstrukce (?{*výraz*}) slouží k vykonání perlového bloku kódu uvnitř vzoru, přičemž výsledek nijak neovlivňuje vzor. To lze použít v případě, kdy chceme už v regulárním výrazu přiřadit zapamatované hodnoty do proměnných.

```
$retezec = "Cena: 120USD";
$retezec =~ /Cena:\s(\d+)(?{$cena = $1})/;
print $cena; #tiskne 120
```

Místo $1 by bylo pohodlnější ve vzoru použít speciální proměnnou $^N. Jejím obsahem je vždy posledně zapamatovaný řetězec. A pro úplnost zmíním ještě proměnnou $^R, která obsahuje návratovou hodnotu posledního bloku vloženého do regulárního výrazu.

Provedení kódu Perlu uvnitř regulárního výrazu s dosazením do vzoru

Zápis (??{*výraz*}) má podobný význam jako ten předešlý. Liší se v tom, že výsledek po vyhodnocení výrazu je dosazen do vzoru.

```
$retezec = "Cena: 120USD";
$kusu = 5;
$cena_za_kus = 24;
print "MATCHED" if $retezec =~ /(??{$cena_za_kus*$kusu})USD/;
```

Vypnutí backtrackingu

Backtracking znamená posouvání pozice v porovnávaném řetězci zpět. To nastává při porovnávání pomocí hladového kvantifikátoru. Nejprve je vždy vlivem hladového kvantifikátoru spolknuta nejdelší možná část textu a poté se pozice posunuje zpět. Právě toto je backtracking.

```
print "MATCHED" if "cokoliv" =~ /.*liv/;    #vyhovuje
print "MATCHED" if "cokoliv" =~ /(?>.*)liv/;#nevyhovuje
```

V prvním případě backtracking normálně funguje. Ve druhém je však vlivem hladového kvantifikátoru spolknut celý text až do konce, backtracking neprobíhá a vzor tedy nemůže uspět.

Podmínky

A na závěr jsem si nechal řídící konstrukci. Je možné se až uvnitř regulárního výrazu rozhodnout, jak bude jeho další část vypadat. Podmínka má dva možné zápisy. Pro podmínku typu if-else to je (?(*test*)*v_případě_true*|*v_případě_false*) a pro samotné if jen (?(test)v_případě_true). Test je výrazem, který se vyhodnocuje na true nebo false. Podle vyhodnocení je aplikována příslušná část regulárního výrazu.

```
$prospel = 0;
$retezec = "nedostatečný";
print "MATCHED" if $retezec =~ /
(?(?{
$prospel == 1;                  #pokud prospěl
})
(^(výborný|chvalitebný|dobrý|dostatečný)$)   #aplikuje se tento regulární výraz
|
^nedostatečný$                  #jinak tento
)
/x;
```

Pokud máte zájem o podrobnější informace o rozšířených vzorech, odkazuji na manuálovou stránku perlre a její oddíl Extended Patterns.

Další escape znaky

V předcházejících dílech jsme již pár escape znaků poznali, ale ještě mnohé další zbývají. Pojďme si představit alespoň některé z nich.

Pomocí escape znaků lze tisknout všechny znaky ASCII tabulky. Libovolný znak se zapisuje buď osmičkově nebo šestnáckově. V 1. případě se píše příslušné trojmístné číslo za zpětné lomítko, u šestnáctkového zápisu je to dvojmístné číslo za \x.

```
print "Perl" =~ /\x50\x65\x72\x6C/; #šestnáctkový zápis
print "Perl" =~ /\120\145\162\154/; #osmičkový zápis
```

Escape znaky lze použít ke změně velikosti písmen. Převedeme řetězec s náhodnou velikostí znaků na řetězec, který velkým písmenem pouze začíná (ostatní budou malá).

```
$retezec = "náhODná VElIkOSt";
$retezec =~ s/(.?)(.*)/\u\1\L\2\E/;
print $retezec; #tiskne "Náhodná velikost"
```

Znak \u Převádí znak, který následuje, na velké písmeno. Podobně znak \l převede následující písmeno na malé. \L změní skupinu znaků na malá písmena. Skupina začíná ihned za \L a končí uvedením znaku \E nebo koncem řetězce (v ukázce tedy není nezbytně nutný). Opět existuje alternativa v podobě \U pro převedení na velká písmena, která také končí znakem \E.

Další možností escape znaků je potlačení metavýznamu části řetězce. Stačí ji jen ohraničit znaky \Q (začátek) a \E (konec). 1. případ nevyhoví, protože otazník má význam kvantifikátoru. \Q tento význam ruší.

```
print "MATCHED" if " ? " =~ /^ ? $/;
print "MATCHED" if " ? " =~ /^\Q ? \E$/;
```

Další věc nesouvisí ani tak s escape znaky, jako spíše s významem speciálních znaků. Vytvoříme vzor, kterému vyhoví jeden z těchto řetězců: "/root", "/home/xdf" nebo "/usr/local/apache2.2/xdf". V nich se vyskytuje řada lomítek, které ale mají v regulárních výrazech speciální význam. Proto jim všem musíme předřazovat zpětné lomítko.

```
/^((\/root)|(\/home\/xdf)|(\/usr\/local\/apache2\.2\/xdf))$/
```

Je to správné řešení, ale právě v těchto případech (typické právě pro adresářové cesty) je užitečné použít jiný uvozovací znak, který není ve výrazu použit. Připomínám, že je potom povinné i uvedení m před uvozujícím znakem.

```
m#^((/root)|(/home/xdf)|(/usr/local/apache2\.2/xdf))$#
```

Příště se už konečně podíváme na nějaké praktické užití regulárních výrazů.

Perl (24) - Regulární výrazy - příklady

Náš miniseriál se pomalu chýlí ke konci. Předposlední díl bude ryze praktický.

Již znáte řadu konstrukcí, které se v regulárních výrazech používají, ale jedna důležitá věc stále chybí. Dosud zde byly regulární výrazy podány výhradně teoreticky. Dnes se to změní, protože celý díl je věnován čistě příkladům.

Testování vstupu

To, že je třeba každý vstup z neověřeného zdroje testovat, je jasné. A právě regulární výrazy nabízejí spolehlivé a pohodlné řešení.

Mechanizmus takového testování si ukážeme. Budeme chtít po uživateli zadat trojciferné číslo a vzápětí zkontrolujeme, zda ho opravdu zadal. Na takové drobnosti se regulární výrazy využívají velmi často.

```
print "Zadej trojciferné číslo: ";
$cislo = <STDIN>;
if ($cislo =~ /^[1-9]\d{2}$/){
print "OK\n";
}else{
die "Chyba! Toto není trojciferné číslo!\n";
}
```

Validace emailové adresy

Vyjádření emailové adresa je další regulární výraz, který patří k těm nejčastějším. Dokonalý zápis je velice složitý. Jeho vytvoření je popsáno v knize Mastering Regular Expressions. My se spokojíme s poměrně jednoduchým zápisem. Před zavináčem povolíme alfanumerické znaky a dále tečku a pomlčku. Část za zavináčem se skládá ze 2 částí. Doménu lze vyjádřit stejně jako jméno před zavináčem a přípona se skládá ze 2-4 písmen. Uvádím pouze samotný regulární výraz:

```
/^[\w\.-]+@[\w\.-]+\.[a-z]{2,4}$/i
```

Hledání odkazů v HTML souboru

Máme HTML soubor. Z něj chceme získat vše, co je mezi a mezi <a...> a . Tedy odkaz a jeho popis. Získaná data vytiskneme.

Budeme postupně načítat řádky souboru (nebudeme brát ohled na odkazy obsahující znak nového řádku). V každém řádku se pokusíme najít HTML odkaz, z něj zapamatujeme adresu a popis a vytiskneme. Protože mohou existovat řádky bez odkazu, je

nutné tisknout pouze v případě, že porovnání vzoru bylo úspěšné. Toho dosáhneme podmíněnním příkazu print. Dále víme, že HTML není case sensitive. Proto přidáme přepínač i.

```
open SOUBOR, "index.html" or die "Nelze otevřít soubor. $!";
while (<SOUBOR>){
print "$1 => $2\n" if $_ =~ /
<A\sHREF=
["']      #uvozovky nebo apostrofy
([^"']*)  #vše mimo uvozovek nebo apostrofů
["']      #koncová uvozovka nebo apostrof
>
([^<]*)   #vše mezi <a href...> a <\/a>
<\/A>
/ixg;
}
```

Vyvstává nám tu několik problémů. Co když je na 1 řádku více odkazů? Na každý řádek se totiž aplikuje regulární výraz pouze jednou, není proto šance najít více než 1 odkaz. Problém by vyřešil další cyklus. Není ale potřeba nic zásadně upravovat. Pouze if zaměníme za while.

Dalším problémem jakým způsobem vyhoví řetězce jako <a href="xxx'...> Jako odkaz se vyseparuje pouze xxx. Jinými slovy musíme zajistit, aby byly oba uvozující znaky stejné. To je úloha jako šitá pro pamatování. Uzavřeme tedy úvodní uvozovací znak do závorek (["']) a místo dalšího ["'] pro uzavření jen \1. Podobně upravíme část ([^"']*), kde nahradíme znaky "' za \1. Po provedení úprav získáváme již funkční program:

```
open SOUBOR, "index.html" or die "Nelze otevřít soubor. $!";
while (<SOUBOR>){
print "$2 => $3\n" while $_ =~ /
<A\sHREF=
(["'])    #uvozovky nebo apostrofy
([^\1]*?) #vše mimo uvozujícího znaku
\1        #koncová uvozovka nebo apostrof, podle toho, který znak uvozuje
>
([^<]*)   #vše mezi <a href...> a <\/a>
<\/A>
/ixg;
}
```

Zvýraznění skalárních proměnných

Vytvoříme nástroj, který přijímá jako parametr soubor, ve kterém zvýrazní všechny skalární proměnné. Bude-li se například vyskytovat v textu řetězec '$promenna', bude nahrazen za '$promenna'.

Každou iterací cyklu načteme řádek zdrojového kódu Perlu, v něm zvýrazníme výskyty proměnných a vytiskneme výsledek. Jediný problém tak spočívá ve vytvoření vzoru, kterému vyhoví identifikátor skalární proměnné. Vzor bude začínat dolarem, následuje libovolné písmeno nebo podtržítko (nebereme ohled na speciální proměnné) a nakonec libovolný počet znaků slova. Celý vzor uzavřeme do kulatých závorek, získáme tak proměnnou $1, kterou použijeme jako část náhrady.

```
while (<>){
s/(\$[_a-zA-Z]{1}[\w_]*)/<font color="800000">$1<\/font>/g;
print;
}
```

Získání adresáře a jména souboru z umístění

Řetězec, který je cestou k souboru, rozdělíme na 2 části. Na adresář, ve kterém je daný soubor a jeho relativní jméno. Víme, že oddělovacím znakem je poslední lomítko. Využijeme tak hladovosti kvantifikátorů.

```
$umisteni = "/boot/grub/menu.lst";
($adresar, $soubor) = $umisteni =~ /^(.*\/)(.*)$/;
print "Adresář: $adresar\nSoubor: $soubor\n";
```

Výpočet výrazů v textu

Na vstupu přijme program textový řetězec, ve kterém se občas může vyskytnout podřetězec <<*výraz*>>. Výraz vyhodnotíme a získanou hodnotu za něj nahradíme.

Budeme tedy postupně načítat řádky textu a v něm takové výrazy hledat. Na 1 řádku se může vyskytnout více výrazů, proto použijeme přepínač g. Další přepínač, který aplikujeme, je e. Umožňuje nám přímo v regulárním výrazu nahrazovat za výsledek nějakého výrazu. A použijeme ho hned 2krát, protože ještě potřebujeme vyhodnotit řetězec, jako by byl částí zdrojového kódu.

```
while (<>){
s/<<([^>]+)>>/$1/gee;
print;
}
```

Spíše jen pro ukázku zde uvádím zápis stejného programu, který používá pouze jeden přepínač e. Funkce eval může ten druhý zastoupit.

```
while (<>){
s/<<([^>]+)>>/eval $1/ge;
print;
}
```

Pošleme programu na vstup řetězec

Týden má <<7>> dní, <<7*24>> hodin, <<7*24*60>> minut nebo <<7*24*60*60>> sekund.

a získáme

Týden má 7 dní, 168 hodin, 10080 minut nebo 604800 sekund.

Upozorňuji, že toto je jen ilustrační příklad na regulární výrazy. Mimo jiné je totiž také velkou bezpečnostní dírou. Umožňuje spuštění příkazů. Například po zadání řetězce "...<<system("rm *soubor*")>>..." bude proveden systémový příkaz rm *soubor*. Speciální techniky, kterými se lze chránit, probereme někdy v budoucnu.

Zdvojení všech znaků slova v řetězci

Problém vyřešíme tak, že vyhledáme všechny (přepínač g) výskyty znaků slova a ty jednoduše zdvojíme. Nejlepší řešení vede přes pamatování, ale abychom vyzkoušeli také něco dalšího, použijeme proměnnou $&.

```
$_ = "LINUX & PERL";
s/\w/$& x 2/eg;
print;
```

Validace HTML tagu

HTML tagem může být <html>, <a href="http://www.linuxsoft.cz">, , nebo i . Možností je celá řada. Pokusíme se je v co největší míře obsáhnout v našem řešení.

Tag bude vždy začínat znakem < a končit znakem >. Za úvodním < následuje případné lomítko a dále samotné jméno tagu. To je složeno z nenulového počtu znaků slova. Potom je možných ještě několik bílých znaků, dále opět nepovinné lomítko a nakonec znak >. To je struktura tagu, který nemá žádné parametry.

Argumenty se píší za jméno tagu. Může jich být libovolné množství. Každý z parametrů začíná bílým znakem. Tu následuje nenulový počet znaků slova. Nyní jsou v našem výrazu implementovány i přepínače. Ale stále chybí pravé parametry. Za přepínačem může nepovinně být rovnítko (případně obalené bílými znaky) a nějaká hodnota - v uvozovkách, apostrofech, nebo jako holé slovo.

U HTML tagů nezáleží na velikosti písmen. O to se ale starat nemusíme, protože jsme nikde konkrétní velikost neuváděli.

Když všechny tyto poznatky aplikujeme na regulární výraz, vznikne nám toto:

```
/
  <                   #začátek tagu
 \/?                  #případné lomítko
   \w+                #jméno tagu
    (                 #parametry
      \s+             #mezera
      \w+             #parametr
 (                    #případná hodnota parametru
                      \s*
        =             #rovnítko
                      \s*
       ("[^"]*")      #v uvozovkách
                |     #nebo
       ('[^\']*')     #v apostrofech
                |     #nebo
       ([^\W]*)       #holé slovo
                )?
              )*
              \s*
 \/?                  #případné lomítko
   >                  #konec tagu
/x
```

Ale nemůže to stále být konečné řešení, protože vyhoví i řetězec </x/>, který tagem rozhodně není. Lomítko tedy může být maximalně jedno - buď na začátku nebo na konci. Toho docílíme rozvětvením.

```
/
                <                                                  #začátek tagu
((\/\w+(\s+\w+(\s*=\s*("[^"]*")|('[^\']*')|([^\W]*))?)*\s*)   #lomítko na začátku
                |                                                  #nebo
(\w+(\s+\w+(\s*=\s*("[^"]*")|('[^\']*')|([^\W]*))?)*\s*\/?))  #případné lomítko na konci
                >                                                  #konec tagu
/x
```

Generování příkazů INSERT z dat textového souboru

Nakonec si uvedeme (po testování vstupu) asi nejpraktičtější příklad. Řekněme si, že máme data v nějakém textovém souboru a chceme je dostat do databáze. Ve zdrojovém souboru je máme k dispozici ve formátu sloupec\tsloupec\tsloupec... (sloupce oddělené tabulátorem), tedy například:

```
20050101  50000
20050102  84000
  20050103  0
20050102  -20000
20050104  47000
```

Právě pro případ dvou sloupců si napíšeme skript, který data převede na INSERT příkazy:

```
INSERT INTO finance (datum, zisk) VALUES ("20050101", "50000");
INSERT INTO finance (datum, zisk) VALUES ("20050102", "84000");
INSERT INTO finance (datum, zisk) VALUES ("20050103", "0");
INSERT INTO finance (datum, zisk) VALUES ("20050104", "-20000");
INSERT INTO finance (datum, zisk) VALUES ("20050105", "47000");
```

Náš příklad je velmi konkrétní. Mělo by to usnadnit pochopení.

Postupně načteme každý řádek, rozdělíme na jednotlivé sloupce a vygenerujeme INSERT příkaz.

```
open DATA, "data" or die "Nelze cist zdroj dat\n";
open W, ">insert.sql" or die "Nelze zapisovat\n";

while (<DATA>){
($datum, $zisk) = $_ =~ /(.+)\t(.+)/;
chomp $zisk;
print W "INSERT INTO finance (datum, zisk) VALUES (\"$datum\", \"$zisk\");\n";
}

close DATA;
close W;
```

Poznámka - samozřejmě by šlo rozdělení řešit jednodušeji přes split:

```
($datum, $zisk) = split "\t", $_;
```

Zkusíme si ještě podobnou věc a to generování INSERT příkazů z dokumentů tabulkových editorů. Přesně toto jsem kdysi potřeboval a někde jsem našel velice hezké řešení. Prvním krokem je uložit dokument v nějakém textovém formátu - například CSV. CSV soubor může vypadat třeba takto:

```
0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0
0;0;0;0;0;0;0;0;0;0;0;0;0;0;1;1;1;1;1;1;1;0;0;0;0;0;0;0;0;0;0
0;0;0;0;0;0;0;0;0;0;1;1;1;1;1;1;1;1;1;1;1;1;0;0;0;0;0;0;0;0;0
0;0;0;0;0;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;0;0;0;0;0;0;0;0;0
0;0;0;0;0;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;0;0;0;0;0;0;0;0;0
0;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;1;0;0;0;0;0;0;0;0;0
```

Středníky jsou oddělovače sloupců a konce řádků oddělovače řádků. V každé buňce máme hodnotu 0nebo 1. Tuto dvojrozměrnou strukturu budeme ukládat do jednoho rozměru. Pro každou buňku budeme tedy chtít v databázi uložit její souřadnice.

```
my $radek;
my @radek;
my $cislo_radku = 0;
my $jmeno_tabulky = "table";

open DATA, "data.csv" or die "Nelze číst zdroj dat\n";
open(W, ">insert.sql") or die "Nelze zapisovat\n";

postupně načítáme jednotlivé řádky
while ($radek = <DATA>){
$cislo_radku++;
@radek = $radek =~ /(?:([^;]*);)/g;#řádek (řetězec) je rozdělen na jednotlivé buňky

#pro každou buňku aktuálního řádku vytvoříme INSERT příkaz
for (my $cislo_sloupce=0; $cislo_sloupce<@radek; $cislo_sloupce++){
chomp $radek[$cislo_sloupce];#je-li hodnota poslední v řádku, obsahuje i znak konce řádku a ten je třeba odstranit
print W "INSERT INTO $jmeno_tabulky (x, y, hodnota) VALUES
($cislo_radku,$cislo_sloupce, $radek[$cislo_sloupce]);\n";
}
}
close DATA;
close W;
```

Poznámka - rozdělení by opět šlo řešit jednodušeji použitím funkce split:

```
@radek = split(";", $radek);
```

Dnes jsem se pokusil ukázat široké využití regulárních výrazů. Doufám, že jsem vám dodal alespoň trochu inspirace. Příští díl bude ze série o regulárních výrazech poslední. Podíváme se na měření rychlosti a debugging.

Perl (25) - Regulární výrazy - závěr

V posledním díle věnovaném regulárním výrazům se podíváme na měření rychlosti a debugging.

Porovnání rychlosti několika regulárních výrazů

Pro měření času potřebného pro vykonání určitého úseku kódu se v Perlu obvykle používá modul Benchmark. Nyní si ukážeme pouze to, jak pomocí něj porovnávat rychlosti regulárních výrazů. To ovšem není zdaleka jediné užití. Benchmark se používá poměrně často a někdy později v seriálu si podrobněji rozebereme i jeho jiné funkce. Modul Benchmark načteme takto:

```
use Benchmark qw(:all);
```

Nyní využijeme jen jeho jedinou funkci a to cmpthese. Ta se používá právě pro porovnávání rychlostí jednotlivých úseků kódu. Jako parametr jí jsou předávány 2 hodnoty:

1. číslo - je-li záporné, znamená minimální počet sekund, po které testování poběží
2. odkaz na hash, jehož každý prvek obsahuje nějaký úsek kódu - právě rychlosti těchto úseků jsou pak porovnávány

My si zkusíme dokázat, že použití závorek s pamatováním hodnoty uvnitř nich je časově náročnější než speciální syntaxe závorek, která si svůj obsah nepamatuje.

Nejprve se musíme konkrétně rozhodnout, které regulární výrazy budeme porovnávat. Jako ukázku si zvolíme například tyto:

```
my $re1 = qr/X(\w)/o;
my $re2 = qr/X(?:\w)/o;
```

Dále je třeba určit s čím se budou tyto regulární výrazy porovnávat. Protože testujeme pouze rychlost závorek, musí být jediným odlišným faktorem v regulárním výrazu právě závorky a vše ostatní musí být stejné. To platí i o řetězcích, které budeme se vzory porovnávat. V opačném případě by to mohlo ovlivňovat rychlost.

Nastavíme tedy parametry funkce cmpthese. Předáme jí hash, jehož klíči budou nějaké unikátní řetězce (jména porovnávaných úseků) a hodnotami anonymní podprogramy, v nichž probíhá test regulárního výrazu.

Testovaný úsek bude obsahovat určitý počet porovnání nějakého řetězce s regulárním výrazem. Proto musíme definovat řetězec a počet porovnání (obě tyto hodnoty budou pro testování obou regulárních výrazů stejné).

Jako řetězec, který bude srovnáván, definujeme podřetězec "X5 " uvnitř dvou stejných podřetězců.

```
my $okoli = "zxcvasdf" x 500;
my $text = $okoli . "X5 " . $okoli;
```

Nastavíme počet opakování:

```
my $pocet_opakovani = 1000;
```

Ještě nám zbývá zadat 1. argument - minimální počet sekund, po které porovnávání poběží:

```
my $delka_testu = -3;
```

A můžeme volat cmpthese:

```
cmpthese($delka_testu, {
"s pamatovanim"  => sub {$text =~ m/$re1/ for 1 .. $pocet_opakovani;},
"bez pamatovani" => sub {$text =~ m/$re2/ for 1 .. $pocet_opakovani;},
});
```

Po chvilce čekání program vytiskne výsledky ve formě tabulky, seřazené podle rychlostí vzestupně. Sloupec Rate udává hodnotu, kolikrát byl proveden daný úsek kódu (tj. 1000 porovnávání) za sekundu. Další sloupce ukazují relativní rozdíl rychlostí každé

dvojice úseků v procentech. V našem případě je tato dvojice jediná (nebereme v potaz pořadí), protože jsme k porovnávání zadali pouze 2 úseky.

```
$ perl cmpthese
                  Rate  s pamatovanim bez pamatovani
s pamatovanim  115/s             --           -31%
bez pamatovani 168/s            46%             --
$
```

Pokud si zkusíte pohrát se vstupním řetězcem (změna délky, pozicí apod.), zjistíte, že poměr rychlostí záleží i na dalších okolnostech a to dost výrazně. Vždy by ale mělo být rychlejší porovnávání bez pamatování.

Zpracování regulárních výrazů

Způsob, kterým se zpracovávají regulární výrazy můžeme kontrolovat. Ne že by se debugging používal masově, ale v případech, kdy potřebujeme opravit nějakou chybu v rozsáhlém regulárním výrazu a nemáme tušení, kde by mohla být, může pomoci. V Perlu máme možnost zapnout sledování překladu a následného vyhodnocování pomocí jedné z následujících direktiv. Direktiva

```
use re "debug";
```

zapíná debugging. Budou tak nalezeny a podrobně rozepsány všechny regulární výrazy. Lze užít také přehlednější variantu se zvýrazněním:

```
use re "debugcolor";
```

Zkusíme tímto způsobem otestovat nějaký jednoduchý úsek kódu.

```
use re "debugcolor";
"xy123456zzz" =~ /^xy\d{6}z*(1)$/;
```

Program jako obyčejně spustíme:

```
$ perl debug.pl
```

Objeví se výpis. My si vysvětlíme pouze to nejdůležitější z něj. Začneme tímto úsekem:

```
  1: BOL(2)
  2: EXACT <xy>(4)
  4: CURLY {6,6}(7)
  6:   DIGIT(0)
  7: STAR(10)
  8:   EXACT <z>(0)
 10: OPEN1(12)
 12:   EXACT <1>(14)
 14: CLOSE1(16)
 16: EOL(17)
 17: END(0)
```

Každá položka nebo skupina položek (můžeme jim říkat uzly) vyjadřuje nějakou část regulárního výrazu. Číslo na začátku každého řádku je id uzlu. Za každým uzlem je v závorce id následujícího uzlu.

BOL znamená prázdný řetězec na začátku řádku - tedy ^. EXACT je přesná shoda řetězce mezi < a > - v našem případě xy. Kvantifikátor {6} je převeden na obecnější zápis {6,6}. 6krát se opakuje vše odsazené - v našem případě jen uzel DIGIT, který označuje numerický znak. Pokračujeme přesným výskytem znaku s libovolným počtem opakování. Dále máme 1. otevírací závorku. Vše mezi OPEN1 a CLOSE1 je uloženo v $1, vše mezi OPEN2 a CLOSE2 v $2 atd. EOL označuje prázdný řetězec na konci řádku - znak $ - a konečně END je vždy na konci.

Toto zatím nemá se srovnávaným řetězcem nic společného. Probíhá pouze překlad regulárního výrazu. Porovnávat se bude až v další fázi.

Uvádím tabulku několika častých uzlů. Kompletní je k vidění v manuálové stránce perldebguts.

Uzel	Význam
Kotvy	
BOL	začátek řádku
END	konec regulárního výrazu
EOL	konec řádku
BOUND	na hranici slova
NBOUND	mimo hranici slova
Znak z množiny znaků	
ALNUM	alfanumerický znak
NALNUM	nealfanumerický znak
DIGIT	číslice
DIGIT	nečíslice
SPACE	bílý znak
NSPACE	nebílý znak
ANYOF	množina definovaná hranatými závorkami
ANY	libovolný znak
Kvantifikátory	
STAR	libovolný počet opakování
CURLY	opakování definované složenými závorkami
PLUS	minimálně 1 opakování
Ostatní	
EXACT	přesná shoda

NOTHING	prázdný řetězec
OPEN*n*	otevírací kulatá závorka
CLOSE*n*	uzavírací kulatá závorka

V tuto chvíli máme regulární výraz zkompilován a můžeme se pustit do porovnávání.

Matching REx `^xy\d{6}z*(1)$' against `xy123456zzz1'

Následují řádky, které postupně srovnávají zkompilovaný regulární výraz se vzorem. Syntaxe se liší podle toho, zda používáte direktivu use re "debug"; nebo use re "debugcolor";. V prvním případě je následovná:

pozice_ve_srovnávaném_řetězci <vyhovující_podřetězec> <zbývající_podřetězec> | id uzel

Konkrétně může vypadat třeba takto:

11 <xy123456zzz> <1> | 10: OPEN1

debugcolor má tu výhodu, že spojuje <vyhovující_podřetězec> a <zbývající_podřetězec>. Vyhovující podřetězec je bíle podbarven (v článku tučně zeleně). Varianta s debugcolor je hlavně díky této vlastnosti o mnoho přehlednější.

Vezměme teď výstup řádek po řádku.

0 <xy123456zzz1> | 1: BOL

0 <xy123456zzz1> | 2: EXACT <xy>

2 <**xy**123456zzz1> | 4: CURLY {6,6}

DIGIT can match 6 times out of 6...

1. řádek ukazuje vždy výchozí stav. Na dalším je již úspěšně nalezen znak ^ (ten je samozřejmě v každém řetězci), který ale není viditelný - proto jsme stále na pozici 0. Zajímavější je to na 3. řádku. Podbarven máme podřetězec xy. Vyhovuje totiž uzlu EXACT <xy>.

8 <**xy123456**zzz1> | 7: STAR

EXACT can match 3 times out of 2147483647...

Právě jsme úspěšně nalezli 6 po sobě jdoucích číslic. Podbarveno již máme 8 znaků.

11 <**xy123456zzz**1> | 10: OPEN1

11 <**xy123456zzz**1> | 12: EXACT <1>

12 <**xy123456zzz1**> | 14: CLOSE1

12 <**xy123456zzz1**> | 16: EOL

12 <**xy123456zzz1**> | 17: END

Dále byly nalezeny 3 znaky z a nakonec ještě znak 1. V okamžiku, kdy se regulární výraz dostane k uzlu END, skončí porovnání úspěchem.

Match successful!

Závěr

Tímto jsme definitivně skončili rozsáhlou kapitolu o regulárních výrazech. Pokud máte zájem o další informační zdroje, zde některé uvádím:

- perlre(1) - popis syntaxe regulárních výrazů v Perlu
- perlrequick(1) - vysvětluje základy regulárních výrazů
- perlretut(1) - tutoriál na téma Perl a regulární výrazy
- perlop(1), oddíly Regexp Quote-Like Operators a Gory details of parsing quoted constructs - operátory v Perlu pro práci s regulárními výrazy
- perlfaq6(1) - časté otázky
- perldebguts(1), oddíl Debugging regular expressions - sledování průběhu vyhodnocování regulárních výrazů
- kniha Mastering Regular Expressions, Jeffrey Friedl, O'Reilly and Associates. Česká recenze vyšla na serveru linuxzone.cz
- existují weby zaměřené přímo na regulární výrazy - namátkou www.regular-expressions.info nebo www.regularnivyrazy.info.
- řada článků a tutoriálů na Internetu, z nichž nemohu opomenout vynikající dílo Pavla Satrapy na www.kit.vslib.cz/~satrapa/docs/regvyr/all.html nebo práci Pavla Dařeny.
- www.regexlib.com - zde je archiv regulárních výrazů. Dříve, než začnete psát složitější regulární výraz, se podívejte sem. Možná už někdo měl stejný problém.

Perl (26) - Podprogramy

Uživatelsky definované funkce jsou základem programování složitějších aplikací. Pojďme si je přiblížit.

Pomocí podprogramů lze program rozložit na kratší úseky, což pomáhá hned v několika aspektech. Některé části programu se až na pár detailů, jimiž jsou vstupní hodnoty, častěji opakují. Když je nutné tyto části nějak změnit (například kvůli vylepšení algoritmu nebo přidání dalších vlastností), musíme jich místo jediného řádku přepisovat třeba desítky. Toto tvrzení však neplatí, pokud použijeme podprogram.

Podprogram je uživatelsky definovanou funkcí. Vykonává nějaký úsek kódu na základě algoritmu, který je pro jeden podprogram vždy stejný, a vstupních hodnot, jenž tento algoritmus používá. Podprogram lze zapsat ve kterémkoliv místě programu, ale pro přehlednost bychom vždy měli zvolit nějaký systém. Vhodné je definovat podprogramy na konci nebo začátku programu, případně v modulu. Z hlavního programu (a samozřejmě i podprogramů) pak můžeme podprogramy volat. Jakmile je podprogram definován, práce s ním je stejná jako práce s předdefinovanými funkcemi, což jsou konec konců také podprogramy.

Neméně významnou výhodou podprogramů je zvýšení srozumitelnosti. Pokud zvolíme vhodné členění programu, lze poměrně rychle poznat, jak program funguje, aniž bychom museli číst kilobajty komentářů. Obzvláště u delších programů je tak mnohem snazší hledat chyby nebo je upravovat. Každý problém je rozložen na menší problémy a ty se řeší jednotlivě.

Další výhoda souvisí s tou prvně jmenovanou a spočívá v tom, že úseky programu, které mají být stejné, opravdu stejné jsou, neboť je píšeme pouze jednou.

Definice podprogramu

Definice pojmenovaného podprogramu vypadá následovně:

sub *jméno* {
příkazy
}

Pomocí jména pak lze podprogram volat. Jak bude podrobněji rozebráno v příštím díle, lze do definice zahrnout i takzvaný prototyp. Pomocí něj si program může vyžádat počet, pořadí a typy argumentů.

sub *jméno*(*prototyp*) {
příkazy
}

Volání podprogramu

Jménu podprogramu se předřazuje znak &. Seznam argumentů podprogramu se uvádí za jeho jménem v závorkách. Použijete-li závorky, uvedení & ve většině případů nebude nutné. Pokud není podprogram použit v souvislosti s odkazy nebo jako parametr některých funkcí, Perl sám rozezná, že jde o podprogram.

```
&prumer(2, 7);
prumer(2, 7);
```

Použití

Velice jednoduchý podprogram může vypadat takto:

```
$jmeno = "uzivatel";
pozdrav();

$jmeno = "root";
pozdrav();

sub pozdrav {
print "Podprogram: Ahoj $jmeno, já jsem podprogram\n";
}
```

Nutno dodat, že tento podprogram nemá vůbec dobré vychování, neboť používá globální proměnné. Přijatelným řešením tohoto nedostatku bude předání argumentů podprogramu, jakmile se to naučíme.

Příklad by fungoval stejně i kdybychom nahradili řádek

```
pozdrav();
```

za

```
&pozdrav;
```

Argumenty a lokální proměnné

Jak již bylo napsáno pod posledním příkladem, použití globálních proměnných podprogramem je více než nešikovné. Kdyby všechny proměnné ve všech podprogramech sdílely tentýž prostor jmen, brzy bychom se zamotali. Proto je mnohem lepší předat podprogramu hodnotu jako argument. Ten v podprogramu uložíme do proměnné, která bude platná jen v rámci tohoto podprogramu (tedy bloku). Potom se uvnitř podprogramu vůbec nemusíme starat o jména globálních proměnných. Obecně platí, že čím víc je proměnná lokální, tím lépe.

Jak již víme, lokální proměnná se definuje pomocí funkce my. Funkcí podobných s myje více, ale zatím je pro jednoduchost zamlčím a někdy v blízké době si rozsahy platnosti rozebereme ve zvláštním díle.

Parametrem funkce je buď proměnná nebo seznam proměnných v kulatých závorkách. Než se pustíme do argumentů, uveďme si ještě příklad pro lepší pochopení funkce my.

```
$x = 7;
  a();
  b();
print "g: $x\n"; #tiskne 7 - lokální proměnná v &b byla platná jen pro
                 #daný blok. Po jeho ukončení má x opět globální hodnotu

sub a {
print "b: $x\n"; #tiskne hodnotu globální proměnné x - tedy 7
}

sub b {
my $x = 15;
print "a: $x\n"; #vytiskne 15 - lokální proměnná přebíjí globální
}
```

Nyní už k věci. V podprogramu je seznam předaných argumentů uložen ve speciálním poli @_. Na začátku podprogramu se obvykle vytváří lokální proměnné, které z tohoto pole hodnoty přebírají. Pokud nejde o vyloženě triviální podprogram, není zrovna přehledné uvnitř něj používat proměnné $_[0], $_[1] a jim podobné.

Nyní, když známe teorii, pojďme si ukázat praxi. Definujeme podprogram tiskni_prumer, který přebírá jako argumenty 3 čísla a vypisuje jejich aritmetický průměr.

```
tiskni_prumer(6, 9, 33); #voláme funkci tiskni_prumer
                         #se třemi argumenty 6, 9, 33

sub tiskni_prumer {
 my($a, $b, $c) = @_;
print (($a + $b + $c) / 3);
}
```

Do pole @_ jsou v našem případě uloženy hodnoty 6, 9 a 33. Ty jsme na 1. řádku podprogramu přiřadili do lokálních proměnných $a, $b a $c.

Další často užívaná možnost, jak získat hodnoty z @_ vede přes funkci shift. shiftmaže první prvek pole a jeho hodnota je výsledkem vyhodnocování.

```
print tiskni_prumer(6, 9, 33);

sub tiskni_prumer {
  my $a = shift @_;
  my $b = shift @_;
  my $c = shift @_;
print (($a + $b + $c) / 3);
}
```

Pokud podprogram voláte bez uvedení seznamu (ať už třeba prázdného), je funkci automaticky předáno pole @_. To demonstruje následující příklad.

```
@_ = (1, 2, 3);
&tisk;

sub tisk {
  print @_;
}
```

Též můžete zkusit místo &tisk volat &tisk(). V takovém případě se @_ nepředává.
Předáváte-li podprogramu 2 pole, slijí se do jednoho. Toto lze řešit pomocí odkazů, které prozatím necháme stranou. Je to též možné pomocí globálních názvů, ale jak již bylo několikrát řečeno, není vhodné, aby funkce jakkoliv zasahovala do svého okolí.
Návratová hodnota podprogramu
Co když nechceme, aby podprogram výsledky vypisoval, ale chceme je přiřadit do nějaké proměnné a následně je použít? Od toho je zde návratová hodnota - tedy hodnota, kterou bude vracet volání podprogramu při vyhodnocování. Právě díky návratové hodnotě tak můžeme podprogram volat na pravé straně od operátoru přiřazení.

```
$prumer = prumer(6, 9, 33);

sub prumer {
my($a, $b, $c) = @_;
my $prumer = ($a + $b + $c) / 3;
}
```

Jako návratová hodnota je automaticky považována poslední hodnota v podprogramu - v našem případě jí byla proměnná $prumer.
Chcete-li návratovou hodnotu výslovně uvést, což doporučuji, použijte klíčové slovo return. Jeho parametrem je návratová hodnota.

```
$prumer = prumer(6, 9, 33);

sub prumer {
my($a, $b, $c) = @_;
return ($a + $b + $c) / 3;
}
```

Program, který bude vracet průměr 3 hodnot asi nebude mít příliš velký význam. Posuneme se proto dále. Daleko větší uplatnění jistě má funkce, která počítá aritmetický průměr nezávisle na počtu argumentů.

```
sub spocitej_prumer {
my $soucet = 0;
$soucet += $_ for (@_);
return ($soucet / scalar @_);
}
```

Deklarace
Jak už bylo řečeno, podprogram může být definován kdekoliv v programu. Pokud ale voláte podprogram dříve než ho definujete a zároveň používáte céčkovský styl volání (bez ampérsandu), musíte argumenty uzavřít do závorek. Spíše pro zajímavost uvádím, že pokud ještě před voláním podprogram deklarujete, lze závorky vynechat.

```
sub prumer; #deklarace

prumer 1, 2, 6;

sub prumer {
my ($a, $b, $c) = @_;
print (($a + $b + $c) / 3);
}
```

Konstanty
Konstanta je speciální typ funkce, která neočekává žádný argument, ale pouze vrací nějakou hodnotu. Jelikož má takový podprogram při všech voláních stejné podmínky, vrací konstantní hodnotu. To, že odmítáme jakýkoliv argument, dáme na vědomí uvedením prázdného prototypu - ten zakazuje veškeré argumenty.

```
sub PI() { return 3.141592654 }

sub obsah_kruhu ($) {
my($r) = @_;
return PI*$r**2;
}

print obsah_kruhu(2);
```

Právě jsme poznali první prototyp. Příště si v tomto směru znalosti rozšíříme, neboť celý díl se bude věnovat právě jim.
Perl (27) - Prototypy

Pomocí prototypů lze kontrolovat datové typy proměnných vstupujících do podprogramu.

Téměř všechny smysluplné podprogramy přijímají nějaké argumenty, které jsou potom v podprogramu přístupné v podobě seznamu. Již z minulého dílu víme, jak argumenty přijímat. Nevíme však, jak si je i vynutit. Právě pomocí prototypů lze určit kolik jakých argumentů má podprogram vyžadovat.
Prototyp obsahuje posloupnost speciálních znaků, podle které se určuje, co za parametry bude podprogram požadovat. Je uveden vždy za názvem podprogramu v jeho deklaraci nebo definici v kulatých závorkách.

```
sub jméno(prototyp) {
příkazy
}
```

U podprogramů s prototypem je nutné, aby byla definice nebo deklarace před samotným voláním. V opačném případě nebude na prototyp brán zřetel.

Jednoduché prototypy
V tabulce jsou znaky, jimiž lze určit prototyp.

Znak	Význam
$	skalár
@	pole
%	hash
&	anonymní podprogram
*	typeglob

Znak $ určuje, že bude očekávána hodnota ve skalárním kontextu. Skalární kontext je vynucen - tzn. při předání seznamu je vrácen počet jeho prvků. Pomocí * se dá předat typeglob, což se používá pro přijímání ovladačů. Prototyp pole a hashe spolkne vše až do konce seznamu argumentů.
Pokud definujeme funkci s prototypem $$$$$$, je očekáváno 6 skalárních argumentů. Prototyp $@ zase vyjadřuje skalár a pole v daném pořadí. Pozor na zápisy jako @@. Pole se slijí, protože 1. pole spolkne vše až do konce seznamu a na 2. pole žádné hodnoty nezbydou.

```
sub f($$$) {
    print @_;
    }

f(1, 2);          #chyba, funkce f požaduje 3 skalární argumenty
    f(1, 2, 3);       #ok
f(1, 2, 3, 4, 5); #chyba, funkce f požaduje 3 skalární argumenty
```

Pokud spustíte tento úsek kódu, bude hlásit chyby. Prohoďme pořadí definice a volání funkcí tak, aby volání bylo před definicí.

```
f(1, 2);          #zdánlivě ok
    f(1, 2, 3);       #ok
f(1, 2, 3, 4, 5); #zdánlivě ok

sub f($$$) {
    print @_;
    }
```

Žádné chyby nejsou hlášeny! Jakoby prototypy nebyly specifikovány. Pokud však budete spouštět program s přepínačem -w,

```
$ perl -w prototyp.pl
```

tedy zapnutými varováními, zobrazí se pro taková volání funkcí

```
main::f() called too early to check prototype
```

Znamená to, že argumenty nebyly zkotrolovány pomocí prototypu. Zabránit se tomu dá, jak již bylo řečeno, buď definicí podprogramů před jejich použitím, nebo dopřednou deklarací podprogramu.
Deklarace popisuje parametry podprogramu před jeho definicí.

```
sub podprogram;
```

Deklarace s prototypem pak vypadá následovně:

```
sub podprogram(prototyp);
```

Ukažme si, použití v praxi. K poslednímu úseku kódu, který hlásil varování, přidáme deklaraci.

```
sub f($$$);

f(1, 2);          #chyba, funkce f požaduje 3 skalární argumenty
    f(1, 2, 3);       #ok
f(1, 2, 3, 4, 5); #chyba, funkce f požaduje 3 skalární argumenty

sub f($$$) {
    print @_;
    }
```

Teď už byl prototyp normálně aplikován. Jen pozor na to, aby nedošlo ke konfliktu prototypů. V deklaraci musí být stejný prototyp jako v definici.
Protože to, zda bude prototyp volán, záleží i na syntaxi volání podprogramu, rozlišíme si volání na 2 případy podle toho, zda bude prototyp aplikován. Tabulka obsahuje 4 možné zápisy volání:

Zápis volání	Prototyp?
&podprogram(parametry);	není aplikován!
&podprogram; (jako &podprogram(@_);)	není aplikován!
podprogram(parametry);	ok
podprogram; (jako bez parametrů)	ok

Jinak řečeno, je-li podprogram volán s ampérsandem, prototyp není nikdy aplikován.
Zde jsou další příklady využití prototypů:

```
sub f(@@);  #nemá smysl - pole se slijí
sub f();    #funkce nepřijímá argumenty
    sub f($@);  #přijme skalár a pole
sub f(&$$$);#přijme anonymní podprogram a 3 skaláry
    sub f(*);   #přijme ovladač souboru
```

Zajímavá situace může nastat, pokud voláme funkci příkazem podobným následujícímu:

```
funkce +10;
```

Jak je vlastně takový příkaz interpretován? Jako funkce s argumentem 10 nebo jako součet návratové hodnoty funkce s hodnotou 10? Je-li definován prototyp, mělo by to být jasné. V případě prázdného prototypu se příkaz bere jako součet, jinak je číslo parametrem podprogramu.

Předávání odkazů a další vlastnosti

Před symbolem, označujícím datový typ, může být znak \. To znamená, že podprogram přijímá datový typ, který označuje symbol, ale předává odkaz na něj. V samotném volání podprogramu pak nesmí být konstantní hodnota.

```
sub f(\@@);   #1. pole bude převedeno na odkaz
sub f(\%@);   #hash bude převeden na odkaz
sub f(\@\@\@);#pole budou převedena na odkazy
```

Na posledním řádku funkce přijímá jako parametr 3 pole, které jsou v podprogramu dostupné jako odkazy (skaláry) a neslijí se tak. Pak je možné volat podprogram klidně takto:

```
sub f(\@\@\@);

@a = (100, 200);
@b = (10, 20, 30);
@c = (1, 2, 3, 4, 5);
f(@a, @b, @c);

sub f(\@\@\@){
print "1. předané pole: ", @{$_[0]};
print "2. předané pole: ", @{$_[1]};
print "3. předané pole: ", @{$_[2]};
}
```

Nepovinné parametry se od povinných oddělují středníkem. Můžeme tak definovat podprogram, který bude přijímat právě 1 nebo 2 skalární hodnoty.

```
sub f($;$){print @_;}

f(10);       # ok
f(10, 20);   # ok
f(10, 20, 30); # chyba
```

Další možnost, kterou prototypy nabízejí, je specifikace více možných datových typů na tutéž pozici. Podprogram s touto deklarací přijme buď skalár nebo pole:

```
sub f(\[$@]);
```

Pojmenované argumenty

V případech, kdy funkce přijímá velké množství argumentů, z nichž jsou navíc některé povinné a některé ne, je těžké udržet si v nich pořádek. Právě s tímto mohou pomoci pojmenované argumenty. Nejde syntakticky o nic nového, jen použijeme novou filozofii předávání argumentů.

Předáme-li funkci hash s pevně danou strukturou, vytváříme vlastní systém správy argumentů. Je pak už na nás, jak si je ošetříme a zpracujeme.

Argumenty se často uvozují znakem -. Uvedeme si jednoduchý příklad.

```
vypis_jmeno("-jmeno" => "Josef", "-prijmeni" => "Adamec");

sub vypis_jmeno(%){
my %args = @_;
print "Jméno: ", $args{"-jmeno"}, "\n";
print "Příjmení: ", $args{"-prijmeni"}, "\n";
}
```

Příští díl se bude zabývat rozsahy platnosti proměnných.

Perl (28) - Rozsahy platnosti proměnných

Jak v Perlu vytvářet lokální a globální proměnné?

Definici proměnných lze provádět třemi funkcemi. V minulých dílech jsme používali pouze funkci my, aniž bychom hlouběji pátrali po jejích účincích. Mimo my existují s podobným významem další 2 funkce - local a our.

S definicemi proměnných souvisí to, že Perl umožňuje užívat pro různé datové typy proměnné stejného názvu. Podprogram &prog, skalární proměnná $prog, hash %prog, pole @prog, formát prog nebo ovladač prog. Všechno to jsou identifikátory se stejným názvem. Existují tzv. typegloby, které sdružují identifikátory se stejným názvem.

Lexikální platnost proměnné

Funkce my lexikálně deklaruje lokální proměnnou. Její rozsah platnosti je omezen jen na blok. To znamená, že je přístupná pouze zde a z jiného bloku není její hodnota žádným způsobem zjistitelná. Hodnota takové proměnné není dostupná ani z podprogramů volaných ze stejného bloku (jde však spíše o podstatu než o praktické využití takových podprogramů). Proměnná pozbývá platnosti s ukončovací složenou závorkou aktuálního bloku. Stejně jako pro bloky za podmínkou nebo v podprogramu to platí i pro holé bloky. Důkazem toho je následující ukázka.

```
{
my $p = 1;
}
print $p;
```

Vytisknuto není nic, protože platnost lokální proměnné $p je omezena jen na blok.

Plně kvalifikované jméno proměnné

Lokální proměnná pro daný blok vždy překryje globální proměnnou stejného názvu a datového typu. Přestože je proměnná překrytá, můžeme k ní přistupovat. Doteď jsme, aniž bychom si to uvědomovali, proměnné nepojmenovávali celými názvy, ale pouze jejich částmi. Plné jméno globální proměnné se skládá nejen z názvu, kterým jsme dosud vždy proměnné identifikovali, ale i z názvu balíčku, ve kterém se nachází.

Pokud příkazem package balíček neměníme, je implicitním balíčkem main. Z libovolného místa pomocí něj můžeme získat hodnotu proměnné $x, která je v tomto balíčku definována - a to zápisem $main::x. Proměnnou balíčku nelze deklarovat pomocí my, protože my deklaruje proměnné lexikálně. To znamená mimo jiné to, že taková proměnná pouze překrývá proměnnou globální. Pokud ve stejném bloku definujeme proměnnou balíčku a následně lexikální proměnnou se stejným názvem, vytvoříme tím 2 různé proměnné! K té lexikálně deklarované přistupujeme stejně jako dosud, ale chceme-li použít globální proměnnou stejného názvu, je třeba ji plně kvalifikovat.

Dynamická platnost proměnné

Další funkcí pro deklaraci proměnných je local. Syntaxe je shodná se syntaxí funkce my. Proměnné takto deklarované mají tzv. dynamicky vymezenou platnost. Při deklaraci funkcí local je případná hodnota proměnné stejného názvu jen odložena a nahrazena do konce bloku jinou hodnotou. Poté se opět vrátí na původní globální hodnotu. Takže i po deklaraci funkcí local jde o globální proměnnou! Jen je dočasně změněna její hodnota. Z tohoto důvodu je vidět i v podprogramech, volaných z bloku s deklarací.

local se užívá jen málo, ale přesto je pro některé úkoly účinným pomocníkem. Kupříkladu při práci s vestavěnými proměnnými. Potřebujeme na chvíli nějakou vestavěnou proměnnou změnit? To je úkol přesně na míru šitý pro funkci local.

```
{
local $, = ", ";
print @pole;
}
```

Vestavěná proměnná je do konce bloku nahrazena novou hodnotou. Dále se o nic nemusíme starat, protože za blokem se obsah vestavěné proměnné opět nahradí původní hodnotou.

Srovnání my a local

Stručně tedy shrňme rozdíl mezi my a local. local nedeklaruje žádnou novou proměnnou, pouze dočasně pozmění hodnotu globální proměnné stejného názvu. Oproti tomu my deklaruje skutečnou lokální proměnnou, která v daném bloku překrývá globální proměnnou.

To byla teorie. Následující příklad se dvěma podprogramy je typickou ukázkou činnosti zmiňovaných funkcí.

```
$my = 1;
$local = 1;
a();

sub a {
local $local = 2;
my $my = 2;
b();
}

sub b {
print "LOCAL:", $local, "\n";#vytiskne 2
print "MY:", $my, "\n"; #vytiskne 1
}
```

Voláme podprogram &a. V něm je pomocí my vytvořena lokální proměnná $my a zároveň je pozměněna hodnota globální proměnné $local. Dále je v tomto podprogramu volán jiný podprogram &b, v němž jsou tisknuty proměnné $my a $local. Protože žádné lokální proměnné takových jmen neexistují, tisknou se globální proměnné. V případě my je vytisknuto 1, protože tato globální hodnota byla určena už na 1. řádku. V případě local se tiskne hodnota 2. To je také hodnota globální proměnné, která však byla v podprogramu &a dočasně pozměněna.

Stejně to funguje například s hashi. Nelze napsat

```
my $hash{"x"} = 1;
```

ale pouze překrýt prvek $hash{"x"} lze. Použijeme-li local, bude jen odložena globální hodnota prvku a dočasně nahrazena novou.

```
local $hash{"x"} = 1;
```

Definice globální proměnné

Funkce our deklaruje globální proměnné. A to úplně kdekoliv. Nezáleží na hloubce zanoření. Jedinou podmínkou je, že deklarace musí proběhnout před použitím proměnné. Proměnné deklarované pomocí our jsou obsaženy v balíku (jsou tedy skutečnými globálními proměnnými), čehož lze využít pro vytváření globálních proměnných v režimu strict.

```
{
our $x = 1;
}
print $x;
```

Příklad tiskne 1, protože funkce our deklarovala globální proměnnou. Ta je globálně platná od okamžiku deklarace. Kdybychom místo our použili my, stala by se proměnná $x naopak lokální.

Zajímavostí je, že lze proměnnou deklarovat pomocí our i local zároveň. Vytváříme tak globální proměnnou, která však má hodnotu pouze v bloku.

```
{
local our $p = 1;
print "V BLOKU: $p\n";
}
print "MIMO BLOK: $main::p\n";
```

V příštím díle se zaměříme na práci se soubory.

Perl (29) - Úvod k práci se soubory

Soubory jsou prvním způsobem komunikace programu s okolím, kterým se bude seriál zabývat.

Soubory jsou místa k ukládání nebo přenosu dat. Můžeme s nimi různými způsoby manipulovat. Právě tím se budeme v několika následujících dílech zabývat.

Ovladače

Práce se soubory spočívá ve vytvoření ovladače, což je datový typ, který potom soubor zprostředkovává. Pomocí něj lze do souboru zapisovat nebo naopak data získávat.

Narozdíl od dosud poznaných datových typů nemá ovladač souboru žádný prefix. To způsobuje, že s ním musíme pracovat v určitých případech trochu jinak - například při předávání podprogramům, kopírování apod. Obvykle musíme použít typegloby. Důležité je, že práce s ovladačem je univerzální, ať se jedná a jakýkoliv zdroj dat. Nezáleží na tom, zda jde o socket, textový soubor, výstup roury nebo standardní vstup.

Existují 3 standardní ovladače, které jsou vždy automaticky otevřeny. Neodkazují na soubor, ale na výstup (obrazovku) nebo vstup (klávesnici): STDIN (standardní vstup), STDOUT (standardní výstup) a STDERR (standardní chybový výstup - jde o výpis hlášení, která nejsou ve zdrojovém kódu - například chyby).

Otevření ovladače

Ovladač je v případě textových souborů vytvářen funkcí open nebo sysopen. Později, až se setkáme například s prostředky meziprocesorové komunikace, zjistíme, že lze ovladače vytvářet i pro jiné soubory než textové.

Nejjednodušší volání funkce open má 2 parametry - jméno ovladače a jméno otvíraného souboru (pozor na přístupová práva). Jméno souboru může být absolutní i relativní. Před jménem souboru je třeba ještě dát najevo, jak bude soubor otevírán. Na výběr je jedna z následujících možností.

čtení

Uvedením < otevřeme soubor jen pro čtení:

```
open(DATA, "<soubor");
```

< je nastaveno implicitně, takže zápis bez určujícího znaku bude ekvivalentní.

```
open(DATA, "soubor");
```

Teď máme k dispozici ovladač DATA, se kterým můžeme pracovat stejně jako s kterýmkoliv jiným ovladačem.

zápis

Chcete-li zapisovat do souboru pomocí znaku > a soubor ještě neexistuje, vytvoří se. Pokud ale již existuje, bude původní obsah bez potvrzení smazán. Existuje metoda, která umožňuje vyšší kontrolu nad souborem. Tou jsou operátory pro zjišťování informací o souborech. V některém blízkém díle se jimi budeme také zabývat.

```
open(DATA, ">soubor");
```

přípis

Skrze ovladač pro přípis můžeme zapisovat, ale nikoliv mazat. Text se zapisuje na konec souboru. Pokud soubor neexistuje, vytvoří se.

```
open(DATA, ">>soubor");
```

standardní výstup

Otevření dalšího ovladače pro standardní výstup se dá využít například tam, kde se program až za běhu rozhoduje, kam bude zapisovat. Ovladač se vytváří jako zápis do souboru -.

```
open(DATA, ">-");
```

standardní vstup

A pro úplnost ještě dodejme, jak se vytvoří ovladač pro standardní vstup:

```
open(DATA, "<-");
```

Znak < je opět nepovinný.

```
open(DATA, "-");
```

roury

Ovladač jako zdroj dat může stejně dobře obsahovat i výstup nějakého shellového příkazu. Nyní přesměrujeme výstup příkazu ls / do ovladače a ovladač tak bude zpřístupňovat jména adresářů v kořenovém adresáři. Tentokrát se mód nepíše před název souboru ale až za něj.

```
open(DATA, "ls / |");
```

Pokud chceme naopak nějaký výstup přesměrovat na vstup - například na vstup příkazu more, použijeme zápis:

```
open(DATA, "| more");
```

Praktickým příkladem použití roury může být program, který odešle na danou adresu email:

```
$to = "nekdo\@nekde.cz";
$from = "ja\@mujpc.cz";
$subject = "Test";
$zprava = "text emailu\n";

open EMAIL, "| mail $to -s $subject -r $from" or die "Nelze spustit příkaz
k odeslání pošty. $!";
print EMAIL $zprava;
close EMAIL;
```

zápis (přípis) a čtení zároveň

Pro vytvoření ovladače umožňujícího zápis i čtení je třeba před módy <, > nebo >>připsat znak +. U textových souborů není výskyt takových ovladačů příliš častý. Tento přístup se používá hojně například při socketové komunikaci, kdy posíláme i přijímáme data jediným kanálem.

mód	význam
+<	kdekoliv v souboru lze číst i zapisovat
+>	kdekoliv v souboru lze číst i zapisovat, ale stávající soubor je přepsán
+>>	kdekoliv v souboru lze číst, připisovat se dá jen na konec, takže soubor není nikdy přepsán

Konkrétně:

```
open(DATA, "+<soubor");
```

verze se 3 argumenty

Mód lze uvést jako samostatný argument. Následující zápis funguje pro všechny módy. Pokud takto chcete použít rouru, musíte dát najevo jestli bude na začátku (|-) nebo na konci (-|).

```
open(DATA, ">>", "soubor");
```

kopie ovladačů

Za módy >, >>, <, +>, +>> a +< lze užít ampersand. Tím vznikne stejný ovladač jako ovladač, jehož jméno je za ampersandem.

```
open(OUT, ">&STDOUT");
```

Ovladač OUT posílá nyní data stejně jako STDOUT na standardní výstup.

Vytváření ovladačů funkcí sysopen

sysopen podobně jako open otevírá soubory, ale poskytuje nad nimi lepší přehled. Jako argumenty přijímá název ovladače, jméno souboru a příznaky oddělené operátorem |, které určují způsob otevření souboru.

Příznak	Význam
O_RDONLY	pro čtení
O_WRONLY	pro zápis

O_RDWR	pro čtení a zápis
O_APPEND	pro přípis
O_EXCL	existuje-li soubor, skončí neúspěchem
O_CREAT	pokud soubor neexistuje, bude vytvořen
O_TRUNC	vymaže obsah
O_NONBLOCK	pouze neblokující otevření

Chceme-li například do souboru připisovat a v případě, že neexistuje, jej vytvořit, funkce sysopen bude mít následující tvar:

```
sysopen(DATA, "soubor", O_WRONLY | O_CREAT | O_APPEND);
```

Zrušení ovladače

K zavření ovladače slouží příkaz close:

```
close DATA;
```

V případě, že je ovladač na konci programu ještě otevřený, měl by se zavřít automaticky sám.
Je dobré soubor nenechávat zbytečně dlouho otevřený a zavřít ho vždy, jakmile to je možné. Nikdy bychom třeba neměli nechávat otevřený soubor, jestliže program čeká na standardní vstup.

Práce s daty

Nyní si na několika příkladech ukážeme práci s otevřenými ovladači. Máme-li soubor otevřený pro čtení, můžeme k datům přistupovat přes nám již známý diamantový operátor. Jako 1. a nejjednodušší příklad napíšeme program, který opíše soubor data.txt na výstup.

```
open(DATA, "data.txt");
print <DATA>;
close DATA;
```

print je zde voláno v seznamovém kontextu. V každém prvku seznamu je řádek. Zkusme to samé s tím rozdílem, že zavoláme print opakovaně ve skalárním kontextu:

```
open(DATA, "data.txt");
print scalar $_ while <DATA>;
close DATA;
```

Teď vytvořme (hodně zjednodušenou) analogii příkazu cp. Bude umět jen kopírovat soubor do jiného. Oba soubory budou zadány. Naše verze zatím nebude přijímat ani argumenty z příkazového řádku. K tomu se dostaneme až v díle o spolupráci s příkazovým řádkem.

```
my $zdroj;
my $cil;

print "Zadejte zdrojový soubor: ";
chomp($zdroj = <STDIN>);
print "Zadejte cílový soubor: ";
chomp($cil = <STDIN>);

open(ZDROJ, $zdroj) or die "Nelze zapisovat do souboru: $!";
open(CIL, ">$cil") or die "Nelze otevřít soubor: $!";

print CIL <ZDROJ>;

close ZDROJ;
close CIL;
```

Prakticky veškerá činnost probíhá na jediném řádku, v němž kopírujeme jeden soubor do druhého.
Jako další ukázku si předvedeme přípis do souboru data.backup, kam přidáme nový řádek.

```
open(SOUBOR, ">> data.backup") or die "Nelze otevřít soubor: $!";
print SOUBOR "20060313 55000 0 0 0\n";
close SOUBOR;
```

Na závěr zjistíme 5 nejčastějších řádků ze souboru .bash_history nebo jiného souboru s historií příkazů. Prvním krokem bude načtení všech příkazů z tohoto souboru do hashe, kde klíčem bude vždy příkaz a hodnotou počet použití.

```
my $historie = $ENV{"HISTFILE"}; #cesta k souboru s historií
my %stat;

open(PRIKAZY, $historie) or die "Nelze otevřít soubor s historií!";

while ($prikaz = <PRIKAZY>){
chomp $prikaz;
$stat{$prikaz}++;
}

close PRIKAZ;
```

Poznámka - zápis cesty jako $ENV{"HISTFILE"} je lepším - tedy obecnějším - řešením než natvrdo zadaná cesta /home/user/.bash_history. V systémové proměnné $HISTFILE je uložena cesta k souboru s historií. Hashová proměnná %ENVsouvisí se spoluprácí s operačním systémem, kterou se teprve budeme zabývat.

Podle hodnot ale nelze řadit hash. Abychom si zjednodušili práci, vytvoříme pole, do jehož každého prvku uložíme text ve formátu *počet_použití_příkazu - příkaz*.

```
foreach my $klic (keys %stat){
$radky[$i] = "$stat{$klic} - $klic\n";
$i++;
}
```

Pole číselně (nemusíme si všímat případného varování) seřadíme a tiskneme požadovaný počet řádků.

```
foreach my $klic (sort {$b <=> $a} @radky){
print $klic;
$pocet--;
last if $pocet == 0;
}
```

Ještě aktualizujeme deklarace proměnných a získáváme celý zdrojový kód.
To byly nejzákladnější příkazy z oblasti práce se soubory, na které příště navážeme.
Perl (30) - Práce se soubory

Rozšiřující informace k minulému dílu.

Zápis na libovolnou pozici
Připisujeme-li do souboru, je zde možné kromě zapisování na jeho konec i přidávání textu kamkoliv jinam. Perl obsahuje funkci seek, která nastavuje pozici, kde má dojít k zápisu. seek má 3 argumenty, které určují ovladač a pozici:

1. ovladač
2. posun - tedy počet znaků, o který se bude pozice posunovat.
3. pozice, odkud se bude posunovat. Tento argument může nabývat následujících hodnot:
   - 0 - nastavuje pozici posunu
   - 1 - k současné pozici přidává posun
   - 2 - pozici nastavuje na konec souboru + posun. Je logické, že posun je pak záporný

Abychom si mohli názorně ukázat, jak funkce seek funguje, vytvoříme textový soubor, ve kterém budou dobře vidět změny a jejich pozice. Takový soubor může vypadat následovně.

```
**********
**********
**********
**********
**********
```

Zkusme v něm nahradit pár hvězdiček na libovolném místě textu:

```
open SOUBOR, "+< soubor"; #otevření pro současné čtení a zápis

seek SOUBOR, 15, 0;      #pozice 15
print SOUBOR "X";

seek SOUBOR, 3, 1;       #pozice 15+3=18
print SOUBOR "X";

seek SOUBOR, -5, 2;      #pozice (počet_znaků_souboru)+(-5)
print SOUBOR "X";

seek SOUBOR, 4, 0;       #pozice 4
print scalar <SOUBOR>;   #jako bonus vypíšeme zbytek řádku
                         #to je ukázka zápisu a čtení zároveň z 1 ovladače
```

Nyní velice hezky vidíme výsledek. Soubor, do kterého jsme zapisovali, má následující obsah.

```
**********
****X***X*
**********
**********
******X***
```

Zamykání souborů
Obzvláště při psaní aplikací, které sdílejí tytéž data - tedy například u webových aplikací, časem narazíme na problém. Může se totiž klidně stát, že několik lidí pošle serveru současně data, která mají být zapsána do téhož souboru. To vede k závažným problémům v podobě ztráty dat. Je třeba nějakou organizaci sdílení souborů. Tuto záležitost nejlépe vyřešíme blokováním přístupu k souboru během práce s ním.
Zde je syntaxe příkazu flock, který obstarává zamykání.

```
flock(soubor, režim);
flock(ovladač, režim);
```

V tabulce jsou možné režimy, do kterých lze soubor přepnout.

Režim	Název	Činnost
1	LOCK_SH	zámek pro čtení (sdílený)
2	LOCK_EX	zámek pro zápis (nesdílený)
4	LOCK_NB	neblokující zámek
8	LOCK_UN	uvolnění zámku

Zámek pro čtení zabraňuje zápisu v okamžiku, kdy soubor někdo jiný čte. Zámek pro zápis zamezuje jakémukoliv pokusu o otevření.
Pokud to není opodstatněné, rozhodně neblokujte soubory na delší dobu. Například pokud program čeká na uživatelský vstup, je soubor otevřen zbytečně, protože program dlouhou dobu nic nedělá.
Ukažme si jednoduchý úsek kódu, jež přibližuje zápis do souboru, který byl předtím uzamčen.

```
open(SOUBOR, ">soubor") or die "Nelze otevřít soubor: $!\n";
flock(SOUBOR, 2) or die "Nelze zamknout soubor\n"; #zamkneme soubor pro zápis
print SOUBOR "text, zapsaný pomocí zámku";
flock(SOUBOR, 8); #uvolníme zámek
close SOUBOR;
```

Jako parametr funkce flock lze použít místo režimu i jeho symbolický zápis, avšak k tomu je třeba zavést modul Fcntl.

```
use Fcntl qw(:DEFAULT :flock);
...
flock(DATA, LOCK_SH);#zamkneme pro čtení
```

Dočasné soubory

Dočasné soubory jsou obyčejné soubory, které slouží programu pouze po dobu jeho vykonávání. Program ho tedy musí vytvořit a smazat. Zmiňuji se o tom proto, že program může spouštět ve stejný okamžik více lidí a je nutné zabezpečit, aby každé spuštění programu mělo vlastní dočasný soubor.

Dočasné soubory jsou odlišným problémem od zamykání. Zde slouží každému uživateli jeden soubor po dobu jeho práce, narozdíl od zamykání, které se používá u veřejných souborů.

Dočasný soubor musí mít jméno charakteristické pro každého uživatele - tedy jméno, které ho jednoznačně identifikuje. Příkladem takového unikátního řetězce je proměnná $$, jež obsahuje ID procesu (PID). To je jedinečné pro každý proces běžící zároveň na stejném operačním systému. Dočasný soubor tak bude v názvu obsahovat PID. Z toho plyne, že vytváření dočasných a běžných souborů se nijak neliší, pouze u dočasných musíme zavést jednoznačná jména.

```
open(TEMP, ">/tmp/$$");
```

Přejmenování a přesouvání souborů

Příkaz rename přijímá jako první argument název existujícího souboru a druhým je nový název. Pokud zdrojový soubor neexistuje, vrací funkce false. rename se, stejně jako všechny funkce zmíněné dále, často používá v jednořádkových skriptech, o kterých bude v seriálu řeč později.

```
rename "zaloha", "20060402zaloha" or die "Soubor nebyl přejmenován. $!";
```

Mazání souborů

Funkce unlink maže všechny ze seznamu souborů (obyčejných nebo odkazů), které jí jsou předány. Pro mazání adresářů nelze unlink na většině systémech použít. Můžeme však použít funkci rmdir.

```
unlink "kopie1.dat", "kopie2.dat" or die "Soubor nebyl smazán. $!";
```

Práva a vlastník souboru

Funkce chmod přijímá mód a seznam souborů. Mód je číslo, vyjadřující práva pro vlastníka, skupinu a ostatní. Obvykle se udává v osmičkové soustavě (0755, 0711, 0644 apod.). Pokud neudáte nulu před osmičkové číslo, bude bráno jako desítkové a práva se nastaví úplně jinak!

```
chmod 0755, "logo.png", "tlacitko.png";
```

Podobná funkce, chown, mění vlastníka. Parametry jsou UID, GID a seznam souborů.

```
chown 1001, 100, "logo.png";
```

Oříznutí souboru

truncate zkrátí soubor na uvedenou velikost (počet znaků). Soubor lze uvést jménem nebo ovladačem.

```
truncate "logo.png", 100;
```

Počítadlo znaků

Napíšeme si program, který ze vstupu načte znak a jmého souboru a následně spočítá podíl (v procentech) výskytu zadaného znaku k celkovému počtu znaků v souboru. Přitom nebude počítat znak nového řádku. Dále, abychom vyzkoušeli i trochu jinou práci s ovladači, bude existovat volba, zda vypsat výsledek na výstup nebo do jiného souboru.

Nejprve tedy načteme název zdrojového souboru, hledaný znak a volbu. Dále podle volby nastavíme, kam bude směřovat ovladač CIL. Jsou 2 možnosti - buď standartní výstup nebo textový soubor. Poté prohledáme znak po znaku zdrojový soubor, přičemž budeme počítat počet výskytů hledaného znaku a také všechny znaky mimo znak nového řádku dohromady. Nakonec spočítáme hledaný podíl a vytiskneme pomocí ovladače CIL. Sice to není příklad, který dělá něco užitečného, ale můžeme si na něm demonstrovat některé konstrukce.

Nejdříve definujeme proměnné a načteme data. Přitom musíme otestovat, zda je hledaný znak opravdu jen 1 znak.

```
my $zdroj;        #jméno zdrojového souboru
my $volba;        #kam se budou tisknout výsledky - na výstup (1) nebo
                  #do souboru (2)
my $znaku = 0;    #celkový počet znaků v souboru
my $hledany;      #hledaný znak
my $hledanych = 0; #počet výskytů hledaného znaku v souboru

print "Zadejte zdrojový soubor: ";
chomp($zdroj = <STDIN>);

print "Zadejte hledaný znak: ";
chomp($hledany = <STDIN>);
if (length($hledany) != 1){
die "Toto není regulérní znak.\n";
}
```

Dále musíme načíst uživatelovu volbu. Jsou 3 možnosti. Zadá-li uživatel 1, bude ovladač CIL směřovat na standartní výstup. Zadá-li 2, bude ten samý (což je výhodné, protože se pak už o výstup nemusíme starat) ovladač směřovat do souboru. Na ten se tedy musíme uživatele zeptat a následně ho otevřít. Poslední případ nastává, pokud není zadána správná volba.

```
print "Chcete vytisknout statistiky na výstup (1) nebo zapsat
do souboru (2)? ";
chomp($volba = <STDIN>);

if ($volba == 1){
open(CIL, ">-") or die "Nelze zapisovat na standartní výstup. $!\n";
}elsif ($volba == 2){
print "Zadejte cílový soubor: ";
my $cil = <STDIN>;
open(CIL, ">>$cil") or die "Nelze otevřít cílový soubor. $!\n";
}else{
die "Toto není regulérní volba.\n";
}
```

Teď můžeme začít se samotným výpočtem. Musíme vymyslet, jak otestovat každý znak. Budeme tedy cyklem while načítat řádek po řádku, každý řádek rozdělíme funkcí split na znaky, každý znak porovnáme s hledaným znakem a inkrementujeme

příslušná počítadla (jsou-li testovaný znak a hledaný znak shodné, inkrementujeme celkový počet znaků i výskyt hledaného znaku, v opačném případě jen celkový počet). Předtím ještě musíme ošetřit, zda není testovaným znakem znak nového řádku.

```
open(ZDROJ, $zdroj) or die "Nelze otevřít zdrojový soubor: $!\n";

while (<ZDROJ>){     my @znaky = split "";

foreach (@znaky){
next if $_ eq "\n";
$hledanych++ if $_ eq $hledany;
$znaku++;
}
}
```

Počítání vyskytnuvších se hledaných znaků uvnitř cyklu lze samozřejmě řešit i jinými způsoby.
Máme celkový počet znaků i počet výskytů hledaného znaku, nic nám již nebrání spočítat podíl a tisknout ho pomocí CIL, který už směřuje na vybraný výstup.

```
my $procent = $znaku==0 ? 0 : $hledanych/$znaku*100;
print CIL "Relativní četnost znaku \"$hledany\"
v souboru $zdroj je $procent%.\n";
```

Příště se podíváme na testování souborů.

Perl (31) - Testování souborů

Hlavní náplní třetího dílu o souborech je zjišťování vlastností souborů pomocí k tomu určených operátorů.

V předcházejících dílech jsme se z oblasti práce se soubory zabývali pouze tím nejzákladnějším. Dnes navážeme a v poněkud praktičtějším díle si představíme poměrně často používané nástroje, které využijeme zejména při ošetřování chyb.

Testování souborů

Zjišťování vlastností souborů se děje skrze operátory pro testování souborů. Vycházejí z interpretu Unixu, proto vypadají podobně jako přepínače. Přijímají argument, jímž je název souboru nebo ovladač. Není-li argument uvedem, automaticky se použije obsah výchozí proměnné.

Typická situace pro použití těchto operátorů nastane, když chceme minimalizovat množství chyb při otevírání souborů. Úkolem pro ně může být například detekce práv. Jindy můžeme chtít zapisovat do souboru, přičemž soubor již existuje. V tom případě se původní obsah souboru smaže, což může mít v určitých případech neblahé následky. Toto lze řešit právě testováním souborů.

Zde je tabulka s výčtem operátorů pro testování:

Test	Význam
-e	soubor existuje
-z	soubor existuje a je prázdný
-s	vrací velikost souboru v bajtech
-d	soubor je adresářem
-l	soubor je symbolickým odkazem
-f	soubor je "obyčejným" souborem (ne tedy adresářem apod.)
-p	soubor je pojmenovanou rourou, nebo je rourou ovladač
-S	soubor je socket
-b	soubor je blokový soubor
-c	soubor je znakový soubor
-t	soubor je prostředkem tty (STDIN)
-x	soubor je spustitelný
-w	do souboru lze zapisovat
-r	soubor lze číst
-B	soubor je binární
-T	soubor je textový
-A	počet dní od posledního přístupu k souboru
-M	počet dní od poslední změny souboru
-C	počet dní od poslední změny i-uzlu
-u	soubor má nastavený setuid bit
-g	soubor má nastavený setgid bit
-k	soubor má nastavený sticky bit

Všechny testy přijímají argument v podobě názvu souboru nebo ovladače. Ten může a nemusí být uveden v závorkách.

```
-r "soubor";
-r("soubor");
```

Tyto operátory se téměř výhradně používají v podmínkových konstrukcích. Je to dáno mimo jiné tím, že kromě operátorů pro získávání časových údajů a operátoru -s, který vrací délku souboru, je má smysl používat pouze k testování pravda-nepravda.

Abychom si předvedli, jak se operátory používají, uvedeme si tři krátké úseky kódu. První a nejjednodušší bude vyšetřovat soubor a.out z hlediska práv.

```
$file = "a.out";
print "Soubor $file:\n";
print "Lze číst\n" if -r $file;
```

```
print "Lze zapisovat\n" if -w $file;
print "Lze spouštět\n" if -x $file;
print "Nastaven SUID\n" if -u $file;
print "Nastaven SGID\n" if -g $file;
print "Nastaven sticky\n" if -k $file;
```

Pokud si zkusíte skript spustit pod uživatelem s omezenými právy, změní se samozřejmě i výstup našeho programu.
Dále si demonstrujme ochranu před nechtěným přepsáním souboru.

```
$zaloha = "zaloha.bck";

if (-e "soubor"){
print "Soubor $zaloha již existuje! Přepsat? (a/n) ";
chomp($volba = <STDIN>);
if (lc $volba ne "a"){
die "Záloha nebyla provedena.\n";
}
}

open (ZALOHA, ">$zaloha") or die "Chyba: $!";
...
```

Soubor je otevřen pro zápis jen v případě, že test byl vyhodnocen jako false nebo pokud program dostal výslovný souhlas soubor přepsat.
Operátor -s získává velikost souboru v bajtech. Napíšeme program, který bude vypisovat velikost zadaného souboru, pokud půjde o obyčejný soubor. Využijeme také toho, že operátory pracují s výchozí proměnnou.

```
while(<STDIN>){
chomp;
next unless -f;
$velikost = -s;
print "$_ má ${velikost}B\n";
}
```

Zjišťování informací z i-uzlů
Fuknce stat, podobně jako stejnojmenný systémový příkaz, zpřístupňuje i-uzel. V něm jsou uloženy důležité informace o každém souboru. Parametrem funkce stat je ovladač nebo jméno souboru. Funkce vrací pole hodnot s následujícími informacemi o souboru.

Index	Název	Význam
0	dev	číslo zařízení souborového systému
1	ino	číslo i-uzlu
2	mode	mód souboru (typ a práva) v desítkovém zápisu
3	nlink	počet pevných odkazů na soubor
4	uid	uid vlastníka
5	gid	gid vlastníka (tedy id skupiny)
6	rdev	identifikátor zařízení (význam má jen u speciálních souborů. U obyčejných vrací 0)
7	size	velikost souboru v bajtech
8	atime	čas posledního přístupu k souboru
9	mtime	čas poslední změny souboru
10	ctime	čas poslední změny i-uzlu
11	blksize	velikost bloku
12	blocks	počet alokovaných bloků

Pokud chceme informace o symbolickém odkazu, použijeme lstat.
S využitím hodnoty mtime a funkce localtime, která čas poslední změny souboru převede do srozumitelného formátu, napíšeme program, který vypisuje datum a čas poslední změny souboru. Fungovat bude prakticky stejně jako náš program na zjišťování velikostí souborů, jen bude vypisovat jinou informaci.

```
while(<STDIN>){
chomp;
next unless -e;
$mtime = (stat)[9];
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime($mtime);
$datum = sprintf("%02d.%02d.%4d", $mday, $mon, $year+1900);
$cas = sprintf("%02d:%02d:%02d", $sec, $min, $hour);

print "Posledni zmena souboru je z $datum, $cas\n";
}
```

ID ovladače
Příkaz fileno vrací deskriptor ovladače. To je nějaké číslo, které je pro shodné ovladače stejné. Používá se právě k testování shodnosti ovladačů.

```
open(IN, "<-") or die "Chyba: $!";
open(OUT, ">-") or die "Chyba: $!";

print fileno IN;  #0
print fileno OUT;  #1
```

```
print fileno STDIN;#0

if (fileno IN == fileno STDIN){
print "Ovladač IN směřuje na standartní vstup";
}
```

Příští díl bude poslední k tématu práce se soubory. Zaměří se na jiné druhy souborů než obyčejné textové a navíc se podíváme na adresářovou rekurzi.

Perl (32) - Jiné typy souborů

V posledním díle o práci se soubory naleznete mimo jiné čtení z adresářů a ukázku rekurze.

Mimo textových souborů, kterými jsme se zabývali v předchozích dílech, existují jiné. Nejdůležitější z nich jsou adresáře, na něž přichází řada právě dnes. Později se v tomto seriálu objeví i další typ souboru, kterým je socket.

Symbolické a tvrdé odkazy

Funkce link (tvrdý odkaz) a symlink (symbolický odkaz) fungují podobně jako unixovský příkaz ln. Obě funkce přijímají jako 1. argument soubor, na který se bude odkazovat a 2. argument jako název odkazu.

```
symlink("soubor", "odkaz") or die "Odkaz nebyl vytvořen: $!";
```

Relativní cestu skutečného umístění souboru po zadání odkazu vrací funkce readlink.

```
print readlink "odkaz";
```

Binární soubory

Pro práci s binárními daty je třeba použít na ovladač funkci binmode, která ho přepne na binární režim. Poté už s ovladačem můžeme normálně pracovat.

```
binmode OVLADAC;
```

Adresáře

S adresáři se stejně jako u ostatních typů souborů pracuje prostřednictvím ovladačů. Opět je nutné adresář nejdříve otevřít. K tomu slouží funkce opendir, jejíž syntaxe se podobá funkci open. Čtení z adresáře provede funkce readdir a zavření closedir. readdir vrací v seznamovém kontextu seznam souborů a ve skalárním kontextu vrátí s každým voláním jednu položku adresáře. Následující kód vypisuje obsah kořenové složky a využívá seznamový kontext.

```
$, = ", ";
opendir(DIR, "/");
print readdir(DIR);
closedir DIR;
```

Zkusíme napsat program, který vypíše všechny soubory ze zadaného adresáře a určí zda se jedná o obyčejný soubor, adresář, symbolický odkaz nebo ještě něco jiného.

```
my $dir;
print "Zadej adresář, jehož strukturu chceš vytisknout: ";
chomp ($dir = <STDIN>);

opendir(DIR, "$dir") or die "Nepodařilo se otevřít $dir: $!";

while (my $pol = readdir DIR){
        if (-f $dir."/".$pol){
                print "F ";
        }elsif(-d $dir."/".$pol){
                print "D ";
        }elsif(-l $dir."/".$pol){
                print "L ";
                }else{
                print "X ";
                }

        print "$pol\n" }

closedir DIR;
```

Nyní tento příklad ještě o něco vylepšíme. V případě, že bude soubor adresářem, vypíšeme i jeho obsah. Obsahem budou soubory, které vždy odsadíme o příslušný počet pomlček podle toho, jak hluboko jsme aktuálně v zanoření. To je mimo jiné i typická ukázka rekurze.

Napíšeme tedy podprogram, který bude vypisovat obsah adresáře a v případě, že nalezne podadresář, volá opět sám sebe. Jak jistě tušíte, problémy nastanou s otevíráním adresářů, u kterých na to nemáme právo. Protože jde pouze o demonstrační program, nebudeme tuto situaci řešit a v takovém případě program necháme ukončit.

```
my $dir;
print "Zadej adresář, jehož strukturu chceš vytisknout: ";
chomp($dir = <STDIN>);
tiskni_adresarovou_strukturu($dir);

sub tiskni_adresarovou_strukturu {
my($dir, $prefix) = @_;
my $fh;
opendir($fh, $dir) or die "Nepodařilo se otevřít $dir: $!";

while (my $pol = readdir $fh){
next if $pol eq "." or $pol eq "..";
        if (-f $dir."/".$pol){
        print "${prefix}F $pol\n";
        }elsif(-d $dir."/".$pol){
        print "${prefix}D $pol\n";
tiskni_adresarovou_strukturu($dir."/".$pol, $prefix."-");
```

```
}elsif(-l $dir."/".$pol){
print "${prefix}L $pol\n";
}else{
print " ";
}
}
}
```

Mimo readdir existují další 3 funkce, které využívají otevřený adresář.

Funkce	Význam
seekdir(*ovladač*, *pozice*)	Nastavuje aktuální pozici v adresáři na *pozici*
rewinddir(*ovladač*)	Nastavuje pozici na začátek
telldir(*ovladač*)	Vrací aktuální pozici

Funkce glob

Pro výpis souborů (ať už obyčejných nebo podadresářů) z adresáře, které vyhovují danému vzoru, lze použít k tomu určenou funkci glob. Ta v seznamovém kontextu načte do pole seznam vyhovujících souborů.

```
@soubory = glob("*.pl"); #pole obsahuje jména souborů s příponou .pl v aktuálním adresáři
```

Ve skalárním kontextu vrací každé volání název dalšího vyhovujícího souboru. Následující příklad vypíše jména všech souborů v aktuálním adresáři:

```
while ($soubor = glob ("*")){
print $soubor."\n";
}
```

Ke stejnému účelu jako funkci glob lze využít operátor <> a to následujícím způsobem.

```
$, = ", ";
@soubory = <*.pl>;
print @soubory;
```

Nicméně operátor <> se touto možností poněkud tříští a většinou se dává přednost funkci glob. Pokud vám však hodně záleží na rychlosti, použijte přednostně kombinaci opendir, readdir, closedir, která je o něco rychlejší.

Další funkce pro práci s adresáři

Funkce	Význam
mkdir(*jméno*, *práva*)	vytvoří adresář
rmdir(*jméno*)	smaže adresář
chdir([*jméno*])	změní pracovní adresář, je-li to možné. Pokud není jméno uvedeno, nastaví aktuální adresář podle $HOME

Následující program bude vypisovat obsah zadaných adresářů, dokud budou zadávány. Každou iteraci cyklu bude změněn funkcí chdir aktuální adresář.

```
$, = " --- ";
print "Adresář: ";
while (<STDIN>){
if ($_ eq "\n"){exit;}
chomp;
print glob("*"), "\n\n" if (chdir $_);
print "Adresář: ";
}
```

Příští díl se bude věnovat možnostem formátování výstupu.

Perl (33) - Formátování výstupu - printf

Popis a užití funkcí pro formátování textu.

Formátovat text lze v Perlu dvěma způsoby. Pro jednodušší, většinou jednořádkový text, je většinou nejvýhodnější použít funkci printf, která je přejatá z jazyka C. Druhou možností formátování je použití formátů. Tato metoda je určena pro komplikovanější výstup.

Funkce printf a sprintf

Funkce printf v jazycích C a Perl funguje prakticky stejně. Lze pomocí ní zaokrouhlovat, formátovat čísla do požadovaného tvaru, převádět mezi základními číselnými soustavami apod.

Obecnou syntaxi této funkce můžeme zapsat následovně.

printf ***OVLADAČ*** (*formát*, *seznam*);

Ovladačem se určuje, kam bude směřovat výstup a formát je šablona, do které budou vloženy aktuální hodnoty proměnných uvedených v seznamu.

sprintf dělá s řetězcem úplně to samé, ale rozdíl oproti printf je v tom, že zatímco printf formátovaný text tiskne, sprintf ho vrací. To je výhodné v okamžiku, kdy s naformátovaným řetězcem budeme ještě chtít pracovat. V seriálu jsme se o potřebnosti této funkce již přesvědčili. V 22. dílu jsme potřebovali zaokrouhlit hodnotu na 2 desetinné číslice a dále s ní pracovat.

Pro srovnání s funkcí printf se podívejme na následující dva zápisy, které jsou ve výsledku ekvivalentní.

printf ***OVLADAČ*** *formát*, *seznam*;

print ***OVLADAČ*** sprintf *formát*, *seznam*;

Formátovací řetězec

Nyní již přistupme k významu parametrů. Formát je speciální řetězec, který obsahuje zástupné znaky. Každému zástupnému znaku je v seznamu hodnot přiřazena nějaká proměnná. Její obsah se vloží vždy na místo příslušného zástupného znaku a zároveň se zformátuje podle určené šablony.

Zde je tabulka se základními formátovacími řetězci, které budeme dále rozšiřovat.

Formát	Význam
%c	znak, odpovídající dané ASCII hodnotě

%s	řetězec
%d	celé číslo v desítkové soustavě
%u	celé nezáporné číslo v desítkové soustavě
%o	celé nezáporné číslo v osmičkové soustavě
%b	celé nezáporné číslo v dvojkové soustavě
%x, %X	celé nezáporné číslo v šestnáctkové soustavě (a-f, A-F)
%f	desetinné číslo
%e, %E	desetinné číslo v semilogaritmickém tvaru
%g	automaticky se nahradí za %f nebo %e
%p	adresa v paměti (šestnáctkově)
%n	má zvláštní význam - do proměnné přiřadí počet dosud vytisknutých znaků
%%	znak procento

Podívejme se na konkrétní příklady.

```
$f = 350 / 3;#v $f je hodnota 116.666666666667
printf("Znak s ASCII hodnotou 116: %c", $f);#t
printf("Řetězec: %s", $f);                  #116.666666666667
printf("Celé číslo: %d", $f);               #116
printf("Celé číslo šestnáctkově: %x", $f);  #74
printf("Desetinné číslo: %f", $f);          #116.666667
printf("Semilogaritmický tvar: %e", $f);    #1.166667e+02
printf("Adresa: %p", $f);                   #816cafc

printf ("Text o 17 znacích%n a další text.", $n);
print $n;                                   #17
```

Toto byly příklady toho nejjednoduššího užití. Řídící sekvence začínající znakem %mají ve skutečnosti mnohem rozsáhlejší syntaxi. Zde je její kompletní zápis.

%[*příznaky*][*minimální_délka*][.*počet_desetinných míst*]typ

To, co jsme si doposud ukázali bylo určení typu. Právě typ musí obsahovat každý formátovací řetězec. Vše ostatní je nepovinné. Příznaky určují způsob, jakým se bude hodnota zarovnávat. Pokud by zabírala hodnota méně znaků, než kolik je v šabloně, lze nastavit na jakou stranu hodnotu zarovnat a případně čím.

Příznak	Význam
0	zleva zarovná nulami
mezera	zleva zarovná mezerami
-	zprava zarovná mezerami
+	vnutí znaménko +, je-li číslo kladné
#	nejedná-li se o číslo v desítkové soustavě, vypisuje před ním 0x, 0, respektive 0b

Minimální délka specifikuje, kolik znaků bude minimálně číslo nebo řetězec zabírat. Pokud je hodnota kratší než minimální délka, doplní se například mezerami. Záleží na nastavení příznaku.

Počet desetinných míst (tedy počet číslic za desetinnou tečkou) určuje přesnost. Začíná vždy tečkou. U celých čísel a řetězců určuje maximální délku.

```
printf("Dvojkové číslo s prefixem. %#b", 9444);#0b10010011100100
printf("Vnucené znaménko. %+d", 99);          #+99
printf(">% 5d<", 10);                         #> 10<
printf(">%-5d<", 10);                         #>10 <
printf(">%05d<", 10);                         #>00010<
printf("Zaokrouhlování: %.3f", 9995.32154532); #9995.322
printf("%+010.3f", 9995.32154532);            #+09995.322
printf("%.5s", "Nevejde se...");              #Nevej
printf("desítkově:
%d\nšestnáctkově:
%x\nosmičkově: %o\ndvojkově: %b\n", 250, 250, 250, 250);
                                              #desítkově: 250
                                              #šestnáctkově: fa
                                              #osmičkově: 372
                                              #dvojkově: 11111010

printf("pi = %.10f\n", 4 * atan2(1, 1));      #3.1415926536

$plan = 11200;
$vyrobeno = 10355;
printf("Plán je splněn z %3.1f%%\n", $vyrobeno / $plan * 100);
                                              #Plán je splněn z 92.5%
```

Další možnosti

Přesnost čísla pomocí proměnných lze zadávat i pomocí znaku *. Tyto zápisy mají stejný význam.

```
printf ">%${pocet}d<", 5;
printf ">%*d<", $pocet, 5;
```

Pomocí sekvence \$ lze určovat, který argument ze seznamu se použije.

```
printf "%2\$d", 11, 22, 33;
```
Do formátovacího řetězce můžeme připsat znak v a řetězec se rozloží na celá čísla oddělená implicitně tečkou.
```
printf "%vd\n", 123;
```
printf umí ještě další věci, ale již jen odkazuji na manuálovou stránku perlfunc(1).

Formátovaný výpis ASCII tabulky

Abychom si ukázali trochu více než pouhé modelové příklady, pokusíme se zformátovat ASCII tabulku. Napíšeme tedy program, který načte ze vstupu jméno souboru a do něj uloží znaky 33 - 126 z ASCII tabulky. Na každém řádku bude znak a jeho desítkový, osmičkový a šestnáctkový zápis.
```
#!/usr/bin/perl
use strict;

my $cil;

print "Kam chcete uložit ASCII tabulku? ";
chomp($cil = <STDIN>);

open CIL, ">$cil" or die "Nelze psát do souboru $cil: $!";

print CIL "ZNAK DEC OCT HEX\n";
for (my $i=33; $i<=126; $i++){
printf CIL ("%4c %3d %3o %3x\n", $i, $i, $i, $i);
}

close CIL;
```
Celý algoritmus je velmi jednoduchý. Zaměříme se hlavně na funkci printf. Všimněme si, že se tiskne čtyřikrát totéž číslo, pokaždé však jinak zformátované. Poprvé jako znak s velikostí 4. Další 3 sloupce mají velikost 3 a určují pořadí znaku v ASCII tabulce.

Soubor, do kterého bude tabulka uložena se nám takto hezky srovná:
```
ZNAK DEC OCT HEX
...
<  60  74  3c
=  61  75  3d
>  62  76  3e
?  63  77  3f
@  64 100  40
A  65 101  41
B  66 102  42
C  67 103  43
D  68 104  44
...
```
Tabulka goniometrických funkcí

Na dalším příkladu si předvedeme použití sprintf. Vytvoříme tabulku se třemi sloupci. V prvním bude hodnota a v dalších její sinus a kosinus s přesností 5 desetinných míst. Použijeme přitom hodnoty z intervalu <0; 2pi> po osminách.

Tabulka musí mít nějakou hlavičku a rámeček. K tomu budeme ale potřebovat znát předem její šířku. Jak ji ale zjistíme dříve, než vypíšeme hlavičku? Naštěstí je tu funkce sprintf. Nic vypisovat nebudeme, formátovanou hlavičku jen uložíme do proměnné a zjistíme její délku pomocí speciální sekvence %n. Až teď můžeme tisknout horní rámeček.
```
my $hlava = sprintf("| %11s | %8s | %8s | %n\n", "x", "sin x", "cos x", my $n);
my $line = "+" . "-" x ($n - 3) . "+\n";
print $line, $hlava, $line;
```
Teď konečně přijde na řadu obsah tabulky. Musíme zachovat šířku buněk.
```
my $osmina_pi = atan2(1,1) / 2;
for (my $i=0; $i<=16; $i++){
printf ("| %11s | %8.5f | %8.5f |\n", "$i / 8 * pi", sin($i*$osmina_pi),
cos($i*$osmina_pi));
}
```
Tabulku nakonec musíme uzavřít.
```
print $line;
```
Tímto získáváme na standardním výstupu formátovanou tabulku.

Perl (34) - Formátování výstupu - formáty

Formát je dalším nástrojem pro tisk formátovaných dat.

Formáty slouží opět k vytváření výstupů podle šablony. Oproti printf se užívají zejména pro šablony větších rozsahů, kdy je užití formátů nejen přehlednější, ale i jednodušší.

Abychom mohli formáty používat, musíme ze všeho nejdříve definovat šablonu, kterou potom můžeme libovolněkrát použít.

Definice šablony

Šablona se definuje klíčovým slovem format. Jeho syntaxe obsahuje název formátu, uvozující rovnítko, masky, hodnoty a ukončující tečku.
```
format NAZEV =
formátovací_řetězec1
hodnoty1
formátovací_řetězec2
hodnoty2
...
.
```
Nezadáte-li název, implicitně se použije název cíle dat - v našem případě většinou STDOUT.

Formátovací řetězec se skládá z prvků, které začínají znakem @. Může obsahovat i text. Zde jsou znaky, které lze používat:

Znak formátovacího řetězce	Význam
@	začíná položku hodnoty
^	začíná položku postupně vkládané hodnoty
>	zarovnání doprava
<	zarovnání doleva
|	zarovnání na střed
...	zobrazí ..., pokud se nevešel celý řetězec
#	zobrazování čísel, ale pokud je uveden jako 1. znak řádku, pak se bere jako začátek komentáře. Implicitně se zarovnává doprava
0	zarovnání nulami zleva
.	desetinná tečka
~	nebudou vytištěny prázdné řádky
~~	vypisuje text po řádcích
@*	vypisuje libovolně dlouhý text
^*	vypisuje libovolně dlouhý řádek textu

Abychom si lépe ujasnily význam těchto znaků, vyjádříme z nich několik konkrétních vzorů. Jak se můžete z tabulky přesvědčit, je jejich vytváření intuitivní.

Formátovací řetězec	Význam
@<<<<	prvních 5 znaků (zavináč také zastupuje znak) zadaného textu, zarovnání doleva
@>>>>>>>>>	hodnota zarovnaná doprava
@>>>>>>>>>...	hodnota zarovnaná doprava, pokud se nevejde, končí třemi tečkami
@|||||||||	centrovaná hodnota
@###.##	desetinné číslo ve tvaru xxxx.xx, zleva případně doplněné mezerami
@0##.##	desetinné číslo ve tvaru xxxx.xx, zleva případně doplněné nulami
@*	libovolně dlouhý řetězec

Jako ukázku si napíšeme pro začátek jednoduchou šablonu. Bude tisknout prvních 20 znaků z dané proměnné. Text přitom bude vycentrován uvnitř hranatých závorek.

```
format =
[@||||||||||||||||||]
$text
.
```

Aplikace šablony

Data zobrazíme funkcí write, které můžeme předat jako parametr ovladač, kam se budou data posílat. Ovladač je zde svázán se jménem šablony. Důsledkem toho je, že použijeme-li pro výstup funkci write, bude vytištěn záznam podle příslušného formátu. Následujícím způsobem aplikujeme šablonu vytvořenou výše.

```
$text = "centrovaný text";
write;

format =
[@||||||||||||||||||]
$text
.
```

Pokud používáme jiný formát než STDOUT a chceme tisknout na STDOUT, musíme ještě před tiskem nastavit proměnnou $~. V té je uchovávána implicitně používaný formát.

```
$~ = "DATA";
$text = "centrovaný text";
write;

format DATA =
[@||||||||||||||||||]
$text
.
```

Příklad užití

Pokusme se ještě o jeden příklad. Vytvoříme formátovaný kurzový lístek. Data, uložená v hashi, vypíšeme pomocí cyklu. Každou jeho iteraci bude volán příkaz write;.

```
my($mena, $cena);
my %kurzy = (
"Austrálie AUD" => 17.283,
"Čína CNY" => 2.808,
"Dánsko DKK" => 3.8,
"EMU EUR" => 28.335
);

foreach my $key (keys %kurzy){
```

```
$mena = $key;
$cena = $kurzy{$key};
write;
}

format =
Měna: @<<<<<<<<<<<<<< Cena v korunách: @####.##
$mena,                $cena
.
```

Výstup se nám přesně podle šablony zformátuje. Nutno však poznamenat, že na takto jednoduchý příklad by stačila i funkce printf.

Rozdělení textu do více řádků

Výměnou @ za ^ lze dosáhnout rozdělení obsahu proměnné na více řádků s pevnou délkou. Zároveň jsou zachovávány slova, je-li to možné. Zvolme například 15 znaků jako délku řádku.

```
$text = "Nějaký text, který chceme rozdělit na více částí.";
write;

format =
[^|||||||||||||||]
$text
.
```

To ještě není rozdělení, ale pouze odříznutí všeho, co je za 15. znakem.

```
$ perl format.pl
[ Nějaký text, ]
$
```

Nyní provedeme skutečné rozdělení.

```
$text = "Nějaký text, který chceme rozdělit na více částí.";
write;

format =
[^|||||||||||||||]
$text
[^|||||||||||||||]
$text
[^|||||||||||||||]
$text
[^|||||||||||||||]
$text
[^|||||||||||||||]
$text
.
```

Text je úspěšně rozdělen.

```
$ perl format.pl
[ Nějaký text, ]
[ který chceme ]
[ rozdělit na ]
[ více částí. ]
[ ]
$
```

Sice jsme dosáhli cíle, nicméně za vysokou daň. Sami asi ze zdrojového kódu vidíte, že takto postupovat nelze. Celý formát je navržen absolutně nepružně. Tento problém však elegantně vyřešíme pomocí již zmíněné sekvence ~~.

```
$text = "Nějaký text, který chceme rozdělit na více částí.";
write;

format =
[^|||||||||||||||]~~
$text
.
```

Nyní již není omezena délka textu, protože se automaticky vytvoří potřebný počet řádků.

Poznámka - Perl implicitně neláme slova. Pokud však příkazem

```
$: = "";
```

nastavíme, že lámat lze všude, dostaneme následující výstup.

```
$ perl format.pl
[Nějaký text, kt]
[erý chceme rozd]
[ělit na více čá]
[ stí. ]
$
```

Výpis řádků s odsazením

Sekvenci ~~ lze užít mimo předchozího i k dalším účelům. Níže uvedený kód vypisuje seznam, jehož položky jsou přehledně pod sebou.

```
$text = "1. položka\n2. položka\n3. položka\n4. položka\n";
write;

format =
Seznam: ^*
```

```
$text
^* ~~
$text
.
```

Nejprve je vypsán 1. řádek textu za řetězcem Seznam:, poté se vypíše o řádek níž další a protože je uvedeno ~~, bude se to opakovat, dokud bude nějaký řádek k dispozici.
Je nutné si uvědomit, že ^* ve vzoru zastupuje 1. položku (resp. řádek) a ^* ~~postupně všechny ostatní položky.
Položky se nám tak srovnají pod sebe, což můžeme vidět na výstupu.

```
$ perl format.pl
Seznam: 1. položka
        2. položka
        3. položka
        4. položka
$
```

Zápis formátovaného textu do souboru

Při přesměrování výstupu do souboru ovladač souboru pojmenujeme stejně jako je název formátu. Parametrem write potom musí být název ovladače. Zapíšeme do souboru tabulku, obsahující 1., 2., 3., 4. a 5. mocniny čísel 0-20.

```
open FILE, ">soubor";

for ($x=0; $x<=20; $x++){
write FILE; #totéž co select FILE; write;
}

close FILE;

format FILE =
@######## @######## @######## @######## @########
$x,     $x**2,   $x**3,   $x**4,   $x**5
.
```

Hlavičky

Dosud jsme produkovali šablony, které mají značnou nevýhodu. Nelze jim vložit hlavičky. To je u různých tabulek nebo sloupcových výčtů nezbytné. Ještě než se začne aplikovat šablona, potřebujeme aby se automaticky aplikovala šablona hlavičky. Perl nabízí následující řešení. Máme formát TABULKA. Vytvoříme další formát s názvem TABULKA_TOP, který bude obsahovat právě formát hlavičky. Příkazem write; se nyní jednou provede hlavička a poté se vypisují už jen data. Platí, že je-li definován formát NÁZEVFORMÁTU_TOP, je aplikován jako hlavička formátu NÁZEVFORMÁTU.
Poslední příklad na zápis tabulky do souboru trochu rozšíříme. Přidáme do něj hlavičku tak, že vytvoříme formát FILE_TOP.

```
format FILE_TOP =
@>>>>>>>>> @>>>>>>>>> @>>>>>>>>> @>>>>>>>>> @>>>>>>>>>
"x",      "x^2",    "x^3",    "x^4",    "x^5"
-------------------------------------------------
.
```

Patičky

Definice patiček je o něco složitější. Je nutné nastavit 2 speciální proměnné. Proměnná $= specifikuje po kolika vypsaných řádcích (do kterých se počítají i řádky hlavičky) bude pata vypisována. V proměnné $^L je pak samotný obsah patky.
Je-li tedy v proměnné $= hodnota 10, každých 10 řádků výstupu formátu se vypíše patka, poté znovu hlavička a dál pokračují data. Po vypsání dat je ale nutné ještě zvlášť vypsat hlavičku, aby na poslední straně nechyběla.

```
$= = 10;
$^L = "---------KONEC---------\n\n";

for ($x=0; $x<=50; $x++){
write; #totéž co select FILE; write;
}
print $^L;

format =
Řádek: @<
       $x
.

format STDOUT_TOP =
----ZAČÁTEK STRANY----
.
```

Je zde jeden nedostatek. Pata na poslední straně může být klidně třeba v polovině stránky. My bychom ale chtěli každou patku přesně na konec strany (tedy na stejné místo jako na ostatních stranách). To obnáší vynechat nějaký proměnný počet řádků. A právě počet řádků, které zbývají do konce strany, je uložen v proměnné $-. Řádek print $^L; nahradíme za:

```
print "\n" x $- . $^L;
```

Stránkování

V proměnné $% je vždy uloženo aktuální číslo strany. Změníme hlavičku tak, aby ho obsahovala. Formát STDOUT_TOP bude vypadat takto:

```
format STDOUT_TOP =
---ZAČÁTEK STRANY @<--
                  $%
.
```

Příště začneme debugging.

Dnes převážně o možných příkazech debuggeru.

Chyby v programech mohou být dvojího druhu. Syntaktické, na ty nás upozorní překladač, nebo logické. Laděním rozumíme hledání a odstraňování logických chyb. Nejčastějším nástrojem pro ladění jsou ladící tisky nebo debugger.
Občas za nás také vyřeší mnoho práce pragma warnings (případně přepínač -w), který upozorňuje na sice syntakticky správné, ale nečekané zápisy.

Vestavěnný debugger

Když spustíme Perl s přepínačem -d, program poběží v režimu debugger. Debuggerem rozumíme nástroj, který postupně krok po kroku prochází program, vypisuje informace o stavu proměnných v určitých okamžicích a umožňuje tak snadněji hledat chyby.

Ovládání vestavěného debuggeru

Abychom si mohli ukázat základní příkazy debuggeru, vytvoříme si krátký, ale pokud možno různorodý program. Dále budeme pracovat s následujícím kódem v souboru program.pl.

```
#!/usr/bin/env perl
use strict;

my $prumer;

$prumer = nacti_data();

pis_vysledky($prumer);


sub nacti_data {
    my $prumer;
    print "Zadej prumer: ";
    $prumer = <STDIN> until ($prumer > 0);
    return $prumer;
}

sub pis_vysledky {
    printf "Obvod kruhu: %6.2f\n", PI()*$_[0];
    printf "Obsah kruhu: %6.2f\n", PI()*($_[0]/2)**2;
}

sub PI() {
    return 4*atan2(1, 1);
}
```

Nyní spusťme program.pl v debuggeru.

```
$ perl -d program.pl

Loading DB routines from perl5db.pl version 1.27
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(program.pl:4):       my $prumer;
DB<1>
```

Debugger čeká na naše příkazy. Ale než-li se k nim dostaneme, objasněme si ještě význam promptu. Poslední dva řádky se skládají z několika údajů.

- main - jméno aktuálního balíku
- program.pl - jméno souboru
- 4 - číslo aktuálního řádku (řádky #!/usr/bin/env perl, use strict;, případně prázdné řádky jsou vynechány)
- my $prumer; - aktuální řádek laděného souboru
- <1> - číslo příkazu pro debugger

Nápověda

Příkazem h se objeví seznam příkazů. Nápovědu k jednotlivým příkazům získáme příkazem h příkaz. Kompletní nápovědu získáme příkazem h h, případně |h h.

Výpis kódu laděného programu

Příkaz l vypisuje 10 řádků programu od aktuální pozice. Dalším zadáním příkazu l se zobrazí opět 10 řádků, ale až od místa, kde minulý výpis skončil.

```
DB<1>
4==>    my $prumer;
5
6:      $prumer = nacti_data();
7
8:      pis_vysledky($prumer);
9
10
11
12      sub nacti_data {
13:         my $prumer;
DB<1>
```

Pokud chceme vidět 6. řádek a nic okolo, použijme příkaz l 6.

```
DB<1> l 6
```

```
6:      $prumer = nacti_data();
DB<2>
```

Je též možné určit řádky od do (příkaz l od-do, například pro řádky 2 až 6 l 2-6). Podobně lze získat daný počet řádků od dané pozice. Například příkaz l 3+5 znamená zobraz 3. řádek a dalších 5 za ním). K zobrazení deseti předcházejících řádků máme příkaz -.

```
DB<2> l 1-5
1       #!/usr/bin/env perl
2:      use strict;
3
4==>    my $prumer;
5
DB<3>
```

Řetězec ==> za číslem řádku v každém víceřádkovém výpisu ukazuje aktuální řádek. Deset řádků okolo aktuálního vypíšeme příkazem v.

```
DB<3> v
1       #!/usr/bin/env perl
2:      use strict;
3
4==>    my $prumer;
5
6:      $prumer = nacti_data();
7
8:      pis_vysledky($prumer);
9
10
DB<3>
```

Hledání v kódu programu

Debugger umožňuje hledat výskyty zadaého řetězce ve zdrojovém kódu. Směrem dopředu se hledá příkazem /řetězec/ a zpět příkazem ?řetězec?. Koncové / případně ? není povinné.

```
DB<4> /prumer
8:      pis_vysledky($prumer);

DB<5> /prumer
13:         my $prumer;

DB<6> ?prumer
8:      pis_vysledky($prumer);

DB<7>
```

Krokování

Příkaz s je jedním z vůbec nejzákladnějších příkazů debuggeru. Po jeho zadání se provede příkaz a řádek kódu se zároveň vypíše. Vykonává se skutečně jen jediný příkaz i přesto, že je řádek vypisován pokaždé celý (na řádku může být více než jeden příkaz). Pokud debugger narazí na podprogram, provádí se od prvního řádku stejně, jakoby byl rozepsán v hlavním programu. Podobnou funkci jako s má příkaz n. Chová se stejně až na to, že volání podprogramu bere jako každý jiný příkaz a nevstupuje do něj.

Prázdný příkaz (stisk ENTER) zopakuje poslední příkaz s nebo n.

```
DB<7> s
main::(program.pl:6):       $prumer = nacti_data();
DB<7>
main::nacti_data(program.pl:13):        my $prumer;
DB<7>
main::nacti_data(program.pl:14):        print "Zadej prumer: ";
DB<7>
main::nacti_data(program.pl:15):        $prumer = <STDIN> until ($prumer > 0);
DB<7>
Zadej prumer: 25
main::nacti_data(program.pl:16):        return $prumer;
DB<7> n
main::(program.pl:8):     pis_vysledky($prumer);
DB<7>
Obvod kruhu:  78.54
Obsah kruhu: 490.87
Debugged program terminated.  Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
DB<7>
```

Z výpisu je hezky vidět, že debugger data vypisuje a přijímá stejně jako kdybychom program normálně spustili.

Pokud krokujeme v podprogramu a chceme, aby se zbytek kódu v něm provedl už bez krokování, slouží k tomu příkaz r. Dalším příkazem, který provádí kód je c [číslo_řádku|podprogram]. Ten provede program až po daný řádek nebo začátek daného podprogramu. c bez parametru provádí program do příští zarážky a nebo, pokud za aktuální pozicí žádná zarážka není, tak do konce. Restartujeme tedy provádění příkazem R a zkusíme provést vše až do 8. řádku.

```
DB<7> R
...
DB<7> c 8
Zadej prumer: 12
main::(deb.pl:8):       pis_vysledky($prumer);
```

DB<8>

Příkaz c program provádí, ale implicitně nezobrazuje zdrojový kód. Příkazem taktivujeme trasovací režim a poté už bude každý provedený příkaz vypisován.

Výpis hodnot

Pomocí příkazu p se tiskne hodnota proměnné na výstup. Aktuální hodnotu proměnné $prumer dostaneme takto.

```
DB<6> p "prumer=$prumer"
prumer=12
```

DB<7>

Dalšími příkazy k výpisu proměnných jsou X, V a x. X vypisuje proměnné definované v aktuálním balíku, V proměnné v balíku, uvedeném jako parametr.

Příkaz x umí znázornit složitější strukturu. Je však třeba předat mu odkaz, protože v opačném případě zobrazí pouze výčet prvků.

```
DB<44> x \%hash
0  HASH(0x8171874)
'klic1' => 'hodnota1'
'klic2' => 'hodnota2'
'klic3' => 'hodnota3'
```

DB<45>

Příkaz S vypisuje dostupné podprogramy. Bez parametru vypíše všechny. Jako parametr přijímá název balíčku (o balíčcích budeme hovořit hned po debuggingu).

```
DB<3> S main
main::BEGIN
main::PI
main::nacti_data
main::pis_vysledky
```

DB<4>

Všechny používané moduly zobrazíme příkazem M.

Nastavení proměnných

Příkazem o zobrazíme seznam proměnných debuggeru a jejich hodnoty. Změnu některé proměnné provedeme příkazem o proměnná=hodnota.

Zarážky

Lze nastavit, aby se skript normálně vykonával do určitého místa, které je označeno zarážkou a poté se zastavil a čekal na další příkazy. Zarážku definujeme příkazem b číslo_řádku (zarážka se nastaví na číslo_řádku) nebo b název_podprogramu(zarážka se nastaví na první příkaz v zadaném podprogramu). Máme-li definované zarážky, zadáme příkaz c. Program se vykoná do řádku, na kterém je 1. zarážka a zastaví se. Dále můžeme opět krokovat nebo znovu stisknout c a dostat se na další zarážku.

Takto jsme definovali zarážky, které budou zarážkami za všech okolností. Lze ale definovat zarážku, která bude zarážkou jen pokud bude platit nějaká podmínka. Například zadáme-li příkaz b 9 $p > 150, bude zarážka aktivní jen v případě, že na jejím místě bude hodnota proměnné $p větší než 150.

Zarážky se mažou podobným způsobem, jen se místo b používá příkaz B. Všechny zarážky smažeme příkazem B *.

```
DB<1> b 7
DB<2> b pis_vysledky
DB<3> l 1-30
...
6
7:b     $prumer = nacti_data();
8
...
20      sub pis_vysledky {
21:b        printf "Obvod kruhu: %6.2f\n", PI()*$_[0];
22:         printf "Obsah kruhu: %6.2f\n", PI()*($_[0]/2)**2;
...
DB<4> c
main::(deb.pl:7):       $prumer = nacti_data();
DB<4> c
Zadej prumer: 12
main::pis_vysledky(deb.pl:21):      printf "Obvod kruhu: %6.2f\n", PI()*$_[0];
DB<4> c
Obvod kruhu:  37.70
Obsah kruhu: 113.10
Debugged program terminated.  Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
DB<4>
```

Akce

Akce je příkaz Perlu, který se provede na určitém řádku. Je to v podstatě zařazení nového kódu do programu. Příkaz na vytvoření akce je ve formátu a číslo_řádku příkaz.

Akce se maže příkazem A číslo_řádku, všechny akce potom A *.

```
DB<3> a 9 print "PRUMER: $prumer";
DB<4> n
main::(deb.pl:7):       $prumer = nacti_data();
DB<4>
Zadej prumer: 54
main::(deb.pl:9):       pis_vysledky($prumer);
PRUMER: 54
DB<4>
```

Proměnné pod kontrolou

Pomocí tzv. sledovaných výrazů (watch-expressions) si je možné nechat automaticky vypisovat hodnotu proměnné, jakmile se tato proměnná změní. Nastavuje se příkazem w proměnná.

Kontrolu proměnné smažeme W proměnná, všechny už tradičně přidáním hvězdičky - W *.

```
DB<3> w $prumer
DB<4> n
main::(deb.pl:5):       my $prumer;
DB<4>
main::(deb.pl:7):       $prumer = nacti_data();
DB<4>
Zadej prumer: 21
Watchpoint 0:   $prumer changed:
    old value:  ''
    new value:  '21
'
main::nacti_data(deb.pl:20):        return $prumer;
DB<4>
```

Přehled nastavení

Abychom se v zarážkách, akcích a sledovaných výrazech neztratili, je zde příkaz L, který zobrazí všechna jejich nastavení.

```
DB<26> L
deb.pl:
6:      $prumer = nacti_data();
    break if (1)
13:         my $prumer;
    action:  $vysledek=15
23:         printf "Obvod kruhu: %6.2f\n", PI()*$_[0];
    break if ($prumer > 100)
    Watch-expressions:
    $stav
    $vysledek
DB<26>
```

Historie příkazů

Kurzorové šipky slouží k pohybu v historii příkazů.

Navíc stejně jako v shellech je tu k dispozici vykřičník. Příkaz ! n provádí *n*tý příkaz a !! opakuje poslední příkaz. Parametrem vykřičníku může být i řetězec. V takovém případě je vykonán poslední příkaz, který na řetězec začíná.

Posledních n příkazů historie vypíšeme příkazem H -n. Historie se maže příkazem H *.

Stejně jako v shellech navíc funguje tabulátorová expanze.

Externí příkaz shellu se spouští pomocí !!příkaz_shellu.

Příkaz promptu

Lze nastavit, aby se spolu s promptem vždy zároveň provedl nějaký příkaz Perlu nebo debuggeru. V případě příkazů Perlu máme 2 možnosti - příkaz se může provádět před nebo po promptu.

příkaz Perlu

Příkaz těsně před promtem se nastavuje zadáním < příkaz. Potřebujeme-li takových příkazů víc, další se přídává pomocí << příkaz. Příkazem < se zobrazí zdrojový kód všech takto nastavených příkazů.

To samé ale až po promptu se nastavuje stejně, jen za < nahradíme >.

příkaz debuggeru

Pro definici příkazu se používá příkaz {, jinak je mechanizmus analogický příkazům Perlu. V tomto případě nemá význam jestli je příkaz před nebo po manuálním zadání.

Aliasy

Pomocí příkazu = si je možné nastavit u často používaných příkazů aliasy. Vytvoříme si příkaz prumer, který bude mít stejný význam jako p "prumer=$prumer".

```
DB<25> = prumer p "prumer=$prumer"
prumer = p "prumer=$prumer"
DB<26> prumer
prumer=11

DB<27>
```

Všechny nastavené aliasy zobrazíme příkazem =.

Příkazy Perlu

Do promptu lze normálně psát i příkazy Perlu. U víceřádkových je nutné psát vždy před nový řádek zpětné lomítko.

```
DB<10> for (1..5) { \
cont: print "CYKLUS: $_\n" \
cont: }
CYKLUS: 1
CYKLUS: 2
CYKLUS: 3
CYKLUS: 4
CYKLUS: 5

DB<11>
```

Modul Devel::DProf

Nakonec ještě zkusíme spustit program tímto příkazem.

```
$ perl -d:DProf program.pl
```

V aktuálním adresáři se vytvořil soubor tmon.out. Obsahuje informace, týkající se zatížení procesoru, které je pak schopen zobrazit příkaz dprofpp. Můžeme tak zjistit například náročnost jednotlivých podprogramů. Je též možné oba příkazy spojit a volat dprofpp -u -p program.pl. Více informací například v manuálu.

V příštím dílu se podíváme na grafické debuggery, které lze použít pro ladění programů psaných v Perlu.

Dnes zabrousíme trochu jinam a podíváme se na grafické debuggery, ve kterých lze ladit perlové programy.

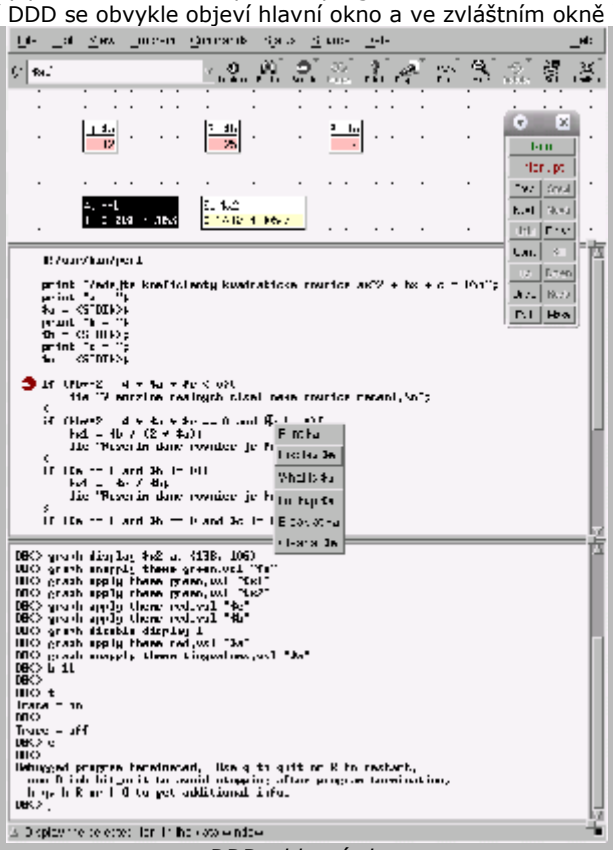
DDD - Data Display Debugger

Data Display Debugger je grafickou nadstavbou debuggerů Perlu, Pythonu, GNU Debuggeru apod. DDD je k dostání na www.gnu.org/software/ddd. Po instalaci ho spustíme příkazem

```
$ ddd --perl program.pl
```

Volba --perl většinou není potřeba, protože DDD by sám měl poznat, o který jazyk v předávaném souboru jde. Případné parametry příkazového řádku pro náš program lze uvést normálně na konec řádku.

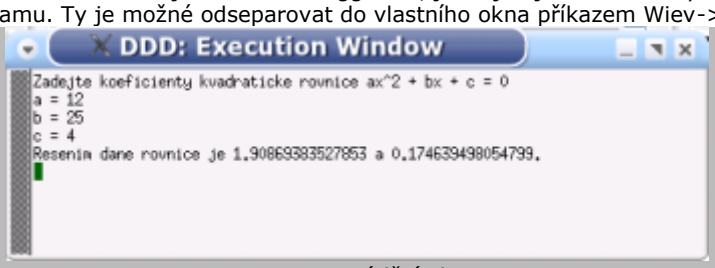
Po spuštění DDD se obvykle objeví hlavní okno a ve zvláštním okně řídící panel.



DDD - hlavní okno

Řídící panel obsahuje několik tlačítek. Pro Perl nejsou aktivní úplně všechny. Tlačítko Run nespouští program, jak bychom asi čekali, ale restartuje (analogické příkazu R v debuggeru Perlu). Příkazy Step (příkaz s) a Next (příkaz n) krokují. Cont (příkaz c) začne provádět program a zastaví se až u přerušení. Tlačítko Edit spustí grafický Vim se souborem, který ladíme.

Hlavní okno je rozděleno na tři části. Část dole je vlastně debuggerem, je stejná jako vestavěný debugger Perlu. Obsahuje mimo jiné i výstupy programu. Ty je možné odseparovat do vlastního okna příkazem Wiev->Execution Window.

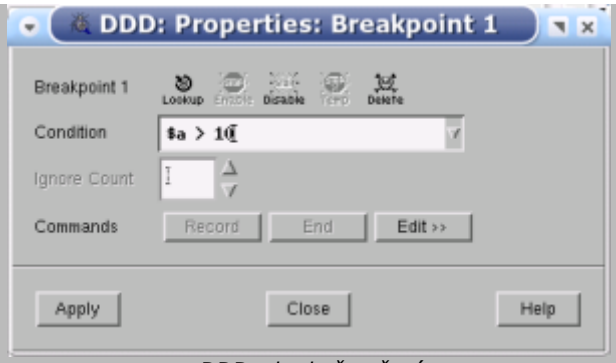


DDD - prováděcí okno

I zde v DDD je možné používat příkazy vestavěného debuggeru Perlu. Zadávají se do dolní části hlavního okna. Prostřední část okna obsahuje zdrojový kód programu s ladícími symboly. Aktuální řádek je označen zelenou šipkou.

Body přerušení

Dobře čitelná je zde značka STOP, která ukazuje místo přerušení. Tu nastavíme kliknutím levým tlačítkem myši na začátek příslušného řádku a příkazem Set Breakpoint. Kliknutím na místo, kde už je přerušení nastaveno se objeví nabídka, kde lze přerušení deaktivovat nebo smazat úplně. Dvojitým kliknutím pravého tlačítka myši na značku STOP se objeví nabídka, kde se nastavují podmínky pro přerušení.



DDD - bod přerušení

Sledované výrazy

V horní části okna máme k dispozici volně editovatelnou plochu, kde mohou být zobrazeny aktuální hodnoty proměnných, což je přehledná verze sledovaných výrazů.

Položka se dá přidat několika způsoby. První možnost je kliknout na plochu levým tlačítkem myši, zvolit New Display a zadat nějaký výraz (nejčastěji jméno proměnné, ale lze vložit cokoliv, co může mít nějakou hodnotu). Tento výraz se bude automaticky s každým příkazem vyhodnocovat a tedy se i v případě změny aktualizovat.

Další možností je kliknout levým tlačítkem na nějakou proměnnou nebo jiný výraz přímo v kódu v prostřední části a zvolit Display.

Výraz se z plochy odstraňuje příkazem Undisplay. Pokud takto odstraníme všechny, plocha zmizí úplně.

Položky na ploše se dají stylovat. V nabídce vyvolané levým tlačítkem myši zvolme Theme a vyberme si z voleb. Volbou Edit Themes se zobrazí okno, ve kterém se dají styly nastavit globálně.

DDD - styly

Výpis hodnot

Vypsání hodnoty proměnné zařídíme opět kliknutím levým tlačítkem a zvolíte příkaz Print. Mělo by to jít i pouhým nastavením kurzoru na proměnnou v kódu - v takovém případě se obsah zobrazí v bublině a stavovém řádku.

Hodnoty proměnných změníme kliknutím na položku plochy v horní části, vybereme příkaz Set Value a objeví se okno, kde se hodnota nastavuje.

Příkaz Data->Display Local Variables zobrazí všechny lokální proměnné.

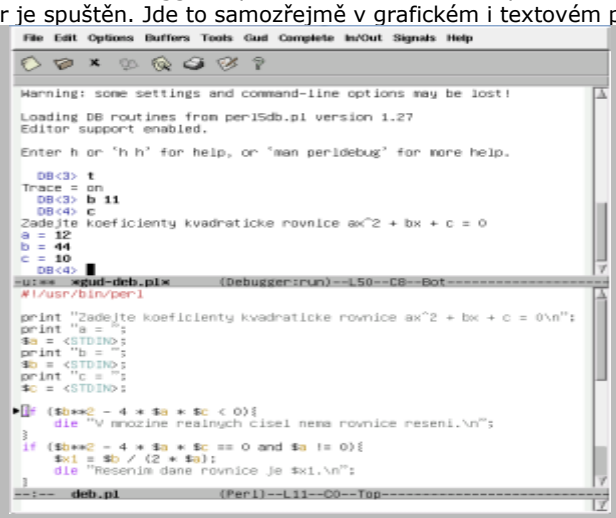
Uložení aktuálního stavu ladění

Jednou z praktických vlastností DDD je možnost přerušit práci na debuggingu a později po jejím obnovení se opět vrátit do stavu, který jsme uložili.

V nabídce File zvolme příkaz Save Session As a uložme sezení do souboru. Kdykoliv potom můžeme aktuální stav obnovit pomocí File->Open Session.

Emacs Debugger

Ještě se velice stručně podíváme na Emacs debugger. Spustíme Emacs, zadáme příkaz M-x perldb, poté perl -d program.pl a debugger je spuštěn. Jde to samozřejmě v grafickém i textovém prostředí.



Emacs debugger

Aktuální pozici ukazuje černá šipka. Ovládání je stejné jako u vestavěného debuggeru Perlu. Příjemné je zvýraznění syntaxe, což zaručuje relativně dobrou přehlednost.

Tento díl byl závěrem k možnostem ohledně ošetřování chyb. Udělali jsme mírné odbočení od tématu seriálu, nicméně příště se již budeme věnovat pouze Perlu - budeme pokračovat výkladem modulů, které jsou branou k objektově-orientovanému programování.

Perl (37) - Začínáme s moduly

Moduly umožňují rozdělovat programy do několika nezávislých souborů nebo sdílet tytéž zdrojové kódy mezi více programy

Modul

Modul je soustavou datových struktur, které souvisejí s určitým problémem. Takový modul můžeme nazvat knihovnou. Modul je kód v separátním souboru, který lze použít ve více programech. Tímto způsobem lze zdrojový kód programu rozdělit na několik logických částí.

Modul si můžeme celkem jednoduše představit. Mějme modul Matematika, ve kterém jsou definovány matematické funkce, které nejsou v základní distribuci Perlu dostupné. Importujeme ho do našeho programu a tyto funkce v něm můžeme používat, aniž bychom se jakkoliv starali o jejich algoritmus. Takže můžeme psát příkazy jako print faktorial(7);. Získali jsme novou funkci, kterou standardní verze Perlu neobsahuje.

Balíky

Každý modul potřebuje svoji sadu jmen, aby názvy proměnných nekolidovaly s jiným kódem. K tomu slouží balíky. Předtím, než budeme v modulech pokračovat, se na ně musíme podívat blíže.

Zamýšleli jste se někdy trochu hlouběji nad významem chybových hlášek? Pokud ano museli jste si všimnout, že před názvem proměnné bylo vždy ještě připsáno main::.

Name "main::promenna" used only once: possible typo at soubor line 1.

main::promenna je totiž celý nebo-li plně kvalifikovaný název proměnné. main je název balíku a promenna jméno proměnné, která je v tomto balíku definována. Oddělují se čtyřtečkou, tedy znaky ::. Je to podobné, jako když voláme do České republiky ze zahraničí a vnitrostátně. Představme si, že každý stát je balíkem s názvem telefonní předvolby. Voláme-li ze zahraničí, musíte uvést předponu 00420. Stejně tak u modulů - z jednoho balíku do jiného musíme volat s předponou (main::promenna). U proměnné, volané z téhož balíku, předpona být nemusí (stačí psát promenna).

Každý balík má svoji vlastní sadu proměnných. V balíku jsou definované proměnné nezávisle na tom, které jsou mimo něj. To znamená, že lze definovat v jediném souboru více proměnných stejného jména. (Vytočením čísla 123456789 se v České republice také dovoláme jinam než třeba v Polsku. V celosvětovém kontextu ale musí být číslo jednoznačné.)

Výchozím balíkem je vždy balík main. Všechny proměnné, které jsme zatím v seriálu definovali, patřily právě sem.

Zápis identifikátoru tedy vypadá následovně.

<typ_proměnné><název_balíku>::<název_proměnné>

Typem proměnné je myšlen znak $ pro skalární proměnné, @ pro pole, % pro hashe a prázdný řetězec pro formáty a ovladače. Všimněme si, že se uvádí už před jméno balíku. Nikoliv až před lokálním názvem, jak by se mohlo čekat.

Příkaz package

K přepínání aktivního balíku uvnitř programu se používá příkaz package, za kterým je uveden název balíku. Platnost příkazu package končí uvedením jiného packagenebo koncem bloku, ve kterém byl package použit.

Nutno podotknout, že to s příkazem package není vhodné v jednom souboru přehánět. Nejlepší a nejpřehlednější použití balíků spočívá v jejich vytvoření tak, aby každý byl v separátním souboru. Příkaz package je v tomto případě použít v každém souboru pouze jednou ihned na začátku zdrojového kódu programu.

Ukážeme si konkrétní úsek kódu, který se pokusí použití funkce packagedemonstrovat.

```
$prom = 1;
package JinyBalik;
$prom = 2;
package main;
print $prom;
```

Nejdříve je do proměnné $main::prom (main je implicitní balík) přiřazena hodnota 1, poté se přepne aktuální balík na JinyBalik (obvykle se počáteční písmeno balíku píše velkým písmenem. Výjimkou bývá výchozí balík main, který je vždy s malým m.) a do proměnné $JinyBalik::prom se přiřadí hodnota 2. Opět přepneme balík, tentokrát zpátky na main a tiskneme proměnnou $main::prom, která má hodnotu 1.

V případě, že je aktuálním balíkem balík JinyBalik, lze se dostat k hodnotě proměnné odjinud uvedením plně kvalifikovaného názvu.

```
$prom = 1;
package JinyBalik;
$prom = 2;
print $main::prom;
```

Ještě se krátce zmiňme o speciálním symbolu __PACKAGE__. Ten obsahuje vždy jméno aktuálního balíku. Lze ho použít podobně jako klasickou proměnnou. Podmínkou je, aby se nedostal do uvozovek, protože pak by nešlo o symbol a Perl by ho identifikoval jako část řetězce.

```
print __PACKAGE__;
```

Moduly v Perlu

Moduly se obvykle ukládají do samostatných souborů s příponou .pm (zkratka Perl Module). Přípona je oproti obyčejným .pl programům důležitá, protože ji předpokládá příkaz use sloužící k zavedení modulu do programu. Část názvu souboru před příponou je shodná s názvem modulu. Modul Data tak bude uložen v souboru Data.pm.

Uvedeme si první a značně nedokonalý příklad konkrétního modulu. Do souboru Zeme.pm uložíme hodnoty, které popisují fyzikální vlastnosti Země.

```
$polomer = 6378000;
$hustota = 5515;
#apod.
1;
```

Poslední řádek je návratovou hodnotou zavedení modulu a obsahuje ji každý modul. Podle této hodnoty se zachová příkaz pro načtení modulu do programu. Pokud je vrácena pravdivá hodnota, bylo načtení úspěšné.

Teď modul Zeme zavedeme do nějakého programu. K tomu slouží příkaz use. V adresáři, ve kterém je Zeme.pm vytvoříme další soubor - samotný program.

```
use Zeme;
print "Hodnota rovníkového poloměru Země je ". $polomer/1000 ." km.";
```

Nyní nám však nastal problém, který by mohl zejména v rozsáhlejších programech nadělat pořádnou neplechu. Co když budeme chtít definovat nějakou proměnnou, která má shodný název jako jiná proměnná v importovaném modulu? Zákonitě dojde ke kolizi názvů proměnných. Modulová hodnota se jednoduše přepíše. Této kolizi je třeba nějakým způsobem předcházet.
Právě z důvodu, že mechanizmus modulů sám o sobě neodděluje jmenné prostory, jsme si zmiňovali balíky. Pomocí nich si jmenné prostory vytvoříme sami.
Dělá se to tak, že v samotném modulu definujeme balík, který bude mít pro přehlednost stejné jméno jako název celého modulu.
V souboru Zeme.pm přidáme na první řádek název balíku.

```
package Zeme;
$polomer = 6378000;
$hustota = 5515;
1;
```

Prostory pro proměnné jsou nyní odděleny. Modul používá jmenný prostor Zeme:: a proměnné v programu jsou v main::. Jeden problém jsme vyřešili a v důsledku se nám objevil další. Spusťme teď samotný program. Proměnná $polomer nebude definovaná. Je to však správné - modul (a tedy i proměnnou $polomer) jsme definovali v balíku Zeme (je definováno $Zeme::polomer), ale proměnné voláme z balíku main (voláme $main::polomer). To jsou dvě různé proměnné. Musíme tedy změnit program a místo $polomer volat $Zeme::polomer.

```
use Zeme;
print "Hodnota rovníkového poloměru Země je ". $Zeme::polomer/1000 ." km.";
```

Moduly nejsou obvykle takto jednoduché (to však neznamená, že by se moduly uchovávající konstanty nepoužívaly). Ve většině modulů jsou definovány i nějaké složitější struktury než obyčejné proměnné. Na závěr si uveďme definici modulu Mat, který bude sčítat, odčítat, násobit a dělit. To znamená definici čtyř podprogramů.

```
package Mat;
use strict;

#vstup: n sčítanců
#vystup: součet
sub soucet {
my $soucet = 0;
$soucet += $_ while $_ = shift @_;
return $soucet;
}

#vstup: n činitelů
#vystup: součin
sub soucin {
my $soucin = 1;
return undef if !@_;
$soucin *= $_ while $_ = shift @_;
return $soucin;
}

#vstup: menšenec, menšitel
#vystup: rozdíl
sub rozdil {
return $_[0] - $_[1];
}

#vstup: dělenec, dělitel
#vystup: podíl
sub podil {
return undef if $_[1] == 0;
return $_[0] / $_[1];
}

1;
```

Teď můžeme modul importovat a používat v něm definované funkce. Systém je stále tentýž.

```
use Mat;
print Mat::soucin(10, 8, 6, 3);
print Mat::rozdil(7, 2);
```

Příště se podíváme na speciální modul Exporter, který otevírá dveře k dalším možnostem v importu modulů.

Perl (38) - Rozhraní modulu

Rozhraní modulu se obvykle definuje pomocí modulu Exporter. Určujeme tak, která data mají být veřejná a která soukromá.

Rozhraní modulu jsou proměnné a podprogramy, které modul poskytuje okolí. K jeho definici se běžně užívá speciální modul Exporter. Ten určuje několik proměnných se zvláštní funkcí, které budou definované uvnitř našeho modulu. Tyto proměnné mají řídící funkci a vytvářejí rozhraní, které bude modul poskytovat. Postupně si tyto proměnné představíme blíže.

Pole @EXPORT

Do tohoto pole přiřazujeme proměnné, které mají být dostupné nejen v balíku modulu (např. Mat::soucin), ale i v balíku, který je aktuální ve chvíli, kdy je modul importován (v našem případě tedy většinou main). Vše, co je v poli @EXPORT, bude importováno do tabulky symbolů souboru, který tento modul zavolal.
Použijeme modul Mat, který jsme definovali v minulém dílu. Pomocí modulu Exporter mu vytvoříme rozhraní. Zde vidíme zdrojový kód nového modulu Mat.

```
package Mat;

use Exporter;
```

```
our @ISA = qw(Exporter);
our @EXPORT = qw(&soucet &soucin);

sub soucet {...};
sub soucin {...};
sub rozdil {...};
sub podil {...};
1;
```

Pole @ISA obsahuje název rodičovské třídy. Pro tuto chvíli to není tolik podstatné a zabývat se jím budeme až v souvislosti s dědičností. Na dalším řádku se přiřazují názvy identifikátorů soucet a soucin do pole @EXPORT. Tyto identifikátory budou v programu, kam je modul importován, viditelné jak v balíku Mat, tak i v balíku main.

To znamená, že po importu modulu Mat máme v programu k dispozici proměnné Mat::soucet, Mat::soucin, Mat::rozdil, Mat::podil a navíc i main::soucet a main::soucin, které bychom bez použití Exporteru neměli.

Zde je ukázka použití rozhraní našeho modulu.

```
use Mat;
print soucin(10, 8, 6, 3);
print Mat::soucin(10, 8, 6, 3);
```

Nicméně, jak jistě někoho mohlo zarazit, pole @EXPORT znečišťuje jmenný prostor pro samotný program. Tím se sami částečně zbavujeme výhod odděleného prostoru jmen. Proto, je-li to možné, je lepší dávat přednost poli @EXPORT_OK.

Pole @EXPORT_OK

Než začneme s výkladem této struktury, je třeba uvést, že všechny identifikátory uložené v poli @EXPORT mají automaticky také vlastnosti, které mají identifikátory uložené v @EXPORT_OK (což ale neznamená, že se do @EXPORT_OK automaticky ukládají). @EXPORT_OK obsahuje identifikátory, které mohou (ale nemusí) být dostupné i z balíku, ve kterém je modul importován. Je totiž na autorovi programu, aby se rozhodl, které identifikátory chce mít importované do svého balíku (proto je vzhledem k autorovi programu používání pole @EXPORT_OK slušnější než @EXPORT).

Uživatel modulu (tedy autor programu) při importu pomocí use dá sám na vědomí, které identifikátory chce importovat. To se dělá přidáním seznamu požadovaných identifikátorů k příkazu use.

```
use Mat qw(&soucin);
```

Žádné jiné identifikátory než &soucin nebudou v daném balíku programu dostupné. Je-li &soucin v @EXPORT nebo @EXPORT_OK, bude se importovat &soucin a nic jiného. V okamžiku, kdy dáváme příkazu use tento parametr totiž jako vedlejším efektem zakazujeme import čehokoliv dalšího bez ohledu na obsah pole @EXPORT. Toto je pro uživatele modulu cesta, jak si snadno ohlídat svůj jmenný prostor.

S tím souvisí uvedení prázdného seznamu identifikátorů. Tímto říkáme příkazu use, že nechceme za žádných okolností cokoliv importovat.

```
use Mat ();
```

To, že nejsou některé proměnné nebo programy importovatelné ale vůbec neznamená, že se k nim nedá dostat! Není-li v modulu proměnná deklarovaná lokálně (pomocí my), vždy ji můžeme z programu, kam je načten modul, přečíst uvedením plně kvalifikovaného názvu.

Odtud plyne, že soukromá a veřejná data Perl nepodporuje. Jak už bylo řečeno, lze to obejít definicí lokálních proměnných. Jinou věcí jsou soukromé podprogramy. Tam je to složitější. Jedinou možností, jak je před uživatelem modulu zaručeně ukrýt, je vytvořeního lokální odkazu na anonymní podprogram. Nicméně vychovaný programátor by nezdokumentované vlastnosti modulů používat neměl.

Hash %EXPORT_TAGS

Pomocí %EXPORT_TAGS můžeme logicky třídit identifikátory. To je výhodné zejména u rozsáhlých modulů, které produkují spoustu identikátorů.

Nechť máme nějaký rozsáhlý modul Mat, ve kterém jsou definované podprogramy soucin, soucet, rozdil, podil, zaokrouhli_dolu, zaokrouhli_nahoru, zaokrouhli, sin, tan, cos, faktorial. Už na první pohled vidíme, že některé spolu souvisejí více, některé méně. První tematický celek tvoří funkce soucin, soucet, rozdil a podil; dalším jsou funkce zaokrouhli_dolu, zaokrouhli_nahoru, zaokrouhli; zbývají goniometrické funkce sin, tan, cos a nakonec ještě faktorial, který necháme nezařazený. Všechny tyto funkce dohromady tvoří další celek.

Klíčem prvku hashe %EXPORT_TAGS je vždy název skupiny a hodnotou odkaz na pole prvků, které do této skupiny patří.

```
%EXPORT_TAGS = (
zakladni_operace => [qw(&soucin &soucet &rozdil &podil)],
zaokrouhlovani => [qw(&zaokrouhli_nahoru &zaokrouhli_dolu &zaokrouhli)],
goniometrie => [qw(&sin &tan &cos)]
);
```

Identifikátory v %EXPORT_TAGS musí být současně i v proměnné @EXPORT nebo @EXPORT_OK.

Implicitně je také nastavena skupina identifikátorů :DEFAULT, která obsahuje všechny identifikátory, které jsou zároveň v @EXPORT.

Ukažme si, jak by vypadalo užití modulu s uvedeným rozhraním. Kdybychom chtěli do vlastního programu importovat skupinu zakladni_operace, funkci faktorial a funkci sin, vypadala by struktura příkazu use takto.

```
use Mat qw(:zakladni_operace &faktorial &sin);
```

Názvy identifikátorů lze uvést i jako regulární výraz. Vložíme všechny, které obsahují řetězec ou.

```
use Mat qw(/ou/);
```

Předřazením vykřičníku můžeme jednotlivé prvky seznamu u příkazu use negovat.

Skalární proměnná $VERSION

Do $VERSION v modulu přiřadíme racionální číslo, které vyjadřuje číslo verze tohoto modulu.

```
$VERSION = 5.12;
```

Při použití modulu v samotném programu lze pak uvést, jaké minimální může být číslo verze. To se uvádí za název modulu v příkazu use. Taková možnost se hodí například v případech, kdy potřebujeme nějakou funkci, která v nižší verzi importovaného modulu nebyla ještě dostupná.

```
use Mat 7.0 qw(:zakladni_operace);
```

Požadujeme minimální verzi 7.0, což ale náš modul nesplňuje, protože ho máme ve verzi 5.12. V takovém případě program přeruší činnost a objeví se chybová hláška.

```
Mat version 7 required--this is only version 5.12 at mat.pl line 1.
```

Je třeba dát pozor na to, jak se čísla verzí porovnávají. Čísla verzí jsou ve skutečnosti racionálními čísly a tedy 5.10 je menší než 5.9! V takových případech musíme verze číslovat jinak. Například 5.09 a 5.10.

Překrytí standardních funkcí

Spíše jako zajímavost si uveďme, že je v modulu možné definovat již implementovanou funkci. Stačí k tomu vytvořit si modul s novou definicí této funkce. Jako ukázku se pokusíme upravit chování funkce chomp. Hodnotu bude přímo tisknout a navíc odstraní nejen znak nového řádku z konce řetězce, ale veškeré bílé znaky na začátku nebo konci řetězce.

Definujeme modul Chomp. Do pole @EXPORT uložíme jméno funkce, kterou chceme překrýt (chomp), aby se automaticky importovala do programu.

```
package Chomp;

use Exporter;
our @ISA = qw(Exporter);
our @EXPORT = qw(chomp);

sub chomp {
my $retezec = shift;
$retezec =~ s/^\s*(.*?)\s*$/$1/;
print $retezec;
}
1;
```

Do programu už jen zavedeme vytvořený modul a použijeme funkci chomp.

```
use Chomp;
$x = "\n  abcd efa sd      \n";
chomp $x;
```

Řešení má háček. chomp změní chování jen v aktuálním balíku. To se dá řešit exportem funkce.

```
sub import {
my($balik, @symboly) = @_;
return unless @symboly; #pokud symboly nejsou, vyskočíme
$balik->export("CORE::GLOBAL", @symboly);
}
```

Přestože je původní funkce překrytá, lze se k ní dostat. Stačí použít speciální balík CORE:: - místo *funkce* volat CORE::*funkce*. V našem případě bychom zavolali standartní chomp $x; jako CORE::chomp $x;.

Kdykoliv importujeme modul pomocí use, provede se speciální procedura import s parametry, jimiž jsou název modulu, a seznam importovaných symbolů. Používá se v něm často také funkce caller, která dává informace o volajícím souboru. Vrací jméno balíku, do kterého je modul importován, jméno souboru a číslo řádku s use.

Perl (39) - Pragma

Pragmata jsou specifická kategorie modulů, které mění chování interpretu Perlu.

Moduly obvykle přidávají do programu funkce nebo proměnné. Pragma je speciální druh modulu, který nám nic nového nedává, ale upravuje stávající chování. Fungují tedy podobně jako přepínače u unixových příkazů.

Názvy pragma modulů zpravidla začínají malým písmenem, čímž se odlišují od běžných modulů.

Další odlišnost s běžnými moduly je v tom, že pragmata obvykle platí pouze lokálně. S koncem bloku končí i platnost vloženého pragma modulu. Lze je nejen zapínat, ale i vypínat. Zapínají se pomocí use *pragma*; a vypínají se příkazem no *pragma*;.

V tabulce jsou uvedeny vybrané pragmatické moduly. Pro více informací o pragmatech navštivte příslušnou část dokumentace.

Pragma	Význam
attributes	nastavuje atributy podprogramům
base	modifikuje pole @ISA (více bude řečeno v souvislosti s dědičností)
bigint	matematické operace jsou přetíženy tak, jako bychom pracovali s modulem Math::BigInt
bignum	jako bigint, ale dokáže pracovat i s Math::BigFloat
bigrat	jako bignum, ale podporuje práci se zlomky
constant	definuje konstantu, která je read only
diagnostics	dlouhé výpisy při chybách; lze použít ještě -verbose
if	na základě výsledku vyhodnocení výrazu dokáže načíst modul
integer	celočíselný režim
lib	mění obsah pole @INC
locale	definuje národní prostředí (více zde)
overload	přetěžování operací; budeme se mu věnovat po probrání objektově-orientovaného programování
re	upravuje chování regulárních výrazů (něco bylo napsáno zde)
sigtrap	zpracování signálů
strict	striktní režim
subs	deklarace podprogramů
utf8	podpora kódování utf8
vars	deklarace proměnných; dnes se nahrazuje deklarací pomocí our
warnings	zobrazuje varování; funguje podobně jako přepínač -w

Nyní si k některým pragma modulům uvedeme více detailů. Kompletní informace jsou opět v dokumentaci.

Další vlastnosti strict

Direktivu strict již běžně používáme. Zmíníme se o ní však ještě jednou, protože jsme dosud přesně nedefinovali, co vlastně zakazuje. Použití příkazu use strict;zakazuje použití potenciálně nebezpečných konstrukcí. Jedná se o tři typy konstrukcí:

1. nedeklarované proměnné - všechny proměnné, které nejsou deklarované pomocí my, our, vars, plně kvalifikované nebo importované
2. holá slova - to jsou neuvozené řetězce, nedeklarované podprogramy apod.
3. symbolické odkazy

Uvedením parametru lze docílit toho, aby byl zakázán jen určitý typ konstrukce. Příkaz use strict 'vars'; zakazuje nedeklarované proměnné, parametr 'subs'zakazuje holá slova a 'refs' symbolické odkazy. Nicméně toto separování použijeme spíše výjimečně.

Konstanty

Pragma constant, jak je již z názvu patrné, vytváří konstanty. Konstanta je symbolem s hodnotou, kterou nelze přepisovat. Definicí konstanty docílíme stejného výsledku, jakého bychom dosáhli definicí podprogramu, jež by vracel konstantní data.

```
use constant PI => 3.141592;
```

Přiřadili jsme symbolickému jménu PI hodnotu. Tato hodnota je neměnná a lze ji pouze číst. Po vytvoření konstanty s ní můžete nakládat podobně jako s obyčejnou proměnnou.

```
print PI;
$polomer = 3;
$obsah = PI * $polomer ** 2;
print $obsah;
```

Přes odkaz lze do konstant ukládat i seznamové hodnoty.

```
use constant SEZNAM => 1, 2, 3;
@pole = SEZNAM;
```

Mohlo by se zdát, že lze ukládat libovolně složité datové struktury a to pomocí odkazů. Ukládat je sice lze, ale má to háček. Konstantní totiž bude odkaz a ne hodnoty, na které ukazuje. Ty můžeme měnit.

```
use constant PLAT => {
"reditel" => 40000,
"zastupce" => 25000
};
PLAT->{"reditel"} = 30000;
print PLAT->{"reditel"};
```

Matematika vyšších čísel

Pro počítání s neomezeně dlouhými čísly lze užít pragmat bigint a bignum. bigintpřetěžuje matematické operace pomocí modulu Math::BigInt. Můžeme tak provést například následující operaci.

```
use bigint;
print 2**128;
```

Získáme výsledek 340282366920938463463374607431768211456. Kdybychom bigint nepoužili, dostaneme pouze zaokrouhlenou hodnotu 3.40282366920938e+38.

bigint a bignum mají několik voleb, které dále upřesňují jeho chování. Volba azapíná podporu zaokrouhlování. Pro nastavení zaokrouhlování na 11 platných čísel zavedeme modul tímto způsobem.

```
use bigint "a", 11;
```

Přesnost lze nastavit i na základě pozice oproti desetinné čárce. Jako parametr volbě p předáme kladné nebo záporné číslo, podle toho na jaké straně od desetinné čárky se bude zaokrouhlovat. Zkusíme zaokrouhlit číslo na 2 desetinná místa.

```
use bignum "p", -2;
print 2**128 + 0.123456789;
```

Nakonec si zmiňme ještě dvě volby. Volba t zapíná trasovací režim a volba vypisuje názvy a verze všech použitých modulů.

Matematika desetinných zlomků

Pragma bigrat je opět rozšířením bignum. Umožňuje počítat se zlomky bez převádění na desetinná čísla. Matematické operátory jsou přetíženy tak, že vracejí řetězec ve formátu zlomku.

```
use bigrat;
my $a = 1/3;
my $b = 2/5;
print $a + $b, "\n"; #vytiskne 11/15
print $a * $b, "\n"; #vytiskne 2/15
```

Celočíselná matematika

Pragma integer umožňuje pracovat s čísly výhradně jako integery. Přepnutí do celočíselného režimu provedeme tímto příkazem.

```
use integer;
```

Nyní bude desetinná část po provedení matematických operací vždy odříznuta. Pro porovnání uveďme též to samé v desetinné aritmetice.

```
use integer; print 7 / 3, "\n"; #vytiskne 2
no integer; print 7 / 3, "\n";  #vytiskne 2.33333333333333
```

Používání systémových zdrojů

Přestože pragma less zatím není implementována a lze ji čekat až ve vyšší verzi Perlu, uvedeme si pro zajímavost její funkci, neboť může leckoho zaujmout. Pomocí ní budeme moci udělat vlastní kompromis mezi mírou použití operační pamětí a časem procesoru. less říká, čeho se má používat méně.

```
use less "memory";
use less "CPU";
```

Tímto máme za sebou většinu těch nejčastějších pragmat. Některé již známe z minulých dílů, ale pár jich na nás ještě v budoucnu čeká.

Perl (40) - Dodatky k modulům

40. díl téměř dokončuje kapitolu o modulech. Hlavními tématy jsou určení cest k adresářům s moduly, speciální procedura AUTOLOAD a něco více k zavádění modulů. Z modulů nám tak zbyde už jen archiv CPAN, který si zaslouží vlastní díl.

Cesty k modulům

Použijeme-li ve svém skriptu příkaz use Modul;, Perl implicitně tento modul hledá v adresářích, které jsou uvedené v poli @INC. Obsahem @INC jsou totiž uložené cesty k modulům.

@INC je už přednastavené, ale mnohdy je třeba ho ještě ručně upravit. Typicky to nastane tehdy, pokud máme někde vlastní úložiště modulů. Je několik způsobů, jak obsah pole @INC ovlivnit.
Prvním z nich je pragma lib, pomocí níž s polem @INC můžeme manipulovat. Podívejme se na tento úsek kódu, který demostruje použití lib.

```
$, = "\n";
use lib qw(/home/instal/perl/lib);
print @INC;
```

Na první místo pole @INC byla přidána cesta /home/instal/perl/lib. Pokusíme-li se nyní načíst nějaký modul, prvním místem, kde ho bude Perl hledat bude právě /home/instal/perl/lib.
A navíc, pokud existuje adresář /home/instal/perl/lib/*architektura*/auto, bude přidán i ten. Bude mít dokonce ještě větší prioritu, protože moduly psané na míru pro konkrétní architekturu mají pochopitelně přednost před moduly obecnými. Názvem architektury je architektura vašeho stroje, tedy například i586-linux.
Poznámka: Poměrně častým problémem při editaci @INC je přidávání adresáře, ve kterém je program. Nelze napsat pouze use lib '.';, protože když pak tento program spouštíme z jiného adresáře, '.' reprezentuje právě jej a nikoliv adresář, ve kterém je umístěn program. Nelze tedy zadat napevno '.'. Problém však vyřešíme odseparováním cesty z proměnné $0 a importem v bloku BEGIN.

```
BEGIN {
  require lib;
my($dir) = $0 =~ /(.*)\//;
lib->import($dir);
}
```

Použitím klíčového slova no namísto use naopak adresář z @INC odstraníme.
Ještě předtím, než je pole @INC změněno, je vytvořena jeho kopie @lib::ORIG_INC. Vždy tak můžeme získat původní @INC.
Poznámka: Možná vás napadlo, že by cestu do pole @INC mělo jít přidat i obyčejným přidáním prvku na začátek tohoto pole. Proč to děláme tak složitě?

```
unshift(@INC, "/nova/cesta/k/modulum");
```

Tak jednoduché to bohužel není. Jak bude vysvětleno dále v tomto díle, import pomocí use se provádí už při kompilaci. Protože ale přiřazujeme cestu až za běhu, modul nalezen nebude. Řešením by teoreticky bylo použití příkazu require nebo uzavření příkazu unshift do bloku BEGIN. Ale proč to dělat, když zde máme pragmu lib...
Jsou i jiné možnosti, jak dostat do @INC další adresář. Například volba -I při spouštění programu. Přepínač -I přijímá seznam, takže jej lze uvést i vícekrát.

```
$ perl -I/home/instal/perl/lib -I/media/sources/perl5.8.8/libs prog.pl
```

Cesty k modulům lze také uchovávat v proměnné prostředí PERL5LIB, případně PERLLIB. To je výhodné zejména tehdy, pokud potřebujeme nějakou cestu přidat dlouhodobě. Stačí jen umístit do konfiguračního souboru shellu řádek podobný následujícímu.

```
export PERL5LIB=/nova/cesta/k/modulum
```

Speciální bloky
Perl rozpoznává takzvaný globální konstruktor modulu (blok BEGIN) a globální destruktor modulu (blok END). V podstatě to jsou zvláštní podprogramy s rezervovanými názvy. Lze je užít v libovolném perlovém programu.
BEGIN
Ne náhodou se bloku BEGIN říká konstruktor. Příkazy uvnitř něj jsou totiž určené k inicializaci. Kód, zapsaný uvnitř BEGIN se provede v okamžiku překladu. Typické je to pro import modulů.
Snadno se o tom můžeme přesvědčit pokusem. Dále v tomto článku poznáme další funkci na zavedení souboru - funkci require. Ta importuje soubor až za běhu programu (tedy v době, kdy už je po překladu). Vytvořme si dva programy. V prvním z nich budeme funkci require volat v bloku BEGIN a v druhém mimo něj. Dále přeložme tyto dva programy pomocí příkazu perlcc (ten převede program pouze do binární spustitelné podoby - tedy přibližně to, co udělá gcc z programu v C). Teď jsou ve stavu, kdy skončil překlad, ale ještě nezačal běh. Znamená to, že soubor s require v bloku BEGIN by měl obsahovat vkládaný soubor a tudíž by měl být větší.
END
Obsah bloku END se vykoná bezprostředně před ukončením běhu programu. Přitom vůbec nezáleží, zda byl program úspěšný. Tedy i v případě volání funkce die. Opět je zde název destruktor na místě.

```
print "Vystup programu\n";
BEGIN { print "V bloku BEGIN\n" }
END { print "V bloku END\n" }
```

Zajímavé je, že bloků jména BEGIN či END může být více v jediném souboru. Potom záleží na jejich pořadí. Bloky BEGIN jsou vykonávány od začátku souboru ke konci, bloky END naopak.

```
END   { print "V bloku END 1\n" }
BEGIN { print "V bloku BEGIN 1\n" }
END   { print "V bloku END 2\n" }
BEGIN { print "V bloku BEGIN 2\n" }
END   { print "V bloku END 3\n" }
BEGIN { print "V bloku BEGIN 3\n" }
BEGIN { print "V bloku BEGIN 4\n" }
END   { print "V bloku END 4\n" }
```

Poslední zmíněné pravidlo je patrné z výstupu tohoto kódu.

```
$ perl bloky.pl
V bloku BEGIN 1
V bloku BEGIN 2
V bloku BEGIN 3
V bloku BEGIN 4
V bloku END 4
V bloku END 3
V bloku END 2
V bloku END 1
$
```

Existují ještě další 2 speciální bloky s podobným významem. Jsou to CHECK a INIT. Tyto bloky slouží k zachycení fáze mezi překladem a během programu. Pořadí vykonávání jednotlivých částí programu je následující:

1. blok BEGIN
2. blok CHECK
3. blok INIT
4. hlavní program
5. blok END

Názorný příklad na všechny 4 bloky je v manuálové stránce perlmod(1).

Zavedení modulu

Existují všeho všudy 2 příkazy na zavedení externího souboru do programu. Jsou to use a require. Mnohé nám o nich napoví to, jaký k sobě mají vzájemně vztah. Zde jsou 3 způsoby, jakými můžeme use použít.

```
use Modul;
use Modul (seznam);
use Modul ();
```

Tyto zápisy lze přepsat tak, že nahradíme use za require. Předchozím ekvivalentní zápisy pak vypadají takto.

```
BEGIN { require Modul; import Modul; }
BEGIN { require Modul; import Modul (seznam); }
BEGIN { require Modul; }
```

Jak je ze zmíněných analogických příkazů patrné, require jsme umístili do bloku BEGIN, aby se vykonával už při překladu. Z toho nám krásně vyplynul hlavní rozdíl mezi use a require. Příkaz use zavádí modul už v době překladu. Oproti tomu require soubory importuje za běhu programu.

Je-li parametrem zaváděcích příkazů slovo bez uvozovek, hledá se stejnojmenný soubor s příponou .pm, ve kterém jsou nahrazeny znaky :: za lomítko. Retězec Math::BigFloat je podle tohoto pravidla převeden na soubor Math/BigFloat.pm. Ten je hledán v adresářích uvedených v poli @INC. Další možností je uvést parametr do uvozovek. Pak se v adresářích @INC hledá soubor, jehož název přesně bez úprav odpovídá tomuto řetězci.

Parametrem příkazů use nebo require může být i číslo verze interpretu Perlu. Program je pak spuštěn jen v případě, že požadovaná verze je starší nebo stejná než aktuální verze interpretu. Jinak se zobrazí chybová zpráva.

```
Perl v6.0.0 required--this is only v5.8.6, stopped at require.pl line 1.
```

Procedura AUTOLOAD

Pokud definujeme podprogram s názvem AUTOLOAD, spustí se tehdy, pokud voláme nedefinovanou proceduru. Název volané neexistující procedury je uvnitř podprogramu AUTOLOAD dostupný v proměnné $AUTOLOAD. Argumenty volané procedury jsou pomocí @_, tedy opět jako argumenty, předány do AUTOLOAD.

```
sub AUTOLOAD {
my(@argumenty) = @_; #obsahy prvků pole @_ jsou "a", "bc", 96
print "Procedura $AUTOLOAD neexistuje!\n";
}

blabla("a", "bc", 96);
```

Protože jsme žádný podprogram blabla nedefinovali, spustí se AUTOLOAD a výsledkem bude hláška Procedura main::blabla neexistuje!.

Pro zajímavost uveďme bez dalšího komentáře použití AUTOLOAD s funkcí system. Ta nebyla dosud v seriálu popsána, ale její funkci lze z programu intuitivně vytušit.

```
sub AUTOLOAD {
$AUTOLOAD =~ s/.*::/;
system($AUTOLOAD, @_);
}

pwd();
ls("-l", "/home");
top();
```

Příště se podíváme na archiv CPAN.

Perl (41) - CPAN

Dnes uzavřeme blok dílů zabývajících se moduly. Tématem 41. dílu je archiv modulů CPAN - jeden z hlavních důvodů popularity Perlu.

Ve standardní distribuci Perlu je již řada modulů k dispozici. Stačí jen nahlédnout do adresářů uložených v poli @INC. Přesto však v praxi budeme relativně často potřebovat i specializovanější moduly, které si budeme muset sehnat sami. Díky archivu CPAN je to velmi snadné.

Akronym CPAN pod sebou skrývá sousloví Comprehensive Perl Archive Network. Na CPANu jsou již přes 10 let shromážděny volně dostupné moduly s dokumentací a vůbec vše okolo Perlu.

Jak napovídá slovo Comprehensive, jde o obrovské množství dat. Počet modulů již překročil číslo 10000. Lze najít vše od opravdu užitečných modulů až po takové rarity jako třeba vyhledávač v koránu.

CPAN je dostupný na stovkách mirrorů po celém světě. V České republice máme zatím 4 oficiální mirrory.

- ftp://ftp.fi.muni.cz/pub/CPAN/
- ftp://ftp.mendelu.cz/perl
- http://ftp.mendelu.cz/perl
- http://archive.cpan.cz/

Každý se do vývoje modulů může sám zapojit a importovat svůj výtvor prostřednictvím PAUSE (Perl Authors Upload SErver). Určitě stojí za to CPAN prohledat skrz naskrz, protože obsahuje spoustu zajímavých informací.

V adresáři /src na CPANu jsou k dispozici distribuce Perlu.

Do adresáře /doc je umístěna dokumentace. Je zde i to, co v oficiální distribuci Perlu nenalezneme. Většina je však neudržovaná. Upozorněme na část FMTEYEWTK (název vytvořen jako akronym k Far More Than Everything You Ever Wanted To Know), jehož autorem je Tom Christiansen. Obsahuje spoustu rad a vyřešených problémů.

V adresáři /authors nalezneme adresáře autorů modulů. Nakonec zde máme adresář /modules, kde je seznam modulů. Ten obsahuje podadresáře, v nichž jsou moduly řazeny podle daných kritérií - jména modulu, kategorie (viz screenshot níže) nebo jmen jejich autorů.

Hledání modulu na CPAN

CPAN obsahuje vyhledávací stroj, který vyhledávání významně zefektivňuje. Na search.cpan.org je možnost vyhledávat podle slova nebo kategorií.

Úvodní strana vyhledávače modulů search.cpan.org

Po nalezení modulu ho jednoduše stáhneme. Teoreticky lze stáhnout i samotný kód (odkaz source v dokumentaci modulu), ale obvykle se kopíruje tar.gz soubor, ze kterého lze modul instalovat.

Detekce modulu

Ještě než ale modul stáhneme, je užitečné se přesvědčit, zda ho náhodou již v systému nemáme.

Asi nejrychlejší způsob, jak ověřit přítomnost daného modulu v adresářích z pole @INC, je zadání následujícího příkazu.

$ perl -e'print "Modul dostupny\n";' -M*JménoModulu*

Buď je vypsána hláška Modul dostupny, pak je vše v pořádku - nic nemusíme instalovat a modul můžeme normálně importovat. Anebo se zobrazí chybová hláška Can't locate Math/Bignt.pm in @INC. V tom případě modul nemáme a nezbývá než ho někde sehnat. Nejlépe v archivu CPAN.

Instalace modulu

Z archivu CPAN jsme získali soubor modul.tar.gz. Rozbalením získáme adresář.

$ tar xvzf modul.tar.gz -C $SOURCE_PATH

V něm jsou soubory distribuce a mimo jiné i soubor README, kde je postup instalace. Nejdříve vytvoříme Makefile.

$ perl Makefile.PL

A poté modul nainstalujeme.

$ make

$ make test

make install

Pokud vše proběhlo, je hotovo a nainstalovaný modul můžeme normálně použít. Spolu s modulem se k němu nainstalovala i dokumentace.

Modul CPAN a automatická instalace modulu

Jestliže máme nainstalován modul CPAN (nebo vylepšený CPANPLUS), výrazně nám to usnadní instalaci modulů. Všechny příkazy, které jsou nutné pro stáhnutí a instalaci modulu se srazí na jediný. Navíc budou automaticky řešeny závislosti.

Než přistoupíme k popisu modulu CPAN, ověřme, zda ho máme v systému.

$ perl -e1 -MCPAN

Pokud dostupný není, stáhneme ho z http://search.cpan.org/~jhi/perl-5.8.0/lib/CPAN.pm a nainstalujeme podle výše uvedeného návodu

Modul CPAN totiž poskytuje interaktivní rozhraní. Stačí zadat jediný příkaz a modul, který chceme instalovat, je stažen a nainstalován. Ke spuštění interaktivního rozhraní je třeba vytvořit skript o těchto dvou řádcích.

```
use CPAN;
shell;
```

Obvykle se tento soubor nevytváří a vše se zahrnuje do příkazu shellu. Aby to bylo ještě jednodušší, je výhodné si na něj udělat alias. Dokonce je možné, že alias máte implicitně nastaven na příkaz cpan.

$ perl -MCPAN -eshell

V interpretu, který se právě spustil můžeme zadávat příkazy. První skupinu příkazů tvoří vyhledávací příkazy.

Příkaz	Význam
a	vyhledávání v autorech
b	vyhledávání v balících
d	vyhledávání v distribucích
i	vyhledávání v autorech, balících, distribucích a modulech
m	vyhledávání v modulech

Vyhledávání může probíhat regulárně nebo ne. Pokud uvedeme jako příkaz pouze hledaný řetězec, hledá se přesná shoda. Umístěním řetězce mezi lomítka se z něj stává regulární výraz a lze tak používat speciální znaky.

```
cpan> a /CHRISTIANSEN/
Author id = TOMC
    EMAIL       tchrist@mox.perl.com
    FULLNAME    Tom Christiansen
```

```
cpan>
```

Dalším a nejdůležitějším příkazem je install. Jak název prozrazuje, stahuje a instaluje moduly. Nainstalujeme si modul Math::Complex.

```
cpan> install Math::Complex
```

Pokud se nevyskytnou žádné problémy (jako třeba práva), měl by se modul sám instalovat. Instalovat lze opět i neinteraktivně přímo přes příkaz shellu.

$ perl -MCPAN -e "install 'Math::Complex'"
Velmi zajímavý je též příkaz r v interaktivním režimu. Najde totiž vaše moduly, u kterých je již k dispozici novější verze.
cpan> r

Package namespace installed latest in CPAN file
Algorithm::Diff 1.15 1.1901 T/TY/TYEMQ/Algorithm-Diff-1.1901.zip
Apache::AuthCookie 3.06 3.08 M/MS/MSCHOUT/Apache-AuthCookie-3.08.tar.gz
...
Chcete-li tyto moduly upgradovat, zadejte tento příkaz.
$ perl -MCPAN -e "CPAN::Shell->install(CPAN::Shell->r)"
Příkaz autobundle vypíše všechny instalované moduly a zapíše je do souboru ~/.cpan/Bundle/Snapshot_RRRR_MM_HH_VV.pm. Pomocí něj lze na jiném stroji instalovat tutéž konfiguraci.
V dokumentaci na CPAN(3pm) je o modulu CPAN mnohem více. Rozhodně se vyplatí ji alespoň zběžně přelétnout
Dokumentace k modulům
Moduly jsou zdokumentovány ve formátu POD přímo v souboru s modulem. Během instalace modulu se POD konvertuje na formát manuálových stránek troff. Informace o modulu CPAN získáme jednoduše příkazem man.
$ man CPAN
Perl (42) - Argumenty příkazového řádku

V dnešním díle se naučíme zpracovávat argumenty příkazového řádku.

Perl by nebyl plnohodnotným jazykem kdyby neuměl obsloužit argumenty programů. Toto téma jsme již naťukli v souvislosti se soubory, kdy jsme zmiňovali ovladač <> (resp. <ARGV> se stejným významem). Pomocí něj jsme byly schopni číst soubory, které byly předány jako argument na příkazovém řádku.
Ovladač <>
Ovladač <> funguje tak, že si Perl nejprve ověří, zda existují soubory se jmény uvedenými jako argumenty. Pokud ano, pak tyto soubory otevře, použije jako zdroj dat a zpřístupní ho ovladačem <>.
Tato metoda je ale vhodná pouze pro úzkou množinu programů. Přesněji řečeno pro ty, které vyžadují jako argument soubor, ze kterého se bude číst. Jistým zjednodušením je i to, že všechny soubory se slijí do jednoho.
Pro určité úlohy je tedy ovladač <> vhodný, ale pro jiné nikoliv. Co kdybychom pouze chtěli zjistit velikost předaného souboru? A co kdyby vůbec nebyl předáván název souboru, ale libovolný řetězec? Na to potřebujeme obecnější metodu pro práci s argumenty.
@ARGV
Perl nabízí pro práci s argumenty příkazového řádku na nejnižší úrovni speciální pole @ARGV. Veškeré parametry předané programu jsou v tomto poli automaticky uloženy.
Další speciální proměnnou s podobným významem, tentokrát skalární, je $0. Ta obsahuje jméno programu. S regulárnímy výrazy nemá nic společného, přestože se tak na první pohled může zdát. Proměnné pro zapamatování však začínají od 1.
Ukažme si, jak to vypadá v praxi. Mějme program ll, kterému předáme jako argumenty jména souborů. U těchto souborů vytiskne příkaz ll informace o i-uzlu, podobně jako unixové ls -l.
ll ghost.png db.sql data install.py
Pokud zavoláme program tímto způsobem, bude obsah proměnných $0 a @ARGVnásledující:

Proměnná	Obsah
$0	ll
$ARGV[0]	ghost.png
$ARGV[1]	db.sql
$ARGV[2]	data
$ARGV[3]	install.py

Příklad
Napíšeme si výše zmíněný program ll. Ten bude přijímat argumenty příkazového řádku. ll bude vypisovat, jakého typu je soubor (adresář, roura, atd.), práva, počet odkazů, vlastníka (uživatel i skupina), velikost, datum, čas a jméno souboru. ll tak bude napodobovat ls -l, jen nebude vyladěn do takových detailů.
Nejprve si udělejme nějakou koncepci. Postup bude následující.
1. Získáme jména souborů předaných z příkazového řádku
2. Pro existující soubory budeme zjišťovat požadované údaje. Každý údaj zjistíme buď přímo pomocí vestavěných funkcí nebo pomocí vlastního podprogramu, který bude vestavěné funkce využívat.
3. Výsledky zformátujeme funkcí printf.

Zde máme hlavní cyklus programu, který v každé iteraci vyšetří 1 předaný existující soubor.

```
#!/usr/bin/perl
use strict;
for (@ARGV){
next unless -e $arg;
...
}
```

V každém cyklu tak zpracujeme 1 argument, který bude uložen vždy v proměnné $_.
Teď budeme postupně zjišťovat informace o i-uzlech. Za prvé zde je typ souboru. To vyřešíme podprogramem zjisti_typ_souboru. Obdobně získáme i řetězec práv. V dalším sloupci máme počet odkazů na soubor. Ten není problém zjistit, protože tuto informaci máme uloženou na 4. pozici v seznamu, který vrací funkce stat. Jméno vlastníka a skupiny dostaneme voláním getpwuid resp. getgrgid. Předposledním sloupcem je velikost, kterou získáme taktéž pomocí funkce stat. A nakonec potřebujeme datum a čas. My máme pouze počet sekund od 1.1.1970. Proto musíme opět použít podprogram.
Nyní máme alespoň teoreticky veškeré potřebné údaje. Vytiskneme je funkcí printf. Oproti příkazu ls -l použijeme pro každou hodnotu pevnou šířku sloupce.
Hlavní cyklus bude na základě těchto údajů vypadat takto.

```
for (@ARGV){
next unless -e $arg;
```

```
my @data = stat;

my $typ_souboru = ziskej_typ_souboru($_);
my $prava = ziskej_prava($data[2]);
my $odkazu = $data[3];
my $user = getpwuid $data[4];
my $group = getgrgid $data[5];
my $velikost = $data[7];
my $cas = zjisti_cas($data[9]);

printf("%1s%9s %3s %7s %7s %10d %16s %s\n", $typ_souboru, $prava,
$odkazu, $user, $group, $velikost, $cas, $arg); }
```

Zbývá nám dopsat jednotlivé podprogramy. Začneme určením typu souboru. K tomu si připomeňme speciální operátory pro soubory. Budeme rozeznávat obyčejné soubory, adresáře, symbolické odkazy, roury, sockety, blokové a znakové soubory. V tomto podprogramu nám postačí jednoduchý switch.

```
sub ziskej_typ_souboru {
my($soubor) = @_;

if    (-d $soubor){return "d";} #adresář
elsif (-l $soubor){return "l";} #symbolický odkaz
elsif (-f $soubor){return "-";} #obyčejný soubor
elsif (-p $soubor){return "p";} #roura
elsif (-b $soubor){return "b";} #blokový
elsif (-c $soubor){return "c";} #znakový
elsif (-S $soubor){return "s";} #socket
else              {return "?";} #neznamý
}
```

Další na řadě máme řetězec práv. To bude vůbec nejsložitější část programu. Ohled musíme brát i na sticky bit, set UID a set GID. A co k tomu vlastně máme k dispozici? Prakticky jen příkaz stat, pomocí kterého lze zjistit desítkový formát přístupových práv.

Vše se tedy bude odehrávat v podprogramu ziskej_prava, který obdrží jako argument desítkový zápis práv.

```
sub ziskej_prava {
my $dec_prava = shift;
...
}
```

Nejdříve musíme získat osmičkový zápis přístupových práv, se kterým se bude lépe pracovat.

```
my $oct_prava = sprintf "%o", $dec_prava & 07777;
```

Dále si celý problém rozdělíme do 3 kroků. Bude to spíše manuální práce než vymýšlení algoritmů. Zde je postup.

- Zjistíme, zda je nastaven sticky bit, set UID a set GID a podle toho nastavíme proměnné
- Získáme výsledný řetězec, do kterého ještě není zahrnut sticky bit, set UID a set GID
- Podle informací z 1. bodu změníme výsledný řetězec.

Pokud je sticky bit, set UID nebo set GID nastaven, je zároveň hodnota $dec_pravavětší než 777 a je čtyřmístná. Takže pokud lze odečíst příslušné hodnoty, uděláme to a zároveň nastavíme proměnné $sbit, $suid a $sgid. Připomeňme si, že pro set UID odečítáme 4000, pro set GID 2000 a pro sticky bit 1000.

```
if (length $oct_prava == 4){
if (($oct_prava - 4000) >= 0){$oct_prava -= 4000; $suid = 1};
if (($oct_prava - 2000) >= 0){$oct_prava -= 2000; $sgid = 1};
if (($oct_prava - 1000) >= 0){$oct_prava -= 1000; $sbit = 1};
}
```

První a nejjednodušší krok máme úspěšně za sebou, zbývají ještě 2. Protože nyní víme, že je hodnota $dec_prava vždy trojmístná, můžeme jednotlivé cifry rozdělit.

```
my($vlastnik, $skupina, $ostatni) = split "", $oct_prava;
```

Od každé z proměnných $vlastnik, $skupina a $ostatni opět budeme postupně odečítat hodnoty (právo pro čtení 4, pro zápis 2, pro spouštění 1) a zároveň přidávat práva (byla-li příslušná hodnota odečtena). V případě absence práva zapíšeme znak -.

```
for my $us ($vlastnik, $skupina, $ostatni){
if (($us - 4) >= 0){$us -= 4; $retezec .= "r";} else {$retezec .= "-";}
if (($us - 2) >= 0){$us -= 2; $retezec .= "w";} else {$retezec .= "-";}
if (($us - 1) >= 0){$us -= 1; $retezec .= "x";} else {$retezec .= "-";}
}
```

V proměnné $retezec nyní máme řetězec práv a musíme do něj zahrnout $sbit, $suid a $sgid.

Existuje-li $suid, nahradíme 3. znak řetězce za s (pokud tam už je x) nebo za S(pokud tam není). Použijeme funkci substr. Její 4. parametr - řetězec, kterým se bude (v našem případě 3. znak) nahrazovat tedy bude právě s nebo S.

```
if ($suid == 1){
substr $retezec, 2, 1, ... s nebo S ...;
}
```

Jestliže je právě na 3. pozici znak x (což zjistíme opět příkazem substr), 4. parametrem bude s. Jinými slovy - prostor pro podmínkový operátor.

```
if ($suid == 1){
substr $retezec, 2, 1, ((substr $retezec, 2, 1) eq "x") ? "s" : "S";
}
```

To samé uděláme i s set GID a sticky bit.

```
if ($sgid == 1){
substr $retezec, 5, 1, ((substr $retezec, 5, 1) eq "x") ? "s" : "S";
}
```

```
if ($sbit == 1){
substr $retezec, 8, 1, ((substr $retezec, 8, 1) eq "x") ? "t" : "T";
}
```
A na závěr vrátíme výsledný řetězec.
```
return $retezec;
```
Posledním podprogramem zjistíme datum a čas. V něm získáme všechny potřebné údaje od funkce localtime. Stačí je pouze vhodně poskládat a vrátit.
```
sub zjisti_cas {
my($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime($_[0]);
return sprintf("%4d-%02d-%02d %02d:%02d", $year + 1900, $mon, $mday,
$hour, $min)
}
```
To je celé. Oproti ls -l se sice ll v mnoha rysech významně liší (stačí porovnat výstupy po zadání adresáře), ale přesto nám posloužil jako názorná ukázka.
```
$ ./ll TEST
-rws--Sr-T   1      jv    users         0 2005-07-07 13:56 TEST
$ ls -l TEST
-rws--Sr-T  1 jv users 0 2005-08-07 13:56 TEST
$
```
Perl (43) - Přepínače

Rozpoznávání přepínačů je jednou ze základních vlastností většiny unixových příkazů. Perl má již v základní distribuci několik modulů, jež práci s přepínači významně usnadní.

Lze říci, že přepínače jsou zvláštním typem argumentů, obvykle začínající znakem -, které mění chování programu. Rozdělme si zápis přepínačů na 4 druhy.

- jednoduchý přepínač (-x)
- zkrácené - několik přepínačů je seskupeno za 1 znakem - (-xyz)
- s hodnotou (-m 1500)
- dlouhé přepínače; mohou být také s hodnotami (--file=getopt.pl)

Ze všeho nejdříve si pro lepší představu napíšeme jednoduchý skript, který pouze vypíše na každý řádek 1 argument.
```
$\ = ">\n";
print for @ARGV;
```
A zavoláme ho s různorodými parametry.
```
$ perl argv.pl -abc -d -o vysledek.txt data1 data2 data3
-abc
-d
-o
vysledek.txt
data1
data2
data3
$
```
Odtud je nutné extrahovat přepínače. Nebudeme to dělat ručně, protože nejlepší cestou, jak toho docílit, jsou moduly Getopt::*. Hlavně Getopt::Std (pro jednoduché přepínače) a Getopt::Long (pro dlouhé přepínače).

Zpracování jednoznakových přepínačů

Importem modulu Getopt::Std získáváme 2 nové funkce getopts a getopt, které nám krátké přepínače hezky roztřídí.
```
use Getopt::Std;
```
Obě funkce jsou v modulu v proměnné @EXPORT, tudíž jsou importovány do aktuálního balíku.
Při jejich používání je třeba mít na paměti, že se při volání veškeré nalezené přepínače a jejich hodnoty mažou z pole @ARGV.

getopts

První jmenovaná funkce, getopts, přijímá jako parametr speciální řetězec, složený z písmen, které mohou být přepínači. Předpokládejme, že náš program přijme 3 přepínače: -x, -y a -z. V takovém případě bude zápis funkce getopts následující.
```
getopts("xyz");
```
Nyní tato funkce podle předaných písmen nastaví příslušné proměnné ve tvaru $opt_*písmeno*. Konkrétně, pokud tedy jsou programu předány některé z přepínačů -x, -y, nebo -z, nastaví se odpovídající proměnné $opt_x, $opt_y, $opt_z na hodnotu 1.
A nyní ještě konkrétněji, tentokrát se podívejme na skript, který rozpozná a vypíše předané argumenty.
```
use Getopt::Std;
getopts("xyz");

print "X: ", $opt_x, "\n";
print "Y: ", $opt_y, "\n";
print "Z: ", $opt_z, "\n";
```
Teď si zkusme, jak bude program reagovat na přepínače. Je důležité, aby byly zadané přepínače před ostatními argumenty, protože by potom nemusely být zpracovány.
```
$ ./getopt.pl -x -z
X: 1
Y:
Z: 1
```
Zdá se, že je vše v pořádku. Zkusme zadat jiný vstup, ve kterém budou nedefinované přepínače.
```
$ ./getopt.pl -yrtg
Unknown option: r
Unknown option: t
Unknown option: g
X:
Y: 1
```

```
Z:
$
```
Z hlášky Unknown option je patrné, že nelze použít žádné jiné přepínače než ty, které jsou v programu uvedeny. Většinou nebude v našem zájmu, aby program Unknown option vypisoval. Abychom toho docílili, je nutné ošetřit během volání funkce getopts její návratovou hodnotu.

```
die "Usage: program [-xyz] file\n" unless getopts("xyz");
```

To byly volby typu true/false. Další věcí, kterou funkce getopts poskytuje, je přijetí hodnoty jednopísmenným přepínačem. Za všechny přepínače, které budou přijímat hodnotu, umístíme ještě dvojtečku.

```
getopts("x:yz:");
```

V našem případě přijímají hodnotu přepínače -x a -z. Díky tomu se do $opt_x, resp. $opt_z nepřiřadí 1, ale přímo hodnota předaná přepínači. Pojďme opět vyzkoušet, jak bude náš program na různá volání reagovat.

```
$ ./getopt.pl -zy10
X:
Y:
Z: y10
$
```

Toto volání se na 1. pohled možná chová trochu nelogicky. Ovšem na druhý pohled lze vše uspokojivě vysvětlit. Do $opt_z je přiřazen řetězec "y10" - tedy hodnota přepínače. Ostatní argumenty nejsou definovány.
Ještě zmatenější může být nadcházející zápis. Ale opět je třeba si uvědomit, že "-y10" bude hodnotou přepínače -z a -y nebude definováno.

```
$ ./getopt.pl -z -y10
```

Zde máme další příkaz s podobným způsobem volání. Liší se v tom, že 1. argument v pořadí, -y, nepřijímá hodnotu a tedy písmeno z je bráno jako název samostatného přepínače.

```
$ ./getopt.pl -yz10
X:
Y: 1
Z: 10
$
```

A na závěr poslední případ, kdy předáváme hodnoty i těm argumentům, které je nepožadují. V tomto případě bude zdánlivá hodnota argumentu brána jako vlastní přepínač.

```
$ ./perl getopt.pl -y50 -x60 -z"delsi retezec"
Unknown option: 5
Unknown option: 0
X: 60
Y: 1
Z: delsi retezec
$
```

Mezi přepínačem a hodnotou může samozřejmě být mezera. Pokud je mezera i v hodnotě přepínače, lze ji celou uzavřít do uvozovek nebo apostrofů, jak je naznačeno v posledním volání.
Funkce pro zpracování přepínačů zároveň veškeré nalezené přepínače a jejich hodnoty mažou z pole @ARGV.

přepínače ve strict režimu

Pokud jste předchozí útržky kódu náhodou zkoušeli s pragmou strict, jistě vám neuniklo, že překladač požaduje deklaraci. K tomu můžeme použít klíčové slovo ourpro deklaraci globálních proměnných.

```
our $opt_x;
our $opt_y;
our $opt_z;
```

Často však narazíme ještě na deklaraci pomocí pragmy vars.

```
use vars qw($opt_x $opt_y $opt_z);
```

getopt

getopt slouží také pro ošetřování argumentů, ale je daleko méně striktní než getopts. Jako parametr přijímá textový řetězec, který obsahuje pouze přepínače, které přijímají hodnotu. Pro každý další jednoznakový přepínač (které se pro funkci getopt vůbec neuvádějí) vytvoří getopt příslušnou proměnnou $opt_*znak*. Funkce nevypisuje žádné chyby.
Tento příkaz vyžaduje, aby byly pro přepínače -a, -b a -C přiřazeny hodnoty. Pro jakýkoliv jiný jednoznakový přepínač přiřadí do příslušné proměnné pravdivou hodnotu.

```
getopt("abC");
```

Jako názorný příklad otestujeme v cyklu pro každou proměnnou $opt_*znak* zda existuje, a pokud ano, tak vypíšeme její hodnotu.

```
use Getopt::Std;
getopt("abC");
for ("a" .. "z", "A" .. "Z", 1 .. 9){
print "\$opt_$_=".${"opt_".$_}."\n" if ${"opt_".$_};
}
```

Teď můžeme zkoušet. Program přijímá přepínače -a, -b a -C s hodnotou a všechny ostatní bez ní.

```
$ perl getopt.pl -F1x -V -C100 -a101 -b102
$opt_a=101
$opt_b=102
  $opt_x=1
$opt_C=100
  $opt_F=1
  $opt_V=1
  $opt_1=1
$
```

Poznámka - předchozí program by šel napsat mnohem přehledněji. Funkce getopt i getopts přijímají ještě další (nepovinný) parametr, jímž je odkaz na hash. Místo do $opt_znak *se pak přepínače zpracovávají do* %prepinac{"znak"}. *Mimochodem tak i odpadají případné problémy s deklaracemi proměnných.*

```
use Getopt::Std;
getopt("abC", \%prepinac);
for (keys %prepinac){
print $_."=".$prepinac{$_}."\n";
}
```

Stručně k modulu Getopt::Regex

Představíme si další modul pro zpracování přepínačů. Getopt::Regex umí podle přepínače nastavit danou proměnnou nebo vykonat podprogram. Zmiňujeme se o něm hlavně proto, že k tomu využívá regulárních výrazů, čímž dosahuje zajímavých výsledků.

Getopt::Regex nabízí jedinou funkci GetOptions, která je importována z pole @EXPORT_OK. Ta dělá veškerou práci.

Nejdříve je nutné stáhnout a nainstalovat příslušný modul.

```
$ perl -MCPAN -e'install Getopt::Regex'
```

Podívejme se na syntaxi příkazu GetOptions. Na první pohled sice vypadá složitě, ale je snadno a rychle pochopitelná.

```
use Getopt::Regex qw(GetOptions);
GetOptions(\@ARGV,[regex, odkaz, hodnota?], ...);
```

regex je regulární výraz, který specifikuje přepínač, *odkaz* je odkaz na proměnnou (bude nastavena) nebo podprogram (bude proveden) a *hodnota* nabývá hodnot true/false a určujeme jí, zda přepínač přijímá hodnotu.

Ukážeme si to prakticky.

```
use Getopt::Regex qw(GetOptions);
GetOptions(\@ARGV, ["-[f|F]", \$p, 0]);
print $p;
```

Co se vlastně děje: Pokud je programu předán argument vyhovující regulárnímu výrazu -[f|F] (tj. buď -f nebo -F), nastaví se proměnná $p na hodnotu 1. V případě, že bychom uvedli místo 0 pravdivou hodnotu, do $p by se přiřadil argument, následující tím za argumentem, který vyhověl.

Vyzkoušíme si ještě možnost vykonat v případě nalezení vyhovujícího argumentu danou funkci. Náš budoucí program bude přijímat volby ve tvaru -p*parametr=hodnota*. Odtud vždy vypíšeme řetězec *parametr = hodnota* podle získaných údajů.

Poznámka - Za -p nesmí být mezera, protože se interpretuje jako oddělovač argumentů.

```
use Getopt::Regex qw(GetOptions);
GetOptions(\@ARGV, ["-p([^=]+)=(.+)", sub {print "$1 = $2\n";}, 0]);
```

Poznámka - sub {...} je tzv. anonymní podprogram a vrací odkaz na podprogram.

Perl (44) - Dlouhé přepínače

Poslední díl týkající se získáváním a zpracováním parametrů příkazového řádku se podrobně zabývá prací s dlouhými přepínači. Zmíníme se i o tom, jak zpracovávat dlouhé a krátké přepínače najednou.

Celý mechanizmus dlouhých přepínačů, který dnes přiblížíme, zajišťuje modul Getopt::Long. Již jsme si představili modul Getopt::Regex, který zpracovávat dlouhé přepínače umí. Nicméně oba tyto moduly se hodí každý na něco trochu jiného a tak se výhodně doplňují.

Dlouhé přepínače budeme rozlišovat do kategorií zejména podle typu argumentu.

- bez hodnoty (například --encode)
- se skalární hodnotou (--file=data.txt)
- s více skalárními hodnotami (--file=2006.txt --file=2007.txt)

Modul Getopt::Long má tu vlastnost, že dokáže zpracovat i volby, které jsou uvedeny za jinými argumenty (jména souborů...). To znamená, že budou všechny volby nalezeny i v případech jako je tento.

```
$ ./program --verbose soubor1 soubor2 --extract
```

Avšak existuje znaménko dvojité pomlčky. Pomocí ní lze dát modulu Getopt::Long na vědomí, že dále už nemá prohledávat. V dalším volání programu ./program, který používá Getopt::Long, se zpracuje už jen volba --verbose.

```
$ ./program --verbose -- soubor1 soubor2 --extract
```

Getopt::Long podporuje i zápis přepínače s jednou pomlčkou. Při volání tak lze psát --verbose i -verbose.

Další vlastností modulu je, že funkce GetOptions implicitně nerozlišuje velikost písmen. Program lze volat s volbou --extract a účinek je stejný jako kdybychom předávali --ExtrACT. Implicitní chování je možné změnit uvedeným způsobem.

```
Getopt::Long::Configure("no_ignore_case");
```

Getopt::Long poskytuje již zmíněnou funkci GetOptions, jejíž syntaxe vypadá obecně takto. Do proměnných uvedených jako hodnoty prvků se ukládají data získaná na základě příslušných přepínačů.

```
GetOptions("prepinac1" => \$promenna1, "prepinac2" => \$promenna2, ...);
```

Dlouhé přepínače bez hodnoty

Nejjednodušším případem dlouhých přepínačů je přepínač bez hodnoty. Napíšeme kód, který bude přijímat volby --get a --verbose.

```
use Getopt::Long;

GetOptions(
"verbose" => \$verbose,
"get" => \$get
);

print "verbose = $verbose\n";
print "get = $get\n";
```

Do proměnných předaných funkci GetOptions se bude ukládat pravdivá nebo nepravdivá hodnota podle toho, zda byl ten který přepínač zadán.

```
$ ./getopt.pl --verbose
verbose = 1
get =
$
```

Inkrementační volby

Inkrementační volby jsou cestou, jak rozlišit, kolikkrát byla zadána tatáž volba.

Přidáním znaménka + (ve zdrojovém kódu podtrženo) na konec jména přepínače docílíme toho, že se hodnota v dané proměnné nenastaví na pravdivou hodnotu, ale inkrementuje se. A inkrementuje se tolikrát, kolikrát byla zadána.

```
use Getopt::Long;
    $verbose = 0;
GetOptions("verbose+" => \$verbose);
    print "Ukecanost = $verbose\n";
```

Zadáme-li tomuto skriptu volbu --verbose, funkce GetOptions přenastaví hodnotu proměnné $verbose na 1. Pokud zadáme --verbose --verbose, $verbose se zvětší na hodnotu 2, apod.

Negované volby

Představme si, že chceme nabídnout 2 volby pro 1 proměnnou - jednu pro výslovné true a druhou pro výslovné false. Tedy například --list a --no-list (resp. --nolist). Je zbytečné uvádět v GetOptions 2 volby, protože stačí jedna. Jestliže za volbu připíšeme vykřičník (opět podtržen), automaticky vzniká další volba s předponou no- (resp. no).

```
use Getopt::Long;
GetOptions("list!" => \$list);
    print "list = $list\n";
```

Uvedením --list se proměnná $list nastaví na 1, naopak --no-list nastavuje 0. Obsahy příslušných proměnných jsou zřejmé z výstupu programu.

```
$ ./getopt.pl
list =
$ ./getopt.pl --list
list = 1
$ ./getopt.pl --nolist
list = 0
$
```

Dlouhé přepínače se skalární hodnotou

Pro to, aby byly zpracovávány i hodnoty voleb, je nutné za volbu v příkazu GetOptionsuvést znak = (má-li být hodnota volby povinná) nebo : (má-li být nepovinná) a za ním datový typ. Ten nabývá těchto hodnot.

Hodnota	Význam
s	řetězec
f	desetinné číslo
i	klasické celé číslo
o	celé číslo; je zde navíc podpora dvojkové (0b), osmičkové (0), šestnáctkové (0x) soustavy a znamének

Uveďme si krátký kód, vyžadující celočíselnou hodnotu přepínače, pokud je tento přepínač uveden.

```
use Getopt::Long;
GetOptions("size=i" => \$size);
    print "size = $size\n";
```

Jestliže volbu nezadáme, nebude program protestovat. Varování ale vypíše tehdy, pokud zadáme volbu bez hodnoty.

```
$ ./getopt.pl --size=11
    size = 11
$ ./getopt.pl --size
Option size requires an argument
    size =
$ ./getopt.pl
    size =
$
```

Je-li použit znak :, lze za něj napsat i hodnotu, která bude dané proměnné přiřazena v případě, že bude na příkazovém řádku zadán poze přepínač bez hodnoty. Přepínači --size nastavíme jako implicitní hodnotu 12.

```
use Getopt::Long;
GetOptions("size:12" => \$size);
    print "size = $size\n";
```

Dlouhé přepínače s více hodnotami

Pokud v GetOptions zaměníme odkaz na skalár za odkaz na pole, bude vloženo tolik hodnot, kolik jich bylo zadáno.

```
use Getopt::Long;
GetOptions("file=s" => \@file);
    print "file = @file\n";
```

Při zadávání více hodnot je nutné volbu opakovat tolikrát, kolik hodnot bude.

```
$ ./getopt.pl --file=create.sql --file=insert.sql
file = create.sql insert.sql
$
```

Funkci GetOptions lze předat dokonce i hash. Poté vyžaduje páry hodnot.

```
use Getopt::Long;
GetOptions("www=s" => \%www);
    print "www = ";
print $_."=".$www{"$_"}." " for keys %www;
    print "\n";
```

V takovém případě se volá program tímto způsobem.

```
$ ./getopt.pl -www www.linuxsoft.cz="Linux Software" --www www.google.com=Google
www = www.google.com=Google www.linuxsoft.cz=Linux Software
$
```

Další možnosti funkce GetOptions

Ukládání voleb do hashe

Podobně jako u jednoznakových přepínačů lze určit hash, do kterého se mají volby souhrnně ukládat. Formát takového hashe je pak $hash{"volba"} = hodnota.

Potom se ale poněkud mění fungování funkce GetOptions. Jako první parametr je nutno zadat odkaz na hash a dalšími parametry jsou jednotlivé volby.

```
use Getopt::Long;
GetOptions("\%volby, "height=f", "width=f", "length=f");
print $_."=".$volby{$_}."\n" for keys %volby;
```

Aliasy

GetOptions umožňuje výskyt synonymních voleb. To znamená, že lze nastavit 1 proměnnou pro více voleb. Takové volby pak mají totožný význam.

K tomu je třeba vepsat do GetOptions obě možnosti a oddělit je znakem |.

```
use Getopt::Long;
GetOptions("paste|insert|ins=s" => \$insert);
print "insert = $insert\n";
```

Nyní bude proměnná $insert nastavena ve všech těchto případech volání.

```
$ ./getopt.pl --insert=zvuk.ogg
$ ./getopt.pl --paste=zvuk.ogg
$ ./getopt.pl --ins=zvuk.ogg
```

Volání podprogramů

Stejně jako odkaz na proměnnou je možné předávat odkaz na podprogram. V takovém případě se žádná proměnná nenastavuje, ale zato se v případě uvedení přepínače provede daný podprogram.

```
use Getopt::Long;
GetOptions("pozdrav" => sub {print "Ahoj!\n";});
```

Kombinace krátkých a dlouhých přepínačů

Getopt::Long umožňuje zpracovávání dlouhých i krátkých přepínačů najednou. A to dokonce tak, aby bylo možné krátké volby seskupovat.

Slouží k tomu funkce Configure, která se stará o nastaveví chování modulu. Pokud jí jako parametr předáme hodnotu bundling (svazkování), bude brát přepínače s jednou pomlčkou vždy jako jednoznakové a se 2 pomlčkami jako dlouhé.

```
use Getopt::Long qw(Configure GetOptions);
Configure("handling");
GetOptions("decode" => \$decode, "x" => \$x, "y" => \$y, "z" => \$z);

print "x: $x\n";
print "y: $y\n";
print "z: $z\n";
print "decode: $decode\n";
```

Nyní se budou zadáním voleb -xyz --decode hledat přepínače -x, -y, -z a --decode. Pokud bude ale zadáno -xyz -decode budou považovány všechny všechny volby za jednoznakové: -x, -y, -z, -d, -e, -c, -o, -d a -e. To ve většině případů nemáme v úmyslu. Proto existuje pro funkci Configure ještě hodnota handling_override, která umí taková zadání rozpoznat. Potom bude fungovat i zápis -xyz -decode podle očekávání.

Jako základ o Getopt::Long by měl tento díl seriálu stačit. Pokud vás však informace v něm stále neuspokojily, pak můžete nahlédnout do dokumentace.

Perl (45) - Odkazy

V dnešním díle otevíráme další velké téma, jímž je problematika odkazů.

Dodnes jsme si pod pojmem datová struktura vybavovali skalár, pole a hash. Odkazy jsou prostředkem, který otevírá dveře ke složitějším. Díky nim budeme moci implementovat takové struktury jako pole polí hashů. Pomocí odkazů lze pracovat s daty daleko lépe než dosud a prakticky neomezeně se v tomto směru rozšíří naše možnosti.

Přes to všechno není odkaz nic jiného než zvláštní typ skalární proměnné. Proto se píše normálně s předponou $.

Odkazy nyní rozdělíme na 2 kategorie: pevné odkazy a symbolické odkazy. V tomto a následujících dílech se budeme věnovat zejména těm pevným; symbolické zmíníme pouze letmo.

Pevné odkazy

Pevný odkaz (dále jen odkaz) má speciální obsah. Je v něm uložena adresa v paměti. Pokud hodnotu odkazu - tedy adresu - vytiskneme, uvidíme řetězec ve tvaru *DATOVÝ_TYP*(*adresa*) - konkrétně například SCALAR(0x8187a3c). Odtud je zřejmé, že Perl zde zavádí typovou kontrolu. To má za následek, že pokud budeme chtít s odkazem na skalár zacházet jako s odkazem na pole, skončí program chybou.

Odkaz na skalár

Teď už k věci. Vytvoříme 1. odkaz. Abychom tak mohli učinit, musíme vytvořit nějakou proměnnou s hodnotou.

```
$x = "HODNOTA";
```

V paměti to nyní vypadá zjednodušeně takto:

Adresa	Hodnota	Proměnná
3		
4		
5	"HODNOTA"	$x
6		
7		
8		
9		
10		
11		

A teď vytvoříme na onu proměnnou odkaz. To se dělá stejně jako přiřazení proměnné, jen se před přiřazovanou hodnotu zapíše zpětné lomítko.

```
$ref_x = \$x;
```

$ref_x je skalární proměnná typu odkaz. Situace v paměti se změnila.

Adresa	Hodnota	Proměnná
3		
4		
5	"HODNOTA"	$x
6		
7		
8		
9		
10	SCALAR(5)	$ref_x
11		

Popišme si, co údaje v tabulce znamenají. Na adrese 10 je uložena hodnota proměnné $ref_x. Tou je odkaz na jinou adresu - konkrétně na adresu 5, na které leží hodnota typu SCALAR s obsahem "HODNOTA". Lze tedy říci, že $ref_x ukazuje na adresu 5, kde je uložena hodnota "HODNOTA".

Nyní můžeme k hodnotě na adrese 5 přistupovat dvěma způsoby. Tak jako dosud, pomocí $x, nebo nově i přes odkaz $ref_x (jediné místo v paměti má 2 jména). Přístupu k hodnotě přes odkaz se říká dereferencování a uskutečňuje se uvedením dolaru mezi stávající dolar a identifikátor. Hodnotu na adrese 5 tedy můžeme tisknout jedním z těchto dvou příkazů, podle dané situace.

```
print $$ref_x;
print $x;
```

Perl si počet pevných odkazů, které ukazují na určité místo v paměti, počítá. Počet odkazů se dokáže nejen zvyšovat (vytvářením odkazů na dané místo v paměti), ale i snižovat (zánik nebo přepsání proměnných). Pokud takto klesne na 0, ztrácí se informace o umístění hodnoty. V takovém případě je hodnota zrušena pomocí automatického mechanizmu (tzv. garbage collection) a paměť se uvolní.

Stejný systém správy funguje v souborovém systému, kde je počet odkazů pro každý i-uzel také udržován. Počet odkazů je jednou z informací získaných příkazem ls -l.

Spiše jako zajímavost si uveďme, že za jistých okolností může nastat situace, kdy proměnná $x ukazuje na proměnnou $y a zároveň $y ukazuje na $x. Tomu se říká cyklický odkaz. Po skončení platnosti obou proměnných nemůže být místo v paměti uvolněno, protože čítač odkazů stále neklesl na 0. Pokud to nastane, nezpůsobí to prakticky žádné problémy. Ty nastávají až tehdy, když se tento efekt kumuluje. Zkuste si tento příklad (ale ještě předtím si vše rozdělané uložte) a sledujte paměťové zatížení například pomocí nástroje top.

```
while(1){
 my($x, $y);
  $x = \$y;
  $y = \$x;
    }
```

Odkaz na pole

Podobně se pracuje s odkazy na pole. Při práci s polem jako s celkem se syntaxe liší pouze nezbytným nahrazením úvodního dolaru za zavináč.

```
@pole = (5, 6, 7, 8);
$r_pole = \@pole;
print @$r_pole; #tiskne 5678
```

Zajímavější je to s prvky polí. Z jediného odkazu získáváme přístup k celému poli. Pouze přidáme na konec zápisu dereferencovaného odkazu index. Prvek s indexem 2 se tiskne takto.

```
print $$r_pole[2]; #tiskne 7
```

Takový zápis může být značně nejasný. Na první pohled (a často ani na druhý) není zřejmé, kde provádění začíná. Vyhodnotí se nejdříve $$r_pole nebo $r_pole[2]? Lze pouze konstatovat, že správně je $$r_pole. Dereference má vždy vyšší prioritu než index v [].

Z tohoto důvodu se zavádí ještě další a na pohled jednoznačná syntaxe. Mezi název proměnné a index nebo klíč se vloží šipka. Tyto zápisy přístupu k prvku pole jsou stejné.

```
print $$r_pole[0];
print $r_pole->[0];
```

Odkaz na hash

Pro hashe platí stejný postup jako při práci s polem. Platí to jak při práci s hashem jako celkem, tak s jednotlivými prvky.

```
%hash = ("a" => 1);  #vytvoření hashe
$r_hash = \%hash;    #vytvoření odkazu na hash
print $$r_hash{"a"}; #přístup k prvku přes odkaz na hash
print $r_hash->{"a"};#to samé s použitím šipkové notace
```

Odkaz na podprogram

Nejen na typy skalár, pole a hash jde vytvořit odkaz. Důkazem toho je možnost odkazu na podprogram. Ačkoliv je tento typ odkazu možná hůře představitelný, systém je úplně stejný, jako v předchozích příkladech. Adresu získáme pomocí operátoru zpětného lomítka, tu přiřadíme do proměnné a pomocí ní můžeme po dereferenci volat podprogram.

```
sub tisk { print "Podprogram\n"; }

$r_print = \&tisk;

&$r_print();
 &tisk();
```

Poslední 2 příkazy volaly tutéž proceduru. Podprogram tisk má nyní 2 jména.
Lze samozřejmě použít šipkovou notaci a případně i předat argumenty.

```
$r_code = \&delej_neco;
$r_code->(1, 2, 3)
```
Ale pozor - za jméno dereferencovávaného podprogramu se nesmí uvést závorky. V takovém případě by se podprogram provedl dříve, než se vytvořil odkaz. Do proměnné by se tak přiřadil odkaz na návratovou hodnotu podprogramu.

Úprava dat pomocí odkazů

Doteď jsme data jen získávali. Pokud chceme data, na které odkazy ukazují, upravovat, musíme si uvědomit, co se děje v paměti. Přestavme si, že máme proměnnou $x a odkaz $r_x. Ten ukazuje na hodnotu proměnné $x.
```
$x = "HODNOTA";
$r_x = \$x;
```
Zápis na místo, kam ukazuje odkaz, provedeme, stejně jako při čtení hodnoty, dereferencí.
```
$$r_x = "JINA HODNOTA";
```
A teď vypišme hodnoty, na které ukazují $$r_x a $x.
```
print $$r_x;#tiskne "JINA HODNOTA"
print $x;   #tiskne "JINA HODNOTA"
```
Přestože jsme upravili pouze $$r_x, novou hodnotu má i proměnná $x. Vysvětlení nabízí reprezentace v paměti. Obě proměnné ukazují na stejnou adresu paměti. Změnou hodnoty na této adrese se tedy změní všechny hodnoty získané dereferencováním proměnných, které na toto místo ukazují. Přesněji řečeno jedna hodnota, protože $$r_x je vlastně jen aliasem pro $x.

Symbolické odkazy

Pokud je zapsána proměnná tak, jako by byla pevným odkazem, ale pevným odkazem není, jde o symbolický odkaz. Máme-li proměnnou $$prom v případě pevných odkazů, samotné $prom obsahuje adresu v paměti. Symbolické odkazy se liší tím, že $prom není odkazem a místo toho obsahuje řetězec. Symbolické odkazy nejsou v pravém smyslu odkazy.
```
$delka = 186;
$prom = "delka";
print $$prom;
```
Hodnota proměnné $prom se stává jménem jiné proměnné. $delka a $$prom jsou aliasy, takže bylo vytisknuto 186.

Úplně stejně to funguje pro jiné datové typy.
```
@xx = (2, 3, 6);
$prom = "xx";
print @$prom[0];
```
Symbolické odkazy nelze v režimu strict (přesněji strict "refs") používat.

Perl (46) - Užití odkazů a anonymní data

Anonymní data jsou nezbytná pro vytváření libovolně složitých datových struktur.

Předávání seznamů podprogramům

Podprogramy v Perlu nejsou schopny sami rozlišit, co jim je předáváno. Vše se slije do pole @_ a sám programátor musí v podprogramu pole znovu rozdělit. Problém je v tom, že seznamy mají libovolný počet prvků. Nelze tak zjistit hranici mezi předanými seznamy.

Existuje jediné (pomineme-li předání počtů prvků polí nebo jiné informace jako dalších argumentů) řešení - nepředávat seznam, ale pouze adresu, kde leží v paměti. Adresa je skalární hodnotou a tak netřeba hledat hranice mezi poli.

Předávání datových struktur pomocí odkazů je také efektivnější - pole zůstává v paměti na svém místě a odpadá tak jeho kopírování. U větších datových objemů to může mít vliv.

Ukažme si konkrétní ukázku podprogramu, který přijímá 2 pole. Napíšeme program na zjištění duplicitních hodnot z předaných polí.
```
my @pole1 = (1, 3, 5, 7);
my @pole2 = (1, 2, 4, 5);
my @spolecne_hodnoty = vrat_duplicity(\@pole1, \@pole2);

print "@spolecne_hodnoty\n";

sub vrat_duplicity {
my($r_p1, $r_p2) = @_;
my @spolecne_hodnoty;

foreach my $a (@$r_p1) {
foreach my $b (@$r_p2) {
push(@spolecne_hodnoty, $a) if $a == $b;
}
}
return @spolecne_hodnoty;
}
```
Podprogram přijímá jako argumenty 2 odkazy. Ty pak dereferencuje a pracuje s nimi jako s obyčejným polem.

Stejný postup - předávání polí pomocí odkazů - lze v případě potřeby uplatnit i při vracení výsledků z procedury. Podprogram procedura vrací odkaz na pole. Jeho hodnoty vytiskneme následovně.
```
$odkaz_na_pole = procedura(...);
print @$odkaz_na_pole;
```
Datový typ, na který odkaz odkazuje

Funkce ref přijímá jako parametr skalární proměnnou. Pokud tato proměnná není odkazem, vrací ref nepravdivou hodnotu. V opačném případě vrací datový typ hodnoty, na kterou odkaz ukazuje. ref vrací vždy jednu z těchto hodnot:

Hodnota	Popis
"" *(prázdný řetězec)*	parametr funkce není odkazem
SCALAR	skalár
ARRAY	pole
HASH	hash

CODE	podprogram
GLOB	typeglob
REF	odkaz ukazuje na hodnotu, která je také odkazem
jméno_balíku	balík, se kterým je odkaz svázán (v případě objektu)

Anonymní data

Anonymní data se používají pro vytváření složitých datových struktur, kde může jediná proměnná zpřístupňovat hluboce vnořená data.

Vytváření anonymních dat je v podstatě vytváření dat, ke kterým nebudeme mít přístup přes obyčejnou proměnnou. Jinými slovy, nemáme-li k určitým datům přímý přístup, ale máme jejich adresu, jsou tato data anonymní. Přistupovat k nim lze jen přes odkaz a ne jinak. Tabulka ilustruje skalární anonymní hodnotu v paměti.

Adresa	Hodnota	Proměnná
3		
4		
5	"HODNOTA"	
6		
7		
8		
9		
10	SCALAR(5)	$r_x
11		

K hodnotě "HODNOTA" neexistuje příslušná proměnná. Existuje ale jiná proměnná, ve které je adresa této hodnoty.

Pro jednoduchost jde v tabulce pouze o anonymní skalární data. Ty se ale prakticky nepoužívají. Mnohem větší význam mají anonymní pole, hashe a občas procedury.

Anonymní pole

Na úvod vytvoříme odkaz na pole.

```
@pole = (2, 3, 4);
$r_pole = \@pole;
```

To je klasické pojmenované pole tak, jak ho známe. A teď zkusíme udělat to samé, ale záměrně opustíme rozsah platnosti proměnné @pole.

```
{
my @pole = (2, 3, 4);
our $r_pole = \@pole;
}
```

Za blokem již neplatí proměnná @pole. Přesto data zůstávají alokována, protože na ně stále ukazuje globální proměnná $r_pole a čítač odkazů je na hodnotě 1. K poli nyní nelze přistupovat přímo, ale pouze přes odkaz. Jinými slovy, pole se stalo anonymním.

Tento postup je značně neohrabaný a pro tvoření rozsáhlých datových struktur ho použít nelze. Je na něm ale hezky vidět, co to anonymní data vlastně jsou.

V praxi to funguje jinak. Perl umožňuje definovat pole od začátku již jako anonymní. Postup je stejný jako u definice klasického pole, jen se seznam hodnot píše místo kulatých do hranatých závorek.

```
$r_pole = [2, 3, 4];
```

Prefixem proměnné $r_pole je dolar, neboť je stále pouze odkazem.

Anonymní hash

Analogicky lze vytvořit anonymní hashe. Tentokrát se používají složené závorky.

```
$r_hash = {"h1" => 2, "h2" => 3, "h3" => 4};
```

Nyní se jen pro zajímavost podívejme na tuto zajímavou situaci. Jak poznáme, kdy složené závorky označují blok a kdy jde o vytvoření odkazu na hash? Někdy to může kolidovat. Jako v tomto případě:

```
sub rret { { (2, 3, 4) } }
```

Teď je otázkou, zda budou vnitřní složené závorky brány jako blok nebo jako anonymní hash. Obojí bude z procedury vracet jinou hodnotu. Správně je v tomhle případě první možnost. Lepší je ale jednoznačně určit co je co:

```
sub rret { +{ (2, 3, 4) } }# + označuje, že jde o anonymní hash
sub rret { {; (2, 3, 4) } }# středník vylučuje anonymní hash - jde o blok
```

Anonymní konstantní skalár

Další možností odkazů je vytvořit odkaz na konstantní skalární hodnotu. Ta je po vytvoření přístupná pouze pro čtení. Značí se zpětným lomítkem před výrazem.

```
$r_a = \11;
$r_b = \(3 + $$r_a);
$r_c = \"HODNOTA";

print $$r_a; #tiskne 11
print $$r_b; #tiskne 14
print $$r_c; #tiskne "HODNOTA"
```

Všechny proměnné $$r_a, $$r_b a $$r_c jsou pouze pro čtení. Jestli se je pokusíte měnit, vyskočí na vás:

Modification of a read-only value attempted

Anonymní podprogram

sub beze jména vrací odkaz na anonymní podprogram. Vyvolat ho lze předřazením znaku &.

```
$r_hello = sub { print "Hello world!\n"; };
&$r_hello();
```

Protože volání podprogramu je příkaz, píše se za ním středník.

Dnes si zadefinujeme první datové struktury skládající se z několika do sebe vnořených polí a hashů.

Vytváření složitých datových struktur

Odkaz na odkaz

Jednou z hodnot, které může vracet funkce ref, je "REF". To znamená, že odkaz může ukazovat na jiný odkaz. Ukažme si to opět na reprezentaci v paměti. Máme nějaký obyčejný odkaz

```
$x = "HODNOTA";
$r_x = \$x;
```

s takovou reprezentací v paměti:

Adresa	Hodnota	Proměnná
3		
4		
5	"HODNOTA"	$x
6		
7		
8		
9		
10	SCALAR(5)	$r_x
11		

Teď vytvoříme odkaz na proměnnou $r_x, která je také odkazem.

```
$rr_x = \$r_x;
```

$rr_x se v paměti uloží na novou adresu.

Adresa	Hodnota	Proměnná
3		
4		
5	"HODNOTA"	$x
6		
7	REF(10)	$rr_x
8		
9		
10	SCALAR(5)	$r_x
11		

$rr_x ukazuje na hodnotu proměnné $r_x, ale ta ukazuje dále na hodnotu "HODNOTA". Všechny tyto 3 proměnné ukazují postupnou dereferencí na stejnou adresu v paměti. A tak můžeme pokračovat stále dál.

Dereferenci pak provádíme postupným přidáváním dolarů. Všechny následující příkazy tisknou hodnotu na adrese 5.

```
print $x;
print $$r_x;
print $$$rr_x;
```

Při vytváření vnořených odkazů na konstantní skalár je přípustné dokonce i použití více zpětných lomítek najednou. Sice se to neužívá, ale pro zajímavost můžeme uvést alespoň ukázku.

```
$rrrr_x = \\\\"HODNOTA";
print $$$$$rrrr_x;
```

Pole odkazů na pole

Jak už víme, pole v Perlu může obsahovat pouze skalární hodnoty. Právě proto zní nadpis Pole odkazů a nikoliv Dvojrozměrné pole. Vícerozměrná pole Perl nepodporuje, ale lze je nasimulovat právě pomocí odkazů. A to až do té míry, že se dá s polem odkazů na pole pracovat prakticky stejně, "jako by se pracovalo s dvojrozměrným polem".

Vytvoření datové struktury

Pole odkazů na pole je vůbec nejjednodušší složitější datová struktura. Nejprve vytvoříme několik obyčejných polí.

```
@pole1 = (1, 2, 3);
@pole2 = (4, 5, 6);
@pole3 = (7, 8, 9, 10);
```

Odkazy na tyto pole uložíme do jiného pole. Dosáhneme toho jedním ze dvou různých zápisů. Ten první asi nikoho nepřekvapí.

```
@pole_odkazu = (\@pole1, \@pole2, \@pole3);
```

Ovšem lze toho docílit i umístěním zpětného lomítka před otevírací závorku seznamu.

```
@pole_odkazu = \(@pole1, @pole2, @pole3);
```

@pole_odkazu je právě ono pole odkazů na jiná pole. Za chvíli si ukážeme, jak se s ním pracuje.

Lepší způsob definice složitých datových struktur však nabízejí, jak víme z předchozího dílu, anonymní data. Díky nim totiž umíme vytvořit odkaz na pole bez toho, abychom toto pole měli. Použití je jednoduché - žádné proměnné @pole1, @pole2 atd. nevytváříme, ale jako prvky pole @pole_odkazu přímo předáme anonymní data.

```
@pole_odkazu = (
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9, 10]
    );
```

Toto pole je identické s posledně vytvořeným, ale zápis je názornější.

Práce s polem odkazů na pole

Nyní, když máme vytvořenou datovou strukturu, z ní můžeme získávat hodnoty. Pole, na něž ukazuje prvek pole @pole_odkazu s indexem 0, získáme a vytiskneme tímto způsobem.

```
print @{$pole_odkazu[0]};
```

Složené závorky jsou tu proto, abychom změnili prioritu vyhodnocování. Potřebujeme získat pro index vyšší prioritu než pro dereferenci. Samotné $pole_odkazu[0] obsahuje odkaz na pole (proto se musí vyhodnotit jako první). Odkaz pak dereferencujeme.

Přístup k jednotlivým prvkům získáme přidáním indexu na konec (a samozřejmě změnou zavináče za dolar).

```
print ${$pole_odkazu[0]}[2];
```

Tento ne příliš přehledný zápis používat nemusíme. Existuje totiž operátor šipky.

```
print $pole_odkazu[0]->[2];
```

Zápis má stejný význam jako ten předchozí. Ale to není vše. Můžeme jít ještě dál a šipky mezi indexy vynechat.

```
print $pole_odkazu[0][2];
```

Zde vidíme, že práce s dvojrozměrným polem z jiných jazyků a polem odkazů na pole se prakticky neliší. Proto budeme dále tyto dva pojmy pro lepší pochopitelnost textu zaměňovat.

Počítání s maticemi

Abychom si procvičili práci s datovými strukturami, ukážeme si dále několik příkladů. Typickou ukázkou pro použití pole polí jsou operace s maticemi. Matice není nic jiného, než obdélníkové dvojrozměrné pole. Rozebereme si součet a součin matic. V archivu CPAN existují moduly, které tyto operace zvládnou samy (například Math::MatrixReal), my si však napíšeme programy vlastní.

Součet matic

Implementujeme si funkci, která sečte 2 matice. Sčítané matice musejí mít stejný počet řádků a sloupců. Pro součet matic A a B platí:

$$c_{ij} = a_{ij} + b_{ij}$$

Součet matic

pro všechny dvojice ij, kde C je výsledná matice a a_{ij}, b_{ij}, resp. c_{ij} jsou odpovídající prvky matice A, B, resp. C na řádku i a sloupci j.

Sčítat matice bude procedura secti_matice. secti_matice přijme jako argument odkazy na 2 matice. V této proceduře musí být také zkontrolováno, zda je vůbec možné matice sečíst. To znamená zkontrolovat, zda mají obě matice stejný počet řádků a sloupců. Budeme předpokládat, že matice jsou obdélníkové.

Než se pustíme do řešení, poznamenejme, že v praxi bychom operace s maticemi nejspíše prováděli pomocí objektů. Jelikož čas objektů v seriálu teprve nastane, spokojíme se s neobjektovým provedením.

Pojďme tedy začít pracovat na funkci secti_matice. Z přijatých odkazů je snadné získat počet řádků. Stačí dereferencovat a skalární vyjádření získaných polí porovnat.

```
sub secti_matice {
my($r_A, $r_B) = @_;

die "Matice A a B maji jiny pocet radku!\n" if @$r_A != @$r_B;

...
}
```

Pro názorost nyní v podprogramech voláme die, ale ve výsledném programu tato volání nahradíme.

Dále musíme zkontrolovat, zda mají matice i stejný počet sloupců. Uděláme to pro nulté řádky matic. (nultý prvek obsahuje odkaz na pole - toto pole musí mít u obou matic stejný počet prvků).

```
die "Matice A a B nemaji stejny pocet sloupcu!\n" if scalar @{$r_A->[0]} != scalar @{$r_B->[0]};
```

Předpokládáme, že všechny řádky mají stejný počet sloupců.

Nyní jsou všechny podmínky splněny, takže můžeme matice sečíst. Máme počet řádků i počet sloupců. Zbývá jen napsat cyklus, ve kterém vytvoříme matici C.

```
my $radku = @$r_A;
my $sloupcu = @{$r_A->[0]};

for (my $i=0; $i<$radku; $i++){
  for(my $j; $j<$sloupcu; $j++){
  $C[$i][$j] = $A[$i][$j] + $B[$i][$j];
  }
}
```

To je podprogram secti_matice. Nyní napíšeme ukázkové tělo programu, který ho používá.

```
my @A = (
  [2, 3, 5, 2],
[7, -8, 12, 4],
  [4, 5, 0, 7]
  );

my @B = (
  [8, 7, 5, 8],
[16, -4, -10, 0],
  [0, -1, 7, 0]
  );

my @C = secti_matice(\@A, \@B);
tiskni_matici(\@C);
```

Podprogram tiskni_matici bude vypadat podobně jako secti_matice. Lišit se bude jen v tom, že odpadnou kontroly a místo sčítání se bude tisknout.

```
sub tiskni_matici {
my ($r_matice) = @_;
```

```
my $radku = @$r_matice;
my $sloupcu = @{$r_matice->[0]};

for (my $i=0; $i<$radku; $i++){
  for (my $j; $j<$sloupcu; $j++){
   printf "%8.1f ", $$r_matice[$i][$j];
   }
    print "\n";    }
   }
```

Zdrojový kód

Součin matic

Vytvoříme další podprogram pro počítání s maticemi. Tentokrát to bude o jeden cyklus složitější, protože půjde o násobení. Součin matic má význam jen tehdy, má-li matice A stejný počet sloupců jako matice B řádků. Připomeňme si opět vztah.

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \times b_{kj}$$

Součin matic

pro všechny dvojice ij, kde C je výsledná matice, a_{ij}, b_{ij}, resp. c_{ij} jsou prvky matice A, B, resp. C na řádku i a sloupci j a m je počet sloupců matice A a zároveň počet řádků matice B.

Stejně jako u součtu matic musíme zkontrolovat zda má násobení smysl. Počet sloupců matice A musí být stejný jako počet řádků matice B.

```
die "Matice A ma jiny pocet sloupcu nez matice B radku!\n" if @{$r_A->[0]} != @$r_B;
```

Pokračujme ale dále v podprogramu vynasob_matice. Teď totiž přichází na řadu hlavní část programu. Vnořeny do sebe budou 3 cykly. Každý řádek matice A (1. cyklus) bude vynásoben se všemi sloupci matice B (2. cyklus). Každé násobení řádek × sloupec bude navíc rozloženo na násobení odpovídajících si prvků (pro každý prvek se bude počítat $a_{ik} * b_{kj}$). Proto musíme použít 3. cyklus o tolika iteracích, kolik má A sloupců, resp. B řádků.

```
for (my $i=0; $i<$A_radku; $i++){
  for (my $j; $j<$B_sloupcu; $j++){
    $prvek = 0;
  for (my $k=0; $k<$A_sloupcu; $k++){
  $prvek += $r_A->[$i][$k] * $r_B->[$k][$j];
    }
  $C[$i][$j] = $prvek;
    }
  }
```

Teď jen upravit tělo programu a vstupní matice a můžeme spustit program. Podprogram tiskni_matici upravovat nemusíme.

Zdrojový kód

Hash polí

Než se pustíme do libovolně složitých datových struktur, musíme si ukázat, jak do jejich tvorby zapojit hashe. Proto se tedy ještě krátce podíváme na další datovou strukturu, kteroužto je hash polí. Takový hash obsahuje jako své hodnoty odkazy na pole. Hash bude obsahovat jednoduchou databázi knih, tříděných podle témat. Tedy každému tématu přiřadíme několik knih. Anonymní pole s názvy knih budou hodnotami hashe.

```
%knihy = (
"Perl" => ["Programování v Perlu pro pokročilé", "Perl pro zelenáče",
"Myslíme v jazyku Perl"],
"C++" => ["Mistrovství v C++", "Myslíme v C++"],
"Java" => ["Java - příručka programátora", "Začínáme programovat"],
);
```

A teď můžeme se strukturou pracovat. Pokud chceme vypsat knihy o Perlu, zapíšeme následující příkaz.

```
print "Knihy o Perlu: ", @{$knihy{"Perl"}};
```

Podobně, chceme-li přidat nějakou další knihu.

```
push(@{$knihy{"Java"}}, "Jáva - dějiny ostrova");
```

Na závěr poznamenejme, že při vytváření anonymních polí je leckdy výhodné použití ozávorkování pomocí qw. V našem případě seznamu knih to bohužel použít nelze, ale pokud by položky byly jednoslovné, mohli bychom psát zhruba toto.

```
%struktura = (
"Téma1" => [qw(polozka1 polozka2 polozka3 polozka4)],
"Téma2" => [qw(polozka1 polozka2 polozka3 polozka4)],
"Téma3" => [qw(polozka1 polozka2 polozka3 polozka4)],
);
```

Perl (48) - Libovolně složité datové struktury

Ukážeme si příklady datových struktur, které lze použít k ukládání dat se složitými vzájemnými vztahy. Popíšeme také způsob prohlížení těchto struktur vhodný pro ladění.

Dvojrozměrné struktury, kteréžto byly tématem minulého dílu, jsou mezistupněm k rozsáhlým datovým stromům. Ty poskytují způsob, kterým se lze poměrně snadno vypořádat se složitými vztahy mezi daty. Složité datové struktury mohou být vytvořeny například jako dlouhý propojený řetězec odkazů a až daleko na jeho konci jsou skalární hodnoty - tedy ta konkrétní data, kvůli kterým celá struktura existuje. Odkazy budou jsou v datové struktuře pouze proto, aby vytvořily v té změti dat nějaký systém. Protože teorii již máme v podstatě kompletně zvládnutou, ukážeme si příklad. Vytvoříme si datovou strukturu, která bude reprezentovat atlas světa. To je velice názorný příklad. Právě atlas totiž obsahuje data setříděná postupně podle řady kritérií, tedy přesně to, na co chceme ukázat.

Nejprve se musíme rozhodnout, která data budeme uchovávat. Rozhodněme se pro informace o jednotlivých státech světa. Každý stát bude zařazen do nějaké části světa (Evropa, Asie apod.). Samotné informace budou obsahovat rozlohu, počet obyvatel, jméno hlavního města a seznam jmen dalších větších měst.

Vše budeme řešit pomocí anonymních dat. Budeme mít jedinou pojmenovanou proměnnou. Pomocí ní musíme být schopni vytisknout jakoukoliv informaci, obsaženou v naší datové struktuře.
Začneme tím, že vytvoříme hash nejvyšší úrovně. Jeho prvky budou tvořeny dvojicí hodnot - klíčem bude jméno části světa a prvkem odkaz na hash, ve kterém bude seznam států.

```
%svet = (
"evropa" => { ... },
"asie" => { ... },
"jizni amerika" => { ... },
"severni amerika" => { ... },
"australie" => { ... },
"antarktida" => { ... },
"afrika" => { ... }
```

Jak už bylo řečeno, v každém z těchto anonymních hashů bude seznam států.

```
%svet = (
"evropa" => {
"albanie" => { ... },
"andorra" => { ... },
"belgie" => { ... },
"belorusko" => { ... },
"bosna" => { ... },
...
},
...
);
```

Každá hodnota prvku je opět odkazem na hash. Tento hash už bude obsahovat konkrétní informace.

```
%svet = (
"evropa" => {
...
"ceska republika" => {
"rozloha" => 79_000,
"lidi" => 10_250_000,
"hlavni mesto" => "Praha",
"dalsi mesta" => [ ... ]
},
...
},
...
);
```

Vnoření ale půjde ještě hlouběji. Do prvku s klíčem "dalsi mesta" přiřadíme odkaz na seznam měst.

```
%svet = (
"evropa" => {
"ceska republika" => {
"rozloha" => 79_000,
"lidi" => 10_250_000,
"hlavni mesto" => "Praha",
"dalsi mesta" => [
"Brno",
"Ostrava",
"Ceske Budejovice",
"Plzen"
]
},
...
},
...
);
```

A tak můžeme pokračovat dále. Ke všemu bychom stále přistupovali pomocí jediné proměnné. Otázka je, jak moc by byl další postup vhodný. Struktura by teoreticky mohla obsahovat například ještě informace o jednotlivých městech apod. Avšak je třeba se s tímto umět ve vhodnou chvíli zastavit a zamyslet se, zda by nebylo lepší řešit problém nějak jinak. Leckdo by namítl, že ani námi demonstrovanou strukturu by přes hash neřešil. V mnoha případech bude například než struktura vhodnější databáze, které si podrobně rozebereme někdy v budoucnu.
Teď již ale zabíháme někam úplně jinam. Smysl předchozího odstavce měl být ten, že rozsáhlou datovou strukturu je dobré použít, pokud je to "rozumné". To znamená, že s dobře navrženou datovou strukturou by se mělo nechat pohodlně pracovat a měla by obsahovat opravdu jen ty informace, pro které je určena. Do jisté míry také záleží na zkušenostech a vkusu programátora.
Tak či onak, metoda vytváření struktury by měla být z výše uvedeného již jasná. Nyní se krátce podívejme, jak se s takovou strukturou pracuje.
Kontinenty jsou klíči proměnné %svet. Použijeme funkci keys k jejich vytisknutí.

```
print "Seznam kontinentu: ", keys %svet;
```

Půjdeme o úroveň dále a vytiskneme seznam evropských států. Na tento seznam ukazuje proměnná $svet{"evropa"}.

```
print "Seznam evropskych statu: ", keys %{$svet{"evropa"}};
```

Teď zkusíme nějaký konkrétní údaj.

```
print "Rozloha Ceske republiky je ", $svet{"evropa"}{"ceska republika"}{"rozloha"};
```

S těmito údaji lze operovat stejně jako s jakoukoliv jinou skalární hodnotou. Poměr počtu lidí České repuliky k její rozloze získáme takto.

```
print "Pomer poctu lidi Ceske republiky k jeji rozloze je ", $svet{"evropa"}{"ceska republika"}{"lidi"}
/ $svet{"evropa"}{"ceska republika"}{"rozloha"};
```

Dále vypíšeme seznam měst České republiky. Nesmíme zapomenout, že Praha jako hlavní město je ve zvláštním prvku. Dále musíme myslet na to, že když tiskneme ostatní města, jde o seznam, a tudíž musíme použít jako předponu zavináč.

```
print "Mesta Ceske republiky: ", $svet{"evropa"}{"ceska republika"}{"hlavni mesto"},
@{$svet{"evropa"}{"ceska republika"}{"dalsi mesta"}};
```

Teď uděláme podobnou věc, ale vytiskneme hned všechna evropská města, která máme zaznamenaná. Použijeme cyklus foreach a v každé jeho iteraci přiřadíme do promněnné $_ jeden odkaz na hash obsahující informace o konkrétním evropském státě. Tento odkaz v tělě cyklu dereferencujeme. Všimněme si, jak je odkaz dereferencován. Je totiž větví celé struktury %svet. Název státu nás nezajímá, proto můžeme použít pro vytvoření pole předávaného cyklu funkci values.

```
print "Seznam evropskych mest: ";
foreach (values %{$svet{"evropa"}}){
print $_->{"hlavni mesto"};  #tiskne hlavní město
print @{$_->{"dalsi mesta"}};#tiskne další města
}
```

Poslední příklad vylepšíme a budeme tisknout i název státu, ve kterém města leží. V předchozím příkladu nás název státu nezajímal. Teď už ale ano. Protože název státu uložen jako klíč hashe, musíme změnit pole předávané cyklu foreach. values vyměníme za keys. Teď už $_ nebude přímo odkazem na hash s informacemi o konkrétním státě, ale pouze řetězec obsahující název státu. Musíme proto dereferencovat už od nejvyšší úrovně proměnné %svet.

```
print "Seznam evropskych mest podle statu: ";
foreach (keys %{$svet{"evropa"}}){
print "$_: ";                          #tiskne název státu
print $svet{"evropa"}{$_}{"hlavni mesto"};  #tiskne hlavní město
print @{$svet{"evropa"}{$_}{"dalsi mesta"}};#tiskne další města
print "\n";
}
```

Nejen získávat je třeba informace. Je nutné občas aktualizovat - přidávat, mazat nebo editovat. Občas vypukne válka a nějaké státy zanikají a jiné vznikají. Přidáme si tedy nový stát.

```
$svet{"evropa"}{"novy stat"} = {
"rozloha" => 10_000,
"lidi" => 1_000_000,
"hlavni mesto" => "mesto X",
"dalsi mesta" => [
"mesto A",
"mesto B"
]
};
```

Funkcí delete můžeme prvky datových struktur naopak mazat.

Moduly pro manipulaci s datovými strukturami

Nyní si představíme moduly Storable a Data::Dumper.

Tisk datových struktur

Modul Data::Dumper se používá k formátovanému tisku datových struktur. Obsahuje funkci Dumper, která přijímá jako parametr odkaz na datovou strukturu a tu zformátovanou tiskne na výstup.

```
use Data::Dumper;
print Dumper $odkaz;
```

V případě datové struktury podobné té, kterou jsme si dnes napsali, se vytiskne toto.

```
$ perl dumper.pl
$VAR1 = {
'evropa' => {
'ceska republika' => {
'lidi' => 10250000,
'dalsi mesta' => [
'Brno',
'Ostrava',
'Ceske Budejovice',
'Plzen'
],
'rozloha' => 79000,
'hlavni mesto' => 'Praha'
},
...
},
'asie' => {}
...
};
$
```

Zapamatujme si, že je nutné, abychom opravdu předali odkaz. Svět na tom sice nestojí, ale je dobré to mít na paměti, když nám Dumper nevypíše to, co chceme. Může to svádět psát pouze

```
print Dumper %svet;
```

Takhle se vytiskne něco trochu jiného. Dumper ve skutečnosti přijímá seznam odkazů, takže by se vytiskly struktury pro každý klíč i hodnotu. Dumper musí být volána takto.

```
print Dumper \%svet;
```

Perzistence datových struktur

Modul Storable zajišťuje perzistenci. Používá se k ukládání datových struktur do souborů a zpětně i k jejich načítání. K uložení se používá funkce store. Umožňuje uložit do souboru jednu datovou strukturu. (Pokud jich chceme uložit více, můžeme předávat odkaz na pole odkazů na datové struktury.)

```
use Storable;
store(\%odkaz, "soubor");
```
store ukládá data do souboru v binární podobě.
Pro načtení ze souboru slouží funkce retrieve. Znovuvytvoří strukturu a vrátí na ní odkaz.
```
use Storable;
$r_svet = retrieve("soubor");
```
Storable nabízí ještě funkce dclone, která klonuje (nikoliv vytváří odkaz na ni!) datovou strukturu. Přesnou kopii tak můžeme získat tímto způsobem.
```
use Storable;
$r_kopie_svet = Storable::dclone(\%svet);
```
Následujícím trikem danou strukturu hned i pojmenujeme.
```
%kopie_svet = %{ Storable::dclone(\%svet) };
```
To bylo stručné, avšak pro běžné účely dostačující seznámení s moduly Storable a Data::Dumper. Zájemci naleznou podrobnější informace v dokumentaci.

Pseudohashe

Na závěr uveďme, opět spíše pro zajímavost, něco o pseudohashích. Pseudohash je speciální datová struktura, se kterou lze pracovat jako s odkazem na pole i jako odkazem na hash. V Perlu je zavedena pouze experimentálně.
Pseudohash je pole, jehož první prvek musí být odkaz na hash. V něm jsou řetězce, které dávají do vztahu indexy pole s klíči hashe.
```
$pseudohash = [{"a" => 1, "b" => 2, "c" => 3}, "1. hodnota", "2. hodnota", "3. hodnota"];
```
Nyní lze k hodnotám přistupovat jako k hodnotám hashe nebo hodnotám pole. Tyto zápisy vytisknou stejnou hodnotu.
```
print $pseudohash->{"c"};
print $pseudohash->[3];
```

Perl (49) - Tabulky symbolů a typegloby

Kde jsou uložena jména proměnných? Co to jsou typegloby a k čemu jsou dobré?

Tabulka symbolů

O tabulce symbolů již jsme se letmo zmiňovali. V ní jsou uloženy jména proměnných, ovladače, formáty, podprogramy a odkazy na ně. Takovou tabulku symbolů má každý balík.
Mimo tabulek symbolů existují ještě lexikální prostory, které mají trochu odlišnou funkci. Tam patří lexikálně vymezené proměnné. Lexikálně (tedy pomocí my) lze deklarovat pouze proměnné; nikoliv už ovladače, formáty nebo podprogramy. Lexikální prostor se vztahuje k aktuálnímu bloku. (Z tohoto důvodu lze v jednom bloku deklarovat 2 proměnné zdánlivě stejného jména - jedna z nich patří do tabulky symbolů, druhá do lexikálního prostoru aktuálního bloku.)
Podívejme se však na strukturu tabulky symbolů. V Perlu lze pro různé datové typy použít stejné identifikátory ($a, @a, &a atd.). Všechny identifikátory jsou uloženy ve speciálním hashi. Protože nemohou být klíče v hashi duplicitní, nemohou tam vedle sebe existovat dva stejné identifikátory, lišící se jen datovým typem. Z tohoto důvodu jsou v hashi jako hodnoty prvků takzvané typegloby.
Typeglob seskupuje stejné identifikátory různých datových typů a díky němu tak může být v tabulce symbolů pro proměnné @p, %p apod. vytvořen jediný prvek.
Tabulka symbolů je tedy uložena v hashi. Ten je v každém balíku dostupný pod jménem *%jméno_balíku*::. Konkrétně - pro balík main existuje tabulka symbolů v hashi %main::. Protože je balík main implicitní, lze použít i zkrácený zápis %::. Vypišme si jako ukázku tabulku symbolů balíku main.
```
foreach (sort keys %::){
print $_, "=", $::{$_}, "\n";
}
```
Zkusme nyní ještě voláním tohoto kódu přidat nějaký nový symbol tím, že v programu použijeme novou proměnnou. Připíšeme například tento řádek.
```
$zzz;
```
V hashi se nám objevil nový prvek. Na výstupu vidíme následující řádek navíc.
```
zzz=*main::zzz
```
Takový tvar mají všechny vytisknuté řádky. Mění se jen jméno typeglobu, tedy v našem případě oba řetězce zzz.
Takto můžeme přidávat další jména. Nejen skaláry, ale i pole, hashe atd. Pro každý nový identifikátor se přidá nový řádek. To platí do doby, než přidáme identifikátor stejného jména, které v tabulce již je, neboť, jak bylo uvedeno, typeglob reprezentuje všechny datové typy dohromady.

Vlastnosti typeglobů

Zápis typeglobu vždy začíná znakem *. Již víme, že typeglob zastupuje jakýkoliv datový typ. Zde je pro shrnutí jejich seznam.

- skalární proměnnou $main::zzz
- pole @main::zzz
- hash %main::zzz
- podprogram &main::zzz
- formát main::zzz
- ovladač main::zzz

Jak se za chvíli přesvědčíme, dá se s typegloby pracovat podobně jako s jinými identifikátory a můžeme je často zaměňovat. Důležitou skutečností je, že typegloby nelze deklarovat pomocí my. Pomocí my se deklarují pouze skaláry, pole a hashe. Lexikální prostory nic jiného obsahovat nemohou.
Neméně důležité je i to, že do typeglobů lze přiřazovat odkazy. Z toho budeme dále často vycházet.
Typegloby je možné předávat jako parametr podprogramům. Uvnitř podprogramu nelze z výše zmíněného důvodu použít k lokalizaci funkci my. Je nutné typeglobu vytvořit dočasný alias pomocí local. Příklad tohoto jevu je uveden níže.

Přiřazení

Přiřazení typeglobu umožňuje hromadné kopírování odkazů datových typů stejného identifikátoru. Po přiřazení typeglobu do typeglobu mohou být data zpřístupněna i novou sadou identikátorů. Zde je krátká ukázka.
```
$zzz = "typeglob";
@zzz = ("a", "b", "c");
```

```
*zaloha = *zzz;
print $zaloha, "\n";
print @zaloha, "\n";
```
Proměnné @zaloha i @zzz nyní ukazují na stejnou hodnotu. To znamená, že když změníme jednu z těchto proměnných, obě budou ukazovat na změněnou hodnotu.
Další možností přiřazení pouze určitého datového typu. Chceme-li přiřadit odkaz na skalár a zároveň také aby se nepřiřadil odkaz na pole, napíšeme toto.
```
$x = 1;
@x = (2, 3);

*y = \$x;

print $y;#tiskne 1
print @y;#nic se netiskne - @y neexistuje!
```
Použití typeglobů
Nyní se podíváme na několik příkladů, kde se dají typegloby použít.
Pojmenování anonymnymních dat
Podívejme se na tento bezejmenný podprogram.
```
$sub = sub {
print "Ja jsem podprogram\n";
};
```
Pokud z nějakého důvodu nechceme volat podprogram příkazem
```
&$sub();
```
lze ho přes typegloby pojmenovat. Stačí přiřadit proměnnou $sub do nějakého typeglobu.
```
*podprogram = $sub;
```
Odteď lze použít i toto volání.
```
&podprogram();
```
Naprosto stejně to funguje i pro jiné datové typy. Uveďme si jen ukázku s poli.
```
$r_pole = [1, 2, 3, 4, 5];  #vytvoření anonymního pole
*pojmenovane_pole = $r_pole;#pojmenování anonymního pole
print @pojmenovane_pole;    #už lze volat vlastním jménem
```
Šablony funkcí
Pomocí typeglobů ve spolupráci se symbolickými odkazy lze v cyklu deklarovat libovolné množství podprogramů s různými jmény, která máme uloženy v poli. Protože jsou k tomu ale potřeba symbolické odkazy, musíme vypnout režim strict "refs".
```
@jmena = qw(f g h);
foreach my $funkce (@jmena) {
no strict 'refs';
*$funkce = sub {print "Volany podprogram: $funkce\nPredane parametry: @_\n\n"};
}
```
Nyní máme vytvořené podprogramy f, g, h.
```
f(10, 20);
g();
h(1, 2, 3);
```
Vytvoření aliasu pro identifikátor
Představme si, že potřebujeme vytvořit 2 stejné podprogramy různého jména. Máme několik možností. Jednou z nich je prosté přiřazení typeglobu.
```
sub podprogram {
print "@_\n";
}

*tiskni_argumenty = *podprogram;
&tiskni_argumenty(7, "X");
```
Tvoření odkazů podle speciální syntaxe
Pomocí syntaxe *identifikátor{DATOVÝ_TYP} lze získávat odkazy na příslušný datový typ.
```
@pole = (2, 3, 4);
$r_pole = *pole{ARRAY};
print @$r_pole;
```
Datový typ může být jedním z těchto řetězců.

- SCALAR
- ARRAY
- HASH
- CODE
- IO
- GLOB

Tímto způsobem můžeme předávat odkazy na ovladače podprogramům, což by si jinak člověk jen těžko představil.
```
open DATA, "ovladac.pl";

tiskni_z_ovladace(*DATA{IO});

sub tiskni_z_ovladace {
my $fh = shift;
print <$fh>;
}
```

Ještě jednou předávání ovladačů podprogramům
Pokud se vám právě uvedená metoda nezamlouvá, můžete místo ovladače předat rovnou typeglob, což je asi logičtější metoda. Jen je třeba pamatovat, že typegloby nelze vymezovat lexikálně.

```
open ZDROJ, "data" or die;

&tiskni_z_ovladace(*ZDROJ);

sub tiskni_z_ovladace {
local(*DATA) = @_;
print <DATA>;
}
```

Odkazy na ovladače
Ovladače jsou specifický datový typ. Nelze je mezi sebou přiřazovat a ani lokalizovat pomocí my nebo local. Tato omezení lze obejít pomocí typeglobů.
Pokud máme ovladač ZDROJ a chceme ho přiřadit do ovladače Z, opět to uděláme přes typegloby (možná si ještě vzpomenete, že kdysi jsme si kopírování ovladačů již ukazovali; umí to funkce open). Napíšeme do programu toto.

```
open ZDROJ, "data" or die;
*Z = *ZDROJ;
print <Z>;
```

Podobně, chceme-li ovladač lokalizovat. Použijeme typeglob.

```
local *ZDROJ;
```

Ovladače mohou být i řetězci. Možné je toto.

```
$fh = *ZDROJ;
print <$fh>;
```

Přesměrování výstupu
Vypisuje-li nějaký program data na obrazovku a my je chceme přesměrovat do souboru, přiřadíme do *STDOUT jiný typeglob.

```
open CIL, ">vystup";
*STDOUT = *CIL;
print "cokoliv";
```

Perl (50) - Uzávěry a iterátory

Dnes si představíme jednu skrytou vlastnost anonymních podprogramů a následně ji aplikujeme na generování posloupností čísel.

Uzávěr je speciální anonymní podprogram, který v době volání používá proměnné, jež už neexistují, ale existovaly v okamžiku, kdy byl tento podprogram definován.
Charakteristika uzávěrů
Ukážeme si nejprve několik příkladů, pomocí kterých pak uzávěry lépe popíšeme. Zde je první.

```
$r_s = sub {
my $h = "uzaver";
print $h;
};

&$r_s();
```

Vytisknut je řetězec "uzaver". To nikoho nemůže překvapit. Teď ale kód zmodifikujeme. Budeme vracet odkaz na podprogram, který definujeme v jiném podprogramu. Podprogram vrácený odkazem zároveň bude používat proměnnou deklarovanou v podprogramu, z něhož je odkaz na tento podprogram vrácen (a tedy v němž je tento podprogram deklarován).

```
sub f {
my $h = "uzaver";
return sub { print $h; }
}

$r_s = f();#$r_s obsahuje odkaz na podprogam vrácený podprogramem f
&$r_s();   #a zde tento odkaz dereferencujeme
```

$r_s teď obsahuje odkaz na vnořený podprogram. Dereferencujeme ho a voláme. Co ale dereferencovaný podprogram udělá? Měl by tisknout hodnotu proměnné $h. Ta ale byla platná jen v těle podprogramu f. V době jeho volání je nedefinovaná. Po ukončení podprogramu f by měli být všechny jeho lexikálně (pomocí my) deklarované proměnné zapomenuty.
Bude se tedy tisknout nedefinovaná hodnota? Je to překvapivé, ale nebude. Vytiskne se řetězec "uzaver", přestože v době volání ani v definici volaného podprogramu tento řetězec uveden není. Je pouze platný v době definice podprogramu. A právě o tom celé téma okolo uzávěrů je.
Anonymní podprogram má tu vlastnost, že si v paměti udržuje všechny proměnné, které ještě bude potřebovat. Po zavolání je pak takový anonymní podprogram schopen použít informaci, která byla platná v době definice. Právě to je případ proměné $h z příkladu.
Další důležitou skutečností je, že uzávěry fungují jen s lexikálně vymezenými proměnnými. S proměnnými deklarovanými pomocí local nebo přímo s globálními nemůže jít o uzávěr. Zkuste si v posledním úseku kódu zaměnit my za local, nebo ho přímo odstranit. Před samotným voláním změňte hodnotu tisknuté proměnné a uvidíte, že se tiskne právě tato hodnota.
Užití uzávěrů - iterátory
Anonymní podprogram vracený formou odkazu jiným podprogramem může být volán hned několikrát. Dále je možné každým voláním hodnotu pamatované proměnné měnit. To je podstata iterátoru.
Pomocí iterátorů lze tvořit například posloupnosti čísel.
Posloupnost 2^n
Vytvoříme si vzestupnou řadu čísel složenou z čísel 2^n. Musíme zařídit, aby anonymní podprogram během každého volání

- vytiskl aktuální hodnotu pamatované proměnné
- a zároveň pamatovanou hodnotu změnil na hodnotu dvojnásobnou

Takto bude vypadat samotný generátor:

```
sub generuj_posloupnost_2_na_ntou {
```

```
my($y) = 1;      #1. pamatovanou hodnotou bude hodnota 1.
#Každým voláním anonymního podprogramu se bude měnit.
return sub {
print "$y\n";#tiskne aktuální pamatovanou hodnotu
$y *= 2;     #násobí pamatovanou hodnotu dvěma

}
}
```

Nyní zavoláme podprogram generuj_posloupnost_2_na_ntou. Tím získáme odkaz na podprogram.

```
$r_s = generuj_posloupnost_2_na_ntou;
```

Podprogram generuj_posloupnost_2_na_ntou už dále používán nebude. Dále už budeme volat jen získaný podprogram.

```
&$r_s(); #tiskne 1
&$r_s(); #tiskne 2
&$r_s(); #tiskne 4
&$r_s(); #tiskne 8
```

A tato volání můžeme třeba zacyklit.

```
for($i=0; $i<40; $i++){
&$r_s();
}
```

Posloupnost x^n

Nyní příklad trochu modifikujeme. Místo řady 2^n budeme generovat obecnější posloupnost x^n, přičemž $x bude zadáno. Podprogram generuj_posloupnost_x_na_ntou se bude lišit hned v několika ohledech. Především - bude nutné si pamatovat 2 hodnoty - $n a $x. Jednotlivá volání budou měnit hodnotu $n, ale nikoliv hodnotu $x. Hodnota $x bude zas přijata jako argument. Generátor bude vypadat takto:

```
sub generuj_posloupnost_x_na_ntou {
my($x) = shift;
my $y = 1;
return sub {
print "$y\n";
$y *= $x;
}
}
```

A teď jsme v nové situaci. Můžeme vytvořit hned několik iterátorů!

```
$r_s1 = generuj_posloupnost_x_na_ntou(2);
$r_s2 = generuj_posloupnost_x_na_ntou(5);
```

Teď lze volat podprogramy &r_s1 a &r_s2. Každý má svůj vlastní generátor - a tedy svoji vlastní hodnotu pamatovaných proměnných $x a $y.

```
&$r_s2(); #     tiskne 1
&$r_s2(); #     tiskne 5
&$r_s1(); #tiskne 1
&$r_s1(); #tiskne 2
&$r_s1(); #tiskne 4
&$r_s2(); #     tiskne 25
```

Sdílení dat více iterátory

Dále je možné, aby více iterátorů sdílelo stejné pamatované proměnné. Je to vlastnost neobvyklá a stojí za to se nad ní zamyslet. Jde o to, že Perl umožňuje z jednoho generátoru vrátit seznam odkazů na anonymní podprogramy. Uveďme si opět příklad.

```
sub generuj_posloupnost {
my $a = shift;

return (
sub {
$a++;       #přičítá k pamatované hodnotě 1
print "$a\n"; #tiskne aktuální pamatovanou hodnotu
},
sub {
$a--;       #odečítá od aktuální hodnoty 1
print "$a\n"; #tiskne aktuální pamatovanou hodnotu
},
sub {
print "$a\n"; #tiskne aktuální pamatovanou hodnotu
}
);
}

($r_nahoru, $r_dolu, $r_tiskni) = generuj_posloupnost(0);

&$r_tiskni; #tiskne 0
&$r_nahoru;     #tiskne 1
&$r_nahoru;     #tiskne 2
&$r_nahoru;     #tiskne 3
&$r_dolu;   #tiskne 2
&$r_dolu;   #tiskne 1
```

Nyní přistupují podprogramy &$r_nahoru, &$r_dolu i &$r_tiskni ke stejné proměnné.

Tímto dílem začíná blok několika článků, které se budou věnovat vztahu s operačním systémem. První z nich má čtenáře seznámit s možnostmi při zpracovávání signálů.

Čas od času nastane situace, kdy operační systém potřebuje programu předat nějaký příkaz. Takovému příkazu budeme říkat signál. V následujících odstavcích se budeme zabývat otázkami jak na ně reagovat a jak je vyvolávat.

Typickým příkladem vyvolání signálu je přerušení programu stisknutím Ctrl-c. V takovém případě je programu předán signál INT (zkratka od interrupt) a ten na to nějakým způsobem zareaguje. Obvykle se program ukončí, ale lze to ovlivnit.

Reakce na signály

Jak již víme, Perl umožňuje na signály vlastním způsobem reagovat. Funguje to tak, že se určí podprogram, který se má při obdržení signálu provést.

Odkazy na tyto podprogramy jsou obsaženy ve speciální proměnné %SIG. Klíči tohoto hashe jsou jména signálů a hodnotami odkazy na podprogramy, které mají být v případě obdržení odpovídajícího signálu volány.

Pro demonstrační účely si vytvoříme program, který zareaguje na INT signál. Měli bychom vymyslet takový program, který hned neskončí - aby bylo možné stisknutí Ctrl-C (nebo jiný způsob vyvolání) stihnout. To znamená použít například čtení ze zdroje dat. Takovým vhodným programem může být toto.

```
<> while 1;
```

Nyní v tomto programu vytvoříme akci pro signál INT - přiřadíme odkaz na podprogram do prvku hashe %SIG s příslušným klíčem.

```
$SIG{"INT"} = \&ctrl_c;
```

A nakonec musíme definovat proceduru ctrl_c. Celý program teď vypadá takto:

```
$SIG{"INT"} = \&ctrl_c;
<> while 1;
sub ctrl_c {
print "Zachyceno Ctrl-c!\n";
}
```

Samozřejmě, že lze do prvků hashe %SIG přiřadit přímo anonymní podprogram vytvořený pomocí sub.

```
$SIG{"INT"} = sub {print "Zachyceno Ctrl-c!\n";};
```

Dodejme ještě, že řádek

```
$SIG{"INT"} = \&ctrl_c;
```

by se měl nacházet na začátku programu. Signály lze totiž zachytávat až od okamžiku, kdy je proměnná %SIG nastavena.

Nyní ale už zkusme spustit posledně vytvořený program. Tučně je zvýrazněn vstup včetně neviditelných kláves.

```
$ perl signal.pl [ENTER]
nejaky vstup[ENTER]
dalsi vstup[ENTER]
[CTRL-C]Zachyceno Ctrl-c!
[CTRL-C]Zachyceno Ctrl-c!
[CTRL-C]Zachyceno Ctrl-c!
[ENTER]
[CTRL-C]Zachyceno Ctrl-c!
[ENTER]
[CTRL-C]Zachyceno Ctrl-c!
[ENTER]
[ENTER]
...
```

Program touto cestou nelze ukončit. Aby to šlo, museli bychom jinak napsat podprogram ctrl_c - tedy použít funkci die.

Jména signálů jsou vzestupně uloženy v proměnné $Config{"sig_name"}, která je exportovaná ze standartního modulu Config. Všechny dostupné signály i s příslušnými čísly tak vypíše tento program.

```
use Config;
@signaly = split(" ", $Config{"sig_name"});
for ($i=0; $i<@signaly; $i++){
print "Signal $i: ", $signaly[$i], "\n";
}
```

Do prvků hashe %SIG lze přiřadit místo odkazů na podprogramy i následující speciální řetězce.

Řetězec	Reakce na vybraný signál
"IGNORE"	obdržený signál se ignoruje
"DEFAULT"	nastavuje zpět implicitní reakci na signál

Na závěr poznamenejme, že některé signály zachytit nelze (KILL, STOP).

Vyvolávání signálů

Funkce kill posílá signál danému procesu. kill přijímá 2 parametry - číslo nebo jméno signálu a seznam procesů, kterým se tento signál má poslat.

Podívejme se na několik ukázkových příkazů. Protože máme k dispozici proměnnou $$, která uchovává ID procesu, se kterým program běží, názorným příkladem může být sebevražda programu. Pošleme našemu programu signál KILL.

```
kill "KILL", $$;
```

Jak je patrné z výstupu programu, který nám vypsal seznam signálů, KILL má číslo 9. Tudíž stejný smysl bude mít tento příkaz.

```
kill 9, $$;
```

Často je u příkazu kill vidět, že se čárka nahrazuje šipkou =>, která má stejný význam.

```
kill "KILL" => $$;
```

Funkce getppid vrací PID rodičovského procesu. Tímto způsobem zabijeme shell.

```
kill "KILL" => getppid;
```

Upozorněme také návratovou hodnotu funkce kill. V případě, že se signál podařilo poslat, vrací funkce pravdivou hodnotu, v opačném případě nepravdivou. Příčinou toho může být například to, že nemáme dostatečná práva nebo prostě proces neexistuje - důvod pak už snadno zjistíme z proměnné $!. Dále v této souvislosti ještě zmiňme signál ZERO, který nedělá nic. Právě pomocí něj lze snadno testovat, zda nějaký proces existuje.

```
print "proces vypadá mrtvě ($!)" unless kill "ZERO" => 5032;
```
Pragma sigtrap

Modul sigtrap nabízí speciální rozhraní pro práci se signály. Zavádí se tímto způsobem.

```
use sigtrap qw(ovladač seznam_signálů);
```
Vše je o tom, že na signály ze seznamu signálů je aplikován ovladač, který určuje, jak na ně zareagovat. Proto jen stručně. Jako ovladač může být uveden jeden z následujících. V posledním případě je *vlastní_ovladač* cokoliv, co by šlo přiřadit jako hodnota do prvku hashe %SIG.

- stack-trace (implicitní hodnota) - vypíše zprávu na STDERR
- die - ukončí se běh programu
- handler *vlastní_ovladač* - vykoná činnost zadanou uživatelem

Seznam signálů je prostě seznam signálů. Lze ale používat i některé speciální hodnoty:

- normal-signals - INT, HUP, PIPE, TERM
- error-signals - ABRT, BUS, EMT, FPE, ILL, QUIT, SEGV, SYS, TRAP
- old-interface-signals - ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT, SEGV, SYS, TERM, TRAP

Není-li uveden seznam signálů, je automaticky použito old-interface-signals.

Budou následovat příklady programů, které zachytávají různé signály. K testovacím účelům bude vhodné programům posílat signály příkazem kill -*SIGNÁL číslo_procesu*. Například

```
$ kill -INT 10538
```
K zjištění PID programu bude nejjednodušší přímo do programu připsat tento řádek.

```
print "Číslo procesu: $$\n";
```
Program s následujícím řádkem reaguje na signály INT a TERM tak, že se ukončí.

```
use sigtrap qw(die INT TERM);
```
To samé, jen pro signály normal-signals - tedy INT, HUP, PIPE, TERM, platí pro tento řádek.

```
use sigtrap qw(die normal-signals);
```
Nakonec zachytíme signály INT, HUP, PIPE, TERM a to tak, že při obdržení některého z těchto signálů bude vypsán jeho název pomocí námi napsaného podprogramu.

```
use sigtrap "handler", \&signal, "normal-signals";

print "Číslo procesu: $$\n";#abychom věděli, kam posílat zkušební signály
<> while 1;

sub signal {
my($signal) = @_;
print "Zachycen SIG$signal!\n";
}
```
Z příkladů by mělo být jasné, jak zachytávání pomocí sigtrap funguje.

Perl (52) - Externí příkazy

Tentokrát o spouštění shellových příkazů uvnitř programu.

Při psaní programů nastanou situace, kdy potřebujeme implementovat něco, co již napsané je, ale je to externí program. Pokud nevadí, že u takového programu nemůžeme ovlivňovat jeho činnost - tj, že ho můžeme pouze spustit a případně vzít výstupy -, pak ho lze snadno použít. Perl totiž má hned několik nástrojů pro spouštění shellových příkazů.

Než se do nich ale pustíme, měli bychom nejprve upozornit na několik rizik.

Spuštění externího programu je potenciální bezpečnostní díra. Je na místě dbát zvýšené pozornosti a všechny příkazy, které jsou shellu posílány, pečlivě kontrolovat. Perl z tohoto důvodu nabízí tzv. mechanizmus nakažení, který žádným potenciálně nebezpečným datům nedůvěřuje. Budeme se jím zabývat v následujícím díle.

To však není jediný důvod proč je lepší se spouštění shellových příkazů vyhnout. Bude také trpět přenositelnost. A nezřídka bude mít takový program vyšší nároky.

Funkce system

Vestavěná funkce system provede externí program tak, že nejprve spustí kopii aktuálního procesu (pomocí fork) a přepíše ji voláním tohoto externího programu. Dále program čeká, až se externí program ukončí.

system očekává jako parametr řetězec. Tento řetězec předá shellu, aby jej provedl jako příkaz.

```
system "ls";
```
Skript i volaný externí příkaz přitom sdílejí stejný standartní vstup a standartní výstup, dále pak i chybový výstup nebo informace prostředí.

Z tohoto důvodu nelze dostat výstup příkazu, protože ten je pouze normálně vytisknut. Jediným řešením, jak to změnit, je použití nějaké funkce shellu - tedy přesměrování. Pomocí něj je možné výstup ukládat do souborů.

```
system "ls > vystup";
```
Nebo, pokud žádný výstup nechceme, přesměrujeme do černé díry.

```
system "ls > /dev/null";
```
Umístěním funkce system na pravou stranu přiřazení tedy žádný výstup nezískáme. Získáme pouze návratovou hodnotu volaného příkazu. I to se však může hodit - například při testování, zda dopadl externí příkaz bez chyb.

```
$navratova_hodnota = system "ls";
```
Shellové příkazy obvykle vracejí v případě úspěchu 0 a v případě opačném jiné číslo. To mimo jiné znamená, že nemůžeme použít konstrukci příkaz or die;. Ale pokud na ni trváme, lze ji následovně modifikovat.

```
system "ls" and die "chyba\n";
```
Pomocí jednoho volání funkce system lze zavolat více příkazů. Přesněji řečeno, je to možné, pokud je v shellu definován znak pro oddělování příkazů. Oddělovači příkazů jsou obvykle středník nebo znak nového řádku.

```
system "echo 'soubory v aktuálním adresáři: '; ls";
system "echo 'soubory v aktuálním adresáři: '\n ls";
```
Pokud je funkci system předán seznam řetězců, jsou druhý a další řetězce brány jako argument pro příkaz, jehož název skrývá první řetězec. Argumenty ale, ostatně jsme to již viděli, lze předávat i ve stejném řetězci jako samotný příkaz.

Vytvoříme pro ukázku primitivní nadstavbu shellu, která nic neumí a bude pouze přijímat příkazy a vykonávat je.

```
while (<>) {
```

```
chomp;
last if $_ =~ /\d*exit\d*/;
system $_;
}
```

Funkce exec

Funkce system ve skutečnosti spustila kopii aktuálního procesu a ta byla přepsána příkazem, který byl externě volán. Tento přepis procesu zajišťovala právě funkce exec.

Pokud použijeme místo system funkci exec, přepíšeme tím stávající proces - externí příkaz se vykoná, ale řízení procesu již nebude vráceno našemu programu (přesněji řečeno - nebude vráceno, pokud je program spuštěn úspěšně). Po vyřízení volaného příkazu celý proces končí.

Zde je ukázka. Text "po exec\n" již nebude vytisknut.

```
print "před voláním exec\n";
exec "echo 'v exec'";
print "po exec\n";
```

Vytisknut by byl v případě, že by volaný příkaz skončil neúspěšně.

```
print "před voláním exec\n";
exec "neexistujici_prikaz";
print "po exec\n";
```

Obrácené apostrofy

Další možností, jak provést shellový příkaz, je umístit ho do obrácených apostrofů (na americké klávesnici je obrácený apostrof nad tabulátorem). Navíc je tímto způsobem možné získat výstup, jež je návratovou hodnotou.

```
$vystup_prikazu = `ls`;
```

Podobně jako uvození apostrofy resp. uvozovkami má i uvození obrácenými apostrofy alternativní syntakční variantu. Lze použít qx//.

```
$vystup_prikazu = qx/ls/;
```

Nyní je celý výstup v proměnné $vystup_prikazu. V seznamovém kontextu se výstup rozdělí na řádky (řádek je definován v proměnné $/).

```
@vystup_prikazu = `ls`;
```

Návratová hodnota provedeného příkazu je uložena ve speciální proměnné $?.

Uvedeme si krátký skript zjisti_jmeno, který na základě uživatelského jména zadaného jako argument na příkazovém řádku vytiskne skutečné jméno. Údaje budeme čerpat z /etc/passwd. Obrácené apostrofy použijeme k získání řádku s daným uživatelským jménem z tohoto souboru. Z tohoto řádku pak získáme skutečné jméno (nachází se mezi 4. a 5. dvojtečkou) pomocí regulárních výrazů.

```
#!/usr/bin/env perl
use strict;

die "Pouziti: zjisti_jmeno nick\n" if @ARGV != 1;
my $user = $ARGV[0];
foreach (`less /etc/passwd | grep $user`) {
/^[^:]+?:[^:]+?:[^:]+?:[^:]+?:([^:]+?):/;
print "$user je $1\n";
exit;
}

print "$user nenalezen!\n";
```

Program vyžaduje z příkazového řádku právě jeden argument. Pokud obdrží jiný počet, vypíše použití.

Spouštění příkazů pomocí open

O další možnosti spouštění shellových příkazů jsme se již letmo zmiňovali v souvislosti s funkcí open.

Jméno otevíraného souboru je lze nahradit příkazem shellu, který musí začínat nebo končit znakem |. Končí-li tímto znakem, výstup příkazu lze číst ze zdroje dat. A naopak začíná-li jím, pak vše, co přes ovladač pošleme, posíláme na vstup zadanému příkazu. Obě roury uvést nelze - v takovém případě bude ta koncová ignorována.

Tento program získá výstup příkazu ls -l, odstraní 1. řádek o počtu souborů a místo něj řádky očísluje.

```
open LL, "ls -l |" or die "Chyba\n";
<LL>;odstraníme 1. řádek
while (<LL>){
++$i;
printf "%3d: %s", $i, $_;
}
```

Perl (53) - Režim nakažení

Programy, které komunikují, jsou k bezpečnostním dírám náchylné více než kterékoliv jiné. Režim nakažení do značné míry omezí rizika.

Proměnné prostředí

Než se pustíme do hlavního tématu, podívejme se ještě na proměnné prostředí, protože jejich znalost se nám bude při jeho studiu hodit.

Proměnné prostředí jsou v Perlu dostupné poměrně jednoduše. Jsou zpřístupněny v hashi %ENV. Klíčem je vždy jméno proměnné a hodnotou její hodnota.

Například seznam cest k manuálovým stránkám je v proměnné MANPATH.

```
print $ENV{"MANPATH"};
```

Obsah všech proměnných prostředí tak vytiskne tento příkaz.

```
print $_, "=", $ENV{$_}, "\n" for keys %ENV;
```

Ještě snažší přístup k proměnným prostředí umožňuje modul Env, který pro každou z nich vytvoří skalární proměnnou o stejném názvu. Seznam umístění manuálových stránek se pak tiskne takto.

```
use Env;
print $MANPATH;
```

Režim nakažení

U každého programu, který mohou spouštět ostatní, je třeba myslet i na kontrolu nedůvěryhodných dat. Ne každý může mít čisté záměry. Lépe řečeno - téměř vždy se najde někdo, kdo se programu pokusí podstrčit data, na jejichž základě se může nějakým způsobem poškodit systém.

Řešením je všechna vstupní data důkladně kontrolovat. Ani tak ale nemusíme mít 100% jistotu. Co když někde není ošetření dokonalé? Nebo se někde dokonce na ošetření úplně zapomnělo? Od toho tu je mode tainted - nakažený režim.

Jeho aktivací se z Perlu rázem stane paranoik, který podezírá vše, co podezírat lze. Všechny uživatelské vstupy jsou považovány za nebezpečné a taková data jsou označena jako nakažená. To v důsledku znamená, že s nimi nebude možné provádět nebezpečné operace, pokud je neošetříme.

Nakažená data jsou veškerá data získaná ze vstupu. Co to znamená? Myslí se toto:

- vstup přečtený z ovladače
- proměnné prostředí (PATH, BASH_ENV, CDPATH, ENV, IFS)

Režim nakažení, ač je velký pomocník, není samospasitelný. Je stále schopen zabránit pouze části programátorových chyb.

Aktivace

Nakažený režim se zapíná uvedením přepínače -T. Skripty je proto třeba spouštět takto.

```
$ perl -T program.pl
```

Případně lze pozměnit 1. řádek skriptu, pokud je spustitelný.

```
#!/usr/bin/perl -T
```

Ukázka nakažených dat

Přistupme teď už k nějaké konkrétní situaci.

```
$nakazena_data = <>;   #načteme hodnotu ze vstupu
$bezpecna_data = "ls"; #načteme danou hodnotu

print $bezpecna_data;  #není důvod, proč by to nemělo fungovat
print $nakazena_data;  #i toto funguje, print je bezpečná operace

system $bezpecna_data; #problém!
system $nakazena_data; #dokonce 2 problémy!!!
```

Jeden problém je zřejmý - $nakazena_data jsou přijata ze vstupu a nejsou bezpečná. Proto má 2. volání funkce system o problém víc.

Ale co první problém? Je snad nějaký důvod, proč by neměla být funkci system předána bezpečná data?

Ano, skutečně existuje ještě další nenápadný důvod. Hodnoty proměnných prostředí v hashi %ENV jsou také nakažené, což je logické. Potom jsou nakažené i hodnoty v $ENV{"PATH"} a $ENV{"ENV"}.

Co kdyby byl v systémové proměnné PATH nějaký adresář, ve kterém by byl program ls, jež by dělal něco úplně něco jiného než klasické ls? Proto je třeba nastavit si vlastní PATH. Stejně tak ENV.

Nyní se podívejme na další kód. Ošetřili jsme (ač nepříliš elegantně) proměnné prostředí, volání system je tedy již bezpečné a vše by mělo fungovat.

```
$ENV{"PATH"} = "/usr/local/bin:/usr/bin:/bin";
$ENV{"ENV"} = "/etc/bash.bashrc";
system $bezpecna_data; #OK
system $nakazena_data; #už "jen" 1 problém
```

Léčení nakažených dat

Jak ale vyřešíme náš první problém? Co když si vstup ošetříme a bude-li vyhovovat kritériím, budeme ho chtít funkci system skutečně předat? Odpověď skýtá léčení nakažených dat.

Léčba probíhá pomocí regulárních výrazů. Je třeba stanovit kritéria, kdy je řetězec nakažený a kdy ne - tedy určit vzor. Pak nakažený řetězec porovnáme s tímto vzorem. Pokud testovaný řetězec vzoru vyhovuje, pak má daný řetězec požadovaný formát. To znamená, že je bezpečný.

Podívejme se na příklad. Pomocí regulárního výrazu z řetězce vyseparujeme části, které vyhovují vzoru - a které jsou tedy bezpečné.

```
$nakazena_data =~ /^(pwd|whoami)$/;
$uzdravena_data = $1;
system $uzdravena_data if $uzdravena_data; #OK
```

Nyní byla data uzdravena pouze v případě, že původní nakažená data obsahovala řetězec pwd nebo whoami. Žádné jiné příkazy nemohou být provedeny. Kdyby byl načten jiný řetězec, proměnná $1 by nebyla definována.

Podívejme se ještě na odstrašující příklad.

```
$nakazena_data =~ /(.*)/;
$uzdravena_ale_nebezpecna_data = $1;
```

Takový zápis je nebezpečný a neměl by se vůbec používat. $nakazena_data budou uzdravena, ať obsahují cokoliv. Je to jako bychom měli režim nakažení vypnutý.

Testování na nakaženost

Pomocí výjimek lze zjistit, zda jsou určitá data nakažena či nikoliv. Zde je podprogram, který této vlastnosti využívá a předaná data testuje.

```
sub is_tainted {
my $arg = shift;
local $@;
eval { eval "# " . substr($arg, 0, 0) };
return length($@) != 0;
}
```

A teď můžeme testovat libovolná data na nakaženost.

```
$data1 = <STDIN>;
print "nakaženo!\n" if is_tainted($data1);
print "nakaženo!\n" if is_tainted("TAINTED?");
print "nakaženo!\n" if is_tainted($ENV{"PATH"});
```

Zde je patrné, že byly nakaženy 1. a 3. řetězec.

Jak vytvářet nové procesy? Jaká je podstata externích příkazů?

Již při probírání funkce system jsme se setkali s dělením procesů. Dnes se této problematice budeme věnovat podrobněji.

Proces

Proces je instancí programu. Každý proces má svoje ID (PID) - to je unikátní číslo, kterým jádro proces označuje.

Vytváření procesů

Perl obsahuje funkci fork, která se shoduje s unixovým forkem. Funkce fork vytváří dceřinný proces. Jeden program tak může běžet v několika kopiích. Po zavolání funkce fork se proces rozdvojí. Vzniknou 2 paralelní procesy - rodič a potomek. Každý s vlastním PID. Na sobě nejsou tyto procesy nijak závislé (oba mají vlastní data, jmenné prostory atd.).

Jaký to má ale všechno význam? fork se v praxi užívá k paralelnímu spuštění určitého množství kopií programu. Bez použití forku by kopie musely být spuštěny sériově.

Typickým příkladem využití je server, který zpracovává požadavky klientů. Ve stejném okamžiku server může obsluhovat více klientů zároveň. Použitím sériového zpracování by musely ostatní klienti čekat až bude hotov klient, který je právě na řadě (To může působit potíže například tehdy, když pro každého klienta musí uživatel několikrát zadat text na vstup. Navíc klientů mohou být desítky, stovky nebo ještě více.). Až se v budoucnu budeme zabývat sockety, nějaký ukázkový server si vytvoříme.

Funkce fork nepřijímá žádný argument. Vytváří dvě totožné kopie původního programu a vrací hodnotu, kterou může být 0, PID nebo nedefinovaná hodnota. Pro potomka vrací 0, pro rodiče PID.

Příklady

Podívejme se na první úsek kódu.

```
my $pid = fork;                     #pro jednoduchost netestujeme na undef
    printf "%-7s - %d\n", "BOD A", $pid;     #provedeno oběma procesy
        if ($pid) {                         #jde o rodicovsky proces
            printf "%-7s - %d\n", "RODIC", $pid;
        } else {                            #jde o potomka
            printf "%-7s - %d\n", "POTOMEK", $pid;
        }
    printf "%-7s - %d\n", "BOD B", $pid;     #provedeno oběma procesy
```

Protože všechny procesy vypisují na stejný standartní výstup, je výsledek smíchán. V našem případě však jsou skripty tak krátké, že než druhý začne, první je už ukončen.

```
$ perl fork.pl
BOD A   - 0
POTOMEK - 0
BOD B   - 0
BOD A   - 8266
RODIC   - 8266
BOD B   - 8266
$
```

Abychom si dokázali, že běží tyto procesy paralelně, nikoliv sériově, použijeme funkci sleep. Ta přerušuje na daný počet sekund vykonávání programu.

```
my $pid = fork;
    printf "%-7s - %d\n", "BOD A", $pid;
if ($pid) {                         #jde o rodicovsky proces
    sleep  1;                       #čekání 1s
    printf "%-7s - %d\n", "RODIC", $pid;
} else {                            #jde o potomka
    sleep  1;                       #čekání 1s
    printf "%-7s - %d\n", "POTOMEK", $pid;
}
    printf "%-7s - %d\n", "BOD B", $pid;
```

Výstup se nám následovně změnil.

```
$ perl fork.pl
BOD A   - 0
BOD A   - 8266
POTOMEK - 0
BOD B   - 0
RODIC   - 8266
BOD B   - 8266
$
```

V bodě A potomek čeká a rodič ho na chvíli dožene. Potom ale čeká i rodič.

Čekání na dceřinný proces

Často se můžeme dostat do situace, kdy je dceřinný proces časově náročný a zároveň ho potřebujeme ukončit dříve, než rodičovský proces. Na to funkci sleep použít nemůžeme. Perl k tomuto účelu nabízí funkci wait nebo waitpid.

wait čeká do doby, než je ukončen libovolný z dceřinných procesů. waidpid navíc přijímá PID procesu, na jehož ukončení se čeká. Vraceno je PID ukončeného procesu.

wait i waitpid vracejí PID ukončeného potomka, na kterého se čekalo. Pokud žádný potomek neběží, vrací tyto funkce hodnotu -1.

```
my $pid = fork;
    if ($pid) {
print "RODIC - ZACATEK\n";
wait; #čeká se na ukončení potomka
print "RODIC - KONEC\n";
    } else {
print "POTOMEK - napis neco:\n";
$_ = <STDIN>; chomp;
print "POTOMEK - napsal jsi $_\n";
```

```
}
```
Odtud je zřejmé, že rodič je ukončen až po potomkovi.
```
$ perl fork.pl
POTOMEK - napis neco:
RODIC - ZACATEK
neconeco
POTOMEK - napsal jsi neconeco
RODIC - KONEC
$
```
Zjištění PID rodičovského procesu

U dceřinných procesů platí, že funkce getppid vrací PID rodiče. Pokud proces dceřinný není, vrací PID shellu, ve kterém je program spuštěn.
```
my $pid = fork;
if ($pid) {
print "PREDEK RODICE(PID $$): ", getppid, "\n";
} else {
print "PREDEK POTOMKA(PID $$): ", getppid, "\n";
}
```
V proměnné $$ je PID aktuálního procesu. Z následujícího výstupu lze vyčíst hierarchii procesů.
```
$ perl fork.pl
PREDEK POTOMKA(PID 13610): 13609
PREDEK RODICE(PID 13609):7325
$ echo $$     #PID shellu
7325
$
```
Spuštění jiné úlohy v procesu

Toto téma jsme již rozebírali u funkce exec. Doplňme jen, že funkce system je kombinací funkcí exec a fork.

Perl (55) - Eval

Eval je dalším silným nástrojem skriptovacích jazyků, Perl nevyjímaje. V Perlu se využívá eval na několik odlišných činností.

Ukážeme si dva významy nástroje eval. První z nich je dynamické vyhodnocování řetězců - určitý řetězec se chová jako posloupnost příkazů (tj. má korektní perlovou syntaxi). Řetězec je normálně zkompilován jako program a spuštěn v rámci jiného programu. A druhou možností použití eval je odchyt výjimek - tedy to, co z jiných jazyků známe jako konstrukci try-throw-catch.

Dynamické vyhodnocování řetězců

eval přijímá jako argument řetězec, ten je proveden jako samotný program a jeho výstup funkce vrací. Ukažme si jednoduchý příklad.
```
$program = 'print "HELLO WORLD";';
eval $program;
```
Posloupnost těchto příkazů vypíše řetězec "HELLO WORLD". Následující dva řádky tak v důsledku udělají to samé.
```
eval 'print "HELLO WORLD";';
print "HELLO WORLD";
```
Uveďme ještě jeden příklad - jednoduchou kalkulačku.
```
while ($prikaz = <>){
    print "Reseni: ";
    print eval $prikaz;
    print "\n";
}
```
Poznámka - eval podporuje výchozí proměnnou. S jejím použitím je následující zápis analogický předchozímu.
```
print "Reseni: ", eval, "\n" while <>;
```
Když program spustíme, můžeme zadávat libovolný kód v syntaxi Perlu a ten je okamžitě řádek po řádku prováděn. Získali jsme tak jednoduchý perlový shell.
```
$ perl kalk.pl
22
Reseni: 22
5*7
Reseni: 35
sin 3.141592
Reseni: 6.53589793076238e-07
for(60..70){print chr $_;}
Reseni: <=>?@ABCDEF
exit
Reseni: $
```
Za zmínku stojí také to, že v řetězci mohou být pouze ty proměnné, které jsou dostupné v době volání eval. Z tohoto důvodu nebude fungovat následující kód, pokud nebude odstraněna lokálnost proměnné $x.
```
{
our $r = 'print "\$x = $x"';
my $x = 10;
}
eval $r;
```
Pokud není řetězec, předávaný funkci eval, syntakticky správný, žádná chyba se na chybový výstup nevypisuje. Místo toho je její text uložen v proměnné $@. Je-li naopak příkaz proveden úspěšně, bude $@ nedefinována.

Eval a uvozování řetězců

Je třeba upozornit také na další věc. Různé možnosti uvození se chovají dvěma způsoby. Podívejme se na následující možnosti.
```
$a = 3; $b = 4;
$p = 'print $a + $b';
```

```
eval $p;
eval '$p';
eval "$p";
eval qq/$p/;
```

Druhý případ nevytiskne nic, ostatní tisknou součet $a + $b, tedy 7. Proč? Jde o to, co je skutečně funkci eval předáváno. U druhého volání eval je to řetězec '$p' a tudíž je vykonán právě takový příkaz ($p;). V ostatních případech se proměnné a escape znaky nahrazují svými skutečnými hodnotami a ve skutečnosti je tak funkci eval předán řetězec 'print $a + $b'. Na toto je třeba dát pozor.

Bezpečnost

Právě díky funkci eval mohou vznikat v programech četné bezpečnostní díry. Co když našemu programu, simulujícímu kalkulačku, zadáme na vstup například následující příkaz?

```
system 'rm -rf ~'
```

Je žádoucí, aby byl proveden? Opět záleží na situaci. Někdy ano, někdy ne. Pokud má uživatel, pod kterým je spuštěna aktuální instance programu, práva na vykonání tohoto příkazu, pak bude vykonán. Obranou je aktivace režimu nakažení.

Odchyt výjimek

Výjimky jsou chyby za běhu programu. Jejich zpracování umí zařídit funkce eval v blokovém tvaru. eval zde využívá toho, že při chybě v syntaxi narozdíl od obyčejného programu neskončí, ale pouze nastaví (nebo nenastaví) proměnnou $@. Typickým příkladem zpracování výjimky je ošetření případu, kdy je děleno nulou.

```
eval {
$delenec = 10;
$delitel = 0;
$podil = $delenec / $delitel;
};

print $@;
print "Program normalne pokracuje dal...\n";
```

Navíc lze v kombinaci s die definovat vlastní chyby. die totiž neukončí celý program, ale pouze blok funkce eval. Řetězec za die se uloží do $@. Funguje to podobně, jako throw v Javě nebo C++.

```
eval {
open DATA, "soubor" or die "Soubor nelze otevrit\n";
};

print $@;
```

Pokud byl soubor soubor bez problému otevřen, je dále proměnná $@ nedefinována. Ale v případě, že příkaz open selže, je v bloku eval volána funkce die, která vyvolá výjimku a v proměnné $@ tak bude řetězec "Soubor nelze otevrit\n". Poté můžeme v případě selhání například otevřít jiný soubor.

Pro usnadnění při odchytávání výjimek existuje na CPAN několik modulů - například Exception nebo Error (viz článek na www.perl.com).

Eval a regulární výrazy

Pouze pro připomenutí si uveďme, že na dynamické vyhodnocování výrazů lze narazit v substitucích regulárních výrazů. Nahrazovací řetězec může být výrazem. Ano, jde o přepínač /e.

Eval a časově omezené čekání vstup

Chceme vytvořit program, který ze vstupu přijme nějaký řetězec. Ale zároveň chceme, aby ho uživatel zadal v nějakém časovém intervalu (tedy konkrétně například do 20 sekund). Pokud uživatel nic neodešle, výzvě na zadání textu vyprší platnost a program je nucen učinit patřičné kroky.

K tomu je třeba použít funkci alarm. alarm má jako parametr počet sekund, za které vyvolá signál ALRM nebo číslo 0 pro zrušení alarmu.

Jestliže uživatel nezadá 20 sekund vstup, signál ALRM vyvolá funkci die a zachytíme výjimku. Pokud je tak za blokem eval v proměnné $@ řetězec "time limit expired on line ...", víme, že uživatel nic nezadal.

```
$SIG{"ALRM"} = sub {die "time limit expired"};
eval {
alarm(3);
<STDIN>;
alarm(0);
};

print "Vyprsel casovy limit na zadani vstupu.\n" if $@;
print "Pokracuji...\n";
```

Chcete-li vědět více o možnostech eval, lze doporučit knihu Programování v Perlu pro pokročilé od Srirama Srinivasana, kapitolu 5.

Perl (56) - Volby příkazu perl

Jaké máme možnosti při spouštění perlových programů ze shellu? Budeme se zabývat přepínači příkazu perl.

Přepínačů, které ovlivňují chování má perl celou řadu. My si je představíme, protože některé z nich budeme ještě v budoucnu potřebovat. Jiné uveďme pouze pro zajímavost jako ilustraci možností.

Charakteristika

perl [*volby*] *zdrojový_soubor* [*parametr1*] [*parametr2*] ... [*parametrn*]

Již známé přepínače

Přepínače v následující tabulce jsme již používali a nebudeme je nyní více rozebírat.

Přepínač	Popis
-d	spustí program v debuggeru
-v	zobrazí informace o verzi, licenci atd.
-M	načte modul

-I	adresář, ve kterém se budou přednostně hledat moduly
-t, -T	aktivuje režim nakažení

Varování a chyby

Zapnutí zobrazování varování

Přepínač -w zapíná varování. Upozorňuje na úseky programu, které sice překladač vezme, ale je zde podezření na chybu. Často tedy najde chybu, aniž bychom ji museli hledat my.

Uveďme jednoduchý příklad - program na jediný řádek. Sledujme, jak zapůsobí přepínač -w.

```
print $i;
```

Porovnejme výstup s volbou -w a bez ní.

```
$ perl volby.pl
$ perl -w volby.pl
Name "main::i" used only once: possible typo at volby.pl line 1.
Use of uninitialized value in print at volby.pl line 1.
$
```

Obdrželi jsme hned dvě varování. První z nich říká, že je proměnná v programu použita jen na jediném místě. To je opravdu vždycky podezřelé (tedy pokud to není speciální proměnná - ovšem ty přepínač -w nehlásí). Druhým varováním dává Perl najevo, že se mu nelíbí náš způsob práce s proměnnou $i. Nic v ní není a už ji tiskneme. V obou případech má Perl více méně pravdu a stálo by za to něco s tím udělat. Oba problémy se vyřeší tak, že se před print přiřadí do $i prázdný řetězec.

Většinou je lepší používat místo -w pragmu warnings, jejíž význam je podobný.

Další přepínač, -W, zapíná všechna možná varování.

Vypínání varování

-X veškerá varování ruší. Funkce die funguje normálně.

Jednořádkové skripty

Podobně jako u jiných skriptovacích jazyků není vždy nutné psát přímo skripty. Je většinou zbytečné ukládat do souboru jen krátký a jednorázový příkaz. Použijeme-li přepínač -e a za něj do uvozovek příkaz, vykoná se stejně, jako šlo o skript. Tímto způsobem lze předávat i delší posloupnosti příkazů, ač to není příliš přehledné.

Pokud potřebujeme rychle přímo ze shellu ASCII hodnotu písmena d, stačí zadat:

```
$ perl -e 'print ord "d"'
100$
```

Uvedeme-li lichý počet uvozovek, apostrofů nebo obrácených apostrofů, příkaz se neprovede. Po odentrování se dostaneme na nový řádek, uvozený sekundárním promptem, a můžeme pokračovat v příkazu.

Problém nastane, když budeme chtít vytisknout apostrof. Samotné zpětné lomítko nestačí. K vypsání apostrofu použijeme takovouto změť znaků.

```
$ perl -e 'print "'\'"; '
' $
```

Přepínačem -e se ještě budeme zabývat. Mimo jednorázových příkazů se také často používá v kombinaci s nástroji shellu, cronem apod.

Přepínače pro ladění

Následovat budou přepínače, které jednorázově přidávají do našeho kódu nějaký další. Moc lidí je nepoužívá, ale představme si je, možná se někomu některý zalíbí.

Volba -n

Tato volba má ten efekt, že se náš program obalí cyklem while (<>){ *... náš program ...* }. Podívejme se na tento program.

```
$i++;
print "$i: $_";
$_++;
```

Spustíme-li ho s volbou -n, bude se tvářit (při aktuálním spuštění) takto.

```
while (<>){
$i++;
print "$i: $_";
$_++;
}
```

Zkusme ho tedy spustit.

```
$ perl -n volby.pl
666
1: 666
6*4
2: 6*4
...
```

Originální program se provádí stále dokola, pokaždé s jiným testovacím vstupem.

Volba -p

Zde se oproti -n $_ tiskne v bloku continue, který volba -p přidává. Struktura simulovaného programu vypadá následovně: while (<>){ *... váš program ...* } continue { print; }.

Zkusme spustit předchozí příklad znovu, jen zvolíme volbu -p. Výstup je trochu nepřehledný ale vstup rozeznáte podle kurzívy:

```
$ perl -np volby.pl
666
1: 666
6676*4
2: 6*4
7
...
```

Je to téměř to samé, co minulý příklad. Liší se jen v tom, že volba -p zajistí vytisknutí hodnoty proměnné $_ po skončení programu (jako by program končil příkazem print $_;). Řádek $_++; neměl v předminulém příkladě žádný efekt na výsledek, nyní již ano.

Volba -pe

Další možností užití -p je v kombinaci s -e. Každý řádek vstupu (soubor v parametru nebo klasicky stdin - podrobnosti za chvíli) je načten a jsou s ním provedeny příkazy, uvedené jako parametr volby -e. Poté se tiskne.

Struktura: while (<>) { ... *příkazy v hodnotě volby -e* ... } continue { print; }

Praktická ukázka: Chceme očíslovat řádky v dnes několikrát zmiňovaném souboru volby.pl a vytisknout.

```
$ perl -pe '$i++;print $i.". ";' volby.pl
1. $i++;
2. print "$i: $_";
3. $_++;
$
```

Je to poměrně rychlé řešení. Chceme-li navíc uložit výsledek do souboru ukazka, použijeme nástroje shellu.

```
$ perl -pe '$i++;print $i.". ";' volby.pl > ukazka
$ more ukazka
1. $i++;
2. print "$i: $_";
3. $_++;
$
```

Volby -na, -F

S volbou -na bude mít program následující strukturu:

```
while (<>){
@F = split (\ \);
... samotný program ...
}
```

Opět vytvoříme ukázkový program. Bude vypadat takto.

```
$, = "*";
$\ = "\n";
print @F;
```

A zkusíme spustit.

```
$ perl -na volby.pl
každá mezera je nahrazena hvězdičkou
každá*mezera*je*nahrazena*hvezdičkou
...
```

Pokud chceme za while přidat ještě continue { print; }, můžeme příkaz rozšířit o přepínač -p.

Nechceme-li nahrazovat mezeru, ale jiný řetězec vyhovující vzoru (zvolil jsem znaky Q a r), přidejme volbu -F (a za ní v apostrofech a lomítkách regulární výraz).

```
$ perl -na -F'/[Qr]/' program.pl
QwertyQwerty
*we*ty*we*ty
...
```

Nahrazování v souboru pomocí regulárních výrazů

Pokud se některé kombinace přepínačů používají opravdu často, pak mezi ně patří -pi -e. Pomocí nich můžeme nahradit části souboru vyhovující regulárnímu výrazu jiným textem. Samotné -i zajišťuje, že se výsledek nahrazuje stávající soubor. Bez -i by se vytiskl na výstup. Máme soubor os s tímto obsahem.

```
Windows Vista
```

Aplikujeme na něj tento příkaz.

```
$ perl -pi -e's/Windows Vista/SUSE Linux/g;' os
```

Hned je obsah os o něco hezčí.

```
SUSE Linux
```

Pokud chceme zároveň starý soubor zálohovat, použijme hodnotu pro přepínač -i. Ta udá příponu zálohy.

```
$ perl -p -i.backup -e's/Windows Vista/SUSE Linux/g;' os
```

Možnosti této konstrukce ještě zvyšuje možnost určení souborů podle žolíkových znaků.

Separátor při čtení ze zdroje dat

Volba -0 mění separátor $\, který například určuje, kam až se bude číst ze zdroje dat <>. Implicitně je oddělovačem znak nového řádku. Zkusme použít nějaký další separátor - uvádí se jako hodnota přepínači -0. Pozor na to, že musíme uvést oktálovou hodnotu. (ASCII tabulka)

Máme textový soubor data, kde jsou jednotlivá slova oddělena hvězdičkami.

```
text*oddělený*hvězdičkami*
```

A dále zdrojový kód programu.

```
@p = <>;

foreach (@p) {
print "$_\n";
}
```

Spusťme program. Jako parametry uveďme přepínač -0052 a soubor data. Hodnota 052 je oktálové vyjádření hvězdičky v ASCII tabulce. Sledujme, co se stane.

```
text*
oddělený*
hvězdičkami*
```

Něco navíc

Přepínač -x

Pouze pro zajímavost zmiňme, že ve skutečnosti nemusí být řádek #!/usr/bin/perl na prvním řádku. Použijeme-li volbu -x, pak vše, co je před tímto řádkem je ignorováno. Použijeme-li -x, musí tam už #!/usr/bin/perl povinně být (a to na začátku řádku). V opačném případě budeme svědky hlášky No Perl script found in input.

Kontrola syntaxe

S volbou -c Perl zkontroluje syntaxi, aniž by se program spouštel.

Program se nedostane do fáze běhu. Proto bude provedeno pouze to, co je v blocích BEGIN a CHECK.

```
BEGIN {
print "Jsem v BEGIN\n";
$i = 1;
}

$i = 10;
print "za BEGIN";
```

Kód za BEGIN se už nevykonává, ale jen kontroluje.

```
$ perl -c volby.pl
Jsem v BEGIN
volby.pl syntax OK
$
```

Přepínače a řádek #!${PATH}/perl

Na některých unixových systémech můžete údajně za #! napsat maximálně jen 32 znaků. Já jsem se s tímto problémem zatím nesetkal.

Do tohoto řádku lze uvést i volby. Takže chceme-li mít zapnutý režim nakažení, bude náš první řádek vypadat nějak takto.

#!/usr/bin/perl -T

perlcc

Jak víme už od úvodního dílu, příkaz perlcc vytváří ze zdrojových kódů binární kód. Následující příkaz vytvoří spustitelný soubor a.out.

```
perlcc program.pl
```

Pomocí přepínače -o ho lze pojmenovat jinak.

```
perlcc -o program program.pl
```

S přepínačem -e lze kompilovat i jednořádkové skripty.

```
perlcc -e 'print "Hello world\n"' -o program
```

Perl (57) - Jednořádkové skripty

Dnešní díl bude ryze praktický. Předvedeme si na konkrétních příkladech, jak využít příkaz perl jako nástroj přímo v shellu.

Pro spouštění skriptů v jazyce Perl není bezpodmínečně nutné, aby byl uložen v souboru. Pro krátké programy, skládající se z minima příkazů, je zde možnost spouštění přímo přes příkazový řádek. Takové skripty můžeme nazývat jednořádkové (někdy se lze setkat s názvem oneliners).

V 56. díle seriálu jsme se zabývali některými z přepínačů, které Perl akceptuje. Zmínka přišla i na volbu -e, která umožňuje spouštět kód Perlu z příkazového řádku.

Jednořádkové skripty se využívají zejména při práci s textem. Předvedeme si několik příkladů. Ne všechny budou praktické, ale pro představu o možnostech uveďme co nejširší záběr.

Hello World

Nejjednodušším příkladem z této kategorie je obyčejný výpis textu.

```
$ perl -e 'print "Hello World\n"'
```

Jak ukazuje následující příkaz, přepínačů -e může být i více.

```
$ perl -e 'print "Hello ";' -e 'print "World\n"'
```

Výpis pole @INC

Pro přehlednou informaci o obsahu pole @INC zadejme následující příkaz.

```
$ perl -e 'for(@INC){print "$_\n";}'
```

Přejmenování všech souborů v adresáři

Pojďme však k příkladům, které mohou být opravdu užitečné. Nejednou se stane, že potřebujeme hromadně změnit jména skupiny souborů. Variantou takového programu je změna velikosti písmen v názvech soborů na malá.

```
$ perl -e 'for (<*>) {rename $_, lc($_)}'
```

Tento skript lze modifikovat tak, aby na malá písmena přejmenoval jen názvy uvedených souborů. Použijeme k tomu pole @ARGV.

```
$ perl -e 'for (@ARGV) {rename $_, lc($_)} soubor1 soubor2
```

S funkcí rename lze vymýšlet i složitější věci. Kupříkladu následující příkaz přejmenuje všechny soubory v aktuálním adresáři tak, aby začínali vždy rokem, ve kterém byly naposledny změněny.

```
$ perl -e 'for (<*>) {rename $_, 1900+(localtime((stat)[9]))[5]."$_"}'
```

A podobně. Takovéto příkazy mohou v mnohém připomínat použití unixového příkazu find, pomocí nějž lze dělat podobné věci.

Nahrazení slov

Možná vůbec nejčastějším onlinerem je nahrazení části textu, která vyhovuje danému regulárnímu výrazu. Vytvoříme skript, který v uvedených soubourech nahradí libovolný počet mezer za sebou mezerou jedinou.

```
$ perl -pi -e 's/ +/ /g' soubor1 soubor2
```

Připomenutí události

Pokud se vám stává, že si necháte vařit čaj, zasednete k počítači a zapomnenete na vše kolem, jistě přivítáte následující skript.

```
$ perl -e 'if (!fork){sleep $ARGV[0]*60;print "\nCAJ UVAREN!!!\a\n";exit;}exit;' 7
```

Ten se od okamžiku spuštění neozývá, ale to je jen zdání, protože zatím nenápadně čeká na pozadí, aby se mohl za zadaný počet minut spustit a vyvést tak uživatele z tranzu.

Je to spíše taková hračka, ale pokud by se vám zalíbila natolik, že byste ji chtěli mít stále po ruce, vytvořte si třeba soubor /usr/bin/wake s právy pro spouštění a následujícím obsahem.

```
#!/usr/bin/env perl
if (!fork){sleep $ARGV[0]*60;print "\n".$ARGV[1]."\a\n";exit;}exit;
```

Nyní zadejme tento příkaz.

```
$ wake 120 'PROBUĎ SE'
```

Za 2 hodiny můžeme čekat upozornění.

Smazání daných řádků v souboru

Pro smazání 3. až 7. řádku v souboru použijme příkaz

```
$ perl -i.old -ne 'print unless 3..7' soubor
```

Konverze měn

Dalším příkladem je vytisknutí souboru, ve kterém zaměníme ceny v amerických dolarech za ceny v českých korunách. Jako kurz budeme brát hodnotu 20.5CZK/USD.

```
$ perl -pe 's/(\d+.?\d*)USD/20.5*$1."CZK"/ge' soubor
```

Získání WWW stránky

Alternativou k příkazu lynx -source je následující, tentokrát poněkud delší zápis.

```
$ perl -MLWP::Simple -e 'mirror("http://www.linuxsoft.cz", "linuxsoft.html")'
```

Nicméně jeho modifikací můžeme smysl tohoto příkazu upravit. Například výpis hlavičky na standardní výstup.

```
$ perl -MLWP::Simple -e'print head "http://www.linuxsoft.cz"'
```

Spolupráce s find

Abychom si předvedli něco trochu jiného, podívejme se na následující příkaz.

```
$ echo soubor | perl -nle unlink
```

Jeho činnost je již asi zřejmá. Maže soubor, který mu byl předán rourou. Příkaz tak lze využít například na základě výstupu příkazu find.

```
$ find . -name '.backup' | perl -nl -e 'unlink'
```

Nutno však podotknout, že stejného efektu bychom docílili i definicí akce přímo pomocí find.

```
$ find . -name '*2*' -exec rm -f {} \;
```

Vytisknutí části souboru

Dalším příkladem, který můžeme v praxi čas od času užít je výpis konkrétních řádků souboru, určených rozsahem. Pro vytisknutí řádků 7-12 v uvedeném souboru lze psát tento příkaz.

```
$ perl -ne 'print if 7 .. 12' soubor
```

Hromadná změna práv

Další příkaz mění práva všech souborů aktuálního adresáře, které začínají písmenem d.

```
$ perl -e 'while(<*>){chmod 0700, $_ if /^d/}'
```

Pro vykonání této činnosti by však asi bylo pohodlnější použít find.

Očíslování řádků

Pro vytisknutí souboru na výstup s tím, že budou jednotlivé řádky označeny svým pořadím, napíšeme toto.

```
perl -n -e 'printf "%3d. %s",$.,$_' soubor
```

Závěr

Ukázali jsme, že jednořádkové skripty zvládnou skutečně mnoho. Ovšem většinou není jejich vytvoření příliš pohodlné a je lepší užít přímo některý shellový nástroj.

Jednořádkové skripty jsou výhodné například při nahrazování textu v souborech nebo když nevíme, jak realizovat daný úkol přímo v shellu. Je to užitečné doplnění technik uživatele příkazového interpretu.

Perl (58) - OOP - úvod

Co to je objektově orientované programování a jak se liší od programování procedurálního? Tomu bude patřit několik následujících dílů.

Objektově orientované programování (object-oriented programming, dále OOP) je způsob programování, při kterém je na řešený problém nahlíženo z jiného hlediska, než jako tomu bylo doposud. Jde o to, že program je soustavou objektů, které spolu komunikují a ovlivňují se (narozdíl od procedurálního programování, kde je programem seznam příkazů a volání funkcí, jež je prováděn "shora dolů"). Ze všeho nejdříve jsou vytvořeny právě tyto objekty a nich je následně program sestaven. Hlavní myšlenkou pro vznik OOP bylo, aby objekty korespondovaly s věcmi z reálného světa.

Vezměmě si telefon jako příklad objektu. Voláme, přijímáme hovory, posíláme a přijímáme SMS atd. Vůbec nás nemusí zajímat nic o tom co je uvnitř a jak to funguje. Není to potřeba k tomu, abychom ho mohli používat. Používáme pouze rozhraní telefonu - tedy funkce, které nabízí (což je nějaká soustava procedur). Stručně řečeno, to, že nevíme, jak telefon funguje, není důležité, ale důležité je, že víme, jak ho používat. To je základní myšlenka OOP.

OOP nebude záležitost pouze několika příštích dílů, ale budeme se s ním setkávat ve zbytku seriálu prakticky na každém kroku. Ať už půjde o práci s databázemi, CGI, návrhem grafických uživatelských rozhraní nebo psaní modulů. Jedná se o důležitý nástroj pro programy většího rozsahu.

Pojmy objektově orientovaného programování

Vysvětleme si nyní podrobněji některé základní pojmy OOP.

Objekt

Jak již z názvu vyplývá, objekt je základem OOP. Objekt reprezentuje jednu konkrétní věc. Každý objekt má nějaké atributy a chování. Atributy jsou vlastnosti, kterými se liší objekty též třídy - například barva a výška. Chování jsou činnosti, které mají všechny objekty dané třídy stejné.

Uveďme příklad. Z funkčního hlediska je úplně jedno, jestli používáme černou nebo bílou CD-ROM mechaniku. Černá a bílá mechanika patří do stejné třídy, ve které je definován atribut barva. Jinak fungují úplně stejně.

Třída

Třída je šablonou pro objekty se stejným chováním. Podobné objekty (lišící se jen v atributech) jsou definovány právě podle této šablony. Jsou zde definovány činnosti, které mohou objekty provádět.

Instance třídy

Každý objekt spadá pod nějakou třídu. K této třídě je příslušný objekt instancí. Pokud máme třídu CD-ROM mechanik, jejími instancemi budou mechanika1, mechanika2, mechanika3... mechanika1 bude například černá a bude číst rychlostí 40x. Všechny CD-ROM mechaniky (ta moje, ta vaše i všechny ostatní) jsou instancemi třídy CD-ROM mechanika. Všechny poskytují stejné služby (čtení CD, vysunutí apod.) a liší se v atributech - tedy záležitosti jako je barva, rychlost čtení apod.

Metoda

Metoda je podprogram, který se vztahuje pouze k určité třídě a může tak být volána pouze pro instance této třídy. Třída CD-ROM mechanik asi bude těžko obsahovat metodu nakresli_elipsu. Tuto metodu budou ale obsahovat třídy pro návrh grafických aplikací, například QPainter z knihovny Qt.

Ač to nyní není nezbytné, zmiňme, že existují dva druhy metod - metody třídy a metody objektu. Příkladem metody třídy je většina konstruktorů. To je metoda, která se nevolá nad žádným objektem, ale objekt vytváří. Metody objektu jsou obvykle volány nad objektem.

Zpráva

Zpráva je prostředek komunikace objektu s okolím. Například pro vysunutí CD z mechaniky stiskneme tlačítko na mechanice. Tím vyvoláme metodu a mechanice předáme zprávu, kterou oznamujeme, že má vysunout CD. Podmínkou toho je, aby mechanika tlačítko pro vysunutí měla - musí existovat příslušná metoda.

Rozhraní objektu

Rozhraní (často interface) objektu je množina zpráv, kterým objekt rozumí. Uživatele objektu zajímá právě jen rozhraní.

Objektově orientované programování v Perlu

V Perlu nejsou pro OOP téměř žádná speciální klíčová slova. Vše je řešeno pomocí nám známých konstrukcí. Připomeňme, že šetření klíčovými slovy se vyskytuje i v jiných oblastech, vzpomeňme na řešení výjimek.

Reprezentace objektu, metody a třídy v Perlu

Třídou je obyčejný balík definovaný pomocí klíčového slova package. V něm jsou definovány metody. Třída se často vytváří v souboru odděleném od programu jako modul.

Objekt, resp. atributy objektu jsou obvykle odkazem na hash. Není to sice pravidlem a teoreticky objekt může být třeba i odkazem na pole, ale toho se využívá ze zřejmých důvodů jen sporadicky.

A nakonec metody jsou obyčejné podprogramy. Prvním argumentem každé metody je odkaz (platí u metody objektu) nebo jméno třídy - balík (u metody třídy).

Vytvoření objektu

Všechnu činnost při vytváření objektu obstarává funkce bless. Ta přijme jako parametr odkaz (obvykle na hash) a označí tento odkaz jako instanci aktuální třídy. Funkci bless lze předat jako další parametr i jméno třídy, se kterou se má objekt svázat.

```
$objekt = {};          #$objekt odkazuje na anonymní hash;
                       #zatím není objektem, ale pouze odkazem

bless $objekt, "Trida"; #$objekt je svázán s třídou Trida
bless $objekt;          #$objekt je svázán s aktuální třídou (např. main)
```

Vzpomeňme si nyní na funkci ref. Pokud je jejím argumentem objekt, vrací jméno třídy, ke které je tento objekt instancí.

```
$objekt = {};
bless $objekt, "Trida";
print ref $objekt; #tiskne Trida
```

Volání metod

Objekt může využívat metody své třídy (nebo později tříd, z nichž tato třída dědí). Metoda se volá takto.

```
$objekt->metoda(parametry);
```

Před všechny předané parametry je na 1. místo implicitně uveden odkaz na objekt, jehož metodu voláme.

Speciální metody - konstruktor a destruktor

Konstruktor je speciální metoda, která je důležitá při vzniku instancí. Konstruktor se obvykle nazývá new. V Perlu to však není pravidlem. Narozdíl od jiných jazyků, jako například C++, lze konstruktor nazvat jakkoliv a dokonce jich může být i více.

Konstruktor vytvoří hash, přiřadí objekt dané třídě a případně inicializuje prvky hashe.

Destruktor je další speciální metodou, která je volána při zániku objektu. Má pevný název DESTROY a je volána vždy, když program opouští rozsah platnosti tohoto objektu. Protože však v Perlu nemusíme dealokovat paměť, což je obvykle hlavní úloha destruktoru, nevyskytují se metody DESTROY až tak často.

Perl (59) - OOP - typické použití

Nyní, když už známe filozofii objektově orientovaného programování, se podíváme na to, jak napsat a používat objektově orientovaný modul.

V úvodním dílu o objektově orientovaném programování jsme se seznámili se základními myšlenkami a z dálky i se syntaxí. Dnes navážeme a ukážeme si postup, jak objekt, potažmo objektový program napsat.

Modul FileAccess pro snadnou práci se soubory

Vlastnosti OOP si předvedeme na třídě FileAccess, která implementuje objektové rozhraní pro práci se soubory. To znamená, že vytvoříme modul FileAccess, pomocí kterého pak budeme moct se souborem manipulovat bez starostí s vytvářením ovladačů a funkcemi, které čtení a zápis zajišťují.

Co to znamená? Uveďme si na příkladu, čeho vlastně budeme chtít dosáhnout. Nejprve vytvoříme objekt, který bude mít na starosti konkrétní soubor. Pak pro přečtení 10 znaků z daného souboru od aktuální pozice budeme psát jen toto.

```
$objekt->cti(10);
```

Podobně budeme chtít napsat metody zapis, pripis a pro nastavení pozice, kde se má číst, metodu nastav_pozici. Narozdíl od klasických modulů nebude potřeba vytvářet rozhraní pomocí Exporteru. V objektových modulech k tomu není důvod, protože všechny podprogramy se využívají jako metody.

Celý modul se tedy bude sestávat pouze z konstruktoru a ostatních metod. Pro shrnutí uveďme předpokládanou kostru programu.

```
package FileAccess;
use strict;
sub new {...}
sub pripis {...}
sub zapis {...}
sub cti {...}
sub nastav_pozici {...}
1;
```

Konstruktor

Ze všeho nejdříve vytvoříme konstruktor. Při vytváření objektu bude třeba zadat jméno souboru. Toto jméno pak bude v proměnné @_. Společně s ním tam implicitně bude ještě na prvním místě seznamu jméno třídy (balíku) - tedy v našem případě "FileAccess". Toto jméno je sem přidáno samovolně a při volání metod se nezadává.

```
package FileAccess;
use strict;

sub new {
my($self, $file) = @_;
...
}
```

V proměnné $self (to je konvenční název) budeme mít jméno balíku, tedy "FileAccess"a $file bude obsahovat název souboru, který budeme znát až v době použití souboru.

Kdybychom zapsali úvodní řádek konstruktoru následovně, tj. kdybychom nepočítali s implicitně předávaným jménem balíku, pak by v proměnné $file byla hodnota "FileAccess".

```
my($file) = @_;
```
Pojďme dokončit konstruktor. Vytvoříme hash, funkcí bless ho označíme jako objekt a poté již můžeme hash plnit výchozími hodnotami. Zatím budeme potřebovat pouze jedinou položku a to jméno souboru. Když jsme hotovi, vrátíme odkaz na hash funkcí return. Podobně vypadá většina konstruktorů.
```
sub new {
my($self, $file) = @_;
my $o = {};
bless $o;
$o->{"file"} = $file;
return $o;
}
```
Konstruktor máme hotov. Ještě jednou si stručně uveďme, jak nám vlastně poslouží. Konstruktor bude první metoda, kterou zavoláme, až budeme modul FileAccess používat v programu. Pomocí něj vytvoříme samotný objekt, který bude asociován s konkrétním souborem.

Ostatní metody

Nyní je třeba vytvořit další 3 metody. Pro zápis, pro přípis a pro čtení.

Nejdříve implementujeme metody pro zápis a přípis. Jako argument budou obě metody přijímat nějaký řetězec, který bude zapsán, resp. připsán do souboru. Prvním argumentem bude opět odkaz na objekt, ke kterému volání metody patří. To je velmi důležité, protože právě pomocí tohoto odkazu budeme přistupovat k datům objektu, tedy k onomu hashi, který byl vytvořen konstruktorem (konkrétně pro nás to znamená, že tak získáme název souboru, kam budeme zapisovat).

Obě metody budou, pomineme-li názvy metod, až na jediný znak (režim otevření souboru) úplně stejné. Uvnitř podprogramu se neděje nic převratného. Vytvoří se pouze ovladač souboru, zapíše se do něj a zase se uzavře. Pokud otevření selže, vykonávání podprogramu přerušíme, takže k zápisu nedojde. Uvedme si obě metody.
```
sub zapis {
my($self, $text) = @_;
my $f = open FILE, ">", $self->{"file"} or return;
print FILE $text;
close FILE;
return 1;
}

sub pripis {
my($self, $text) = @_;
my $f = open FILE, ">>", $self->{"file"} or return;
print FILE $text;
close FILE;
return 1;
}
```
Metoda pro čtení bude nejsložitější, ale jen z technického hlediska. Musíme brát ohled na počet znaků, které chce uživatel přečíst a také na startovní pozici. Právě kvůli pozici budeme potřebovat ještě další metodu nastavPozici a do hashe s daty objektu musíme přidat položku pozice. Počáteční pozice bude 0, takže do konstruktoru připíšeme následující řádek.
```
$o->{"pozice"} = 0;
```
Nyní můžeme napsat metodu pro nastavení pozice.
```
sub nastavPozici {
my($self, $pozice) = @_;
$self->{"pozice"} = $pozice;
}
```
To je velmi jednoduchá metoda a právě na ní je hezky vidět jak se s objektem pracuje. Zavoláním této metody nad příslušným objektem dojde ke změně položky "pozice" v tomto objektu (tedy hashi).

A jako poslední metodu napíšeme již zmiňovaný podprogram cti. Jako argument přijme počet znaků od dané pozice v souboru, které budou vráceny. Takže otevřeme soubor, funkcí seek nastavíme pozici na hodnotu, kterou uchováváme v hashi, od této pozice přečteme daný počet znaků a tyto znaky vrátíme funkcí return.
```
sub cti {
my($self, $pocet_znaku) = @_;
my $precteno;
my $f = open FILE, $self->{"file"} or return;
seek FILE, $self->{"pozice"}, 0;
read FILE, $precteno, $pocet_znaku;
$self->{"pozice"} += $pocet_znaku;
return $precteno;
close $f;
}
```
Použití modulu

Soubor FileAccess.pm je hotov. Nyní ho můžeme používat jako modul v jiných programech. Pokud je umístěn v některé z lokací v @INC.
```
use FileAccess;
```
Zavoláme konstruktor, který vrací odkaz na objekt. Jako parametr bere jméno souboru.
```
my $soubor1 = FileAccess->new("data1");
```
Tímto způsobem můžeme vytvořit libovolné množství instancí třídy FileAccess (tj. objektů). Pro každý soubor se vytvoří jedna.
```
my $soubor2 = FileAccess->new("data2");
```
A teď můžeme zapisovat a číst soubor data1 resp. data2 pomocí objektu $soubor1 resp. $soubor2, jak potřebujeme. Uveďme několik ukázkových volání.
```
$soubor1->zapis("Zapisuji text do souboru data1\n");
$soubor1->pripis("Pripisuji text do souboru data1\n");
$soubor2->zapis("Zapisuji text do souboru data2\n");
$soubor2->zapis("Zapisuji novy text do souboru data2\n");
```

```
print "1. cteni:", $soubor1->cti(10), "\n"; #cteme 10 znaku z data1 od pozice 0
print "2. cteni:", $soubor1->cti(30), "\n"; #cteme 30 znaku z data1 od pozice 10
print "3. cteni:", $soubor2->cti(10), "\n"; #cteme 10 znaku z data2 od pozice 0
$soubor1->nastavPozici(5);                  #pozice pro data1 je 40, zmenime na 5
print "4. cteni:", $soubor1->cti(15), "\n"; #cteme 15 znaku z data1 od pozice 5
```

Na okraj poznamenejme, že pro každý zápis je soubor otevírán stále znovu. To je neefektivní v případech, kdy je zapisováno (nebo čteno) opakovaně z jednoho souboru (například v nekonečném cyklu). Nicméně je to pohodlné pro uživatele modulu.

Perl (60) - OOP - dědičnost

Dědičnost umožňuje vytvářet mezi třídami vztahy, díky nimž se pak celý systém tříd stává daleko přehlednějším.

Biologicky řečeno, dědičnost (inheritance) je nějaký děj, ve kterém organizmus získá vlastnosti svého rodiče. Když zanedbáme evoluci (chyby dědičnosti) a druhy organizmů budeme považovat za dané, pak vztahy mezi druhy vytvářejí představu dědičnosti v objektově orientovaném programování.

Dědičnost tedy znamená, že skupina tříd je organizována ve stromu na základě vzájemné podobnosti - podobnosti v tom smyslu, že jedna třída je nějakým zobecněním druhé. Pak první třídu nazveme potomkem druhou rodičem. V takovém případě se třída potomka nepíše znovu, ale je prohlášena za dědící od rodiče a získá automaticky jeho vlastnosti, ke kterým si pak může přidat další.

Máme-li podobné třídy, které ale nemají vztah dědičnosti, pak je možné, že společně dědí od jiné třídy. Třeba i třídy, která nemusí mít smysl, tj. u které nemá smysl hovořit o instancích. V takovém případě hovoříme o tzv. abstraktní třídě. Význam abstraktních tříd uvidíme na následujícím příkladu.

Příklad - vlastnosti dědičnosti, abstraktní třídy

Pojďme se ještě na chvíli vrátit k biologii. Pokusíme se srovnat dědičnost s biologickým členěním organizmů a ukázat si na větším teoretickém příkladě, jak vlastně dědičnosti využít.

Organizmus obecně bude popisovat nějaká třída Organizmus. Její definice musí být natolik obecná, aby vyhovovala naprosto všem organizmům. Dejme tomu, že všechny organizmy musí dýchat a definujme metodu dýchej(). Každý organizmus je v určitém okamžiku na nějaké pozici souřadného systému a tak má svůj atribut pozice. Třída Organizmus je abstraktní, protože nemá smysl vytvářet objekty typu Organizmus.

První specializací třídy Organizmus je dělení na živočichy, rostliny, houby apod. Pro každou tuto kategorii bude existovat jedna třída. O každé z těchto tříd (Živočich, Rostlina, Houba) můžeme říci, že je organizmem. Mají vlastnosti Organizmu (dýchej() atd.) a k tomu si přidávají některé svoje další. Například v třídě Rostlina bude implementována metoda fotosyntéza() nebo Živočich se bude moci přijímat potravu metodou přijmi_potravu(). Každý živočich bude mít orgán na vnímaní světla a tedy i definován atribut ostrost_zraku. Toto všechno jsou také abstraktní třídy.

A takto můžeme jít stále dál. Z třídy živočich dědí další třídy jako Pták, Savec nebo Plaz. Třída Pták sdílí všechny vlastnosti třídy Živočich (přijmi_potravu(), dýchej() atd.) a přidává si jiné, například metodu leť(x, y, z).

Udělejme poslední krok, abychom se dostali z oblasti abstraktních tříd. Z třídy pták dědí třídy Kos a Vrabec. Nyní můžeme vytvářet objekty, které jsou instancemi těchto tříd. A právě zde uvidíme vlastnosti dědičnosti. Objekt $vrabec1 i objekt $lípa1 mohou dýchat. Přesto jsme nemuseli metodu dýchej() od základů definovat vícekrát. Dýchání je obecná funkce třídy Organizmus, z níž oba naše objekty dědí. Oba objekty také mají v souřadném systému nějakou pozici. Ale ostrost_zraku $lípa1 už nemá a naopak $vrabec1 neumí fotosyntézu.

Vícenásobná dědičnost

Dědičnost lze rozdělit na jednoduchou a vícenásobnou. Liší se v tom, od kolika tříd je děděno. Pokud naše třída dědí z jedné třídy, jde o jednoduchou dědičnost. Pokud jich je více nazývá se vícenásobná. Opět lze hledat analogie v biologii, tentokrát uvnitř jednoho druhu. Některé druhy organizmů mohou mít více než jednoho rodiče (tedy obvykle dva).

Perl podporuje oba typy dědičnosti, ale u všech objektových jazyků tomu tak není. Některé podporují pouze jednoduchou. Vícenásobná dědičnost má své zastánce i odpůrce. Pravdou je, že se s ní člověk příliš často nesetká, ale na druhou stranu ji, pokud je potřeba, nelze jen tak něčím nahradit.

Dědičnost v Perlu

Formální označení třídy jako potomka jiné třídy obnáší jedinou věc. Rodičovskou třída je třeba v odvozené třídě označit pomocí pole @ISA (anglické "is a" vyjadřuje vztah náležení). Přesněji řečeno, toto pole obsahuje jména všech tříd, ze kterých aktuální třída dědí, což je obvykle (tedy v případě jednoduché dědičnosti) jedna třída. Rodičovská třída může obsahovat pole @ISA také (jinak řečeno může být zároveň potomkem ještě obecnější třídy). Tímto způsobem pak můžeme tvořit různé hierarchické vztahy. Ovšem většinou budeme chtít provádět i jiné věci. Nezbytné bude zejména předefinovávání metod předků. Pojďme se podívat na další syntaktické jevy u dědičnosti a v dalším díle si ukážeme jejich užití.

Třída SUPER

SUPER je speciální třídou, která zastupuje předka. Zavoláním třídy SUPER se rekurzivně spouští hledání metody v rodičovských třídách. SUPER::new má tedy v případě jednoduché dědičnosti ve výsledku podobný smysl jako Předek::new. SUPER::new použijeme, pokud nechceme použít konkrétní třídu, ale nechat si vybrat podle hierarchie definované v @ISA.

Využití SUPER je poměrně široké. Často se s ní setkáme v konstruktorech potomků. Voláním metody SUPER::new totiž vyvoláváme konstruktor předka, který vytvoří objekt, a ten pak pomocí new upravíme.

Předefinování metody

Metoda třídy může upravovat metody tříd, ze kterých tato třída dědí. Ukažme si nesmyslnou, ale jednoduchou ukázku. Máme třídu A, ve které je definována metoda tiskni_atribut. Ta tiskne aktuální hodnotu konkrétního atributu. Třída B, která dědí z A bude chtít metodu tiskni_atribut používat, ale jiným způsobem. Tisknout se bude pouze v případě, že atribut je posloupností číslic. V opačném případě vytiskneme chybovou hlášku. Takto bude vypadat metoda B::tiskni_atribut.

```
sub tiskni_atribut {
    my $self = shift;
if ($self->{"atribut"} =~ /^\d+$/){
    die "Atribut musi byt cislo!\n";
    }
    $self->SUPER::tiskni_atribut();
    }
```

V příštím díle se předefinováváním budeme zabývat podrobněji.

Názvy dědících tříd

Další věcí, která stojí za pozornost, je pojmenování třídy za pomoci čtyřtečky ::. Tento symbol má dva významy. Už z názvu třídy je z takového zápisu patrný příbuzenský vztah. Čtyřtečka si také vynucuje přehlednější adresářovou strukturu. Modul Math::Functions se bude ve skutečnosti hledat pod názvem Math/Functions.pm.

Univerzální třída a její metody

Z třídy s názvem UNIVERSAL dědí všechny třídy mimo UNIVERSAL. UNIVERSAL je jediná třída, která nemá předka. To má několik zajímavých důsledků - například to, že každá námi definovaná třída bude mít alespoň jednoho předka. UNIVERSAL obsahuje tři metody. Ihned z definice třídy UNIVERSAL plyne, že tyto metody zdědí každá naše třída. Pojďme se na ně tedy podívat.

Metoda can

Tato metoda slouží ke zjištění, zda lze nad daným objektem volat danou metodu. Jako argument předáme metodě can jméno metody. Pokud taková metoda skutečné existuje (lze ji volat nad daným objektem bez použití AUTOLOAD), can vrací pravdivou hodnotu.

```
print "ANO" if Trida->can("delej"); #existuje ve třídě Trida metoda delej?
print "ANO" if $trida->can("delej");#může objekt $trida volat metodu delej?
```

Metoda isa

Pokud je argument platným názvem třídy, ze které třída, nad níž je metoda isa volána, dědí, je vrácena pravdivá hodnota.

```
print "ANO" if Rostlina->isa("Organizmus");#dědí třída Rostlina z třídy Organizmus?
print "ANO" if $trida->isa("Organizmus");  #dědí objekt $trida z třídy Organizmus?
```

Metoda VERSION

Ve všech modulech existuje speciální proměnná $VERSION. Do ní můžeme přiřadit jakékoliv desítkové číslo, které označuje verzi modulu.

```
our $VERSION = 3.3;
```

Uživatel modulu si nyní může stanovit jeho nejnižší verzi, která je pro něj únosná. Pokud bude chtít verzi minimálně 2.3, zadá pro import tento příkaz.

```
use Modul 2.3;
```

To, že předchozí řádek funguje, obstarává právě metoda VERSION. Je-li zavolána, ukončí program, pokud je verze zadaná jako argument větší než dostupná verze modulu.

Systém hledání volané metody

Pojďme si stanovit přesná pravidla, jakými se řídí hledání metody po jejím zavolání. Budeme uvažovat obecný případ, tedy vícenásobnou dědičnost.

Ze všeho nejdříve je hledána metoda v třídě, jehož je náš objekt instancí. Pokud taková třída neexistuje, prohledávají se rekurzivně všechny rodičovské třídy, podle toho, v jakém pořadí jsou v poli @ISA uvedeny. Třída UNIVERSAL je implicitně uvedena vždy naposled. Pokud není metoda nalezena, hledá se stejným způsobem speciální metoda AUTOLOAD. Pokud žádná vyhovující metoda není nalezena, dojde k chybě.

Ukažme si pro lepší představu konkrétní příklad. definujeme si třídu Smrk.

```
package Smrk;
@ISA = qw(Strom OkrasnaDrevina);
```

Nyní nad třídou Smrk nebo její instancí zavoláme metodu metoda. Smrk::metoda tedy bude hledána postupně tak, jak je uvedeno v následujícím seznamu, dokud některá z nich nebude nalezena.

1. Smrk::metoda
2. Strom::metoda
3. metoda rekurzivně v třídách z nichž Strom dědí
4. OkrasnaDrevina::metoda
5. metoda rekurzivně v třídách z nichž OkrasnaDrevina dědí
6. UNIVERSAL::metoda
7. Smrk::AUTOLOAD
8. Strom::AUTOLOAD
9. AUTOLOAD rekurzivně v třídách z nichž Strom dědí
10. OkrasnaDrevina::AUTOLOAD
11. AUTOLOAD rekurzivně v třídách z nichž OkrasnaDrevina dědí
12. UNIVERSAL::AUTOLOAD

Perl (61) - OOP - přínos a užití dědičnosti

Dnes si ukážeme hlavní výhody dědičnosti a srovnáme její použití s jinými postupy.

Vlastnosti dědičnosti si ukážeme na následující třídě. Implementujeme si třídu, která bude popisovat majitele mobilního čísla. Nejdříve si zkusíme ukázat, jak by to vypadalo bez dědičnosti, poté ji zapojíme a porovnáme výsledky. Kód nebudeme rozepisovat do detailů, půjde nám pouze o myšlenku.

Co tedy bude hlavním ukolem? Musíme nějak oddělit zákazníky s předplacenou službou od tarifních zákazníků, kteří platí pravidelně za každé zúčtovací období paušální poplatek.

Dva konstruktory

První možností, jak daný úkol řešit, je použití jednoho druhu objektu pro oba typy zákazníků. Tedy budeme mít třídu Zakaznik a vytvoříme konstruktor. Vytvoříme dva konstruktory - jeden pro každý typ zákazníka.

```
package Zakaznik;
sub new_predplacena_sluzba {
my($cislo, $id_majitele, $stav_konta, $cena_za_minutu) = @_;
}

sub new_tarifni_program {
my($cislo, $id_majitele, $stav_konta, $pausal, $volnych_minut,
$cena_za_minutu_po_vycerpani_volnych_minut) = @_;
}
```

Poznámka - Teoreticky by v takto zjednodušeném příkladě mohl být zákazník s předplacenou službou speciálním případem tarifního zákazníka s nulovým paušálem a bez volných minut. V tomto a následujících příkladech půjde hlavně o názornost.

Užití dvou konstruktorů samo o sobě není chybou. V našem případě je ale problém v tom, že nezůstane jen u toho. Co když budeme chtít vytvořit funkci, která počítá cenu za daný počet provolaných minut? Buď bychom museli uchovávat další atribut a složitě ošetřovat podmínky nebo bychom mohli napsat dvě zvláštní metody.

```
sub spocitej_cenu_u_predplacene_sluzby {...}
sub spocitej_cenu_u_tarifniho_programu {...}
```

Chybou je už to, že uživatel tohoto modulu musí po vytvoření objektu vědět, s jakým typem zákazníka pracuje. Pokud by zavolal funkci spocitej_cenu_u_predplacene_sluzby nad tarifním zákazníkem, došlo by k chybě. Uživatel by to musel pravděpodobně řešit podmínkou a to je znak špatně napsaného modulu.

Dalším problémem je, že funkcí typu spocitej_cenu, které fungují pro každý typ zákazníka na jiném mechanizmu by mohlo být mnohem více. Stejně tak pokud by přibyl další typ zákazníka - například program pro pracovníky mobilního operátora. Pak by nám počet metod vzrůstal lineárně. Takové řešení zdaleka není to pravé.

Dvě nezávislé třídy

Možnost vytvořit místo dosavadního řešení dva nezávislé moduly se zdá být poměrně moudrou myšlenkou. Rozeberme si podrobněji klady a zápory této volby.

```
package Zakaznik_predplacena_sluzba;
sub new {
my($cislo, $id_majitele, $stav_konta, $cena_za_minutu) = @_;
}

sub spocitej_cenu {...}


package Zakaznik_tarifni_program;
sub new {
my($cislo, $id_majitele, $stav_konta, $pausal, $volnych_minut,
$cena_za_minutu_po_vycerpani_volnych_minut) = @_;
}

sub spocitej_cenu {...}
```

Rozdíl mezi oběma postupy je okamžitě patrný. Velká výhoda druhého řešení spočívá v tom, že neklade na uživatele modulu nutnost pamatovat si ke každému zákazníkovi i jeho typ a ani nebudeme muset řešit žádné podmínky. Tím odpadá největší problém prvního postupu.

Ovšem přijatelné toto řešení rozhodně není. Odstranění zmíněného problému s sebou přineslo problém jiný. Oba moduly mají společné některé některé atributy ($cislo, $id_majitele, $stav_konta) a metody (například dej_cislo, zmen_majitele, navys_stav_konta apod.). Co když přibude atribut $datum_zalozeni_uctu? Budeme muset upravit oba balíky. A co když nepůjde jen o 2 balíky, ale o 20 nebo více balíků? Jak vidíme, ani rozdělení do nezávislých modulů není elegantním řešením.

Abstraktní třída a dědičnost

Zopakujme si, čeho vlastně chceme docílit. Potřebujeme vytvořit objektově orientovaný modul mobilních zákazníků s těmito vlastnostmi.

- Modul nesmí svému uživateli klást jakékoliv nároky. Pokud nad objektem zavoláme metodu spocitej_cenu, modul ji spočítá bez ohledu na to, jestli uživatel ví, ke kterému balíku objekt patří.
- Atributy a metody, které jsou společné pro všechny typy zákazníků musí být definovány pouze na jediném místě, nehledě na to, o který typ zákazníka jde.

Protože tytéž atributy nesmějí být ve více třídách, budeme problém řešit pomocí abstraktní třídy. Půjde o to vytvořit třídu obecného mobilního zákazníka, který bude obsahovat jen vlastnosti společné všem zákazníkům. Soubor Zakaznik.pm vypadá takto.

```
package Zakaznik;
sub new {
my($pkg, $cislo, $id_majitele, $stav_konta) = @_;
my $o = {};
bless $o, $pkg;
$o->{"cislo"} = $cislo;
$o->{"id_majitele"} = $id_majitele;
$o->{"stav_konta"} = $stav_konta;
return $o;
}

sub dej_cislo {...}
sub zmen_majitele {...}
sub navys_stav_konta {...}
```

Vše, co charakterizuje obecného zákazníka, je definováno zde.

Dále můžeme tvořit už konkrétní typy zákazníků. Nejdříve zákazníka s předplacenou službou (Zakaznik/PredplacenaSluzba.pm). Zde předefinujeme konstruktor tím způsobem, že nejprve vytvoříme objekt obecného zákazníka a poté ho upravíme.

```
package Zakaznik::PredplacenaSluzba;
use Zakaznik;
use vars qw(@ISA);
@ISA = qw(Zakaznik);

sub new {
my($pkg, $cislo, $id_majitele, $stav_konta, $cena_za_minutu) = @_;
my $o = $pkg->SUPER::new($cislo, $id_majitele, $stav_konta);
$o->{"cena_za_minutu"} = $cena_za_minutu;
return $o;
}
```

```
sub spocitej_cenu {...}
```
Ukažme si ještě, jak by mohl vypadat zákazník s tarifním programem (Zakaznik/TarifniProgram.pm).
```
package Zakaznik::TarifniProgram;
use Zakaznik;
@ISA = qw(Zakaznik);

sub new {
my($pkg, $cislo, $id_majitele, $stav_konta, $pausal,
$volnych_minut, $cena_za_minutu_po_vycerpani_volnych_minut) = @_;
my $o = $pkg->SUPER::new($cislo, $id_majitele, $stav_konta);
$o->{"pausal"} = $pausal;
$o->{"volnych_minut"} = $volnych_minut;
$o->{"cena_za_minutu_po_vycerpani_volnych_minut"} =
$cena_za_minutu_po_vycerpani_volnych_minut;
return $o;
}

sub spocitej_cenu {...}
```
Uživatel bude mít nyní k dispozici moduly Zakaznik::PredplacenaSluzba a Zakaznik::TarifniProgram, které lze do programu importovat pomocí use. Takto vypadá použití vytvořeného modulu.
```
my $zakaznik = Zakaznik::PredplacenaSluzba->new(...);
my $cena = $zakaznik->spocitej_cenu();
```
Splnili jsme tedy oba požadavky. Náš model je přijatelný z hlediska uživatele - ten s ním může intuitivně pracovat; i z hlediska programátora - neměl být velký problém s rozšiřováním modulu.

Perl (62) - OOP - přetěžování

Podíváme se na to, jak lze stávající operátory naučit pracovat s novými datovými typy. Taktéž si ukážeme přetěžování konstant.

Perl je poměrně tolerantní vůči datovým typům operandů. Ale pouze do jisté míry. Lze sčítat čísla a řetězce, ale už ne objekty. Přetěžování je cestou, jak to vestavěné operátory naučit.

To je velmi užitečné pro snažší manipulaci s objektem. Představme si nějakou datovou strukturu, se kterou má smysl provádět nějakou operaci. Vezměme si jako příklad matice. Přetěžování nám zajistí, že dva objekty $a a $b typu Matice budeme sčítat příkazem $a + $b, tisknout pomocí print v přehledné tabulce nebo počítat determinant jako abs($a).

Potřeba je k tomu pragma overload a napsání několika podprogramů na obsloužení operátorů.

Syntaxe přetěžování

Pro aktivaci přetěžování je potřeba zavolat pragmu overload s příslušnými parametry. Ty jsou tvořeny dvojicemi v následujícím formátu.
```
"operátor" => \&obslužný_podprogram
```
Uveďme si konkrétní příklad použití pragmy overload. Význam operátorů si vysvětlíme později.
```
use overload
"+"       => \&plus,
"-"       => \&minus,
"nomethod" => sub {die "To neumím."},
"fallback" => 0;
```
Jakmile nyní použijeme nad objektem operátor, vyvolá se podprogram, jež tento operátor reprezentuje, a jako parametry mu budou předány operandy prováděné operace (v případě unárního operátoru je druhá hodnota v poli parametrů nedefinovaná) a dále jejich pořadí (pořadí je pravdivá hodnota pro opačné pořadí a nepravdivá pro zachované pořadí).

Postup hledání podprogramu pro operátor

Podívejme se, co se děje po požadavku na provedení operace nad objektem.

1. Pokud byl pomocí pragmy overload nastaven pro danou operaci jako ovladač nějaký podprogram, pak se zavolá právě ten.
2. Pokud jsme prováděné operaci žádný ovladač nepřiřadili, zkusí Perl tento ovladač na základě ostatních námi definovaných operací sám vymyslet. Tento krok lze vynechat pomocí fallback.
3. Jako další v pořadí Perl zkouší volat metodu nomethod.
4. Pokud neuspějeme ani napotřetí, dojde k chybě.

Speciální operátory fallback a nomethod

fallback určuje, jak se bude postupovat v případě, kdy nebyl nalezen ovladač pro danou operaci. To, že Perl v druhém kroku hledání vhodného podprogramu zkouší vymyslet ovladač sám, nemusí být v našem zájmu. Proto je možné takové chování zakázat a druhý krok vynechat. Podívejme se, jaké máme možnosti pro nastavení fallback.

Hodnota	Popis
undef	Perl zkouší odvodit ovladač, až poté se volá nomethod a poté je vyvolána výjimka
defined true	stejné jako u undef, ale není vyvolána výjimka
defined false	Perl neodvozuje, hned je voláno nomethod a následuje výjimka

Funkce nomethod je volána tehdy, když není definována ani odvozena požadovaná operace. nomethod dostává stejné parametry jako ostatní operátory a navíc také operátor ve tvaru řetězce.

Seznam operátorů s možností přetížení

Podívejme se, které všechny operátory můžeme přetížit. Všechny takové jsou v proměnné %overload::ops roztříděny přehledně do kategorií. Podívejme se tedy na její obsah.
```
%overload::ops = {
'3way_comparison' => '<=> cmp',
'dereferencing'   => '${} @{} %{} &{} *{}',
'str_comparison'  => 'lt le gt ge eq ne',
'with_assign'     => '+ - * / % ** << >> x .',
```

```
'binary'          => '& | ^',
 'iterators'      => '<>',
'unary'           => 'neg ! ~',
'special'         => 'nomethod fallback =',
'num_comparison'  => '< <= >  >= == !=',
'assign'          => '+= -= *= /= %= **= <<= >>= x= .=',
'mutators'        => '++ --',
'func'            => 'atan2 cos sin exp abs log sqrt int',
'conversion'      => 'bool "" 0+'
};
```

Nyní se podívejme na vybrané operátory, které Perl buď umí odvodit nebo které si žádají jiný komentář.

Operátory	Popis	Automatické odvození
neg	unární minus	pomocí binárního -
.	zřetězení	pomocí ""
""	použití objektu jako řetězce (tisk, zřetězení, klíč v hashi)	
0+	použití objektu jako čísla (u aritmetického operátoru, index pole, při definici rozsahu)	
bool	použití objektu jako pravdivostního výrazu	
! not	negace	pomocí "", bool nebo 0+
+=		pomocí +
.=		pomocí "", .
++ --	inkrementace	pomocí += -=
< <= != == >= >	relace pro čísla	pomocí <=>
lt le gt ge eq ne	relace pro řetězce	pomocí cmp
abs		pomocí neg a < nebo <=>
=	kopírovací konstruktor, používá se automaticky například u inkrementace	

Přetěžování a dědičnost

Potomci dědí přetížené operace po svých předcích. Pravidla pro pořadí jsou stejnájako při volání metod.

Příklad - reprezentace vektorů

Uveďme si konkrétní příklad na přetěžování. Budeme mít objekt reprezentující vektor. Pro názornost to bude modul reprezentující pouze třídimenzionální vektory.

Budeme přetěžovat několik operací. Vytvoříme ovladače pro součet, rozdíl, skalární a vektorový součin, násobení skalárem, kopírování, tisk a normu vektoru.

Nejprve tedy uvedeme direktivu autoload s operacemi, které budeme chtít dodefinovat pro vektory.

```
package Vektor3D;
use overload
 '""'  => \&tisk,
"-"   => \&minus,
 "+"   => \&plus,
 "="   => \&copy,
"."   => \&skalarni_soucin,
"x"   => \&vektorovy_soucin,
 "*"   => \&nasobek,
"abs"   => \&absolutni_hodnota,
"nomethod" => sub {die "Operaci ", $_[3], " neumím."};
```

Nyní budeme implementovat jednotlivé metody. Je třeba si uvědomit, že téměř každá z předcházejících metod bude zároveň konstruktorem. Musí tedy vracet objekt vytvořený pomocí bless.

Nejprve vytvoříme klasický konstruktor new. Objekt budeme reprezentovat polem.

```
sub new {
my($pkg, $r_vektor) = @_;
return bless $r_vektor, $pkg;
}
```

Nyní vytvoříme ovladač pro tisk. Chceme, aby se vektor tiskl ve formátu (x;y;z). Vrátíme tedy zformátovaný řetězec.

```
sub tisk {
my($self) = @_;
return sprintf("(%i;%i;%i)", $$self[0], $$self[1], $$self[2]);
}
```

Násobení skalárem vytvoří z daného vektoru nový vektor, jehož složky budou vynásobeny skalární hodnotou. Protože výsledkem bude nový vektor, je třeba vrátit objekt. Ať je násobení zapsáno zleva nebo zprava, v podprogramu dostaneme jako první parametr vektor. Násobení skalárem je komutativní a nezajímá nás tedy třetí parametr, který udává pořadí.

```
sub nasobek {
my($u, $n) = @_;
return bless [$n*$$u[0], $n*$$u[1], $n*$$u[2]], ref $u;
}
```

Obdobně napíšeme také součet dvou vektorů.

```
sub plus {
my($u, $v) = @_;
```

```
return bless [$$u[0]+$$v[0], $$u[1]+$$v[1], $$u[2]+$$v[2]], ref $u;
}
```
Další metody fungují analogicky.
```
sub skalarni_soucin {
my($u, $v) = @_;
return bless [$$u[0]*$$v[0], $$u[1]*$$v[1], $$u[2]*$$v[2]], ref $u;
}

sub vektorovy_soucin {
my($u, $v) = @_;
return bless [$$u[1]*$$v[2]-$$u[2]*$$v[1], $$u[2]*$$v[0]-$$u[0]*$$v[2],
$$u[0]*$$v[1]-$$u[1]*$$v[0]], ref $u;
}

sub copy {
return bless $_[0], ref $_[0];
}

sub absolutni_hodnota {
my($u) = @_;
return sqrt($$u[0]**2 + $$u[1]**2 + $$u[2]**2);
}
```
Nyní máme k dispozici sadu operátorů pro manipulaci s vektory.
```
use Vektor3D;

my @p1 = (2, 3, 5);
my @p2 = (4, 8, 5);

$u = Vektor3D->new(\@p1);
$v = Vektor3D->new(\@p2);

my $w = $u + $v; #je vytvořen objekt $w = (6;9;10)
print $w;       #tiskne řetězec (6;9;10)
```
Všimněme si ještě automatického odvozování operátorů. Pro naše vektory lze využít i operaci +=, ač nebyla definována. Perl si ji totiž odvodil na základě naší definice operátoru +.

Přetěžování a konstanty

Pragma overload poskytuje také funkci constant, kterou využijeme k přetěžování konstant. To znamená možnost upravit jejich výstupní formát.

Podívejme se, které typy konstant je možné přetížit.

Konstanta	Popis
integer	celé číslo
float	desetinné číslo
binary	číslo v jiných číselných soustavách
q	řetězec
qr	regulární výraz

Opět budeme definovat nové metody, které ve výsledku upraví výsledný formát konstanty. Zde se to dělá uvnitř funkce import v modulu, kde chceme přetížení zavést. Opět daným typům konstant přiřadíme obslužné podprogramy.
```
package Pretizeni;
use overload;

sub import {
overload::constant
"float" => sub {...},
"q"     => sub {...}
}

1;
```
Obslužné podprogramy dostanou jako parametry implicitně hodnotu konstanty, jak byla zapsána a hodnotu, jak ji zpracuje Perl. Pro konstanty typu q a qr se předává ještě 3. parametr a místo použití. Ten může nabývat těchto hodnot.

Hodnota	Popis
tr	řetězec použitý u nahrazení u operátorů tr, y
s	řetězec použitý u nahrazení u operátoru s
qq	řetězec, ve kterém je symbol s expanzí
q	řetězec, ve kterém není symbol s expanzí

Jako příklad si napišme dvě metody. První bude zajišťovat, že u desetinných čísel se bude tisknout místo desetinné tečky desetinná čárka. Na druhé si ukážeme, jak a na kterých místech Perl zpracovává řetězce. Tento podprogram s řetězcem nic dělat nebude, ale na místech vyhodnocování vytiskne tento řetězec a typ použití. Uveďme si, jak tedy bude vypada funkce import.
```
sub import {
overload::constant
"float" => sub {
```

```
$_ = shift;
s/\./,/i;
return $_;
},
"q" => sub {
print "@_\n";
return shift;
}
};
```

Nyní můžeme modul použít a zkusit nějaký testovací program. V tom našem dojde celkem 4× ke zpracování řetězce.

```
use Pretizeni;
$x='bez expanze $x'."s expanzi $x";
$x=~tr/abcd/ABCD/;
print 1.59;
```

Pokud by někoho zajímaly další příklady, pak na perl.com vyšel hezký článek o reprezentaci zlomků. Řada dalších vlastností přetěžování je také v dokumentaci.

Perl (63) - OOP - závěr

Dnešní díl zakončí sérii článků o objektově orientovaném programování. Podíváme se na to, co se nevešlo do předchozích pěti dílů.

Dnes se rozloučíme s objektově orientovaným programováním tím, že probereme některé dosud nezmíněné aspekty. Ovšem nebude to rozlučka dlouhá, neboť od příštího dílu si uvedeme několikadílný (a objektový) rozsáhlejší příklad.

Přímý přístup uživatele modulu do objektu

Perl narozdíl od některých jiných jazyků umožňuje přistupovat k atributům třídy z programu i bez přístupové metody.

```
$o = Trida->new();
print $o->{"atribut"};
$o->{"atribut"} = "nova hodnota";
```

Tento způsob přístupu k atributům bychom za žádných okolností neměli používat, neboť to jde proti myšlence objektově orientovaného programování.

Proč je to tedy tak nebezpečné? Vnitřní strukturu objektu by uživatel modulu teoreticky vůbec neměl znát, resp. měl by s ním zacházet tak, že ji nezná. Implementace modulu včetně složení a vůbec existence atributů zkrátka není jeho věc. Pro práci s objektem tady je rozhraní, které by mělo být jedinou možností, jak s objektem komunikovat. V opačném případě půjde o narušení zapouzdření.

Toto je důvod, proč přímo k datům modulu nepřistupovat. Může mít velmi nepříjemné důsledky. Uveďme si ty nejčastější.

- K zajištění konzistence dat jde nezřídka ruku v ruce s změnou atributu vyvolání nějaké jiné akce. Typicky může jít o zápis do databáze. Pokud bychom přistupovali k atributu přímo, pak se může stát, že tato akce provedena nebude.
- Dále je také možné, že v atributech je ukládáno něco jiného, než co chceme zapisovat nebo je to ukládáno v jiné formě. To se může občas stát například u uchovávání různých veličin. Vezměme si metodu nastav_rychlost(36), kde 36 je hodnota v km/h. Ale co když je v objektu uchovávána hodnota v m/s, tedy 10?
- Pokud není modul který používáme zrovna mrtvý, může se stát, že se v příští verzi modulu změní jména atributů nebo celá struktura hashe. Potom je nutné přepsat nejen modul, ale i programy, které přistupují k jeho atributům přímo. V případě respektování myšlenky OOP k tomu však nedojde

Závěrem lze říci, že přístup k atributům dobře napsaného objektu by nikdy nemělo dojít. U těch atributů, kde to má smysl totiž vždy bude existovat nějaká metoda typu dej_atribut nebo zmen_atribut, které potřebné operace zajistí.

Soukromí

Soukromí modulu je množina dat, ke kterým uživatel modulu nemá přístup. Jiné jazyky jako C++ umožňují explicitně označit vybraná data jako soukromá - tedy neposkytnout je uživateli.

Smysl toho všeho lépe pochopíme na následujícím příkladu. Modul, který bude poskytovat uživateli služby s manipulaci s databází si bude uvnitř definovat funkci is_in_array. Ta dělá pouze pomocnou činnost, která nemá co dělat v sadě nástrojů pro práci s databází. Proto autor modulu označí funkci jako soukromou.

Perl toto neumožňuje. Všechna data, která jsou dostupná v tabulce symbolů, může uživatel získat. Nerozlišují se soukromá a veřejná data.

Na tomto místě je ale třeba říci, že zvykem dobře vychovaného uživatele je nepoužívat metody, které nejsou zmíněny v dokumentaci. Důvody jsou podobné jako důvody proti přístupu k atributům. Je to tedy hlavně v jeho zájmu.

Jak bylo řečeno, Perl nepodporuje soukromá data. Pokud však opravdu trváme na tom, aby uživatel neměl vůbec žádnou možnost zasáhnout do vybraných dat, lze to zařídit jednoduše tak, že dotyčné metody nebudou dostupné v tabulce symbolů. Jinými slovy, je třeba tyto metody definovat v lexikálním prostoru. Podívejme se stručně jak na to.

```
my $soukroma_metoda = sub {
...
}
```

Z jiných metod pak soukromou metodu voláme takto.

```
&$soukroma_metoda(parametry);
```

Alternativní syntaxe pro OOP

Někteří programátoři možná ocení další způsob zápisu, kterým lze pracovat s objekty.

```
$objekt = new Trida(parametry_konstruktoru);
metoda $objekt parametr1, parametr2, ..., parametrn;
```

Stanovení dědičnosti pomocí pragmy base

Pomocí base lze následujícím příkazem zajistit, že bude provedeno zavedení modulů předků v době kompilace.

```
use base qw(Trida);
```

Pro srovnání se podívejme na zápis, který ve výsledku udělá totéž.

```
BEGIN {
require Trida;
our @ISA;
@ISA = qw(Trida);
}
```

Objekt jako odkaz na skalár

Již jsme si ukázali objekt jako odkaz na hash, v minulém dílu i jako odkaz na pole, tak si uveďme ještě třetí variantu. Ta se používá když uchováváme jedinou položku. Tedy v našem případě regulární výraz, nad kterým můžeme provádět různé testy.

```
package TestRegex;

sub new {
my($pkg, $regex) = @_;
bless \$regex, $pkg;
}

sub test_rychlosti {...}
sub navrh_optimalizace {...}
```

Metoda AUTOLOAD

Připomeňme, že pokud není nalezena volaná metoda, automaticky se místo ní zavolá metoda AUTOLOAD.

```
sub AUTOLOAD {
return if $AUTOLOAD =~ /::DESTROY/;
print "Metoda $AUTOLOAD neni dostupna\n";
}
```

Další zdroje

Na internetu jsou o objektově orientovaném programování popsány gigabajty. Uveďme jen několik odkazů, kde hledat další informace na toto téma.

Téměř vše potřebné je k dispozici na manuálových stránkách. perlobj(1) se zabývá syntaxí, v perlbot(1) jsou některé triky a příklady, perltoot(1) a perltooc(1) jsou pak tutoriály. Kvalitním zdrojem je také tradičně Wikipedie, kde jsou uvedeny další odkazy.

Ovšem ani v tomto seriálu objektově orientované programování nekončí. Od příštího dílu po několik dalších začneme psát větší program, jehož hlavní součástí bude objektově orientovaný modul.

Perl (64) - Projekt - čtečka sportovních výsledků

Protože jsme se dosud zabývali výhradně teorií, věnujeme následujících 6 dílů na napsání snad prvního smysluplného programu v tomto seriálu.

Dosud jsme v seriálu uváděli pouze minimum příkladů. Byly to příklady pro příklady - tedy vesměs co nejstručnější vyumělkované programy bez většího smyslu. Jejich účelem bylo pouze demonstrovat určitou část syntaxe.

Dnes zahájíme práci na textové čtečce sportovních výsledků, což by mělo přinést zpetření seriálu.

Práci na každém větším projektu lze rozdělit do několika etap. Programování je pouze jednou z nich. Podle velikosti projektu se těmto etapám přisoudí důležitost, která se projeví zejména na časové organizaci. Čím větší projekt, tím je obvykle čas na programování v porovnání s analyzováním a přípravou menší. Na velikosti projektu také záleží to, z kterých fází se bude postup skládat.

Nás toto členění však zase tolik trápit nemusí, protože půjde pouze o poměrně malý projekt. Spoustu fází, jako například hardwarové nároky tak nemusíme řešit vůbec.

První věcí, kterou bychom si tedy nyní měli ujasnit je, co vlastně budeme psát.

Cíl projektu

Jak již bylo řečeno, napíšeme program pro čtení výsledků kopané. Výsledky budeme získávat extrakcí z vybrané internetové stránky. Zmiňme rovnou, že to bude napevno www.livescore.com. Toto řešení má sice několik zásadních nevýhod, ale ukážeme si na něm některé zajímavé postupy.

Prvotní představa o tom, co by náš program měl umět, je taková:

- vypsat všechny zápasy, které se hrají dnes (tedy v podstatě seznam zápasů na www.livescore.com/default.dll?page=home)
- vypsat všechny zápasy z národních resp. mezinárodních soutěží (získávat je budeme z www.livescore.com/default.dll?page=*jméno_soutěže*)
- každý zápas může být v době volání v jedné ze tří fází.
  1. zápas ještě nezačal
  2. zápas se právě hraje
  3. zápas se již dohrál

  Tyto fáze musíme ve výpisech zápasů zřetelně odlišíme
- u každého zápasu by měla být informace o názvu soutěže, údaj kdy zápas začne a pokud již začal tak buď průběžný nebo konečný výsledek.
- zobrazit detail vybraného zápasu - střelci branek, případně karty apod. (tedy to, co je dostupné na www.livescore.com/default.dll/Game?comp=*soutěž*&game=*id_hry*)
- sledovat zápas online, pokud se hraje

Použití programu by mělo být nanejvýš jednoduché a intuitivní. Jediným zadaným příkazem do shellu bychom měli obdržet požadovanou informaci. Abychom získali alespoň jistou představu, uveďme si, jak by taková volání mohla vypadat.

Program pojmenujeme live a bude se spouštět stejnojmenným příkazem. Bude k dispozici několik možností volání. Pro zobrazení zápasů, které se hrají dnes bude sloužit tento příkaz.

```
$ live
```

Pokud zadáme argument, vypíše se pouze zápasy, které vyhovují vzoru.

```
$ live manchester
```

Dále bude program přijímat několik argumentů z příkazového řádku. Pomocí volby -league nebo -l zadáme stát nebo mezinárodní soutěž, jejíž zápasy se mají vypisovat a to v takovém tvaru, v jakém ji uvádí livescore.com. Tedy například tento příkaz zobrazí zápasy v Anglii.

```
$ live -l england
```

Dále zde bude volba -o nebo -online, která bude zobrazovat detail zápasu. V případě, že se zápas hraje, spustí se online přenos. Pokud se již dohrál, zobrazí se střelci branek, průběh apod.

```
$ live -o
```

Pokud nezadáme žádný argument, program zobrazí seznam zápasů a zeptá se, který z nich chceme zobrazit. Zadávat mu budeme část názvu týmu, podle kterého již program požadovaný zápas určí. Pokud nebude zadání jednoznačné, zeptá se na další informaci. Pokud však hodnotu vložíme, program podle ní zápas určí automaticky.

```
$ live -o manchester
```

A nakonec zde bude volba -r nebo -refresh, pomocí které nastavíme sekundový interval pro aktualizaci při online přenosu. Nicméně protože nastravíme nějakou rozumnou implicitní hodnotu, bude používán jen zřídka.

```
$ live -r 30 -o manchester
```

Rozbor

Ponechme stranou pro nás irelevantní (jinak však nezbytné) otázky typu "v čem psát" a pojďme udělat několik důležitých rozhodnutí o tom "jak psát".

Nejprve poznámku o stylu programování. Program nebudeme ladit až k naprosté dokonalosti. Budeme dbát zejména na názornost. Pokud narazíme na problém, který bude chod programu ovlivňovat jen minimálně, ale jehož řešení by zabralo relativně mnoho času, pouze ho zmíníme, případně nastíníme řešení a většinou necháme být.

Koncepce

K návrhu programu použijeme objektově orientované programování. Program se bude sestávat z několika částí. Je třeba se rozhodnout pro konkrétní variantu.

Předně vytvoříme modul s názvem Livescore, který se bude starat o získávání a zpracovávání dat. Nebude řešit věci jako je zobrazování dat, data bude pouze poskytovat.

Zpracování dat můžeme udělat několika způsoby. Buď můžeme napsat mezičlánek mezi modulem Livescore a programem live, např. modul Livescore::Text, který bude řešit zobrazování dat podle požadavků programu live. Výhodou tohoto postupu je, že bychom později mohli dopsat např. moduly Livescore::Qt, Livescore::Curses apod., které bychom mohli snadno použít při tvorbě uživatelských rozhraní.

A nebo napíšeme pouze program live, který bude zpracovávat data od Livescore sám.

Druhá varianta je jednodušší a protože našemu zadání plně postačuje, zvolíme právě ji.

Modul Livescore

Co by měl umět modul? Shrňme požadavky na něj do několika bodů.

- Základním požadavkem na modul je získat data. Proto zde bude existovat funkce ziskej_zdrojovy_kod, která stáhne z www.livescore.com tu stránku, která bude zrovna požadována.
- S předcházejícím bodem tane na mysli další otázka. Otázka perzistence. Budeme stahovat pokaždé, když bude program volán nebo si stránku "někam" uložíme? Tento problém je třeba důkladně zvážit, neboť bude program pravděpodobně často volán opakovaně. Rozhodněme se pro následující řešení. Každý stažený soubor uložíme do adresáře /tmp a po každém volání se podíváme, zda náhodou již vhodný soubor nemáme. Budeme muset brát zřetel na stáří souboru. Další podotázka je, jak se budeme zbavovat starých tmp souborů? Vyřešíme to jednoduše. Nijak. Moc místa zabírat nebudou a čas od času každý stejně tempy maže...
- Dále by měl modul extrahovat data na základě zdrojového kódu. K tomu budeme muset pečlivě prostudovat HTML kódy z livescore.com.
- Další vlastností bude hledání zápasu podle daných kritérií.
- A na závěr se nabízí otázka, jak udělat online přenos? Protože o formu zobrazování dat se modul nestará, tak se tato otázka vyřeší sama. Odpovědnost za online přenos bude mít program live. Modul Livescore bude pouze na požádání posílat průběžná (nebo konečná) data.

Program live

Jak již bylo řečeno, live se stará o kontakt uživatelem a formu zobrazování dat. Vytvořme opět nějakou osnovu, na které budeme moci stavět. I když tentokrát nebude příliš konkrétní.

- Je třeba obsloužit volání uživatele.
- Na základě dat z modulu Livescore bude třeba zobrazit aktuální zápasy, případně detail zápasu. Online přenos bude řešen cyklickým doplňováním nových informací.
- V zadání projektu jsme rozdělili zápasy na 3 typy, podle toho, v jakém jsou stavu z hlediska časového postupu. Tyto druhy rozlišíme barvami.

Správa projektu

Budeme se zabývat pouze programováním. Nicméně nepíšeme program na 10 řádků a asi by tedy bylo žádoucí použít nějaký program na správu verzí.

Dokumentace

Použití našeho programu bude poměrně jednoduché, u modulu to bude složitější. Měli bychom vytvořit dvě manuálové stránky ve formátu POD. Jednu pro modul a další pro příkaz live. Dokumentací POD se v rámci seriálu ještě budeme podrobně zabývat. Co se týče komentářů v kódu, tak je samozřejmě dobré komentovat nejasná místa. Protože používáme objektově orientované programování, neměly by být u dobře navrženého projektu nároky na komentáře příliš vysoké. Nelze říct, že dobrý kód komentáře nepotřebuje, ale že v dobrém kódu je na první pohled jasné, co se právě děje a nejasná místa vznikají daleko méně.

Závěr

Úspěšně jsme dokončili naši minianalýzu a příště se pustíme již do programování.

Perl (65) - Projekt - získání dat

Dnes v rámci našeho projektu stáhneme potřebná data z webu a pomocí regulárních výrazů z nich vyextrahujeme data o zápasech.

Pojďme tedy začít se samotným programováním. Ovšem nejdříve si musíme vše naplánovat a rozhodnout se, kde začneme. Vzhledem k tomu, že jsme zvolili objektově-orientovanou koncepci, začneme určitě modulem Livescore.pm. Napíšeme konstruktor a potom můžeme klidně postupovat podle bodů, které jsme si vytýčili posledně. Nuže, dejme se do doho.

Konstruktor

Nyní je třeba učinit ještě jedno rozhodnutí. Co bude uchovávat objekt? Určitě bude třeba v nějaké formě uložit stránku, ze které budeme stahovat data. Dnešní zápasy jsou k dispozici na http://www.livescore.com/default.dll?page=home, zápasy v rámci České republiky na http://www.livescore.com/default.dll?page=czechia apod. V první verzi konstruktoru tedy uchováme home, czechia apod., které přijde jako parametr konstruktoru od uživatele. Konstruktor zatím necháme být, protože se ještě může spousta věcí změnit a jméno soutěže je asi jediná jistota. Prozatím vypadá náš konstruktor takto.

```
sub new {
my($self, $liga) = @_;
```

```
my $f = {};
bless $f;
$f->{"liga"} = $liga;
return $f;
}
```

Získání zdrojového kódu

Prvním úkolem, který by měl modul Livescore učinit na základě požadavku od uživatele je získání dat. Data získáme na základě položky liga v objektu. Tato funkce nebude veřejná (resp. zdokumentovaná). Jejím úkolem bude vrátit data, o jejichž zpracování se postará zase někdo další.

Jak ale stáhneme data z webu? Nejjednodušší je prohledat CPAN. Jedním z modulů, který to umí je WWW::Mechanize, jež obsahuje metodu get($url).

```
use WWW::Mechanize;
```

Nyní můžeme napsat poměrně jednoduchou metodu ziskej_zdrojovy_kod. Je třeba si uvědomit, že dříve nebo později ji budeme muset přepsat kvůli perzistenci. Zatím to však řešit nebudeme.

```
sub ziskej_zdrojovy_kod {
my($self, $liga) = @_;
my $url;
my $zdroj = undef;

$url = "http://www.livescore.com/default.dll?page=$liga";

my $mech = WWW::Mechanize->new();
$zdroj = ($mech->get($url))->{"_content"};

return $zdroj;
}
```

Extrakce dat

Toto bude možná nejtvrdší oříšek celé aplikace. Co všechno budeme potřebovat za data? Nahlédněme do zdrojového kódu. Vidíme, že lze získat toto.

- účastníci zápasu
- případné skóre
- soutěž a země
- minuta u probíhajícího zápasu
- to, zda se zápas hraje
- čas výkopu
- protože budeme potřebovat i odkaz na detail zápasu, musíme uchovat ID zápasu a jméno soutěže tak, jak je uvedeno v odkazu

Úkolem je vytvořit na základě staženého zdrojového kódu pole hashů, které bude obsahovat zmíněné informace o jednotlivých zápasech. Bude to mechanická práce, ovšem i tu je dobré si ozkoušet.

Tato metoda bude veřejná. To znamená, že uživatel bude nucen volat při použití modulu Livescore nejprve konstruktor a následně metodu ziskej_zapasy_dane_ligy, kterou právě píšeme. Díky tomu si sám uživatel bude řídit, kdy data aktualizovat. Pro jednoduchost metoda vrátí seznam vyhovujících zápasů, se kterým bude nakládat dle uvážení uživatel. Zápasy tak nebudou součástí objektu.

Podíváme-li se na zdrojový kód, zjistíme, že to nebude vůbec tak jednoduché, protože každý zápas může být zobrazen v několika formátech. Pokud nejsou dostupné žádné podrobné informace k zápasu, nalezneme jako jeho reprezentaci ve zdrojovém kódu z livescore.com toto.

```
<tr bgcolor="#dfdfdf"><td width="45" height="18"> 23:00</td><td align="right"
width="118">Genemuiden</td><td align="center" width="50">? - ?</td><td width="118">FC
Omniworld</td></tr><tr><td colspan="4" height="1"></td></tr>
```

Pokud však již byla zaznamenána branka nebo jiná událost, vytvoří uvnitř odkaz a rázem se celý zdrojový kód pro zápas změní.

```
<tr><td colspan="4" height="1"></td></tr><tr bgcolor="#dfdfdf"><td width="45"
height="18"> FT</td><td align="right" width="118">Blackburn R.</td><td
align="center" width="50"><a class="scorelink" target="match_details"
onclick="window.open('','match_details','width=400,height=239,menubar=no,status=no,location=no,
toolbar=no,scrollbars=no,resizable=yes')" href="/default.dll/Game?comp=england1&game=359276">4
- 2</a></td><td width="118">Manchester C.</td></tr><tr><td colspan="4"
height="1"></td></tr>
```

Nehledě na to, že k zápasu musíme přidávat další dvojici údajů, která je rozmístěna mezi zápasy. Jsou to datum a čas výkopu a soutěž. Čas výkopu získáme z tohoto úseku kódu. Navíc může být čas změnen lokálně u jednotlivých zápasů.

```
<tr bgcolor="#333333"><td class="match-light" width="45" height="18"> 13:55</td><td
class="match-light" align="right" width="286" colspan="3">October 19 </td></tr>
```

A nakonec jméno soutěže a stát získáme odtud.

```
<tr bgcolor="#333333"><td class="title" colspan="4" height="18"> <b>England</b> -
League Cup</td></tr>
```

Všechny tyto úseky se v podstatě náhodně vyskytují uvnitř staženého zdrojového kódu. Je tedy třeba postupně projít celý zdrojový kód a hledat výskyty zmíněných úseků. Přitom musíme dodržet jejich pořadí, protože jinak bychom nebyli schopni správně určit čas výkopu a soutěž.

Všimněme si, že každý údaj - ať již datum konání, národní soutěž a zápas jsou vždy na jednom řádku. Tudy povede cesta. Alespoň pro naše řešení.

Napišme si tedy podrobnější postup extrakce dat.

1. Než začneme, je třeba získat zdrojový kód pomocí funkce ziskej_zdrojovy_kod, kterou již máme.
2. Nejprve ze zdrojového kódu vyextrahujeme všechny řádky. Řádek vždy začína tagem <tr> a končí </tr>. Musíme přitom zachovat jejich pořadí.
3. Dále budeme řádky třídit a získávat z nich data. Určíme tedy, jakou informaci řádek poskytuje. Máme 4 možnosti.

- Obsahuje informaci o zápase. Získáme odtud názvy týmů, skóre, minutu a případně změníme čas výkopu. Získaná si případně upravíme data k obrazu svému a všechny údaje zapíšeme do pole zápasů.
- Obsahuje informaci o čase pro nadcházející zápasy - změníme obsah proměnných uchovávajících čas.
- Obsahuje informaci o soutěži pro nadcházející zápasy - změníme obsah proměnných uchovávajících soutěž.
- Neobsahuje žádnou z hledaných informací a je pro nás bezcenný

4. Vrátíme pole zápasů.

Nejprve získáme zdrojový kód pomocí již napsané metody.

```
sub ziskej_zapasy_dane_ligy {
my($self) = @_;
my @zapas; #bude obsahovat informace o zápasech
my $zdroj = $self->ziskej_zdrojovy_kod($self->{"liga"});

#hlavní část funkce

return @zapas;
}
```

Z něj odseparujeme veškeré úseky, které začínají <tr> a končí </tr>. Jsou to pro nás potenciální užitečné informace.

```
my $i=0;
my @radek;
$radek[$i++] = $1 while $zdroj =~ /(<tr.*?>.*?<\/tr>)/g;
```

Další bod je úspěšně za námi. Teď ale přijde na řadu to nejhorší. Každý řádek budeme muset pečlivě prozkoumat.

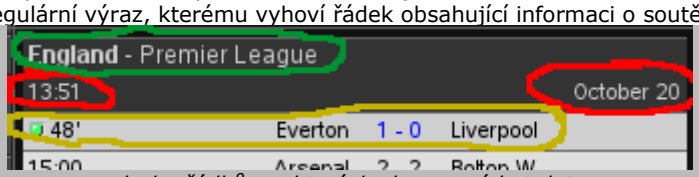
```
for (@radek){
#extrakce
}
```

Hlavním "work horse" tohoto problému budou regulární výrazy. Pomocí nich zajistíme veškerou extrakci.
Předně budeme zjišťovat, zda řádek je pro nás cenná informace. Jak to poznáme? Vzpomeňme na úryvky ze zdrojového kódu na začátku tohoto oddílu. Budeme muset vytvořit pro každý úsek vzor a ten porovnat s řádkem. Bude to vypadat takto.

```
if ($_ =~ /regex1|regex2|regex3/g){
#kód je pro nás cenný; zpracujeme ho
}
#kód nás nezajímá, přejdeme na další iteraci
```

Musíme napsat následující tři regulární podvýrazy.

1. regulární výraz, kterému vyhoví řádek obsahující informaci o zápase (na obrázku žlutá)
2. regulární výraz, kterému vyhoví řádek obsahující informaci o datu a čase výkopu (červená)
3. regulární výraz, kterému vyhoví řádek obsahující informaci o soutěži (modrá)



druhy řádků, ze kterých chceme získat data

Tento regulární výraz tady nebudeme kompletně odvozovat, protože je to spíš manuální práce a popis by zabral několik stránek. Je třeba najít co nejvíc variant formátu těchtýž dat ve staženém zdrojovém kódu a pokusit se vytvořit regulární výraz, který je všechny zahrne. Každý řádek se zápasem na www.livescore.com má totiž trochu jiný formát a tento rozdíl musíme vyeliminovat. Úkol tedy zní: Nalezněme regulární výraz, kterému vyhoví, všechny řádky obsahující informaci o zápase. Stejně potom budeme postupovat i u získávání času a soutěže.
Uveďme si několik obecných metod, kterými lze regulární výraz tvořit.

- Všechny údaje, které chceme extrahovat dáme do závorek
- Pokud se v jedné variantě objeví navíc nějaký kód (např. u hrajícího zápasu se zobrazuje vždy obrázek blikající tečky), použijeme na něj toto uvození: (?:*regex*)?.
- Parametry tagů, které se mohou lišit je dobré nahradit sekvencí (?:[^>]*). Místo <td *parametry*> tak napíšeme do regulárního výrazu <td(?:[^>]*)>.

Pod nějaké době získáme tento nebo jemu podobný regulární výraz pro řádek se zápasem.

```
(?:>tr bgcolor=\"#.....\">>td(?:[^>]*)> (?:<(i)mg[^>]*> )?([\w\:.]*)'?<\/td>>td
(?:[^>]*)>(.[^>]*)>\/td>>td(?:[^>]*)>(?:\>a class=\"scorelink\" target=\"match_details\"
onclick=\"window.open\('','match_details','width=400,height=\d*,menubar=no,status=no,lo
cation=no,toolbar=no,scrollbars=no,resizable=yes'\)\" href=\"\/default.dll\/Game\?comp=
(\w*)&game=(\d*)\">)?([?\d]+) - ([?\d]+)(?:>\/a>)?>td>>td width=\"118\">(.[^<]*)<\/td><\/tr>)
```

Podobně získáme další dva regulární výrazy, spojíme je alternací a vepíšeme do podmínky.

```
if ($_ =~ /(?:>tr bgcolor=\"#.....\">>td(?:[^>]*)> (?:<(i)mg[^>]*> )?([\w
\:.]*)'?<\/td>>td(?:[^>]*)>(.[^>]*)>\/td>>td(?:[^>]*)>(?:\>a class=\"scorelink\" targe
t=\"match_details\" onclick=\"window.open\('','match_details','width=400,height=
\d*,menubar=no,status=no,location=no,toolbar=no,scrollbars=no,resizable=yes'\)\
" href=\"\/default.dll\/Game\?comp=(\w*)&game=(\d*)\">)?([?\d]+) - ([?\d]+)(?:>\/
a>)?>td>>td width=\"118\">(.[^<]*)<\/td><\/tr>)|(?:>tr bgcolor=\"#.....\">>td class=\"tit
le\" colspan=\"4\" height=\"18\"> >b>([^>]*)>\/b> - ([^>]*)>\/td>>\/tr>)
|(?:>tr bgcolor=\"#.....\">>td class=\"match-light\" width=\"45\" height=\"18\"> ([^>]*
)?>\/td>>td class=\"match-light\" align=\"right\" width=\"286\" colspan=\"3\">(\w+) (\d
+) >\/td>>\/tr>)/g){

#zpracování dat

}
```

Poznámka - kvůli sazbě byly výše uvedené zdrojové kódy rozděleny do řádků. Znaky nových řádků ovšem do programu nepatří.

Nyní máme jistotu, že data na řádku, jež vyhovuje výše uvedenému regulárnímu výrazu jsou pro nás cenná. Nyní bychom se měli zamyslet, jak je správně dostaneme do proměnných. Zde je tabulka extrahovaných hodnot.

Typ získané informace	Proměnná	Informace
o zápase	$1	hraje se?
	$2	před výkopem čas výkopu, po výkopu minuta
	$3	název domácího týmu
	$4	(pouze je-li dostupná nějaká událost) liga podle livescore.com
	$5	(pouze je-li dostupná nějaká událost) ID zápasu podle livescore.com
	$6	skóre domácích
	$7	skóre hostů
	$8	název hostujícího týmu
o soutěži	$9	název státu
	$10	název soutěže
o času výkopu	$11	čas výkopu nebo poslední aktualizace
	$12	měsíc výkopu
	$13	den výkopu

Tyto informace uložíme do výsledného pole. Ještě předtím však několik údajů pozměníme. Jsou to většinou věci, které bychom dělali až během testování výsledného programu, ale protože je na to třeba delší zkušenost s daty na livescore.com, uveďme je pro lepší orientaci hned.

Proměnná, kterou budeme upravovat	Podmínka úpravy	Nová hodnota
$1	pokud obsahuje i	změníme na PROBIHA
$1	pokud není definováno	změníme na PRED_VYKOPEM nebo UKONCEN podle probíhající minuty
$2	pokud obsahuje čas výkopu	změníme na --
$2	pokud obsahuje delší řetězec - tedy AET, Pen., Postp. nebo Susp.	zaměníme za dvojznaková OT, PN, XO a XS
$3 a $8	pokud obsahuje &	zaměníme tento podřetězec za &
$6 a $7	pokud je skóre "?"	nahradíme za "-"
$11	pokud získáme čas v proměnné $2	má vyšší prioritu než $10
$12	vždy	anglický název měsíce nahradíme jeho pořadovým číslem

U proměnné $1, nahrazujeme původní hodnotu konstantou. Tyto konstanty je třeba definovat.

```
use constant {
UKONCEN => 0,
PROBIHA => 1,
PRED_VYKOPEM => 2
};
```

Nyní nám zbývá vytvořit z obou tabulek zdrojový kód. Pokud tedy narazíme na řádek s informacemi o soutěži (zjistíme to tak, že jsou definované proměnné $9 a $10), nastavíme proměnné $soutez a $zeme.

```
if($9){
$zeme = $9;
$soutez = $10;
}
```

V případě řádku s informacemi o čase (jsou definované proměnné $11 až $13), nastavíme proměnné $cas, $den a $mesic. $mesic zkonvertujeme (ne příliš elegantně) na pořadové číslo příslušného měsíce.

```
if($12){
$cas = $11;
$mesic = $12;
$den = $13;

$mesic eq "January" and $mesic=1;
$mesic eq "February" and $mesic=2;
$mesic eq "March" and $mesic=3;
$mesic eq "April" and $mesic=4;
$mesic eq "May" and $mesic=5;
$mesic eq "June" and $mesic=6;
$mesic eq "July" and $mesic=7;
$mesic eq "August" and $mesic=8;
$mesic eq "September" and $mesic=9;
$mesic eq "Octomber" and $mesic=10;
$mesic eq "November" and $mesic=11;
$mesic eq "December" and $mesic=12;
}
```

A pak zde máme informace o zápase. Na tomto řádku nejen, že nastavíme proměnné, ale všechna data zaznamenáme. Navíc musíme udělat větší množství úprav v datech. Je třeba vyřešit ampersandy a obsah proměnné $skore. Dále je třeba upravit obsah proměnné $minuta a $hraje_se.

```
if($1){
my $minuta=$1;
my $tym1 = $2;
my $tym2 = $7;
my $skore1 = $5;
my $skore2 = $6;
my $liga = $3;
my $id = $4;
$tym1 =~ s/(&amp;)/&/;
$tym2 =~ s/(&amp;)/&/;

if($skore1 eq "?"){$skore1 = $skore2 = "-";}
if($minuta =~ /^\d\d:\d\d$/){$cas = $minuta; $minuta="--";}

$minuta eq "AET" and $minuta = "OT";
$minuta eq "Pen." and $minuta= "PN";
$minuta eq "Postp." and $minuta= "XO";
$minuta eq "Susp." and $minuta= "XS";

if($1 eq "|"){$hraje_se = PROBIHA;}
elsif($minuta eq "--"){$hraje_se = PRED_VYKOPEM;}
else{$hraje_se = UKONCEN;}

push(@zapas, {
"tym1"       => $tym1,
"tym2"       => $tym2,
"skore1"     => $skore1,
"skore2"     => $skore2,
"liga"       => "$zeme: $soutez",
"odkaz_liga"=> $liga,
"odkaz_idzapasu" => $id,
"minuta"     => $minuta eq "" ? "??" : $minuta,
"vykop"      => $den ? "$den.$mesic $cas" : "KONEC",
"hraje_se"   => $hraje_se
});
}
```

A jsme hotovi. Nyní náš modul již umí získat informace o zápasech.

```
return @zapas;
```

Perl (66) - Projekt - výběr zápasů a podrobnosti

Zdokonalíme náš modul o funkci, která ze všech dostupných zápasů vybere jen ty, které vyhovují uživatelem zadaným kritériím. Mimoto se podíváme na další extrakci dat z webu - tentokrát si opatříme detailní informace o zvoleném zápasu.

Výběr zápasů podle kritérií

Nyní se zaměříme na výběr zápasů podle kritérií. Opět si na to vytvoříme zvláštní metodu, která bude volána uživatelem. Jako parametr bude přijímat odkaz na seznam zápasů vrácených metodou ziskej_zapasy_dane_ligy a odkaz na pole kritérií. Metoda vrátí seznam zápasů, jež vyhovují daným kritériím.

```
sub najdi_zapas_podle_kriterii {
my($self, $p_zapasy, $p_kriteria) = @_;
my @vyhovujici;

#tělo metody

return @vyhovujici;
}
```

Pokud kritéria zadána nebudou, vrátíme tentýž seznam, který jsme získali.

```
return @$p_zapasy if scalar @$p_kriteria == 0;
```

Nyní se dostáváme k jednomu zajímavému místu. Budeme chtít zajistit, aby byl každý zápas porovnán se všemi kritérii (podmínkami). Co to ale je *všemi*? To může být jedna podmínka, dvě podmínky, nebo jakýkoliv přirozený počet podmínek. A otázkou je, jak do testu podmínky vepsat libovolný počet kritérií.

Zdánlivě je to těžko řešitelný problém, ale existuje řešení. Tím je pěkná aplikace nástroje eval.

Vytvoříme tedy podmínku jako řetězec a poté v cyklu tento řetězec porovnáme postupně se všemi vstupními zápasy. Pokud zápas vyhovuje kritériím, přidáme ho k vyhovujícím zápasům.

```
my $eval = 'if(/'.join("/i and /", @$p_kriteria).'/i){push @vyhovujici, $zapas;}';
foreach my $zapas (@$p_zapasy){
$_ = $zapas->{"tym1"}." ".$zapas->{"tym2"};
eval $eval;
}
```

Detail zápasu

Další veřejnou metodou bude prenos, který bude poskytovat informace o událostech v jednotlivých zápasech. Abychom mohli jednoznačně určit, odkud informace o zápasu získat, potřebujeme znát dva údaje. Za prvé soutěž a za druhé ID zápasu. Samotné ID by na to nestačilo.

Co se týká výstupu tak tam to bude složitější. Budeme muset vracet nějakou složitou datovou strukturu. Je třeba, aby obsahovala veškeré informace o událostech a navíc probíhající minutu. Pouvažujme však nad uživatelem této metody. Pro online přenos je třeba vrátit ještě informaci o tom, zda se zápas hraje. Proto vrátíme ještě hodnotu, jež určuje, zda zápas stále probíhá, zda již skončil nebo zda ještě nezačal. Tato informace se sice dá vyvodit z probíhající minuty, avšak později uvidíme, že její uvedení má své opodstatnění. Program live se podle ní zařídí a spustí buď online přenos nebo jen zobrazí výsledky.

A nyní se již pusťme do detailu zápasu.

```
sub prenos {
my($self, $liga, $odkaz_idzapasu) = @_;
my($probihajici_minuta, $hraje_se);
my @udalost;

#tělo metody

return ($probihajici_minuta, $hraje_se, \@udalost);
}
```

Stažení dat

Naším prvním úkolem je stáhnout zdrojový kód. URL ke zdrojovým kódům zápasů je zde: http://www.livescore.com/default.dll/Game?comp=*soutěž*&game=*id_zápasu*. Soutěž je zde přitom hodnota v prvku odkaz_liga, nikoliv liga.

Použijeme na to opět metodu ziskej_zdrojovy_kod, kterou ale budeme muset upravit. Pokud dostane druhý argument - číslo zápasu - upravíme její chování. Bude stahovat z uvedeného URL a je tedy třeba nastavit proměnnou $url odpovídajícím způsobem.

```
my($self, $liga, $odkaz_idzapasu) = @_;

...

if ($odkaz_idzapasu){
$url = "http://www.livescore.com/default.dll/Game?comp=$liga&game=$odkaz_idzapasu";
}else{
$url = "http://www.livescore.com/default.dll?page=$liga";
}
```

Nyní opět přeskočme do metody prenos. Díky úpravám ve funkci ziskej_zdrojovy_kod nyní můžeme napsat následující. Podotkněme ještě, že tato funkce je stále kompatibilní s posledně použitými voláními.

```
my $zdroj = $self->ziskej_zdrojovy_kod($liga, $odkaz_idzapasu);
```

Extrakce dat a získání podrobných informací o zápasu

Na základě tohoto zdrojového kódu musíme opět získat extrakcí několik údajů. Tyto údaje je pak třeba ještě rozdělit na dva druhy - údaje obecné a údaje o události. Údaje obecné jsou následující.

- hraje se? (na obrázku označení červenou barvou)
- probíhající minuta (modrá)
- názvy týmů - s každou událostí bude třeba posílat jméno týmu, kterého se událost týká, proto si oba názvy uložíme (žlutá)

údaje o stavu zápasu

A co se týká událostí, tak budeme sledovat toto:

- typ události - tedy gól, žlutá karta, červená karta (na obrázku označení červenou barvou)
- jméno hráče, kterého se událost týká (modrá)
- minuta události (žlutá)
- nové skóre (fialová)
- tým, kterého se událost týká (hnědá; určíme podle toho, zda je událost vlevo nebo vpravo)

údaje o událostech

Nejprve vyextrahujeme ze zdrojového kódu obecné údaje.

```
my($hraje_se, $probihajici_minuta, $domaci, $hoste) = ($zdroj =~
/<font[^>]*><b>(?:<(i)mg[^>]*>)?\s?([\w\:.]*)'?<\/b><\/font><\/td><t[^>]*>([^[]*)
```

```
\s\[[^\]]*\]\s([^<]*)<\/td>/);
$hraje_se = ($hraje_se ? 1 : 0);
```

Nyní přichází složitější část. V cyklu musíme zpracovávat jednotlivé události. To znamená, že regulární výraz umístíme do podmínky cyklu a uvnitř něj data postupně zpracujeme do pole hashů @udalosti. Budeme zpracovávat řádek po řádku - tedy od <tr> k </tr>. Nicméně bude to o to složitější, že na každém řádku mohou být i dvě události - jedna pro hostující a jedna pro domácí tým.

Na extrakci minuty události, skóre a jména hráče není nic nového. Typ události získáme z názvu obrázku. Začátek jeho názvu totiž je jedním z řetězců yellow, red, yellow-red a goal.

Zde je celý regulární výraz. Pomocí komentářů v něm by měl být jasnější smysl jednotlivých jeho úseků a metoda konstrukce.

```
while ($zdroj =~
/
<tr[^>]*><td[^>]*>
(\d+)'                  #minuta události
<\/td><td[^>]*><b>
(?:\[(\d+)\s?-\s?(\d+)\])?   #nové skóre
<\/b><\/td>

#událost domácích - buď je nebo není
<td[^>]*>
(?:([^<]*)\s<img\ssrc="http:\/\/www1.livescore.com\/img\/([\w-]+)\.gif"[^>]*>)?
<\/td>   #typ události a hráč


#událost hostů - buď je nebo není
<td[^>]*>
(?:([^<]*)\s<img\ssrc="http:\/\/www1.livescore.com\/img\/([\w-]+)\.gif"[^>]*>)?
<\/td>   #typ události a hráč

<td\swidth=4><\/td><\/tr><tr><td\scolspan=\"4\"\swidth=\"400\"\s
height=\"1\"><\/td><\/tr>
/gx){
}
```

Dokončení metody prenos

Nyní nám už zbývá pouze naplnit prvek pole @udalosti získanými hodnotami. Způsob zapisování dat musíme rozdělit hned na tři možnosti - buď máme událost hostů, událost domácích nebo události obě.

Ještě uveďme, že jméno hráče nemusí být v případě gólů vždy uvedeno. Proto, pokud jméno skutečně nebude, nahradíme ho neznámým střelcem.

Pokud se tedy týká událost domácích, zapíšeme to takto.

```
if($4){
$tym = $domaci;
$strelec = ($4 ? $4 : "neznamy strelec");

$udalost[$i++] = {
"id_udalosti" => $i-1,
"udalost" => $5,
"minuta" => $1,
"hrac" => $strelec,
"tym" => $tym,
"skore1" => $2,
"skore2" => $3
}
}
```

A jde-li o hosty použijeme analogický způsob.

```
if($6){
$tym = $hoste;
$strelec = ($6 ? $6 : "neznamy strelec");

$udalost[$i++] = {
"id_udalosti" => $i-1,
"udalost" => $7,
"minuta" => $1,
"hrac" => $strelec,
"tym" => $tym,
"skore1" => $2,
"skore2" => $3
}
}
```

Půjde-li o oba týmy, program sám projde obě podmínky.

Perl (67) - Projekt - dokončujeme modul

Dnes dokončíme základní modul, abychom ho vzápětí mohli použít k vytvoření samotného programu

Perzistence

Když se podíváme na seznam úkolů, které by měl modul zvládat, zbývá poslední. Perzistentní uchovávání dat. Půjde jen o to, jak vhodně přepsat podprogram ziskej_zdrojovy_kod. Nadefinujeme si tedy proměnnou $cil, která bude jednoznačně identifikovat každý stažený soubor. $cil bude obsahovat cestu k souboru, kam zdrojový kód uložíme. Její tvar bude následující, podle toho, zda půjde o seznam zápasů nebo podrobnosti ke konkrétnímu zápasu.

/tmp/livescore_*liga_odkaz_idzapasu*
/tmp/livescore_*liga*
Metoda ziskej_zdrojovy_kod bude vypadat takto.

```
sub ziskej_zdrojovy_kod {
my($self, $liga, $odkaz_idzapasu) = @_;
my $url;
my $cil;
my $zdroj = undef;
local $/ = "";

if ($odkaz_idzapasu){
$url = "http://www.livescore.com/default.dll/Game?comp=$liga&game=$odkaz_idzapasu";
$cil = "/tmp/livescore_".$liga."_".$odkaz_idzapasu;
}else{
$url = "http://www.livescore.com/default.dll?page=$liga";
$cil = "/tmp/livescore_$liga";
}

#získání dat ze souboru nebo stažení; uložení dat do souboru

return $zdroj;
}
```

Za jakých okolností nyní budeme data stahovat a za jakých pouze kopírovat ze zálohy? Soubor musí splňovat dvě podmínky. Musí existovat a musí být mladší než nějaký interval definovaný uživatelem. Ještě předtím, než dokončíme metodu ziskej_zdrojovy_kod musíme do objektu přidat v konstruktoru parametr refresh, který implicitně nastavíme na 30 sekund. Pokud bude soubor starší, než kolik je $refresh bude stažen znovu.

```
$f->{"refresh"} = (int $refresh < 30) ? 30 : $refresh;
```

Vraťme se k metodě ziskej_zdrojovy_kod. Nyní již známe podmínky, za kterých soubor pouze zkopírujeme. Proměnnou $/ jsme nastavili proto, abychom načetli celý kód ze souboru najednou.

```
if (-e $cil and time() - (stat($cil))[9] <= $self->{"refresh"}){
open(R, $cil) or die "Nelze otevrit datovy soubor\n";
$zdroj = <R>;
close R;
}else{
#stáhneme data znovu a uložíme do zálohy
}
```

Nejsou-li tyto podmínky splněny, nezbývá, než data získat znovu z internetu. Vytvořenému souboru poskytneme plná práva, aby mohli zálohu načítat a aktualizovat jiní uživatelé. To sice dává možnost falšovat data, ale vzhledem k tomu, že je stejně získáváme z webu, tak se na ně spolehnout nelze.

```
unlink $cil;
my $mech = WWW::Mechanize->new();
$mech->get($url);
$zdroj = ($mech->get($url))->{"_content"};
open(W, ">$cil") or die "Nelze otevrit datovy soubor\n";
print W $zdroj;
close W;
chmod 511, $cil;
```

Modul hotov

Až na dokumentaci jsme právě dokončili základní část modulu. Zrekapitulujme si stručně, co všechno nyní vlastně dokáže.

- jsme schopni získat data - ať už seznam zápasů nebo seznam událostí k jednotlivým zápasům
- tato data dokážeme perzistentně uchovat
- z těchto surových dat dokážeme získat potřebné informace
- tyto informace dokážeme třídit na základě kritérií

Ukázka použití modulu

Sice tušíme, jak se bude modul používat, ale měli bychom si to pro lepší pochopení alespoň stručně ukázat. V úvodu je třeba volat dvě metody - konstruktor a metodu na získání zápasů.

```
my $live = Livescore->new($liga);
my @vyhovujici = $live->ziskej_zapasy_dane_ligy;
```

Dále zpravidla vybereme pouze zápasy vyhovující nějakým kritériím

```
@kriteria = qw(manchester liverpool);
@vyhovujici = $live->najdi_zapas_podle_kriterii(\@vyhovujici, \@kriteria);
```

Nyní můžeme udělat několik věcí. Buď data jen zobrazit za základě dat v poli @vyhovujici nebo získat podrobnosti k zápasu a zobrazit až je. Je též možné tyto podrobnosti periodicky doplňovat a tím získat program na online sledování.

```
my @udalosti = $live->prenos($vyhovujici_liga, $vyhovujici_idzapasu);
```

Tím vším se ale budeme podrobně zabývat až příště. Pokud si chcete vytvořit vlastní program využívající modul Livescore ještě dříve, než to uděláme společně v dalším pokračování seriálu, můžete si celý modul stáhnout.

Perl (68) - Projekt - zobrazení zápasů

To nejdůležitější máme zdárně za sebou a můžeme se tak pustit do konzolového programu, který našeho modulu využije. Cílem je uživatelsky přívětivý a intuitivně ovládatelný program.

Užití modulu

Dalo by se snad říci, že to nejtěžší máme za sebou. Nyní se budeme zabývat zpracováním získaných dat. Soubor live vytvoříme ve dvou fázích. Dnes napíšeme kompletní live bez podpory online přenosu. To bude nejsložitější část programu a dopíšeme ji až příště.

Zauvažujme nyní, co budeme potřebovat a udělejme si pár poznámek.

- Budeme přijímat argumenty z příkazového řádku a zpracovávat je pomocí Getopt::Long. Též bychom měli myslet na argument --help.
- Zavoláme modul Livescore a získáme data o zápasech.
- Pokud byla zadána kritéria, vytřídíme nevyhovující zápasy.
- Dále data o zápasech, pokud nebyl zvolen online přenos, zobrazíme. Jak bylo požadováno v zadání, odlišíme stav zápasu barvami.
- Poté vyřešíme online přenos

Nyní můžeme začít psát.

```
#!/usr/bin/env perl
use strict;
use Livescore;
use Getopt::Long;
use Term::ANSIColor qw(colored);
```

Jeden importovaný modul ještě neznáme. O možnostech modulu Term::ANSIColor se zmíníme až budeme chtít tisknout barevný text.

Ze všeho nejdříve obstaráme argumenty příkazového řádku. Funkci GetOptions není třeba rozebírat podrobněji, neboť tak již bylo učiněno dříve, když jsme se zpracováváním argumentů zabývali.

```
GetOptions("league|l=s" => \$liga, "online|o" => \$online, "refresh|h=s" => \$refresh, "help|h" => \$help);
```

Aby nemohl zadat uživatel jako ligu cokoliv, vytvoříme pro jednoduchost jejich pevný seznam

Pokud se vám takové řešení nelíbí, tak se zas tolik nestane, když tuto kontrolu úplně vynecháte.

Metoda ziskej_zdrojovy_kod totiž možná v případě zadání neplatné soutěže stáhne nějakou stránku, ale z ní by se nemělo nic vyextrahovat - to jest žádné zápasy se nenajdou. Další a asi nejlepší možností by bylo zjistit seznam soutěží přímo v modulu a poskytovat jej společně s daty. Nyní se však pro jednoduchost spokojíme s již avizovaným pevným seznamem.

Na livescore.com, odkud data získáváme, jsou k dispozici tyto soutěže.

```
my @ligy = sort qw(soccer home euro2008 wc2010 u21_euro eurocups royal intl
england italy spain germany france holland belgium portugal scotland austria
denmark finland greece iceland ireland norway sweden switzerland turkey bulgaria
croatia czechia hungary israel poland romania russia yugoslavia slovakia slovenia
ukraine southamerica argentina bolivia brazil chile colombia ecuador paraguay
peru uruguay venezuela concacaf mexico usa costarica elsalvador guatemala asia
japan china armenia azerbaijan georgia kazakhstan australia africa algeria egypt
morocco southafrica tunisia);
```

Pokud uživatel zadá neexistující jméno soutěže (nebo přepínač --help), vypíšeme nápovědu.

```
if (not in_array($liga, \@ligy) and $liga or $help) {die << "EOF"
live - online monitoring of football matches
Usage: live [OPTIONS] [condition1 condition2 ... conditionn]

Options:
-l, --league=<liga>        matches will be searched in this league
-o, --online, --verbose    online monitoring on
-h, --help                 display this help and exit
-r, --refresh              number of second denouncing refresh interval

conditions are patterns, which are occured in demand match

Available leagues: @ligy

soccer - all matches
home - home matches
EOF
}
```

Chybí nám podprogram in_array. Buď ho můžeme importovat například z modulu Array::PAT, ale než instalovat nový modul, možná bude rychlejší, když si in_array sami napíšeme.

```
sub in_array {
my($hodnota, $p_pole) = @_;
my $vysledek = undef;

foreach my $p (@$p_pole){
if ($p eq $hodnota){
$vysledek = 1;
last;
}
}

return $vysledek;
}
```

Pokud uživatel na vstup ligu nebo refresh nezadá, doplníme je implicitními hodnotami.

```
$liga = "home" if !$liga;
$refresh = 60 if !$refresh;
```

A posledním údajem, který nám uživatel poskytuje, jsou kritéria. Ta jsou obsažena v poli @ARGV.

```
my @vsechna_kriteria = @ARGV;
```

Nyní můžeme začít pracovat s modulem. Již minule jsme si ukázali základní použití. Začít můžeme úplně stejně.

```
my $live = Livescore->new($liga, $refresh);
my @vyhovujici = $live->ziskej_zapasy_dane_ligy();
@vyhovujici = $live->najdi_zapas_podle_kriterii(\@vyhovujici, \@vsechna_kriteria);
```

Nyní je třeba data vhodně zobrazit. Způsob bude záležet na tom, zda byla zadána volba -online. Online přenos dnes ještě dělat nebudeme.

```
if(!$online){
tiskni_vyhovujici(@vyhovujici);
exit;
}else{
die "Online přenos zatím není dostupný!\n";
}
```

Poslední věc první fáze bude napsání podprogramu tiskni_vyhovujici.

```
sub tiskni_vyhovujici {
my @vyhovujici = @_;
#zpracování dat získaných z modulu Livescore a tisk
}
```

Zpracování dat bude probíhat v cyklu. Projdeme tedy pole @vyhovujici a v každé iteraci zpracujeme jeden záznam.

```
for (@vyhovujici){
#zpracování a vytisknutí jednoho záznamu
}
```

Protože nyní máme všechna data přímo dostupná, můžeme začít s formátováním. Nejprve vytvoříme proměnnou $skore, do které zahrneme najednou skóre obou soupeřů, oddělené dvojtečkou.

```
my $skore = $_->{"skore1"}.":".$_->{"skore2"};
```

Nyní pomocí sprintf zformátujeme všechny informace na řádek a výsledek uložíme do stejnojmenné proměnné.

```
my $vysledek = sprintf "\r%02s' %-40s %-3s %10s (%s) %s\n", $_->{"minuta"},
$_->{"tym1"} ." - ". $_->{"tym2"}, $skore, $_->{"vykop"}, $_->{"liga"};
```

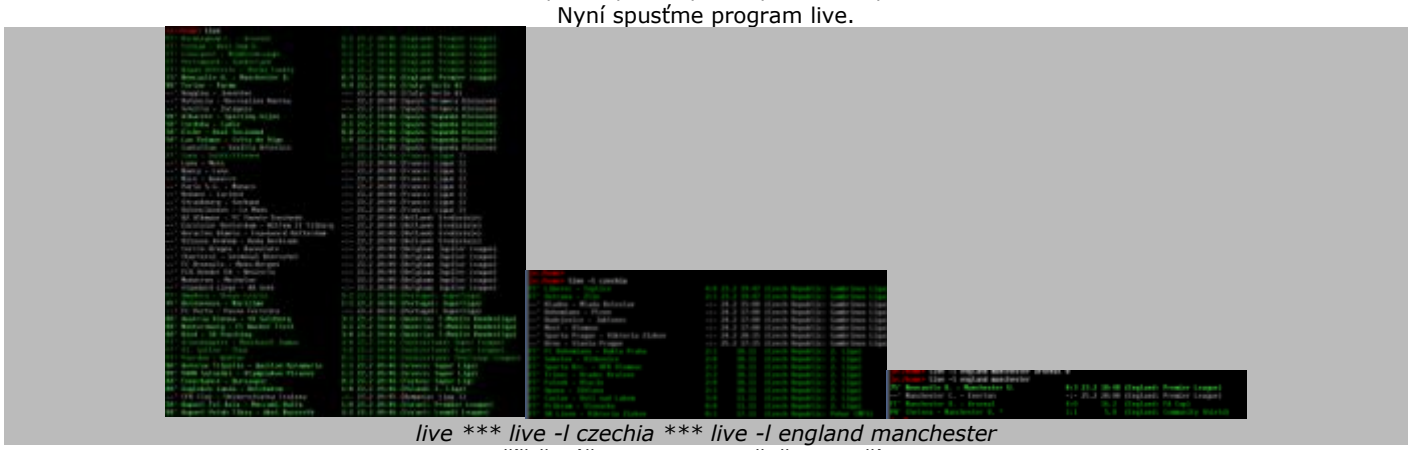
Data jsme nevytiskli ale pouze zformátovali. To proto, že je ještě musíme příslušně obarvit. Zvolíme tmavě zelenou barvu pro zápasy, které již skončily, světle zelenou pro právě probíhající zápasy a zápasy ukončené tiskneme bíle. Term::ANSIColor nabízí několik funkcí z nichž jednou je colored a právě tu jsme importovali. Obarvíme pomocí ní hodnotu proměnné $vysledek a vytiskneme.

```
if ($_->{"hraje_se"} == Livescore::PROBIHA){ print colored $vysledek,
"bold green";}
elsif ($_->{"hraje_se"} == Livescore::UKONCEN){ print colored $vysledek,
"green";}
else {print colored $vysledek, "white";}
```

První výsledky

Nyní zkopírujme modul Livescore.pm někam do cesty, kde se hledají moduly. Skriptu livepak nastavíme atribut spustitelnosti a překopírujeme do PATH. Tedy například /usr/bin a nebo, pokud používáte zvláštní cestu pro své výtvory, použijte třeba /home/bin či /home/*username*/bin.

Nyní spusťme program live.



*live *** live -l czechia *** live -l england manchester*

Příště náš program konečně dokončíme.

Perl (69) - Projekt - online přenos

Dnes dokončíme projekt, jehož obsahem bylo vytvořit uživatelsky přívětivou čtečku sportovních výsledků pro příkazový řádek.

Dnes máme za úkol dokončit práce na projektu. Chybí nám napsat poslední věc a tou je online přenos.

Online přenos

Načrtněme si opět předběžný postup.

- Abychom mohli spustit online přenos, musíme nejprve vybrat právě jeden zápas. Až pokud se to úspěšně podaří, můžeme pokračovat dále.
- Vybraný zapas vytiskneme pomocí funkce tiskni_vyhovujici.
- Na základě ligy a čísla zápasu určíme zápas.
- Nyní budeme periodicky volat metodu prenos a na jejím základě aktualizovat zobrazená data.
- Na závěr zobrazíme konečný výsledek

Pojďme se do toho pustit. Bude třeba napsat tuto podmínku, kterou jsme nechali minule neimplementovanou.

```
if($online){
#implementace online přenosu
}
```

Předně musíme ze všech zápasů vybrat pouze jediný. K tomu si napíšeme zvlášť podprogram.

```
@vyhovujici = vyber_zapas(@vyhovujici);
```

To je v podstatě opakované volání metody najdi_zapas_podle_kriterii, která na základě kritérií postupně zužuje výběr. Jakmile klesne na jediný zápas, můžeme výběr ukončit a postupovat dále.

```
sub vyber_zapas {
my @vyhovujici = @_;
while(scalar @vyhovujici > 1){
print "Byly nalezeny zapasy (".scalar @vyhovujici."):\n";
tiskni_vyhovujici(@vyhovujici);
print "Musite zadat zpresnujici kriterium: ";
my $kriteria;
chomp($kriteria = <STDIN>);
my @kriteria = split(" ", $kriteria);
@vsechna_kriteria = (@vsechna_kriteria, @kriteria);
@vyhovujici = $live->najdi_zapas_podle_kriterii(\@vyhovujici, \@kriteria);
}
return @vyhovujici;
}
```

Je ovšem možné, že doplňující kritérium vyloučí všechny zápasy. V takovém případě musíme program ukončit, protože nemáme jak pokračovat. Jinou možností by bylo nebrat poslední kritérium v úvahu.

```
die "Zadny vyhovujici zapas\n" if scalar @vyhovujici == 0;
```

Nyní zjistíme ligu a číslo zápasu, které je třeba předat metodě prenos. Ale je třeba si dát pozor na tuto věc: Číslo zápasu a liga (parametry odkaz_idzapasu a odkaz_liga v hashi s informacemi o zápase) nemusí být vůbec dostupné - resp. nejsou dostupné, pokud zatím v zápase nedošlo ke sledované události. Nastává zde tedy problém. Jak můžeme získat data, když nevíme odkud? Tuto otázku nemá smysl řešit - tato data nezískáme, protože neexistují.

Vyřešíme to trochu jinak a to tak, že si počkáme, až budou dostupná. To znamená průběžně kontrolovat, zda nenastala událost a neobjevil se nám odkaz na seznam událostí. Jakmile událost nastane, získáme odkaz a budeme ji moci zobrazit. Bude to sice o něco náročnější na programování, ale v zásadě tím nic neztrácíme.

Periodicky tak budeme stahovat informace o zápasech. Vždy v nich najdeme ten náš zápas, který sledujeme. Ten nyní může být ve čtyřech stavech.

- hraje se a nejsou dostupné detailní informace (nedošlo k události)
- hraje se a už jsou dostupné detailní informace (došlo k události)
- zápas byl ukončen
- zápas je před výkopem

Je-li zápas před výkopem, budeme v intervalech zjišťovat, zda to stále platí, dokud se nezačne hrát. Když se začne hrát, nejsou zpravidla dostupné žádné události. Proto budeme v intervalech tisknout pouze probíhající minutu. Jakmile ale dojde k události, vyskočíme z cyklu, protože v tomto okamžiku budeme schopni stáhnout podrobnosti k zápasu - již známe odkaz (prvky odkaz_idzapasu a odkaz_liga). Pokud zápas skončil, ukončíme cyklus také. Pokud tyto požadavky zahrneme do další části programu, získáme toto.

Nejprve tedy zkusíme, zda již jsou detaily dostupné.

```
my $vyhovujici_liga = $vyhovujici[0]{"odkaz_liga"};
my $vyhovujici_idzapasu = $vyhovujici[0]{"odkaz_idzapasu"};
my $probihajici_minuta;
```

Pokud ne, budeme si na ně muset počkat. Periodicky budeme kontrolovat, zda již jsou dostupné.

```
while(!$vyhovujici_idzapasu){
@vyhovujici = $live->ziskej_zapasy_dane_ligy();
@vyhovujici = $live->najdi_zapas_podle_kriterii(\@vyhovujici, \@vsechna_kriteria);

$probihajici_minuta = $vyhovujici[0]{"minuta"}, "\n";
last if($vyhovujici[0]{"hraje_se"} == Livescore::UKONCEN);
if($vyhovujici[0]{"odkaz_idzapasu"}){
$vyhovujici_idzapasu = $vyhovujici[0]{"odkaz_idzapasu"};
$vyhovujici_liga = $vyhovujici[0]{"liga"};
}

if($vyhovujici[0]{"hraje_se"} == Livescore::PROBIHA){
print "Probiha $probihajici_minuta'          \r";
}elsif($vyhovujici[0]{"hraje_se"} == Livescore::PRED_VYKOPEM){
print "Zapas zatim nezacal\r";
}
sleep $refresh;
}
```

K tomu, aby se nám neuchovávala data ve výstupním bufferu zatímco je v činnosti funkce sleep je třeba připsat na začátek skriptu tyto dva řádky. Vyprazdňování bufferu musí být automatické.

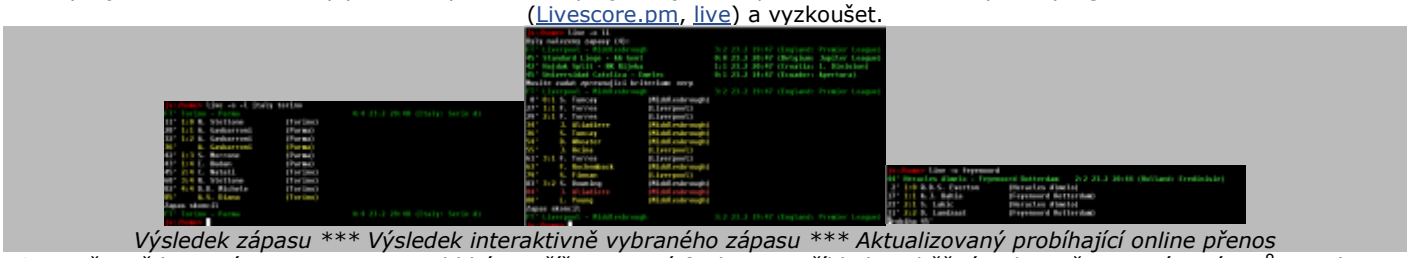
```
use IO::Handle;
STDOUT->autoflush(1);
```

Pokud skončil předchozí cyklus, znamená to, že buď zápas skončil nebo probíhá a jsou dostupné události. V obou případech vytiskneme dostupné události. Jestliže zápas probíhá, budeme navíc tisknout přibyvší události dokud zápas také neskončí.

```
do {
my @udalosti = $live->prenos($vyhovujici_liga, $vyhovujici_idzapasu);
$probihajici_minuta = shift @udalosti;
$hraje_se = shift @udalosti;

#vytiskneme nové události

print "Probiha $probihajici_minuta'\r";
sleep $refresh if $hraje_se == Livescore::PROBIHA;
```

```
}while($hraje_se == Livescore::PROBIHA);
```
Nyní projdeme všechny události a ty, které jsou nové zobrazíme. ID starých událostí budeme kopírovat do pole @zobrazene_udalosti, abychom měli přehled o tom, co již bylo zobrazeno.
```
for (@{$udalosti[0]}){
next if in_array($_->{"id_udalosti"}, \@zobrazene_udalosti);

#zobrazíme událost a zapíšeme do @zobrazene_udalosti

}
```
Nyní zbývají poslední dva kroky. Zvolíme vhodnou formu zobrazení a opět pomocí funkce coloredvytiskneme. Máme 3 druhy události - gól, žlutá karta a červená karta. Na základě prvku udalost můžeme u všech událostí rozhodnout, o který případ jde. Poté aktualizujeme pole @zobrazene_udalosti.
```
if ($_->{"udalost"} eq "yellow"){
print colored(sprintf("%2s'     %-20s (%s)\a\n", $_->{"minuta"},
$_->{"hrac"}, $_->{"tym"}), "bold yellow");
}elsif ($_->{"udalost"} eq "red" or $5 eq "yellow-red"){
print colored(sprintf("%2s'     %-20s (%s)\a\n", $_->{"minuta"},
$_->{"hrac"}, $_->{"tym"}), "bold red");
}elsif ($_->{"udalost"} eq "goal"){
print colored(sprintf("%2s' ", $_->{"minuta"}), "bold white");
print colored(sprintf("%d:%d ", $_->{"skore1"}, $_->{"skore2"}),
"bold yellow");
print colored(sprintf("%-20s (%s)\a\n", $_->{"hrac"}, $_->{"tym"}),
"bold white");
}
push @zobrazene_udalosti, $_->{"id_udalosti"};
}
```
Za cyklus do-while se program dostane pouze v jediném případě - pokud skončil zápas. Tuto informaci tedy vytiskneme a na závěr zopakujeme aktualizovaný výsledek.
```
print "Zapas skoncil\n";
$live->ziskej_zapasy_dane_ligy;
tiskni_vyhovujici($live->najdi_zapas_podle_kriterii(\@vyhovujici, \@vsechna_kriteria));
exit;
```
Závěr

Nyní jsme hotovi. Všechny požadavky ze zadání projektu jsme úspěšně dokončili. Kompletní program si můžete stáhnout (Livescore.pm, live) a vyzkoušet.



*Výsledek zápasu *** Výsledek interaktivně vybraného zápasu *** Aktualizovaný probíhající online přenos*

Samozřejmě by nyní tento program mohl být rozšířen o nové funkce. Například souběžné online přenosy více zápasů najednou, lepší členění pomocí přepínače -l a s tím související vylepšení cachování nebo přepis do Curses, Qt či jiného pokročilejšího uživatelského rozhraní.

Nicméně náš původní cíl je nyní splněn a proto tato šestidílná série, jejíž cílem bylo předvést některé techniky probrané v seriálu i z trochu jiného pohledu, dneškem končí.

Perl (70) - Plain Old Documentation

Perl nabízí vlastní systém tvorby dokumentace. Lze ji psát přímo do zdrojového kódu programů a snadno se exportuje do jiných formátů.

Plain Old Documentation (POD) je jednoduchý značkovací jazyk, používaný jako dokumentace v jazyce Perl. Specialitou formátu POD je, že se dokumentace píše do stejného souboru jako samotný zdrojový kód a může se s ním libovolně prolínat. (Nicméně není to podmínkou a lze vytvořit také samostatný .pod soubor.)

Nespornou výhodou plynoucí z této skutečnosti je, že můžeme generovat dokumentaci (například ve formě manuálových stránek) přímo ze skriptů.

Ve formátu POD je dnes zdokumentována řada aplikací v Perlu - zejména pak moduly v archivu CPAN.

Prohlížení POD

POD dokumentaci většinou neprohlížíme, ale konvertujeme na nějaký jiný formát dokumentace. Nicméně základním příkazem pro zobrazování POD dokumentace je příkaz perldoc, kterému se předává název POD stránky s dokumentací nebo přímo zdrojový soubor. Například pro zobrazení dokumentace k modulu Net::POP3 zadáme příkaz

```
$ perldoc Net::POP3
```

Mezi nejzajímavější přepínače příkazu perldoc patří tyto.

Přepínač	Význam
-m	zobrazí se celý zdrojový kód modulu
-f *funkce*	zobrazí se popis příslušné funkce podle perlfunc
-q *klíčové_slovo*	zobrazí se FAQ otázky, odpovídající zadanému klíčovému slovu

Více informací lze nalézt v manuálové stránce perldoc(1).

Struktura

Pro modul zdokumentovaný v POD existuje jistá nezávazná konvence, která určuje jména možných oddílů. Samozřejmě lze podle potřeby volit jiné názvy. Vždy bychom však měli pojmenování pečlivě zvážit, neboť na něm do značné míry záleží srozumitelnost dokumentace.

Oddíl	Význam	Popis
NAME	název	jméno modulu a několikaslovný popis
SYNOPSIS	charakteristika	rychlá informace, jak program použít
DESCRIPTION	popis	rozsáhlejší pojednání o funkci programu
BUGS nebo CAVEATS	chyby	problémy, jež modul zatím obsahuje
SEE ALSO	příbuzná témata	odkazy na související manuálové stránky, WWW adresy apod.
AUTHOR	autor	autoři programu a kontakty
HISTORY	vývoj	předchozí verze modulu
COPYRIGHT nebo LICENSE	licence	podmínky užívání, vlastnická práva apod.

Syntaxe

POD se do zdrojového kódu Perlu vnořuje pomocí speciálních klíčových slov. Před každým klíčovým slovem POD je znak =. Pro studium jazyka POD je nejlepší cestou prohlížení zdrojových kódů zdokumentovaných modulů. My si zde představíme základní příkazy.

Začátek bloku dokumentace

POD lze začít jakýmkoliv POD příkazem. Protože někdy nemusíme chtít začít klasickým příkazem, ale pouze pokračovat v textu, existuje zde příkaz =pod, který nedělá nic. Je tedy užitečný právě k signalizaci začátku bloku dokumentace.

Konec bloku dokumentace

Uvedením =cut na samostatný řádek ukončíme blok dokumentace. Některé POD parsery vyžadují také prázdný řádek před =cut.

Titulky

Nadpisy se vytvářejí příkazem =head*n*, kde *n* je úroveň nadpisu. Titulky nejvyšší úrovně se píší velkými písmeny.

Styl písma

Pro změnu stylu textu zde jsou jednopísmenné příkazy, které se vztahují na text v lomených závorkách, jež za příkazem následují. Lze využít následující příkazy:

Zápis	Význam
B<text>	programy, přepínače (tučné)
C<text>	zdrojový kód
I<text>	zvýraznění, proměnné (kurzíva)
L<stránka>	odkaz na jinou stránku
E<jméno znaku>	escape znak
X<položka>	buď ignorováno nebo pro vytváření indexů
F<jméno>	zvýraznění názvu souboru
S<text>	mezery v textu jsou nedělitelné
Z<>	prázdný znak

Pokud se někde ve formátovaném textu vyskytuje znak >, musíme zajistit, aby se nepletl se znakem, jež ukončuje formátování. Nejjednodušším řešením je jeho nahrazením escape sekvencí E<gt>.

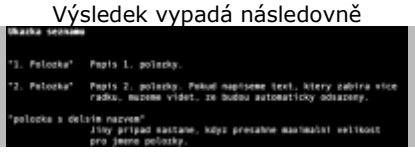
Seznamy

Seznamy se uvádějí mezi příkazy =over a =back. Příkaz =over přijímá jako argument počet znaků, o které budou položky výčtu odsazeny. Jednotlivé položky výčtu jsou pak uvozeny příkazem =item.

Následuje krátký příklad seznamu, který zachycuje jeho možnosti.

```
=head2 Ukazka seznamu

=over 15

=item C<1. Polozka>

Popis 1. polozky

=item C<2. Polozka>

Popis 2. polozky. Pokud napiseme text, ktery zabira vice radku, muzeme videt, ze
budou automaticky odsazeny.

=item C<dalsi polozka>

Jiny pripad nastane, kdyz presahne maximalni velikost pro jmeno polozky

=back
```

Výsledek vypadá následovně



Ukázka seznamu

Specifické bloky

Jak uvidíme níže, POD lze konvertovat do jiných formátů dokumentace. POD nám umožňuje definovat bloky kódu pouze pro určitý formát.

Pomocí klíčových slov =begin *formát* a =end *formát* lze takový blok textu napsat. Například pokud budeme chtít uvést v dokumentaci nějaké schéma, pro HTML dokumentaci použijeme PNG obrázek, pro textovou dokumentaci semigrafiku atd.

```
=begin html

<IMG SRC="schema.png">

=end html

=begin text

+-------------+      +-------------+
    |           |      |           |
|   NODE 1    |-------|    NODE 2   |
    |           |      |           |
+-------------+      +-------------+

=end text
```

Koverze POD do jiných formátů

Je-li zdrojový kód zdokumentovaný pomocí POD, lze z něj kdykoliv generovat jiné formy dokumentace pomocí nástrojů pod2*formát*. Mezi takové příkazy patří například tyto:

Příkaz	Význam
pod2html	vytváří HTML dokumentaci
pod2man	vytváří dokumentaci ve formátu troff užívaného pro manuálové stránky
pod2text	vytváří prostý text
pod2latex	vytváří dokumentaci ve formátu LaTeX

Uložme do souboru priklad.pod náš poslední příklad a zkusme následující dva příkazy.

```
$ pod2text priklad.pod
$ pod2html priklad.pod
```

Příklad - dokumentace programu live

Ukažme si jen velice stručný příklad POD stránky. Takto by mohla vypadat dokumentace k programu live, který jsme vytvořili v předcházející šestidílné sérii.

```
=head1 NAME

live - online monitoring of soccer

=head1 SYNOPSIS

live [-o] [-l league] [pattern]

=head1 DESCRIPTION

live is based on Livescore perl module. live acquires data from Livescore and
displays informations about match in text format.

=head1 OPTIONS

=item B<    -o, --online>

online transmission

=item B<    -l, --league> I<region>

display only matches from selected region.

=item B<    -r, --refresh>

time interval for refresh (default 60 sec.)

=item B<    -h, --help>

display help message and exit

=head1 FILES

temporary files /tmp/livescore_*

=head1 SEE ALSO

Livescore(3pm)
```

```
=head1 AUTHOR

Jiri Vaclavik

=head1 COPYRIGHT AND LICENSE

This program is free software; you can redistribute it and/or modify it under the
terms of the GNU General Public License as published by the Free Software Foundation

=cut
```

Příkazem perlpod dostaneme následující výstup

Dokumentace k programu live

Funkce tie umožňuje přetěžovat takové operace jako čtení z proměnné nebo zápis. Obyčejné přiřazení tak může vykonat daleko více než jen přiřazení.

Mechanizmus tie umožňuje při běžné manipulaci s proměnnými provádět definované akce. Sami si tak obsluhujeme takové rutiny jako je ukládání dat do proměnné nebo čtení jejich obsahu.
V praxi to funguje tak, že si pomocí tie svážeme libovolnou proměnnou s nějakým námi definovaným objektem a při manipulaci s touto navázanou proměnnou se místo obvyklé akce volá určená metoda z objektu. Proměnná tak již nebude řízena Perlem, ale námi napsanou třídou.
Obecná syntaxe příkazu vypadá následovně.
tie $proměnná, "Třída" [, @seznam];
Návratovou hodnotou příkazu tie je již konkrétní objekt, se kterým je proměnná svázána. @seznam je seznam hodnot, které budou předány konstruktoru.
Proměnná je skalár, pole, hash nebo ovladače souboru, který chceme svázat s třídou Třída.
Provádění příkazu tie zahrnuje několik důležitých kroků. Nejprve si Perl určí datový typ předávané proměnné. Dále je volána metoda objektu - pro skaláry je to metoda TIESCALAR, pro pole TIEARRAY, pro hashe TIEHASH a pro ovladače TIEHANDLE. Tyto čtyři metody jsou v podstatě konstruktory, které vrací nyní už konkrétní objekt. Právě tento objekt je s proměnnou svázán.
Nyní je při každé akci s danou proměnnou (uložení hodnoty do proměnné apod.) vyvolána určitá metoda třídy. Která metoda se vyvolá záleží na typu akce (jiná metoda se vyvolá při ukládání a jiná při čtení).
Navázání proměnné můžeme zrušit příkazem untie. Jakmile zavoláme untie, provede se metoda UNTIE. Nakonec po zrušení všech odkazů na proměnnou (při zániku proměnné) se volá destruktor - metoda DESTROY.
Nyní si rozebereme jednotlivé metody pro všechny dostupné datové typy.

Navazování skalárů

Jak již víme, konstruktorem navázaného objektu pro skaláry je metoda TIESCALAR.
Pokud se nyní pokusíme do navázané proměnné uložit hodnotu, vyvolá se další speciální metoda STORE. Ona ukládaná hodnota se však ve skutečnosti nikam neuloží, ale dostaneme ji jako argument metody STORE.
Podobně, když obsah proměnné čteme, vyvolá se metoda FETCH.

Užití navazování

Zkusme si nyní představit, jak bychom užili mechanizmus tie v praxi. tie bychom teoreticky mohli použít například jako řízení regulačních tyčí jaderného reaktoru. Při zápisu do proměnné by se hodnota poslala do centra, které řídí regulační tyče a ty by se automaticky o tuto hodnotu posunuli. Naopak při čtení proměnné bychom si požádali o aktuální stav.
Další analogií by mohlo být měření stavu počasí - typicky například barometr. Při čtení bychom dostávali informaci o aktuální hodnotě tlaku například v nějaké nádobě. Zápis do proměnné by naopak sloužil pro jeho regulaci, případně bychom ho mohli ignorovat.

Příklad

Barometr ani teplotní senzory však většina lidí k počítači připojeny nemá. Napíšeme si tedy o něco jednodušší program. Při zápisu do proměnné budeme chtít přepsat získanou hodnotou určený soubor. Naopak při čtení z tohoto souboru data získáme.
Toho by se dalo teoreticky využít například pro uchovávání obsahu proměnných mezi několika instancemi programu.
Prvním úkolem tedy je napsat modul StoreToFile, jež bude manipulaci s proměnnými řídit. Náš modul bude obsahovat tři metody. První z nich bude konstruktor TIESCALAR. TIESCALAR bude vyžadovat název souboru jako argument. Pokud ho neuvedeme, implicitně se použije soubor tiefile. Název souboru vzápětí zkonvertujeme na objekt.

```
package StoreToFile;

sub TIESCALAR {
my($pkg, $soubor) = @_;
```

```
$soubor = "tiefile" unless $soubor;
bless \$soubor, $pkg;
return \$soubor;
}
```

Dále definujeme metodu STORE. Ta bude volána vždy při zápisu do proměnné. Získaná hodnota (hodnotu získáme jako argument metody) bude zapsána do souboru, jehož jméno uchovává objekt.

```
sub STORE {
my($r_fh, $hodnota) = @_;
open FILE, ">$$r_fh" or die;
print FILE $hodnota;
close FILE;
}
```

A nakonec ještě napíšeme metodu FETCH, jež bude vracet obsah souboru ve formě řetězce.

```
sub FETCH {
my($r_fh) = @_;
open FILE, "$$r_fh" or die;
my $data = <FILE>;
close FILE;
return $data;
}
```

To je téměř vše. Definice třídy je již hotova. Jak se můžeme přesvědčit, jde o normální třídu, kterou bychom mohli běžným způsobem používat. Nápadné jsou pouze názvy metod. Nicméně je třeba připomenout, že tyto metody se chovají speciálně pouze pokud instanci třídy svážeme pomocí tie.

Nyní třídu použijeme tak, že ji svážeme s proměnnou příkazem tie. Zároveň konstruktoru předáme název souboru parametr.

```
tie $a, "StoreToFile", "soubor";
```

A následně můžeme testovat. Uložíme do proměnné $a jakoukoliv hodnotu.

```
$a = "Obsah svázané proměnné";
```

Můžeme se přesvědčit, že vznikl soubor soubor, který obsahuje text přiřazený proměnné $a. To je důkazem skutečnosti, že se vykonala metoda STORE svázaného objektu.

Poznámka - Předcházející přiřazení do proměnné $a bychom mohli přepsat jako volání metody nad objektem. Tento objekt získáme pomocí příkazu tied, jež právě navázaný objekt vrací (a nebo jako návratovou hodnotu příkazu tie). Následující dva příkazy tedy dělají to samé.

```
$a = "Obsah svázané proměnné";
(tied $a)->STORE("Obsah svázané proměnné");
```

A dále zkusíme proměnnou přečíst. Nyní se vyvolá metoda FETCH, která přečte daný soubor a získaná data vrátí.

```
print $a;
```

Přetížení pouze vybraných operací pomocí modulu Tie::Scalar

Pokud se už rozhodneme přetížit operace se skalárními proměnnými, musíme přetížit všechny. To je zbytečné v případech, kdy od jedné nebo více metod chceme implicitní chování.

Přímo se tedy nabízí vytvořit nějakou třídu s metodami, jejichž chování bude stejné s tím, jak manipuluje s proměnnými Perl. Taková třída již existuje, jmenuje se Tie::StdScalar a patří do standartní výbavy modulů. Pokud tedy necháme námi napsanou třídu dědit od Tie::StdScalar, můžeme přetížit jen požadované metody.

Ukážeme si to na krátkém příkladu. Napíšeme třídu, která zaokrouhlí výstupní hodnotu na dvě desetinná místa. Importujeme tedy Tie::Scalar, ve kterém je i definice Tie::StdScalar. A jediné, co dále potřebujeme je nastavení předka naší třídy.

```
package StoreToFile;
use Tie::Scalar;
@ISA = ("Tie::StdScalar");

sub FETCH {
return sprintf("%.2f", ${$_[0]});
}
```

Perl (72) - Navazování složitějších datových typů

Pokračujeme v tématu předchozího dílu. Budeme se zabývat méně obvyklými záležitostmi při navazování proměnných.

Podobně jako skaláry můžeme navazovat složitější datové typy. Nejčastěji se používá navazování hashů, které je používáno zejména systémem DBM. Více bude o DBM řečeno v příštím díle. Nicméně dnes se budeme zabývat i navazováním ovladačů souborů.

Na začátku uveďme, že stejně jako pro skaláry je možné pro přetížení jen vybraných operací použít třídy Tie::StdArray a Tie::StdHash definované v modulech Tie::Array a Tie::Hash.

S hashi a poli lze provádět mnohem zajímavější věci než se skaláry. Představme si, že bychom přes hash přistupovali například k encyklopedii. Takový hash bychom vůbec nemuseli definovat - stačilo by ho pouze svázat s modulem, který by se staral o získávání informací. Ten by mohl data získávat třeba i z Wikipedie. Okamžitě bychom tak měli dostupná hesla pod příslušnými klíči.

Navazování hashů

Navazování hashů je to o něco složitější než navazování skalárů, nicméně tato složitost tkví zejména v počtu metod, které musíme vytvořit. Je to logické, protože si stačí uvědomit, co všechno lze s hashem provádět.

Konstruktorem třídy bude metoda TIEHASH. Dále zde opět máme metody FETCH a STORE. Jako argumenty dostávají klíč hashe pro FETCH, resp. klíč a hodnotu pro STORE.

Narozdíl od skaláru zde však máme navíc další metody. V hashovém kontextu můžeme využívat funkce exists a delete. Jejich chování určují metody EXISTS a DELETE. Obě dostávají jako argument klíč hashe.

Dále zde máme metodu CLEAR, která je volána tehdy, pokud do hashe přiřazujeme prázdný seznam. Pokud přiřazujeme neprázdný seznam, není třeba definovat zvláštní metodu, neboť takové volání se převede na volání CLEAR a STORE.

A nakonec je třeba definovat metody FIRSTKEY a NEXTKEY, které obsluhují volání funkce keys (funkce values a each jsou pouze kombinací FIRSTKEY, NEXTKEY a FETCH). Metoda FIRSTKEY by měla vracet název prvního klíče a NEXTKEY postupně klíče

následující. Pokud tedy nad navázaným hashem zavoláme funkci keys, provede se metoda FIRSTKEY. Pokud zavoláme keys znovu, provede se NEXTKEY. Atd.

Příklad

Nyní si pro jasnější představu napíšeme konkrétní třídu, ke které budeme moci hash navázat. Obsah nenavázaných proměnných se uchovává v paměti RAM. Naše třída bude proměnné navázané na ní uchovávat v adresáři. Budeme tedy pracovat s hashem a veškeré operace s ním se budou odehrávat na úrovni práce se soubory.

Tedy konkrétně. Vytvořením prvku hashe se vytvoří v daném adresáři soubor o stejném názvu jako je klíč hashe. Obsahem souboru pak bude hodnota prvku. Prvky budeme moci přepisovat, mazat apod.

Přikročme tedy k definici třídy. Konstruktor TIEHASH bude přijímat název adresáře, z něhož vytvoří objekt. Dále tento adresář vytvoří. Pro jednoduchost nebudeme ošetřovat případy, kdy by bylo třeba vytvořit více podadresářů.

```
package TieHash;

sub TIEHASH {
my($pkg, $adresar) = @_;
$adresar = "hash" unless $adresar;
mkdir $adresar unless -e $adresar;
bless \$adresar, $pkg;
return \$adresar;
}
```

A nyní již budeme definovat metody reprezentující operace s hashem. STORE na základě přijatých argumentů vytvoří v daném adresáři soubor s daným obsahem.

```
sub STORE {
my($r_dir, $soubor, $obsah) = @_;
open FILE, ">$$r_dir/$soubor" or die;
print FILE $obsah;
close FILE;
}
```

Metoda FETCH přečte soubor, reprezentující daný index a vrátí jeho obsah.

```
sub FETCH {
my($r_dir, $soubor) = @_;
return "" unless -e "$$r_dir/$soubor";
open FILE, "$$r_dir/$soubor" or die;
my $obsah = <FILE>;
close FILE;
return $obsah;
}
```

Metoda EXISTS pouze vrátí pravdivou nebo nepravdivou hodnotu, podle toho, zda daný soubor existuje.

```
sub EXISTS {
my($r_dir, $soubor) = @_;
return -e "$$r_dir/$soubor" ? 1 : "";
}
```

Volání funkce delete má za následek smazání daného prvku. Zároveň zajistíme, aby byla vrácena hodnota tohoto prvku.

```
sub DELETE {
my($r_dir, $soubor) = @_;
return "" unless -e "$$r_dir/$soubor";
my $obsah = $r_dir->FETCH($soubor);
unlink "$$r_dir/$soubor";
return $obsah;
}
```

Přiřazením prázdného seznamu do hashe se všechny klíče hashe vymažou.

```
sub CLEAR {
my($r_dir) = @_;
unlink <$$r_dir/*>;
}
```

A nakonec nám zbývají metody FIRSTKEY a NEXTKEY. Zde vyvstává otázka, jak uchovávat informaci mezi jednotlivými voláními metody. Potřebujeme totiž vědět, které klíče již byly vráceny a které zatím ne. Nejčastější možností je uchovávání celého hashe v objektu. Potom bychom mohli pomocí funkce each volat pokaždé následující prvek. Nicméně my v objektu máme pouze jméno adresáře. Pro jednoduchost definujeme globální proměnnou $posledni, která říká kolikátý byl poslední vrácený klíč.

```
our $posledni;
```

Nyní definujeme metodu FIRSTKEY, která vrací první název souboru z daného adresáře.

```
sub FIRSTKEY {
my($r_dir) = @_;
my($s) = <$$r_dir/*>;
my($a) = $s =~ /$$r_dir\/(.*)/;
$posledni = 0;
return $a;
}
```

A nakonec metodu NEXTKEY, která nejprve do pole načte všechna jména souborů a poté z nich vybere podle obsahu globální funkce $posledni.

```
sub NEXTKEY {
my($r_dir) = @_;
my @files;
push @files, /$$r_dir\/(.*)/ while <$$r_dir/*>;
return $files[++$posledni] ? $files[$posledni] : undef;
}
```

Třída je hotova. Nyní ji můžeme začít navazovat na konkrétní hashe.

```
my %a;
tie %a, "TieHash";
```
Dále můžeme zkoušet pracovat s prvky hashe %a a zároveň sledovat obsah vzniklého adresáře hash. Například následující příkaz má za následek vytvoření souboru hash/klic s obsahem hodnota.
```
$a{"klic"} = "hodnota";
```
Další příkaz smaže obsah adresáře a poté v něm vytvoří soubor novy s obsahem hodnota.
```
%a = ("novy" => "hodnota");
```
A nakonec si vypíšeme obsahy všech souborů adresáře hash.
```
print values %a;
```
Navazování polí

Konstruktorem je opět metoda TIEARRAY. Dále máme k dispozici nám již známé FETCH a STORE. A dále lze ještě využít metody DELETE, EXISTS, CLEAR, POP, PUSH, SHIFT, UNSHIFT a SPLICE, jejichž význam je intuitivní. Metoda FETCHSIZE se volá po požadavku na délku pole (to může být buď $#pole nebo scalar $pole). STORESIZEnaopak definuje chování pro změnu jeho velikosti.

Příklad

V příkladu, který si uvedeme, budeme definovat pouze některé z výše uvedených metod, neboť rozsáhlejší příklad jsme si již předvedli u navazování hashů.

Do prvku pole budeme přiřazovat vždy název souboru. Naše třída zajistí, že hodnotou prvku bude obsah tohoto souboru a ne už jeho název.
```
package TieArray;
use Tie::Array;
our @ISA = ("Tie::StdArray");

sub TIEARRAY {
my($pkg, $adresar) = @_;
my $self = [];
bless $self, $pkg;
return $self;
}

sub STORE {
my($self, $index, $soubor) = @_;
return "" unless -r $soubor;
open FILE, "$soubor";
$self->[$index] .= $_ while <FILE>;
close FILE;
}

sub PUSH {
my($self, @nove_hodnoty) = @_;;
my $i = -1;
$self->STORE($#$self+1, $nove_hodnoty[$i]) while $nove_hodnoty[$i++];
}
```
Nyní můžeme naši třídu použít. Po přiřazení do proměnné se její hodnotou ve skutečnosti stane obsah souboru nebo prázdný řetězec.
```
my @a;
tie @a, "TieArray";

$a[0] = "/home/user/soubor";
print $a[0];
```
Navazování ovladačů

Odpovědnost za práci s ovladači můžeme jako u předcházejících datových typů přenést z Perlu na sebe. Opět to spočívá ve vytvoření speciální třídy, uvnitř které jsou definovány speciální metody.

Protože ovladač nemá vlastní prefix, předává se funkci tie typeglob.

Máme také k dispozici modul Tie::Handle, ve kterém je definována třída Tie::StdHandle.

Konstruktorem třídy, na kterou budou ovladače navázány, je metoda TIEHANDLE. Dále máme určeny pro jednotlivé operace s ovladačem další názvy metod.

Metody PRINT, PRINTF, GETC, OPEN, CLOSE, BINMODE, FILENO, SEEK, TELL, EOFkorespondují s funkcemi o stejném názvu, pouze malým písmem, WRITE s funkcí syswrite a READ s funkcemi read a sysread.

Dále zde máme funkci READLINE, která je volána při čtení z ovladače pomocí operátoru <>.

Perl (73) - DBM

Jednou z aplikací funkce tie je také systém DBM, který slouží k perzistentnímu uchovávání datových struktur.

Běžným způsobem uložení dat je soubor. Soubory se výborně hodí pro předávání dat. Horší to však už je s jejich úpravou. Vytvoření, smazání a ještě oříznutí zvládneme celkem snadno. Ale jak třeba přidat nebo ubrat text z jeho prostředku? Nebo jak ze souboru získat jednu konkrétní informaci, o níž netušíme, na které pozici je? Tady už máme problém. Je zřejmé, že si s pouhými soubory vždy nevystačíme.

Abychom vyřešili náš problém, můžeme použít databázi. Databáze jsou obecně nějaká data s prostředky pro manipulaci s nimi, které jsou dostupné oprávněným uživatelům.

Pro pouhé ukládání seznamů ve formátu klíč - hodnota lze použít databázi DBM. Pokud potřebujeme něco komplexnějšího, použijeme některou z nabídky SQL databází.

Ukládání hashů

Jak již bylo zmíněno, DBM (DataBase Manager) slouží pro uchovávání dat o struktuře klíč - hodnota. To je v podstatě perzistentní uchovávání hashových proměnných.

Tato jednoduchost má své výhody i nevýhody. Největší výhodou je náročnost. Mimo Perlu a několika modulů nepotřebujeme nic.

Nevýhodou jsou zejména malé možnosti struktury ukládání dat. Tuto nevýhodu částečně eliminují různé postupy, pomocí kterých lze ukládat složitější datové struktury do řetězce.
DBM pro svoji činnost využívá mechanizmu tie. Pomocí něj napojíme nějaký hash na příslušnou třídu: DB_File nebo jí podobnou.
Předvedeme si jednoduché použití knihovny Berkeley DB - tedy modulu DB_File. Vytvoříme program pro správu dat. Po jeho spuštění se objeví menu, které nabídne tři možnosti - Přidat nebo odebrat položku a výpis položek. Tyto datové položky budou perzistentní - budou uloženy i poté, co program skončí a při dalším spuštění je opět načteme.

```
use DB_File;

my %hash;
$SIG{"INT"} = sub {untie %hash; exit;};

tie %hash, "DB_File", "data.db" or die;
while(1){
print "1. Přidat\n2. Odebrat\n3. Zobrazit\n";
my $p = <STDIN>;
my($polozka, $hodnota);
if(/^1/){
print "Zadej položku: "; chomp($polozka = <STDIN>);
print "Zadej hodnotu: "; chomp($hodnota = <STDIN>);
$hash{$polozka} = $hodnota;
}elsif($p =~ /^2/){
print "Zadej položku: "; chomp($polozka = <STDIN>);
delete $hash{$polozka};
}elsif($p =~ /^3/){
printf "%-20s %-20s\n", $_, $hash{$_} for sort keys %hash;
}else{
last;
}
}
untie %hash;
```

Mimo Berkeley DB lze použít ODBM (modul ODBM_File), Simple DBM (SDBM_File), New DBM (NDBM_File) nebo GNU DBM (GDBM_File). Tyto knihovny se mimo jiné liší formátem zápisu, takže knihovnu SDBM nelze číst přes GDBM apod. Nicméně práce s nimi je velmi podobná. Také existuje modul AnyDBM_File, který použije tu nejlepší dostupnou variantu. V jeho dokumentaci najdete mimo jiné i srovnání těchto variant podle kritérií.
Pokud nepoužíváme Berkeley DB, je třeba zadat mód otevření a přístupová práva, jež budou aplikována na datový soubor.

```
use POSIX;
tie %hash, "NDBM_File", "data.db", O_RDWR|O_CREAT, 0644 or die;
```

Další možnost, jak pracovat s DBM vede přes zvláštní funkci dbmopen, která funguje podobně jako tie.
Pro smazání všech položek seznamu stačí přiřadit do navázaného hashe prázdný seznam.

Ukládání složitých datových struktur

Pokud chceme jako hodnoty hashe v DBM použít něco jiného než skaláry, musíme sáhnout k tomu určeným nástrojům. V 48. dílu seriálu jsme si představili dva moduly - Storable a Data::Dumper. Tyto moduly umožňují převést libovolnou datovou strukturu Perlu na řetězec a zpět. A abychom nemuseli ručně používat jeden z těchto modulů, existuje přímo modul s názvem MLDBM (MultiLevel DBM), který to dělá automaticky.
MLDBM ke své činnosti potřebuje jeden z modulů Data::Dumper, FreezeThaw nebo Storable. Data::Dumper je implicitní. Nastavit lze také který modul implementace DBM bude používán. Na výběr máme DB_File, SDBM_File (implicitní), NDBM_Filenebo GDBM_File.
Nejprve tedy vyzkoušíme uložení do souboru. Použijeme přitom modul Storable a DB_File.

```
use MLDBM qw(DB_File Storable);
my %hash;
tie %hash, "MLDBM", "data.db";

%hash = (
"klic0" => ["hodnota0A", "hodnota0B"],
"klic1" => ["hodnota1A", "hodnota1B", "hodnota1C"],
"klic2" => ["hodnota2A", "hodnota2B", "hodnota2C"],
"klic3" => ["hodnota3A", "hodnota3B"]
);
untie %hash;
```

Nyní můžeme ze souboru data.db číst. Vytiskneme celou uloženou strukturu pomocí Data::Dumper.

```
use MLDBM qw(DB_File Storable);
use Data::Dumper;

my %hash;
tie %hash, "MLDBM", "dbm_struct.db";
print Dumper \%hash;
untie %hash;
```

Menší problém nastane, pokud budeme chtít měnit zanořená data. V takovém případě nelze přistupovat přímo přes strukturu, ale musíme pomocí jiných proměnných změnit odkazy. Tedy asi takto.

```
my $tmp = $hash{"klic2"};
$tmp = "NOVA HODNOTA";
$hash{"klic2"} = $tmp;
```

Perl (74) - Sockety

Náplní dnešního dílu bude meziprogramová komunikace. Osvětlíme si pojem socket, jeho vlastnosti a nezbytnou součástí bude i názorná ukázka použití.

Socket je virtuální spojení dvou procesů. Důležitě je zde slovo spojení. Z něj vyplývá to, že sockety jsou způsobem meziprocesorové komunikace. V praxi to znamená, že můžeme napsat dva programy, které spolu budou moci vzájemně komunikovat. První program - server - bude naslouchat na určeném portu (port je číslo reprezentující vstupní cestu do počítače) a bude čekat na přípojení druhého programu - klienta. V okamžiku, kdy se klient připojí, vznikne spojení a od této chvíle mohou obě strany komunikovat. Oba programy mohou běžet na různých počítačích, které jsou síťově propojeny.
Existují dva způsoby komunikace. Se spojením nebo beze spojení - tedy TCP nebo UDP. Typickým příkladem komunikace bez spojení je email. Mezi odesílatelem a příjemcem se nevytváří žádné spojení - odeslání emailu není závislé na příjemci a přijetí na odesílateli. Není zaručeno pořadí jednotlivých zpráv ani to, že někdo zprávu skutečně přijme. Naopak mezi komunikaci se spojením patří telefonní hovor. Oba účastnící jsou v jednom okamžiku spojeni komunikačním kanálem, který zaručuje správné pořadí zpráv.

Sockety v Perlu

V Perlu máme pro práci se sockety k dispozici moduly Socket a IO::Socket. My si představíme objektově orientovaný IO::Socket. Ten obsahuje dvě podtřídy. INET a UNIX. Prvně jmenovaná pro síťovou komunikaci a UNIX pro komunikaci lokální. O ní se však zmíníme jen letmo na konci. Máte-li zájem o modul Socket, nahlédněte do stránky man perlipc(1), jež se zabývá meziprocesorovou komunikací.

Příklad - jednoduchý chat

Jako nejjednodušší příklad aplikace s využitím socketů si napíšeme jednoduchý chat, pomocí kterého budou moci dva lidé připojení u různých počítačů ve stejné síti střídavě posílat zprávy.
Ještě než začneme si musíme ujasnit, co vlastně budeme psát. Potřebujeme dva programy - server a klienta.
Činnost serveru spočívá v naslouchání na daném portu. V okamžiku, kdy se na tento port přípojí klient, mohou oba programy vzájemně komunikovat. Komunikace bude probíhat následovně: Server obdrží zprávu od klienta a pošle mu odpověď. A to se bude opakovat donekonečna.

Server

Začneme tedy serverem. Nejprve vytvoříme socket pomocí metody new, jejíž parametry jsou patrné z následujícího zdrojového kódu.

```
use IO::Socket;

my $port = 7777;
my $ip = "192.168.0.10";

my $server = IO::Socket::INET->new(
    Proto     => "tcp",
    LocalPort => $port,
    LocalAddr => $ip,
    Listen    => 1,
    Reuse     => 1
) or die "Chyba pri vytvareni serveru! $!";
```

Právě jsme vytvořili socket server na ip adrese 192.168.0.10, který bude naslouchat na portu 7777. Port musíme zvolit tak, aby ho nepoužívalo více aplikací, jinak dojde ke konfliktům. Parametrem Listen určujeme nejvyšší možný počet klientů. Reusezajistí, že po ukončení programu bez uzavření socketu se port opět uvolní.
Teď přichází chvíle pro naslouchání. Budeme naslouchat tak dlouho, dokud se nepřipojí nějaký klient. Naslouchání se provádí metodou accept, jež vrací socket, kterým lze komunikovat s klientem.

```
$klient = $server->accept();
```

Ve skalárním kontextu metoda accept vrací nový socket, ale pokud bychom přiřazovali do pole, druhým prvkem by byla ip adresa klienta.
Nyní přichází na řadu samotná komunikace s klienty. Směr komunikace si určuje samotná aplikace. Proto může dojít k případům, kdy oba programy spojené socketem čtou nebo zapisují do kanálu zároveň, což způsobuje uváznutí. To by se ve správně napsané aplikaci nemělo nikdy stát.
Vytvoříme tedy cyklus, uvnitř kterého budeme nejprve očekávat nějakou zprávu od klienta a následně pošleme odpověď. Se socketem pracujeme stejně jako s ovladačem.
V této souvislosti je dobré poznamenat, že by odesílaná data měli vždy končit znakem nového řádku, případně jiným znakem, který jeho funkci zastupuje.

```
while (1){
$prichozi = <$klient>;
print "PRIJATO: $prichozi";
print "Zadej text, ktery chces odeslat: ";
$odchozi = <STDIN>;
print $klient $odchozi;
}
```

Klient

Vytvoření klienta, jak bude za chvilku patrné, je podobné.

```
use IO::Socket;

my $port = 7777;
my $ip_serveru = "192.168.0.10";

my $vzdaleny = IO::Socket::INET->new(
    Proto    => "tcp",
    PeerAddr => $ip_serveru,
    PeerPort => $port
) or die "Nelze spojit! $!";
```

Tato část klienta má za úkol zjistit, zda na portu 7777 počítače 192.168.0.10 naslouchá nějaký server a pokud tam skutečně naslouchá, připojit se k němu. Pokud ne, program se jednoduše ukončí.
Dále následuje už komunikace, která je až na směr stejná jako u serveru. První akce serveru bylo přijetí zprávy. Proto klient nejprve data odešle a poté bude očekávat odpověď.

```
while (1){
```

```
print "Zadej text, ktery chces odeslat: ";
$odchozi = <STDIN>;
print $vzdaleny $odchozi;
$prichozi = <$vzdaleny>;
print "PRIJATO: $prichozi";
}
```

Na závěr by běžný klient ukončil spojení. Ten náš ovšem běží nekonečně dlouho (nedostane se za cyklus), takže následující řádek ve zdrojovém kódu nepoužijeme.

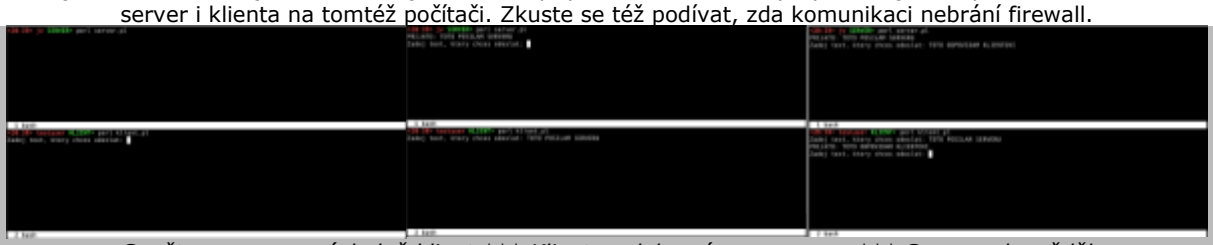
```
close $vzdaleny;
```

Výsledky

Zkuste si též změnit směr jednoho z programů - například, aby oba nejdříve posílaly data. Program uvázne, neboť oba data vysílají, ale nikdo je nepřijímá.

Nyní jsme hotovi a můžeme zkoušet. Zdrojové kódy klienta i serveru si můžete stáhnout. Zkusíte-li nyní na počítači 192.168.0.15 spustit server a na jiném počítači klienta, měl by náš chat fungovat. V okamžiku, kdy spouštíte klienta, musí pochopitelně již běžet server.

Pokud to z nějakého důvodu nejde, testování je v mnoha případech dobré začít přepsáním jména počítače na localhost a spustit server i klienta na tomtéž počítači. Zkuste se též podívat, zda komunikaci nebrání firewall.



*Spušten server a následně klient *** Klient poslal zprávu na server *** Server odpověděl*

Další možnosti

Se sockety lze mnoha způsoby experimentovat. Za vyzkoušení stojí připojení se na HTTP port 80. Napišme si zde jednoduchý skript, který vypíše systém, na kterém běží server. Tedy odešleme HTTP požadavek na nějakého vzdáleného hostitele a budeme čekat odpověď.

```
use IO::Socket;
use strict;

my $host = $ARGV[0];
my $sock = new IO::Socket::INET(
PeerAddr => $host,
PeerPort => 80,
Proto    => "tcp"
) or die "Nelze vytvorit socket: $!";

print $sock "GET / HTTP/1.0\n\n";

while (<$sock>) {
if (/^Server: *(.*)/) {
print "$1\n";
last;
}
}
```

Program přijímá jako argument hostitelský server.

```
$ ./server.pl 127.0.0.1
Apache/2.2.4 (Unix) mod_perl/2.0.2 Perl/v5.8.7
$ ./server.pl www.linuxsoft.cz
Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 PHP/4.4.4-8+etch6 mod_ssl/2.2.3
OpenSSL/0.9.8c
$
```

Lokální sockety

V Unixu existuje speciální typ souborů, které jsou nazývány sockety. Právě ty nyní budeme vytvářet.

Jako ukázku si vytvoříme jednorázový server, který se pro jednoduchost ukončí hned po vyřízení prvního požadavku (vynecháme tedy cyklus).

```
use IO::Socket;

my $server = IO::Socket::UNIX->new(
Local  => "/tmp/sock",
Listen => 1
) or die "Chyba pri vytvareni serveru! $!";

my $klient = $server->accept();
my $prichozi = <$klient>;
chomp $prichozi;
print $klient ($prichozi=~/^\d{3}$/ ? "OK\n" : "CHYBA\n");
```

Server zkoumá, zda obdržel trojmístné číslo. Klient bude vypadat takto:

```
use IO::Socket;

my $vzdaleny = IO::Socket::UNIX->new(
Peer => "/tmp/sock"
) or die "Nelze spojit! $!";
```

```
print "Zadej text, ktery chces odeslat: ";
my $odchozi = <STDIN>;
print $vzdaleny $odchozi;
my $prichozi = <$vzdaleny>;
print "PRIJATO: $prichozi";
```

Vytvořili jsme socket /tmp/sock, skrz který probíhá veškerá komunikace. Práce s unix sockety je prakticky stejná jako s síťovými sockety. Pro programátora se tyto dva typy liší pouze vznikem.

Perl (75) - Obsluha více klientů

Ukážeme si dvě metody, které vylepšují server z minulého dílu tak, aby obsluhoval více klientů zároveň.

Náš jednoduchý klient-server mechanizmus z minulého dílu uměl najednou obsluhovat jediného klienta. To však v praxi většinou nestačí. Dnes se budeme zabývat právě servery, které jsou schopny odpovídat na požadavky většího množství klientů zároveň.

Představíme si následující dva způsoby, jak lze obsluhovat více klientů.

- fork - spuštění více instancí serveru
- select - přepínání mezi klienty

Rozvětvení serveru

První možností jak reagovat na požadavky více klientů je použití forku. fork dokáže větvit proces, což přesně potřebujeme. Jak již bylo řečeno, správný server se neptá, ale pouze naslouchá požadavkům klientů a odpovídá, je-li tázán. Naprogramujeme si tedy nějaký server, který bude této koncepci odpovídat.

Jako ukázku si napíšeme server, kterému budou klienti posílat nějaké řetězce. Ty server zpracuje a klientům pošle řetězce, v nichž bude přehozené pořadí písmen. Obsluhovaných klientů bude moci být několik zároveň a všechny požadavky budou neprodleně vyřizovány.

Začátek skriptu serveru bude stejný jako u serveru z minulého dílu.

```
use IO::Socket;

my $port = 7779;
my $ip = "192.168.0.10";
my $pid;

my $server = IO::Socket::INET->new(
    Proto     => "tcp",
    LocalPort => $port,
    LocalAddr => $ip,
    Listen    => 1,
    Reuse     => 1
) or die "Chyba pri vytvareni serveru! $!";
```

Zajímavé to začíná být od naslouchání. Po každém připojení klienta ho vždy odštěpíme jako zvláštní proces pomocí forku. Tím si zajistíme, že původní proces se může bez vyrušování stále soustředit na naslouchání novým klientům, zatímco odštěpený synovský proces může obsluhovat příslušného klienta.

```
while(my $klient = $server->accept()){
#odštěpíme proces, který sám klienta obslouží a vrátíme se zpět k naslouchání
next unless $pid = fork;

#zde bude probíhat komunikace s odštěpeným klientem

exit; #komunikace skončila - odštěpený proces ukončíme
}
```

Je-li ukončen synovský proces, obdržíme signál CHLD. Příkazem wait odstraníme případné zombie procesy.

```
$SIG{"CHLD"} = sub {wait();};
```

Dále je už vše takové, tak to známe z minulého dílu. Zbývá už jen samotná komunikace s klientem. Přijmeme tedy data, převrátíme v nich znaky a odešleme zpět klientovi. Navíc vytiskneme pro informaci hlášku, která říká co a komu se posílá.

```
while (my $prichozi = <$klient>){
    chomp $prichozi;
    print "Prijat retezec $prichozi od klienta ($pid)\n";
    my $odchozi = reverse $prichozi;
    print "Posilam retezec $odchozi klientovi ($pid)\n";
    print $klient "$odchozi\n";
}
```

Tento přístup nám v zásadě neříká nic nového. Pouze jsme aplikovali znalosti z minulých dílů a dali je dohromady.

Přepínání mezi klienty

Nyní si vyzkoušíme jiný přístup k obsluze více klientů. Tentokrát nebudeme spouštět více instancí programu, ale postačí nám jediné vlákno. Mezi klienty budeme přepínat. Na to už ale musíme použít něco nového.

Modul IO::Select nám umožňuje zjistit, kteří z klientů jsou aktuálně schopni poslat požadavek. Jakmile takový klient existuje, obsloužíme ho, okamžitě se vrátíme a opět zjišťujeme, zda se chce nějaký klient ptát. Důležité je, že klienta obsluhujeme až je-li připraven odesílat data. Máme tak zaručeno, že nebudeme čekat, protože pokud se dostaneme až sem, klient data již odesílá a čeká pouze na to, až je server přijme.

Nejdříve opět vytvoříme socket, a pomocí něj dále definujeme instanci objektu IO::Select.

```
use IO::Select;
use IO::Socket;
use strict;

my @read;
my $sock = new IO::Socket::INET(
```

```
    LocalPort => 7778,
    Listen    => 1,
    Reuse     => 1
);
my $select = new IO::Select($sock);
```

Dále budeme v cyklu čekat na klenty, z nichž lze číst. Zjistíme je metodou can_read z modulu IO::Select. Tyto klienty si uchováme a v cyklu s každým z nich provedeme nějakou operaci.

```
while(@read = $select->can_read) {
foreach my $fh (@read) {
#zde zpracujeme zpravu od klienta
}
}
```

Při prvním průchodu vytvoříme nový klientský socket a při dalších už zpracováváme jeho požadavky. Pokud klient ukončil spojení, uděláme totéž.

```
#vytvoříme socket a přidáme do selectu
if($fh == $sock) {
my $new = $sock->accept;
$select->add($new);

#můžeme číst data z klienta - hned tedy vyřídíme jeho požadavek
}else{
my $zprava;
chomp($zprava = <$fh>);
if($zprava){
$zprava = reverse $zprava;
print $fh "$zprava\n";
}else{
$select->remove($fh);
$fh->close;
}
}
```

Princip tohoto serveru spočívá v tom, že jakmile obdrží od některého klienta zprávu, zpracuje ji a dále na tohoto klienta nečeká. Server jím tak není blokován a okamžitě po vyřízení požadavku je k dispozici ostatním.

Perl (76) - Síťová hra v kostky

Dnes využijeme znalosti nabyté v předchozích dílech a napíšeme si jednoduchý server pro hru v kostky, který bude organizovat hru libovolnému množství hráčů.

Cílem dnešního dílu bude vytvořit základ pro v podstatě libovolnou síťovou hru. V článku půjde konkrétně o hru v kostky. To proto, že hra v kostky je velice jednoduchá na programování. Budou tak více vidět obecnější myšlenky a síťová komunikace.

Plán

Nejprve si tedy uvědomme, co vlastně budeme psát.

Hru v kostky zná snad každý - spočívá v opakovaném házení kostkou dvěma hráči, přičemž ten, kdo hodí v *n*tém hodu více ok, získává bod. V případě shodnosti počtu ok nezíská bod nikdo. Kdo získá 5 bodů, vítězí.

Úkolem tedy bude napsat nějaký kostkový server. Ten bude čekat na klienty. Jakmile se přihlásí klient, zaregistruje ho. Tento klient bude čekat na to, až se přihlásí nějaký další klient, s kterým by mohl začít hru. Jakmile budou k dispozici dva nehrající klienti, založíme jim automaticky hru. A stále budeme čekat na další.

Jinými slovy, v tuto chvíli budeme dělat několik věcí zároveň:

- pro každou již přihlášenou dvojici budeme organizovat hru
- pro další klienty budeme hledat vhodné protějšky na přihlášení do hry

Jde o krátký program, takže to nyní není třeba do detailů rozebírat. Vše bude jasné v průběhu.

Server

Hned teď si ukažme celý skript server.pl. Budeme využívat modulu Kostky::Server, který budeme muset ještě dodělat.

```
#!/usr/bin/perl -w
use strict;
use warnings;
use Kostky::Server;

while (1){
my $vitez;
my $kostky_server = new Kostky::Server;
$kostky_server->spust;
}
```

Nyní se pojďme podívat, jak bude vypadat základ modulu Kostky::Server (tj. soubor Kostky/Server.pm).

```
package Kostky::Server;
use strict;
use warnings;
use IO::Socket;

sub new {
my($self) = @_;
my $f = {};
bless $f;

my $hrac = IO::Socket::INET->new(
Proto     => "tcp",
```

```
        LocalPort => 9005,
        LocalAddr => "localhost",
            Listen    => 2,
            Reuse     => 1
    ) or die "Nelze vytvorit socket! $!";

    $f->{"hrac"} = $hrac;
    $f->{"skore1"} = 0;
    $f->{"skore2"} = 0;

    return $f;
    }

    sub spust {
        ...
    }
```

Zatím šlo pouze o rutinní záležitosti a není zde nic, co bychom nečekali. Tj. v kostruktoru jsme vytvořili spojení (jsme serveru), počet klientů nastavili na 2 (každou hru budou hrát pouze 2 hráči) a skóre jsme nastavili na 0:0.
Nyní bude naším úkolem implementovat metodu spust, která bude dělat veškerou práci serveru.
Máme za úkol řídit hru libovolnému množství hráčů, tj. použijeme fork na odštěpení jednotlivých instancí hry. Tentokráte půjde dokonce o jednu instanci pro dva hráče. Schéma metody bude tedy vypadat takto.

```
sub spust {
while(1){

    #připojení dvou hráčů

    next unless $pid = fork;

    #obsloužíme odštěpený proces

    exit;
    }
    }
```

Hledání dvojic hráčů

Připojení dvou hráčů znamená naplnění proměnných $self->{"vzdaleny1"}, $self->{"vzdaleny2"}, $self->{"nick1"}, $self->{"nick2"} hodnotami. To znamená, že musíme získat hodnoty pro spojení na oba hráče a jejich jména. Na ty se zeptáme klientů, jakmile se připojí - a to tak, že jim odešleme zprávu s obsahem "1\n". Už bude na klientovi, aby tomuto protokolu rozumněl (tj. o to se budeme starat později).
Oba hráče připojíme pomocí cyklu o dvou iteracích. Samozřejmě to lze rozepsat do jednotlivých iterací.

```
for(my $i=1; $i<=2; $i++){
$vzdaleny = $self->{"hrac"}->accept();
    $vzdaleny->autoflush(1);
    print $vzdaleny "1\n";
    chomp($nick = <$vzdaleny>);
    do_logu("-", "$nick pripojen");
$self->{"vzdaleny$i"} = $vzdaleny;
    $self->{"nick$i"} = $nick;
    }
```

Všimněme si tohoto řádku.

```
print "$nick pripojen\n";
```

Pokud použijeme print směrovaný jinam než do socketu, bude to něco jako logování. Nepoužíváme zde log soubor, ale směřujeme výstup na standardní výstup. Logovat budeme po každé akci.
Vhodnější by bylo logování nějak sjednotit. Odteď budeme v lozích zaznamenávat také čas a PID (neboť mícháme výstup všech her do jednoho logu). Pro tento účel si napíšeme jednoduchou funkci do_logu, která pošle jeden log záznam ve formátu 'PID čas zpráva'na standardní výstup.

```
sub do_logu {
my($pid, $zprava)=@_;
my($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
print "$pid $hour:$min:$sec $zprava\n";
}
```

Řízení hry

Nyní již pojďme vyřídit odštěpený proces. Zahájíme hru. Při té příležitosti zapíšeme do logu následující zprávu.

```
do_logu($pid, "V teto hre hraji ".$self->{"nick1"}." a ".$self->{"nick2"});
```

Před tím, než napíšeme hlavní smyčku hry, nastavme kritéria pro konec hry. Funkce konec nám vyhodnotí, zda někdo vyhrál a případně kdo. Při naší hře v kostky je kritérium jediné - stačí dosáhnout 5 bodů.

```
sub konec {
my($self) = @_;
my $vitez;

return 1 if $self->{"skore1"}==$SKORE_VITEZE;
return 2 if $self->{"skore2"}==$SKORE_VITEZE;
return -1; #není vítěz
}
```

A teď konečně přistupme k jádru serveru - napíšeme onu hlavní smyčku. Bude vypadat nějak takto.

```
while(($vitez = $self->konec) == -1) {
$cislo_tahu++;
```

```
#nalosujeme, kdo bude v tomto tahu začínat
#necháme hráče "hodit" kostkou
#vyhodnotíme hody, a pošleme hráčům výsledek
}
```
Začneme tedy losem, kdo bude začínat aktuální tah. My to uděláme tak, že v polovině případů prohodíme hráče a v polovině ne. Toto řešení má své výhody i nevýhody, nicméně nám plně postačuje.
```
if((rand(2)>1 ? 1 : 0)==0){
($self->{"vzdaleny1"}, $self->{"vzdaleny2"}, $self->{"nick1"},
$self->{"nick2"}) = ($self->{"vzdaleny2"}, $self->{"vzdaleny1"}, $self->{"nick2"},
$self->{"nick1"});
}
```
Pro jednoduchost nyní zavedeme proměnné $klient1 a $klient2 jako aliasy pro spojení na hráče.
```
$klient1 = $self->{"vzdaleny1"};
$klient2 = $self->{"vzdaleny2"};
```
Dále musíme hráčům oznámit, kdo bude začínat a kdo bude hrát jako druhý. Pošleme tedy zprávu "1\n" hráči, který bude začínat a zprávu "2\n" hráči, který musí počkat na jeho výsledek.
```
print $klient1 "1\n";#hráč1 začíná
print $klient2 "2\n";#hráč1 čeká
```
Dále čekáme na hráče, který má začínat, až hodí. Jakmile se tak stane, podíváme se, jaké číslo je kostce pomocí funkce rand (tj. v okamžiku, kdy hodí, stopneme čas a podle něj určíme počet ok).
```
do_logu($pid, "Cekame, az hodi ".$self->{"nick1"});
my $hod = <$klient1>;
my $vysledek_hodu1 = int(rand(6))+1;
```
Výsledek oznámíme oběma hráčům.
```
print $klient1 "$vysledek_hodu1\n";
print $klient2 "$vysledek_hodu1\n";
do_logu($pid, $self->{"nick1"}." hodil $vysledek_hodu1");
```
Následně čekáme na hod od druhého hráče.
```
do_logu($pid, "Cekame, az hodi ".$self->{"nick2"});
$hod = <$klient2>;
my $vysledek_hodu2 = int(rand(6))+1;
```
Výsledek taktéž oznámíme.
```
print $klient2 "$vysledek_hodu2\n";
print $klient1 "$vysledek_hodu2\n";
do_logu($pid, $self->{"nick2"}." hodil $vysledek_hodu2");
```
Na závěr zaktualizujeme skóre (přidáme tomu hráči, který hodil na kostce více) a výsledky pošleme.
```
#aktualizujeme skore
$self->{"skore1"}++ if $vysledek_hodu1>$vysledek_hodu2;
$self->{"skore2"}++ if $vysledek_hodu1<$vysledek_hodu2;

print $klient1 $self->{"nick1"}." vs. ".$self->{"nick2"}." ".
    $self->{"skore1"}."-".$self->{"skore2"}."\n";
print $klient2 $self->{"nick1"}." vs. ".$self->{"nick2"}." ".
    $self->{"skore1"}."-".$self->{"skore2"}."\n";
do_logu($pid, $self->{"nick1"}." vs. ".$self->{"nick2"}." ".
    $self->{"skore1"}."-".$self->{"skore2"});
```
Tím je hlavní cyklus hotov. Za něj se program dostane pouze tehdy, je-li znám vítěz. Proměnnou $vitez již máme naplněnou - z hlavičky cyklu. Stačí tedy odeslat celkové výsledky a jsme hotovi.
```
$porazeny = $vitez==1?2:1;
print {$self->{"vzdaleny$vitez"}} "-1\n";
print {$self->{"vzdaleny$porazeny"}} "-2\n";
```
Tím jsme dokončili herní server.

Klienti

Pojďme se podívat na druhou část a tou bude napsání klienta, který bude umět s naším serverem komunikovat. Pro jednoduchost nebudeme psát žádný speciální modul, ač by to šlo (a při náročnější hře by to bylo vhodné), ale spokojíme se se skriptem, který bude vše řešit sám.
```
#!/usr/bin/perl -w
use strict;
use warnings;
use IO::Socket;
```
Nejprve od uživatele zjistíme, kde server běží.
```
print "Zadej IP adresu serveru: ";
chomp($ip_serveru = <STDIN>);
```
Server po nás taktéž bude chtít zadat jméno, takže se na něj hned také zeptáme.
```
print "Zadej nick: ";
chomp($nick = <STDIN>);
```
Teď se již k serveru můžeme připojit a odešleme mu naše jméno. Pokud obdržíme od serveru odpověď, je to dobrá zpráva, protože se nám připojit se podařilo.
```
my $vzdaleny = IO::Socket::INET->new(
    Proto    => "tcp",
    PeerPort  => 9005,
    PeerAddr  => $ip_serveru
) or die "Nelze vytvorit socket! $!";

print $vzdaleny "$nick\n";
```

```
print "Jsme pripojeni!\n" if <$vzdaleny>;
```

Nyní to již nebude těžké - budeme pouze reagovat na to, na co se server ptá. Půjde jen o zesynchronizování se se serverem. Program bude samozřejmě opět ve formě smyčky. Na začátku každého tahu nám server posíla informaci o tom, zda začínáme nebo ne. Proto od něj očekáváme zprávu.

```
while(1){
my $na_tahu = <$vzdaleny>;
chomp $na_tahu;

if($na_tahu == 1){
#hrajeme jako první
}elsif($na_tahu == 2){
#hrajeme až po soupeři
}
...
}
```

Místo tří teček ještě vepíšeme následující kód, neboť právě tímto způsobem server oznamuje vítězství či porážku.

```
elsif($na_tahu == -1){
print "JSI VITEZ\n";
last;
}elsif($na_tahu == -2){
print "JSI PORAZENY\n";
last;
}
```

Na závěr dokončíme první dvě podmínky. Zde jen recipročně reagujeme na server.

```
if($na_tahu == 1){
print "Mas v ruce kostku. Stiskni ENTER pro hod!";
$hod = <STDIN>;
print $vzdaleny "1\n";
chomp($vysledek_hodu = <$vzdaleny>);
print "Hodil jsi $vysledek_hodu. Cekame na soupere.\n";
chomp($vysledek_soupere = <$vzdaleny>);
print "Souper hodil $vysledek_soupere.\n";
$skore = <$vzdaleny>;
print $skore;
}elsif($na_tahu == 2){
print "Cekame na hod soupere.\n";
chomp($vysledek_soupere = <$vzdaleny>);
print "Souper hodil $vysledek_soupere.\n";
print "Mas v ruce kostku. Stiskni ENTER pro hod!";
$hod = <STDIN>;
print $vzdaleny $hod;
chomp($vysledek_hodu = <$vzdaleny>);
print "Hodil jsi $vysledek_hodu.\n";
$skore = <$vzdaleny>;
print $skore;
}
```

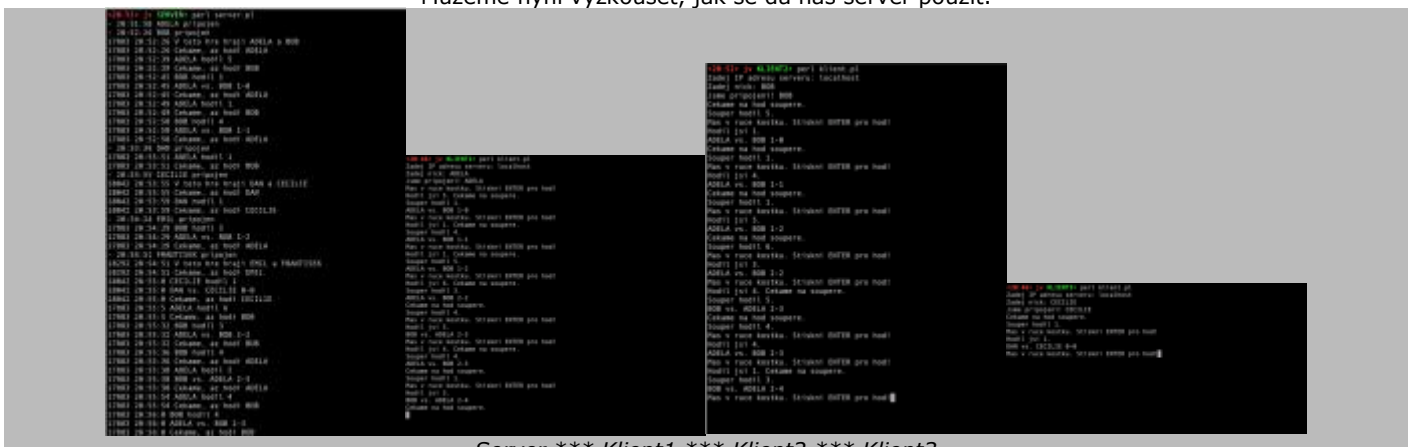
Tím jsme hotovi s klientem.

Závěr

Nyní máme server i klienty hotové. Myšlenkově nešlo o nic náročného, ovšem při těchto aplikacích zabere často hodně času ladění.

Všechny tři soubory si můžete stáhnout a vyzkoušet: server.pl, klient.pl, Kostky/Server.pm.

Můžeme nyní vyzkoušet, jak se dá náš server použít.



*Server *** Klient1 *** Klient2 *** Klient3*

*Klient4 *** Klient5 *** Klient6*

Na základě této hry bychom nyní mohli analogicky napsat cokoliv jiného - například síťové šachy, piškvorky a jiné. Z hlediska programování nebudou o moc složitější. Půjde pouze o více psaní.

Perl (77) - Služby internetu

Dnes si představíme dva moduly z archivu CPAN, které dovedou využít internetových služeb FTP a POP3.

Existuje několik modulů, které poskytují rozhraní pro komunikaci s některými síťovými službami. Můžeme si tím pádem vytvořit jednoduché emailové klienty nebo FTP klienty.

FTP klient

FTP je protokol, který, jak již je z názvu File Transfer Protocol patrné, umožňuje pohodlný přenos dat mezi počítači. Funguje na mechanizmu klient - server. Pro Perl je v archivu CPAN k dispozici modul nazvaný Net::FTP, jež právě FTP komunikaci poskytuje. Na příkladu si ukážeme, jak se připojit k nějakému vzdálenému serveru a jak z něj získávat a posílat mu data.

Abychom si mohli vyzkoušet funkce, jež budeme dále popisovat, je třeba mít nějaký vzdálený počítač, kam se budeme moci přes FTP připojit. Dále budu předpokládat, že máme server ftp.server.cz, na kterém existuje uživatel user s heslem pAssW0Rd.

Práce s modulem Net::FTP zahrnuje několik kroků. Jsou to následující.

1. Spojení s FTP serverem
2. Přihlášení se k serveru pod uživatelským jménem
3. Uživatelské akce - samotná práce se soubory na serveru
4. Ukončení spojení

Spojení a příhlášení

Ze všeho nejdříve vytvoříme proměnné s přihlašovacími údaji.

```
use Net::FTP;
my $hostname = "ftp.server.cz";
my $username = "user";
my $password = "pAssW0Rd";
```

Dále již můžeme volat FTP server. Vytvoříme tedy instanci modulu Net::FTP.

```
my $ftp = Net::FTP->new($hostname) or die "Nelze se spojit s FTP serverem.";
```

A nyní se přihlásíme.

```
$ftp->login($username, $password) or die "Nelze se přihlásit. ";
```

Uživatelské akce

V tomto okamžiku jsme ve spojení se serverem. Pokud znáte shellový příkaz ftp, tak pro vás nebudou další informace již ničím novým. Metody modulu Net::FTP mají totiž stejné názvy jako příkazy nástroje ftp.

Nejdříve si představíme metody pro práci s adresářovou cestou. Funkce cwd mění aktuální adresář - tedy funguje podobně jako příkaz cd v shellu.

```
$ftp->cwd("pub") or die "Nelze změnit adresář. "; #nyní jsme na serveru v adresáři /pub
```

Budeme-li naopak chtít jméno aktuálního adresáře získat a vytisknout, použijeme příkaz pwd.

```
print $ftp->pwd; #tiskne /pub
```

Příkazy ls a ls -l pro výpis obsahu adresáře jsou v příkazovém řádku nepostradatelnými. Modul Net::FTP nám samozřejmě poskytuje i příkazy, které jsou jejich analogiemi. Začneme jednodušším ls. Následující úsek kódu uloží do pole názvy souborů v aktuálním adresáři na serveru a následně každý z nich vytiskne na samostatný řádek.

```
my @ls = $ftp->ls(".") or die "Nelze získat seznam souborů \n";
$" = "\n";
print "@ls\n";
```

Uvedení parametru není pro získání obsahu aktuálního adresáře nezbytné. Jinak lze zadat libovolný dostupný adresář na serveru.

Implementace příkazu ls -l je stejná až na to, že místo metody ls se použije dir.

```
my @ls = $ftp->dir(".") or die "Nelze získat seznam souborů \n";
$" = "\n";
print "@ls\n";
```

Přenos souborů

Nyní se dostáváme k tomu nejdůležitějšímu. Protokol FTP se využívá právě pro přenos souborů. Můžeme posílat soubory na server, ale také stahovat. Slouží k tomu dvě jednoduché metody: get a put.

Již z názvu je patrné, že put provádí upload souboru na server. Budeme-li chtít poslat soubor letter.txt uložený v aktuálním adresáři lokálního stroje do aktuálního adresáře na server, použijeme následující příkaz.

```
$ftp->put("./letter.txt") or die "Nelze provést upload. \n";
```

Pro tok dat opačným směrem, tedy ze serveru na klienta, použijeme funkci get.

```
$ftp->get("send.php") or die "Nelze provést download. \n";
```

Ukončení spojení

A nakonec uzavřeme spojení.

```
$ftp->quit;
```

Existují samzřejmě další a další příkazy a pokročilejší funkce, ale když o ně někdo bude mít zájem, jistě už sám nahlédne do dokumentace.

POP3 klient

Další službou, ke které se pokusíme pomocí Perlu přistupovat, bude email. POP (Post Office Protocol) verze 3 je protokol, který umožňuje stahovat data z emailového serveru. Nevýhodou POP protokolu může být to, že po přečtení emailu se na serveru zpráva smaže. Je však možné přečíst pouze část zprávy a email na serveru zůstane. Další nevýhodou je to, že POP protokol stahuje všechny zprávy na serveru včetně spamu.

V Perlu poskytuje rozhraní pro POP protokol modul Net::POP3, na kterém si práci s POP3 předvedeme.

Předpokladem pro to, aby vám následující kód fungoval, je přístup k emailové adrese. Od poskytovatele emailu si je třeba si zjistit server příchozích zpráv POP3. Předpokládejme, že máme emailovou adresu user@server.cz, POP3 server je pop3.server.cz, uživatelské jméno user a heslo pAssW0Rd.

```
use Net::POP3;
my $hostname = "pop3.server.cz";
my $username = "user";
my $password = "pAssW0Rd";
```

Spojení

Prvním úkolem je připojení k POP3 serveru. Zavoláme tedy konstruktor třídy Net::POP3.

```
my $pop = Net::POP3->new($hostname) or die "Nelze se připojit. $!\n";
```

Přihlášení

Dalším krokem je přihlášení. Metoda login vrací počet nových zpráv ve schránce, takže tuto hodnotu hned přiřadíme.

```
my $zprav = $pop->login($username, $password) or die "Nelze se přihlásit. $!\n";
print "POČET ZPRÁV: $zprav\n\n";
```

Uživatelské akce

Pokud nenastal nějaký problém, můžeme nyní se schránkou pracovat. Metoda listzpřístupňuje seznam zpráv.

```
my $r_zpravy = $pop->list() or die "Nelze získat seznam zpráv. $!\n";
```

Dále v cyklu zobrazíme z každé zprávy hlavičku a 5 úvodních řádků zprávy. K tomu použijeme metodu top.

```
foreach (keys %$r_zpravy) {
print "-----------------------------------\n";
print "Zpráva číslo $_:\n";
my $r_z = $pop->top($_, 5);
print @$r_z;
}
```

Ukončení spojení

A na úplný závěr ukončíme spojení.

```
$pop->quit;
```

Závěr

Jak je vidět, práce se službami internetu není v Perlu díky modulům nic složitého. Dnes jsme si ukázali pouze služby FTP a POP3, nicméně existují moduly pro řadu dalších, které obvykle začínají prefixem Net::. Namátkou uveďme Net::IMAP a Net::HTTP.

Perl (74) - Sockety

Náplní dnešního dílu bude meziprogramová komunikace. Osvětlíme si pojem socket, jeho vlastnosti a nezbytnou součástí bude i názorná ukázka použití.

Socket je virtuální spojení dvou procesů. Důležitě je zde slovo spojení. Z něj vyplývá to, že sockety jsou způsobem meziprocesorové komunikace. V praxi to znamená, že můžeme napsat dva programy, které spolu budou moci vzájemně komunikovat. První program - server - bude naslouchat na určeném portu (port je číslo reprezentující vstupní cestu do počítače) a bude čekat na připojení druhého programu - klienta. V okamžiku, kdy se klient připojí, vznikne spojení a od této chvíle mohou obě strany komunikovat. Oba programy mohou běžet na různých počítačích, které jsou síťově propojeny.

Existují dva způsoby komunikace. Se spojením nebo beze spojení - tedy TCP nebo UDP. Typickým příkladem komunikace bez spojení je email. Mezi odesílatelem a příjemcem se nevytváří žádné spojení - odeslání emailu není závislé na příjemci a přijetí na odesílateli. Není zaručeno pořadí jednotlivých zpráv ani to, že někdo zprávu skutečně přijme. Naopak mezi komunikaci se spojením patří telefonní hovor. Oba účastnící jsou v jednom okamžiku spojeni komunikačním kanálem, který zaručuje správné pořadí zpráv.

Sockety v Perlu

V Perlu máme pro práci se sockety k dispozici moduly Socket a IO::Socket. My si představíme objektově orientovaný IO::Socket. Ten obsahuje dvě podtřídy. INET a UNIX. Prvně jmenovaná pro síťovou komunikaci a UNIX pro komunikaci lokální. O ní se však zmíníme jen letmo na konci. Máte-li zájem o modul Socket, nahlédněte do stránky man perlipc(1), jež se zabývá meziprocesorovou komunikací.

Příklad - jednoduchý chat

Jako nejjednodušší příklad aplikace s využitím socketů si napíšeme jednoduchý chat, pomocí kterého budou moci dva lidé připojení u různých počítačů ve stejné síti střídavě posílat zprávy.

Ještě než začneme si musíme ujasnit, co vlastně budeme psát. Potřebujeme dva programy - server a klienta.

Činnost serveru spočívá v naslouchání na daném portu. V okamžiku, kdy se na tento port přípojí klient, mohou oba programy vzájemně komunikovat. Komunikace bude probíhat následovně: Server obdrží zprávu od klienta a pošle mu odpověď. A to se bude opakovat donekonečna.

Server

Začneme tedy serverem. Nejprve vytvoříme socket pomocí metody new, jejíž parametry jsou patrné z následujícího zdrojového kódu.

```
use IO::Socket;

my $port = 7777;
my $ip = "192.168.0.10";

my $server = IO::Socket::INET->new(
Proto     => "tcp",
LocalPort => $port,
LocalAddr => $ip,
Listen    => 1,
Reuse     => 1
) or die "Chyba pri vytvareni serveru! $!";
```

Právě jsme vytvořili socket server na ip adrese 192.168.0.10, který bude naslouchat na portu 7777. Port musíme zvolit tak, aby ho nepoužívalo více aplikací, jinak dojde ke konfliktům. Parametrem Listen určujeme nejvyšší možný počet klientů. Reusezajistí, že po ukončení programu bez uzavření socketu se port opět uvolní.

Teď přichází chvíle pro naslouchání. Budeme naslouchat tak dlouho, dokud se nepřipojí nějaký klient. Naslouchání se provádí metodou accept, jež vrací socket, kterým lze komunikovat s klientem.

```
$klient = $server->accept();
```

Ve skalárním kontextu metoda accept vrací nový socket, ale pokud bychom přiřazovali do pole, druhým prvkem by byla ip adresa klienta.

Nyní přichází na řadu samotná komunikace s klienty. Směr komunikace si určuje samotná aplikace. Proto může dojít k případům, kdy oba programy spojené socketem čtou nebo zapisují do kanálu zároveň, což způsobuje uváznutí. To by se ve správně napsané aplikaci nemělo nikdy stát.

Vytvoříme tedy cyklus, uvnitř kterého budeme nejprve očekávat nějakou zprávu od klienta a následně pošleme odpověď. Se socketem pracujeme stejně jako s ovladačem.

V této souvislosti je dobré poznamenat, že by odesílaná data měli vždy končit znakem nového řádku, případně jiným znakem, který jeho funkci zastupuje.

```
while (1){
$prichozi = <$klient>;
print "PRIJATO: $prichozi";
print "Zadej text, ktery chces odeslat: ";
$odchozi = <STDIN>;
print $klient $odchozi;
}
```

Klient

Vytvoření klienta, jak bude za chvilku patrné, je podobné.

```
use IO::Socket;

my $port = 7777;
my $ip_serveru = "192.168.0.10";

my $vzdaleny = IO::Socket::INET->new(
Proto    => "tcp",
PeerAddr => $ip_serveru,
PeerPort => $port
) or die "Nelze spojit! $!";
```

Tato část klienta má za úkol zjistit, zda na portu 7777 počítače 192.168.0.10 naslouchá nějaký server a pokud tam skutečně naslouchá, připojit se k němu. Pokud ne, program se jednoduše ukončí.

Dále následuje už komunikace, která je až na směr stejná jako u serveru. První akce serveru bylo přijetí zprávy. Proto klient nejprve data odešle a poté bude očekávat odpověď.

```
while (1){
print "Zadej text, ktery chces odeslat: ";
$odchozi = <STDIN>;
print $vzdaleny $odchozi;
$prichozi = <$vzdaleny>;
print "PRIJATO: $prichozi";
}
```

Na závěr by běžný klient ukončil spojení. Ten náš ovšem běží nekonečně dlouho (nedostane se za cyklus), takže následující řádek ve zdrojovém kódu nepoužijeme.

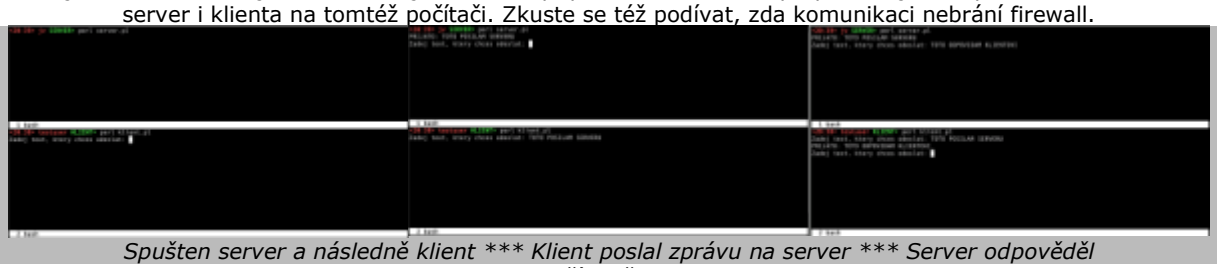
```
close $vzdaleny;
```

Výsledky

Zkuste si též změnit směr jednoho z programů - například, aby oba nejdříve posílaly data. Program uvázne, neboť oba data vysílají, ale nikdo je nepřijímá.

Nyní jsme hotovi a můžeme zkoušet. Zdrojové kódy klienta i serveru si můžete stáhnout. Zkusíte-li nyní na počítači 192.168.0.15 spustit server a na jiném počítači klienta, měl by náš chat fungovat. V okamžiku, kdy spouštíte klienta, musí pochopitelně již běžet server.

Pokud to z nějakého důvodu nejde, testování je v mnoha případech dobré začít přepsáním jména počítače na localhost a spustit server i klienta na tomtéž počítači. Zkuste se též podívat, zda komunikaci nebrání firewall.



*Spušten server a následně klient *** Klient poslal zprávu na server *** Server odpověděl*

Další možnosti

Se sockety lze mnoha způsoby experimentovat. Za vyzkoušení stojí připojení se na HTTP port 80. Napišme si zde jednoduchý skript, který vypíše systém, na kterém běží server. Tedy odešleme HTTP požadavek na nějakého vzdáleného hostitele a budeme čekat odpověď.

```
use IO::Socket;
use strict;

my $host = $ARGV[0];
my $sock = new IO::Socket::INET(
PeerAddr => $host,
PeerPort => 80,
Proto    => "tcp"
) or die "Nelze vytvorit socket: $!";

print $sock "GET / HTTP/1.0\n\n";

while (<$sock>) {
if (/^Server: *(.*)/) {
print "$1\n";
last;
}
}
```

Program přijímá jako argument hostitelský server.

```
$ ./server.pl 127.0.0.1
Apache/2.2.4 (Unix) mod_perl/2.0.2 Perl/v5.8.7
$ ./server.pl www.linuxsoft.cz
```

Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 PHP/4.4.4-8+etch6 mod_ssl/2.2.3
OpenSSL/0.9.8c
$
Lokální sockety
V Unixu existuje speciální typ souborů, které jsou nazývány sockety. Právě ty nyní budeme vytvářet.
Jako ukázku si vytvoříme jednorázový server, který se pro jednoduchost ukončí hned po vyřízení prvního požadavku (vynecháme tedy cyklus).

```
use IO::Socket;

my $server = IO::Socket::UNIX->new(
    Local  => "/tmp/sock",
    Listen => 1
) or die "Chyba pri vytvareni serveru! $!";

my $klient = $server->accept();
my $prichozi = <$klient>;
chomp $prichozi;
print $klient ($prichozi=~/^\d{3}$/ ? "OK\n" : "CHYBA\n");
```

Server zkoumá, zda obdržel trojmístné číslo. Klient bude vypadat takto:

```
use IO::Socket;

my $vzdaleny = IO::Socket::UNIX->new(
    Peer => "/tmp/sock"
) or die "Nelze spojit! $!";

print "Zadej text, ktery chces odeslat: ";
my $odchozi = <STDIN>;
print $vzdaleny $odchozi;
my $prichozi = <$vzdaleny>;
print "PRIJATO: $prichozi";
```

Vytvořili jsme socket /tmp/sock, skrz který probíhá veškerá komunikace. Práce s unix sockety je prakticky stejná jako s síťovými sockety. Pro programátora se tyto dva typy liší pouze vznikem.
Perl (75) - Obsluha více klientů

Ukážeme si dvě metody, které vylepšují server z minulého dílu tak, aby obsluhoval více klientů zároveň.

Náš jednoduchý klient-server mechanizmus z minulého dílu uměl najednou obsluhovat jediného klienta. To však v praxi většinou nestačí. Dnes se budeme zabývat právě servery, které jsou schopny odpovídat na požadavky většího množství klientů zároveň.
Představíme si následující dva způsoby, jak lze obsluhovat více klientů.

- fork - spuštění více instancí serveru
- select - přepínání mezi klienty

Rozvětvení serveru
První možností jak reagovat na požadavky více klientů je použití forku. fork dokáže větvit proces, což přesně potřebujeme.
Jak již bylo řečeno, správný server se neptá, ale pouze naslouchá požadavkům klientů a odpovídá, je-li tázán. Naprogramujeme si tedy nějaký server, který bude této koncepci odpovídat.
Jako ukázku si napíšeme server, kterému budou klienti posílat nějaké řetězce. Ty server zpracuje a klientům pošle řetězce, v nichž bude přehozené pořadí písmen. Obsluhovaných klientů bude moci být několik zároveň a všechny požadavky budou neprodleně vyřizovány.
Začátek skriptu serveru bude stejný jako u serveru z minulého dílu.

```
use IO::Socket;

my $port = 7779;
my $ip = "192.168.0.10";
my $pid;

my $server = IO::Socket::INET->new(
    Proto     => "tcp",
    LocalPort => $port,
    LocalAddr => $ip,
    Listen    => 1,
    Reuse     => 1
) or die "Chyba pri vytvareni serveru! $!";
```

Zajímavé to začíná být od naslouchání. Po každém připojení klienta ho vždy odštěpíme jako zvláštní proces pomocí forku. Tím si zajistíme, že původní proces se může bez vyrušování stále soustředit na naslouchání novým klientům, zatímco odštěpený synovský proces může obsluhovat příslušného klienta.

```
while(my $klient = $server->accept()){
#odštěpíme proces, který sám klienta obslouží a vrátíme se zpět k naslouchání
next unless $pid = fork;

#zde bude probíhat komunikace s odštěpeným klientem

exit; #komunikace skončila - odštěpený proces ukončíme
}
```

Je-li ukončen synovský proces, obdržíme signál CHLD. Příkazem wait odstraníme případné zombie procesy.

```
$SIG{"CHLD"} = sub {wait();};
```

Dále je už vše takové, tak to známe z minulého dílu. Zbývá už jen samotná komunikace s klientem. Přijmeme tedy data, převrátíme v nich znaky a odešleme zpět klientovi. Navíc vytiskneme pro informaci hlášku, která říká co a komu se posílá.

```
while (my $prichozi = <$klient>){
chomp $prichozi;
print "Prijat retezec $prichozi od klienta ($pid)\n";
my $odchozi = reverse $prichozi;
print "Posilam retezec $odchozi klientovi ($pid)\n";
print $klient "$odchozi\n";
}
```

Tento přístup nám v zásadě neříká nic nového. Pouze jsme aplikovali znalosti z minulých dílů a dali je dohromady.

Přepínání mezi klienty

Nyní si vyzkoušíme jiný přístup k obsluze více klientů. Tentokrát nebudeme spouštět více instancí programu, ale postačí nám jediné vlákno. Mezi klienty budeme přepínat. Na to už ale musíme použít něco nového.

Modul IO::Select nám umožňuje zjistit, kteří z klientů jsou aktuálně schopni poslat požadavek. Jakmile takový klient existuje, obsloužíme ho, okamžitě se vrátíme a opět zjišťujeme, zda se chce nějaký klient ptát. Důležité je, že klienta obsluhujeme až je-li připraven odesílat data. Máme tak zaručeno, že nebudeme čekat, protože pokud se dostaneme až sem, klient data již odesílá a čeká pouze na to, až je server přijme.

Nejdříve opět vytvoříme socket, a pomocí něj dále definujeme instanci objektu IO::Select.

```
use IO::Select;
use IO::Socket;
use strict;

my @read;
my $sock = new IO::Socket::INET(
LocalPort => 7778,
Listen    => 1,
Reuse     => 1
);
my $select = new IO::Select($sock);
```

Dále budeme v cyklu čekat na klenty, z nichž lze číst. Zjistíme je metodou can_read z modulu IO::Select. Tyto klienty si uchováme a v cyklu s každým z nich provedeme nějakou operaci.

```
while(@read = $select->can_read) {
foreach my $fh (@read) {
#zde zpracujeme zpravu od klienta
}
}
```

Při prvním průchodu vytvoříme nový klientský socket a při dalších už zpracováváme jeho požadavky. Pokud klient ukončil spojení, uděláme totéž.

```
#vytvoříme socket a přidáme do selectu
if($fh == $sock) {
my $new = $sock->accept;
$select->add($new);

#můžeme číst data z klienta - hned tedy vyřídíme jeho požadavek
}else{
my $zprava;
chomp($zprava = <$fh>);
if($zprava){
$zprava = reverse $zprava;
print $fh "$zprava\n";
}else{
$select->remove($fh);
$fh->close;
}
}
```

Princip tohoto serveru spočívá v tom, že jakmile obdrží od některého klienta zprávu, zpracuje ji a dále na tohoto klienta nečeká. Server jím tak není blokován a okamžitě po vyřízení požadavku je k dispozici ostatním.

Perl (76) - Síťová hra v kostky

Dnes využijeme znalosti nabyté v předchozích dílech a napíšeme si jednoduchý server pro hru v kostky, který bude organizovat hru libovolnému množství hráčů.

Cílem dnešního dílu bude vytvořit základ pro v podstatě libovolnou síťovou hru. V článku půjde konkrétně o hru v kostky. To proto, že hra v kostky je velice jednoduchá na programování. Budou tak více vidět obecnější myšlenky a síťová komunikace.

Plán

Nejprve si tedy uvědomme, co vlastně budeme psát.

Hru v kostky zná snad každý - spočívá v opakovaném házení kostkou dvěma hráči, přičemž ten, kdo hodí v *n*tém hodu více ok, získává bod. V případě shodnosti počtu ok nezíská bod nikdo. Kdo získá 5 bodů, vítězí.

Úkolem tedy bude napsat nějaký kostkový server. Ten bude čekat na klienty. Jakmile se přihlásí klient, zaregistruje ho. Tento klient bude čekat na to, až se přihlásí nějaký další klient, s kterým by mohl začít hru. Jakmile budou k dispozici dva nehrající klienti, založíme jim automaticky hru. A stále budeme čekat na další.

Jinými slovy, v tuto chvíli budeme dělat několik věcí zároveň:

- pro každou již přihlášenou dvojici budeme organizovat hru
- pro další klienty budeme hledat vhodné protějšky na přihlášení do hry

Jde o krátký program, takže to nyní není třeba do detailů rozebírat. Vše bude jasné v průběhu.

Server

Hned teď si ukažme celý skript server.pl. Budeme využívat modulu Kostky::Server, který budeme muset ještě dodělat.

```
#!/usr/bin/perl -w
use strict;
use warnings;
use Kostky::Server;

while (1){
my $vitez;
my $kostky_server = new Kostky::Server;
$kostky_server->spust;
}
```

Nyní se pojďme podívat, jak bude vypadat základ modulu Kostky::Server (tj. soubor Kostky/Server.pm).

```
package Kostky::Server;
use strict;
use warnings;
use IO::Socket;

sub new {
my($self) = @_;
my $f = {};
bless $f;

my $hrac = IO::Socket::INET->new(
Proto     => "tcp",
LocalPort => 9005,
LocalAddr => "localhost",
Listen    => 2,
Reuse     => 1
) or die "Nelze vytvorit socket! $!";

$f->{"hrac"} = $hrac;
$f->{"skore1"} = 0;
$f->{"skore2"} = 0;

return $f;
}

sub spust {
...
}
```

Zatím šlo pouze o rutinní záležitosti a není zde nic, co bychom nečekali. Tj. v kostruktoru jsme vytvořili spojení (jsme serveru), počet klientů nastavili na 2 (každou hru budou hrát pouze 2 hráči) a skóre jsme nastavili na 0:0.

Nyní bude naším úkolem implementovat metodu spust, která bude dělat veškerou práci serveru.

Máme za úkol řídit hru libovolnému množství hráčů, tj. použijeme fork na odštěpení jednotlivých instancí hry. Tentokráte půjde dokonce o jednu instanci pro dva hráče. Schéma metody bude tedy vypadat takto.

```
sub spust {
while(1){

#připojení dvou hráčů

next unless $pid = fork;

#obsloužíme odštěpený proces

exit;
}
}
```

Hledání dvojic hráčů

Připojení dvou hráčů znamená naplnění proměnných $self->{"vzdaleny1"}, $self->{"vzdaleny2"}, $self->{"nick1"}, $self->{"nick2"} hodnotami. To znamená, že musíme získat hodnoty pro spojení na oba hráče a jejich jména. Na ty se zeptáme klientů, jakmile se připojí - a to tak, že jim odešleme zprávu s obsahem "1\n". Už bude na klientovi, aby tomuto protokolu rozumněl (tj. o to se budeme starat později).

Oba hráče připojíme pomocí cyklu o dvou iteracích. Samozřejmě to lze rozepsat do jednotlivých iterací.

```
for(my $i=1; $i<=2; $i++){
$vzdaleny = $self->{"hrac"}->accept();
$vzdaleny->autoflush(1);
print $vzdaleny "1\n";
chomp($nick = <$vzdaleny>);
do_logu("-", "$nick pripojen");
$self->{"vzdaleny$i"} = $vzdaleny;
$self->{"nick$i"} = $nick;
}
```

Všimněme si tohoto řádku.

```
print "$nick pripojen\n";
```

Pokud použijeme print směrovaný jinam než do socketu, bude to něco jako logování. Nepoužíváme zde log soubor, ale směřujeme výstup na standardní výstup. Logovat budeme po každé akci.
Vhodnější by bylo logování nějak sjednotit. Odteď budeme v lozích zaznamenávat také čas a PID (neboť mícháme výstup všech her do jednoho logu). Pro tento účel si napíšeme jednoduchou funkci do_logu, která pošle jeden log záznam ve formátu 'PID čas zpráva'na standardní výstup.

```
sub do_logu {
my($pid, $zprava)=@_;
my($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
print "$pid $hour:$min:$sec $zprava\n";
}
```

Řízení hry
Nyní již pojďme vyřídit odštěpený proces. Zahájíme hru. Při té příležitosti zapíšeme do logu následující zprávu.

```
do_logu($pid, "V teto hre hraji ".$self->{"nick1"}." a ".$self->{"nick2"});
```

Před tím, než napíšeme hlavní smyčku hry, nastavme kritéria pro konec hry. Funkce konec nám vyhodnotí, zda někdo vyhrál a případně kdo. Při naší hře v kostky je kritérium jediné - stačí dosáhnout 5 bodů.

```
sub konec {
my($self) = @_;
my $vitez;

return 1 if $self->{"skore1"}==$SKORE_VITEZE;
return 2 if $self->{"skore2"}==$SKORE_VITEZE;
return -1; #není vítěz
}
```

A teď konečně přistupme k jádru serveru - napíšeme onu hlavní smyčku. Bude vypadat nějak takto.

```
while(($vitez = $self->konec) == -1) {
$cislo_tahu++;

#nalosujeme, kdo bude v tomto tahu začínat
#necháme hráče "hodit" kostkou
#vyhodnotíme hody, a pošleme hráčům výsledek
}
```

Začneme tedy losem, kdo bude začínat aktuální tah. My to uděláme tak, že v polovině případů prohodíme hráče a v polovině ne. Toto řešení má své výhody i nevýhody, nicméně nám plně postačuje.

```
if((rand(2)>1 ? 1 : 0)==0){
($self->{"vzdaleny1"}, $self->{"vzdaleny2"}, $self->{"nick1"},
$self->{"nick2"}) = ($self->{"vzdaleny2"}, $self->{"vzdaleny1"}, $self->{"nick2"},
$self->{"nick1"});
}
```

Pro jednoduchost nyní zavedeme proměnné $klient1 a $klient2 jako aliasy pro spojení na hráče.

```
$klient1 = $self->{"vzdaleny1"};
$klient2 = $self->{"vzdaleny2"};
```

Dále musíme hráčům oznámit, kdo bude začínat a kdo bude hrát jako druhý. Pošleme tedy zprávu "1\n" hráči, který bude začínat a zprávu "2\n" hráči, který musí počkat na jeho výsledek.

```
print $klient1 "1\n";#hráč1 začíná
print $klient2 "2\n";#hráč1 čeká
```

Dále čekáme na hráče, který má začínat, až hodí. Jakmile se tak stane, podíváme se, jaké číslo je kostce pomocí funkce rand (tj. v okamžiku, kdy hodí, stopneme čas a podle něj určíme počet ok).

```
do_logu($pid, "Cekame, az hodi ".$self->{"nick1"});
my $hod = <$klient1>;
my $vysledek_hodu1 = int(rand(6))+1;
```

Výsledek oznámíme oběma hráčům.

```
print $klient1 "$vysledek_hodu1\n";
print $klient2 "$vysledek_hodu1\n";
do_logu($pid, $self->{"nick1"}." hodil $vysledek_hodu1");
```

Následně čekáme na hod od druhého hráče.

```
do_logu($pid, "Cekame, az hodi ".$self->{"nick2"});
$hod = <$klient2>;
my $vysledek_hodu2 = int(rand(6))+1;
```

Výsledek taktéž oznámíme.

```
print $klient2 "$vysledek_hodu2\n";
print $klient1 "$vysledek_hodu2\n";
do_logu($pid, $self->{"nick2"}." hodil $vysledek_hodu2");
```

Na závěr zaktualizujeme skóre (přidáme tomu hráči, který hodil na kostce více) a výsledky pošleme.

```
#aktualizujeme skore
$self->{"skore1"}++ if $vysledek_hodu1>$vysledek_hodu2;
$self->{"skore2"}++ if $vysledek_hodu1<$vysledek_hodu2;

print $klient1 $self->{"nick1"}." vs. ".$self->{"nick2"}." ".
$self->{"skore1"}."-".$self->{"skore2"}."\n";
print $klient2 $self->{"nick1"}." vs. ".$self->{"nick2"}." ".
$self->{"skore1"}."-".$self->{"skore2"}."\n";
do_logu($pid, $self->{"nick1"}." vs. ".$self->{"nick2"}." ".
$self->{"skore1"}."-".$self->{"skore2"});
```

Tím je hlavní cyklus hotov. Za něj se program dostane pouze tehdy, je-li znám vítěz. Proměnnou $vitez již máme naplněnou - z hlavičky cyklu. Stačí tedy odeslat celkové výsledky a jsme hotovi.

```
$porazeny = $vitez==1?2:1;
```

```
print {$self->{"vzdaleny$vitez"}} "-1\n";
print {$self->{"vzdaleny$porazeny"}} "-2\n";
```
Tím jsme dokončili herní server.

Klienti

Pojďme se podívat na druhou část a tou bude napsání klienta, který bude umět s naším serverem komunikovat. Pro jednoduchost nebudeme psát žádný speciální modul, ač by to šlo (a při náročnější hře by to bylo vhodné), ale spokojíme se se skriptem, který bude vše řešit sám.

```
#!/usr/bin/perl -w
use strict;
use warnings;
use IO::Socket;
```
Nejprve od uživatele zjistíme, kde server běží.
```
print "Zadej IP adresu serveru: ";
chomp($ip_serveru = <STDIN>);
```
Server po nás taktéž bude chtít zadat jméno, takže se na něj hned také zeptáme.
```
print "Zadej nick: ";
chomp($nick = <STDIN>);
```
Teď se již k serveru můžeme připojit a odešleme mu naše jméno. Pokud obdržíme od serveru odpověď, je to dobrá zpráva, protože se nám připojit se podařilo.
```
my $vzdaleny = IO::Socket::INET->new(
    Proto    => "tcp",
    PeerPort  => 9005,
   PeerAddr  => $ip_serveru
) or die "Nelze vytvorit socket! $!";

print $vzdaleny "$nick\n";
print "Jsme pripojeni!\n" if <$vzdaleny>;
```
Nyní to již nebude těžké - budeme pouze reagovat na to, na co se server ptá. Půjde jen o zesynchronizování se se serverem. Program bude samozřejmě opět ve formě smyčky. Na začátku každého tahu nám server posíla informaci o tom, zda začínáme nebo ne. Proto od něj očekáváme zprávu.
```
while(1){
my $na_tahu = <$vzdaleny>;
chomp $na_tahu;

if($na_tahu == 1){
#hrajeme jako první
}elsif($na_tahu == 2){
#hrajeme až po soupeři
}
...
}
```
Místo tří teček ještě vepíšeme následující kód, neboť právě tímto způsobem server oznamuje vítězství či porážku.
```
elsif($na_tahu == -1){
  print "JSI VITEZ\n";
  last;
}elsif($na_tahu == -2){
print "JSI PORAZENY\n";
  last;
  }
```
Na závěr dokončíme první dvě podmínky. Zde jen recipročně reagujeme na server.
```
if($na_tahu == 1){
print "Mas v ruce kostku. Stiskni ENTER pro hod!";
    $hod = <STDIN>;
    print $vzdaleny "1\n";
  chomp($vysledek_hodu = <$vzdaleny>);
print "Hodil jsi $vysledek_hodu. Cekame na soupere.\n";
  chomp($vysledek_soupere = <$vzdaleny>);
  print "Souper hodil $vysledek_soupere.\n";
    $skore = <$vzdaleny>;
      print $skore;
  }elsif($na_tahu == 2){
  print "Cekame na hod soupere.\n";
  chomp($vysledek_soupere = <$vzdaleny>);
  print "Souper hodil $vysledek_soupere.\n";
print "Mas v ruce kostku. Stiskni ENTER pro hod!";
    $hod = <STDIN>;
    print $vzdaleny $hod;
  chomp($vysledek_hodu = <$vzdaleny>);
    print "Hodil jsi $vysledek_hodu.\n";
    $skore = <$vzdaleny>;
      print $skore;
  }
```
Tím jsme hotovi s klientem.

Závěr

Nyní máme server i klienty hotové. Myšlenkově nešlo o nic náročného, ovšem při těchto aplikacích zabere často hodně času ladění.

Všechny tři soubory si můžete stáhnout a vyzkoušet: server.pl, klient.pl, Kostky/Server.pm.
Můžeme nyní vyzkoušet, jak se dá náš server použít.

*Server *** Klient1 *** Klient2 *** Klient3*

*Klient4 *** Klient5 *** Klient6*

Na základě této hry bychom nyní mohli analogicky napsat cokoliv jiného - například síťové šachy, piškvorky a jiné. Z hlediska programování nebudou o moc složitější. Půjde pouze o více psaní.

Perl (77) - Služby internetu

Dnes si představíme dva moduly z archivu CPAN, které dovedou využít internetových služeb FTP a POP3.

Existuje několik modulů, které poskytují rozhraní pro komunikaci s některými síťovými službami. Můžeme si tím pádem vytvořit jednoduché emailové klienty nebo FTP klienty.

FTP klient

FTP je protokol, který, jak již je z názvu File Transfer Protocol patrné, umožňuje pohodlný přenos dat mezi počítači. Funguje na mechanizmu klient - server. Pro Perl je v archivu CPAN k dispozici modul nazvaný Net::FTP, jež právě FTP komunikaci poskytuje. Na příkladu si ukážeme, jak se připojit k nějakému vzdálenému serveru a jak z něj získávat a posílat mu data.

Abychom si mohli vyzkoušet funkce, jež budeme dále popisovat, je třeba mít nějaký vzdálený počítač, kam se budeme moci přes FTP připojit. Dále budu předpokládat, že máme server ftp.server.cz, na kterém existuje uživatel user s heslem pAssW0Rd.

Práce s modulem Net::FTP zahrnuje několik kroků. Jsou to následující.

1. Spojení s FTP serverem
2. Přihlášení se k serveru pod uživatelským jménem
3. Uživatelské akce - samotná práce se soubory na serveru
4. Ukončení spojení

Spojení a příhlášení

Ze všeho nejdříve vytvoříme proměnné s přihlašovacími údaji.

```
use Net::FTP;
my $hostname = "ftp.server.cz";
my $username = "user";
my $password = "pAssW0Rd";
```

Dále již můžeme volat FTP server. Vytvoříme tedy instanci modulu Net::FTP.

```
my $ftp = Net::FTP->new($hostname) or die "Nelze se spojit s FTP serverem.";
```

A nyní se přihlásíme.

```
$ftp->login($username, $password) or die "Nelze se přihlásit. ";
```

Uživatelské akce

V tomto okamžiku jsme ve spojení se serverem. Pokud znáte shellový příkaz ftp, tak pro vás nebudou další informace již ničím novým. Metody modulu Net::FTP mají totiž stejné názvy jako příkazy nástroje ftp.

Nejdříve si představíme metody pro práci s adresářovou cestou. Funkce cwd mění aktuální adresář - tedy funguje podobně jako příkaz cd v shellu.

```
$ftp->cwd("pub") or die "Nelze změnit adresář. "; #nyní jsme na serveru v adresáři /pub
```

Budeme-li naopak chtít jméno aktuálního adresáře získat a vytisknout, použijeme příkaz pwd.

```
print $ftp->pwd; #tiskne /pub
```

Příkazy ls a ls -l pro výpis obsahu adresáře jsou v příkazovém řádku nepostradatelnými. Modul Net::FTP nám samozřejmě poskytuje i příkazy, které jsou jejich analogiemi. Začneme jednodušším ls. Následující úsek kódu uloží do pole názvy souborů v aktuálním adresáři na serveru a následně každý z nich vytiskne na samostatný řádek.

```
my @ls = $ftp->ls(".") or die "Nelze získat seznam souborů \n";
$" = "\n";
print "@ls\n";
```

Uvedení parametru není pro získání obsahu aktuálního adresáře nezbytné. Jinak lze zadat libovolný dostupný adresář na serveru.

Implementace příkazu ls -l je stejná až na to, že místo metody ls se použije dir.

```
my @ls = $ftp->dir(".") or die "Nelze získat seznam souborů \n";
$" = "\n";
print "@ls\n";
```

Přenos souborů

Nyní se dostáváme k tomu nejdůležitějšímu. Protokol FTP se využívá právě pro přenos souborů. Můžeme posílat soubory na server, ale také stahovat. Slouží k tomu dvě jednoduché metody: get a put.

Již z názvu je patrné, že put provádí upload souboru na server. Budeme-li chtít poslat soubor letter.txt uložený v aktuálním adresáři lokálního stroje do aktuálního adresáře na server, použijeme následující příkaz.

```
$ftp->put("./letter.txt") or die "Nelze provést upload. \n";
```

Pro tok dat opačným směrem, tedy ze serveru na klienta, použijeme funkci get.

```
$ftp->get("send.php") or die "Nelze provést download. \n";
```

Ukončení spojení

A nakonec uzavřeme spojení.

```
$ftp->quit;
```

Existují samzřejmě další a další příkazy a pokročilejší funkce, ale když o ně někdo bude mít zájem, jistě už sám nahlédne do dokumentace.

POP3 klient

Další službou, ke které se pokusíme pomocí Perlu přistupovat, bude email. POP (Post Office Protocol) verze 3 je protokol, který umožňuje stahovat data z emailového serveru. Nevýhodou POP protokolu může být to, že po přečtení emailu se na serveru zpráva smaže. Je však možné přečíst pouze část zprávy a email na serveru zůstane. Další nevýhodou je to, že POP protokol stahuje všechny zprávy na serveru včetně spamu.

V Perlu poskytuje rozhraní pro POP protokol modul Net::POP3, na kterém si práci s POP3 předvedeme.

Předpokladem pro to, aby vám následující kód fungoval, je přístup k emailové adrese. Od poskytovatele emailu si je třeba si zjistit server příchozích zpráv POP3. Předpokládejme, že máme emailovou adresu user@server.cz, POP3 server je pop3.server.cz, uživatelské jméno user a heslo pAssW0Rd.

```
use Net::POP3;
my $hostname = "pop3.server.cz";
my $username = "user";
my $password = "pAssW0Rd";
```

Spojení

Prvním úkolem je připojení k POP3 serveru. Zavoláme tedy konstruktor třídy Net::POP3.

```
my $pop = Net::POP3->new($hostname) or die "Nelze se připojit. $!\n";
```

Přihlášení

Dalším krokem je přihlášení. Metoda login vrací počet nových zpráv ve schránce, takže tuto hodnotu hned přiřadíme.

```
my $zprav = $pop->login($username, $password) or die "Nelze se přihlásit. $!\n";
print "POČET ZPRÁV: $zprav\n\n";
```

Uživatelské akce

Pokud nenastal nějaký problém, můžeme nyní se schránkou pracovat. Metoda listzpřístupňuje seznam zpráv.

```
my $r_zpravy = $pop->list() or die "Nelze získat seznam zpráv. $!\n";
```

Dále v cyklu zobrazíme z každé zprávy hlavičku a 5 úvodních řádků zprávy. K tomu použijeme metodu top.

```
foreach (keys %$r_zpravy) {
print "-----------------------------------\n";
print "Zpráva číslo $_:\n";
my $r_z = $pop->top($_, 5);
print @$r_z;
}
```

Ukončení spojení

A na úplný závěr ukončíme spojení.

```
$pop->quit;
```

Závěr

Jak je vidět, práce se službami internetu není v Perlu díky modulům nic složitého. Dnes jsme si ukázali pouze služby FTP a POP3, nicméně existují moduly pro řadu dalších, které obvykle začínají prefixem Net::. Namátkou uveďme Net::IMAP a Net::HTTP.

Perl (78) - Databáze - úvod

Tímto dílem začínáme miniseriál o práci s SQL databázemi. Dnes se budeme zabývat některými teoretickými aspekty a také se k databázi poprvé připojíme.

DBI nebo-li DataBase Interface je nástroj pro přístup k SQL databázím. Je to rozhraní, které umožňuje přístup ke všem známým a některým méně známým databázovým systémům.

Cílem DBI je vytvoření jednotného rozhraní pro přístup k různým databázovým systémům. Jeho architektura se skládá ze dvou částí. Samotné DBI definuje pouze rozhraní pro uživatele. O přístup ke konkrétním databázovým systémům se starají takzvané DataBase Drivery (DBD).

Pro každou podporovanou databázi existuje speciální DBD ve formě modulu. Tento mechanizmus je výhodný z hlediska přenositelnosti. Protože DBI je na databázovém systému nezávislá a DBD závislá část, někdy se také akronymu DBI přiřazuje význam DataBase Independent a akronymu DBD DataBase Dependent.

Všechny nejznámější databáze již mají svůj DBD a lze je tedy ve spolupráci s Perlem používat. Seznam všech dostupných DBD modulů je například v archivu CPAN.

K tomu, abychom mohli používat některý z databázových systémů, ho musíme mít zvlášť nainstalovaný. Pokud žádný nemáte nebo se jen chcete dozvědět něco více o databázích, pak využijte zde na Linuxsoftu speciální seriály k tomu určené: Postgres od Marka Olšavského nebo MySQL od Petra Zajíce.

Práce s databází

Pokud máme již vše připraveno (tedy hlavně dostupnou databázi a modul DBI), můžeme začít psát Perl skripty, které umějí s databází pracovat. Prvním krokem k tomu nezbytným je navázání spojení s tou kterou databází. Poté můžeme klást dotazy v jazyce SQL a nakonec databázové spojení opět uzavřeme.

Kontrola ovladačů

Ještě než se začneme připojovat, měli bychom vědět, pro které databáze máme k dispozici ovladače (drivery). Jako první krok tedy necháme tedy provést následující příkaz.

```
$ perl -MDBI -e 'print DBI->available_drivers'
```

Často ještě lepší je zobrazení s čísly instalovaných verzí.

```
$ perl -MDBI -e 'DBI->installed_versions'
```

Pokud je námi požadovaný systém mezi těmi jmenovanými, můžeme se již připojit. V opačném případě bude ještě nutné sehnat příslušné DBD moduly, nejsnadněji z CPAN.

Navázání spojení

Abychom se mohli připojit, musíme zpravidla znát několik údajů.

- jméno databázového systému, resp. jemu příslušného ovladače
- jméno konkrétní databáze
- případně umístění databáze (počítač, port)
- login a heslo uživatele, který má přístup k této databázi

Poznámka - Slovo databáze má dva významy. Prvním je nějaká uložená množina dat a druhým je tato množina i s konkrétními softwarovými nástroji pro jejich manipulaci. Tam, kde je to zásadní, budeme tu druhou variantu nazývat databázovým systémem.

Samotné připojení k databázi nám zajistí metoda connect, která vrací ovladač. Tato metoda vyžaduje jako argument právě výše uvedené identifikační údaje a navíc je možné přidat hash speciálních atributů. Obecný zápis volání metody connect tedy vypadá takto.

```
$dbh = DBI->connect($zdroj_dat, $uzivatel, $heslo, \%atributy);
```

Alespoň pro představu uveďme některé příklady jako obsah prvního argumentu této metody.

```
"dbi:JMÉNO_DRIVERU:JMÉNO_DATABÁZE"
"dbi:JMÉNO_DRIVERU:dbname=JMÉNO_DATABÁZE"
"dbi:JMÉNO_DRIVERU:dbname=JMÉNO_DATABÁZE;host=HOST;port=PORT"
"dbi:JMÉNO_DRIVERU:JMÉNO_DATABÁZE@HOST:PORT"
```

Volání metody connect má nejen z tohoto hlediska řadu rozmanitých možností. Většinou si však vystačíme jen se základními znalostmi. Pro více informací na toto téma odkažme na dokumentaci.

Nyní se již podívejme, jak by tento příkaz vypadal konkrétně pro systém Postgres, databázi testovaci_db, uživatele user s heslem password. Navíc přidáme or die podmínku a příkaz pro odpojení.

```
use DBI;
my $dbh = DBI->connect("dbi:Pg:dbname=testovaci_db", "user", "password") or die "Nelze se spojit s databází!";
...
$dbh->disconnect;
```

Pro ukázku si ještě ukažme, jak bychom se připojili k systému MySQL.

```
my $dbh = DBI->connect("dbi:mysql:dbname=testovaci_db", "user", "password") or die "Nelze se spojit s databází!";
```

Pro toho, kdo používá standardního řádkového klienta pro systém MySQL, lze poslední příkaz přirovnat k následujícímu příkazu zadaného do shellu.

```
$ mysql -uuser -ppassword
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.51a SUSE MySQL RPM

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use testovaci_db
Database changed
mysql>
```

Na závěr uveďme, že první argument předávaný metodě connect lze získat metodou data_sources, vyvolanou nad objektem $dbh.

Ošetřování chyb

Existují tři hlavní metody, které vracejí informace o chybách. Tři proto, protože každá vrací chybu v jiném formátu. První z nich, metoda err, vrací chybový celočíselný kód. Tento kód je dostupný pouze do volání další metody (to platí pro skoro všechny metody, výjimkou je pochopitelně metoda err a pár dalších), poté je nahrazen výsledkem tohoto volání. Ke kódu, který vrací err lze přistupovat také přes proměnnou $DBI::err.

Další a podobnou metodou je errstr nebo $DBI::errstr. Tato metoda vrací řetězec, který chybu popisuje.

Ještě existuje jedna metoda pro detekci chyb. state vrací kód podle standardu SQLSTATE.

Jak již jsme se zmínili, metoda connect přijímá jako poslední volitelný argument odkaz na hash atributů. Právě dvěma z těchto atributů jsou atributy RaiseError a PrintError. Oba je můžeme nastavit na hodnoty 1 (zapnuto) nebo 0 (vypnuto). Pomocí zapnutého RaiseError dáme na vědomí, že při chybě má dojít k okamžitému volání příkazu die. Naopak PrintError pouze tiskne chybu pomocí warn na standartní chybový výstup.

Zajímavá situace nastane, pokud zkusíme oba atributy vypnout - tedy pokud zavoláme metodu connect následovně.

```
$dbh = DBI->connect("dbi:mysql:dbname=testovaci_db", "user", "password", {RaiseError => 0, PrintError => 0})
or die "Nelze spojit s mysql. ".$db->errstr;
```

Nyní nebudou hlášeny žádné chyby a všechna volání metod nad objektem $dbh si musíme ošetřit sami pomocí konstrukce or die a metody errstr.

Další možností vlastního ošetřování je atribut HandleError. Tomu lze přiřadit odkaz na podprogram, který bude proveden po vyvolání chyby. Je tak možné si například měnit text chybových hlášení.

Atributů je celá řada a jejich kompletní popis lze nalézt v dokumentaci.

Tento díl obsahuje nejdůležitější metody, jichž lze využít při práci s daty.

Nyní se již umíme připojit k databázi, takže můžeme v podstatě hned teď začít posílat SQL dotazy.

Posílání příkazů, od nichž nečekáme vrácená data

K posílání příkazů databázi použijeme metodu do. Jejím parametrem je SQL dotaz a návratovou hodnotou počet ovlivněných řádků nebo hodnota -1. Metoda do je používána většinou pro jiné než SELECT dotazy, které nebudou opakovány.

```
$dbh->do("DELETE FROM tabulka WHERE rating>100");
```

Kdybychom metodu do použili na SELECT dotaz, neexistuje cesta, jak získat výsledná data - a ta v takovém případě většinou získat chceme. Také v případě, že bude podobný dotaz opakován, existují jiné a někdy vhodnější metody volání.

Získávání dat příkazem SELECT

Metoda do je ve skutečnosti pouze spojením dvou jiných metod. Můžeme totiž také nad příslušným databázovým ovladačem dotaz nejprve obecně připravit (metodou prepare) a až poté ho provést s konkrétními hodnotami (metodou execute). Jednou připravený dotaz lze provést i vícekrát, což snižuje náročnost programu.

Příprava dotazu

Metoda prepare vrací objekt, který vznikl podle předaného SQL dotazu. SQL dotaz v této chvíli ještě nemusí být konkrétní, ale lze ho zadat pouze v obecném tvaru. Pokud bychom totiž následně volali podobný dotaz, pouze s upravenou podmínkou, nemusíme ho připravovat podruhé, ale bude možné ho hned provést.

Voláním metody prepare by mohlo vypadat následovně.

```
my $q1 = $db->prepare("SELECT * FROM tabulka WHERE rating=?");
```

Co je vlastně myšleno tím "obecným tvarem SQL dotazu"? Jak vidíme v posledním řádku kódu, dotazujeme se na řádky tabulky tabulka podle hodnoty sloupce rating. Otazník nebo lépe zástupný znak (placeholder) je na místě hodnoty právě proto, že máme v úmyslu tento dotaz volat několikrát po sobě, pokaždé s jinou hodnotou v podmínce. Má to ten důvod, že ho lze dále nahradit konkrétní hodnotou.

Poznámka - Práce se zástupným znakem závisí na konkrétním ovladači pro databázi. Následkem toho existují v tomto směru pro různé ovladače drobné odlišnosti. Některé ovladače například umožňují zástupný znak vložit i za "méně obvyklé hodnoty", jiné pracují místo otazníku i s jinými symboly, které mohou mít nějaký vedlejší význam. Nevýhoda těchto odlišností je v nepřenositelnosti a proto se budeme držet obvyklých pravidel.

Otazníků může být v jediném dotazu i více. Ani v případě, že otazník je zástupným znakem pro řetězec, kolem něj nepíšeme uvozovky.

```
my $q2 = $db->prepare("SELECT * FROM tabulka WHERE rating=? AND kod=?");
```

Nicméně zástupným znakem nelze nahrazovat cokoliv. Například dotazy následujícího typu projdou pouze výjimečně (závisí na ovladači).

```
SELECT * FROM ?
SELECT sloupec1, ? FROM tabulka
```

My si v dalším průběhu seriálu vždy vystačíme s nahrazováním hodnot u porovnávání nebo vkládání dat. Více informací o tom, za co lze psát zástupný znak lze nalézt opět v dokumentaci.

Vykonání připraveného dotazu

Nyní můžeme volat nad objektem vráceným metodou prepare další metody. Tou nejdůležitější je metoda execute, která již posílá databázi konkrétní hodnoty. Metodou finish můžeme deklarovat předčasný konec odebírání dat.

Po volání execute již máme k dispozici vrácená data. Chceme-li znát výsledky dotazu SELECT * FROM tabulka WHERE rating=100, tedy předchozího dotazu s hodnotou 100 místo otazníku, přidáme tuto hodnotu jako parametr metodě execute.

```
$q1->execute(100);
```

Pokud máme v dotazu otazníků více, předáme metodě execute seznam zástupných hodnot.

```
$q1->execute(100, "2007AA");
```

Pokud nedodáme požadované množství argumentů nebo dodáme nedefinované hodnoty, doplní se seznam hodnotami NULL. Pro doplňování hodnot za zástupné znaky lze užít také metodu bind_param.

Získání vrácených dat

V tomto okamžiku je již SQL příkaz proveden a nyní jen zbývá vydolovat z objektu $q1 vrácená data. K tomu existuje další metoda fetchrow_array nebo fetchrow_arrayref. Obě tyto metody postupně vrací vyhovující záznamy (tzn. voláme je opakovaně a pokaždé dostaneme další řádek). V případě, že již byly vráceny všechny záznamy a není žádný další, tak metody vrací hodnotu undef. Odlišnost spočívá v tom, že fetchrow_array vrátí data ve formě pole a fetchrow_arrayref jako odkaz na něj.

Obě metody jsou užívány často v hlavičce cyklu while. V každé iteraci cyklu přiřadíme do příslušných proměnných další vyhovující data. Takto vypadá typické užití metody fetchrow_array.

```
while(my($id, $kod, $rating) = $q->fetchrow_array){
print "Záznam: $kod, $id, $rating\n";
}
```

Počet ovlivněných řádků

Často se také může hodit metoda rows, která vrací počet posledním dotazem ovlivněných řádků.

Další metody pro získávání záznamů

S výše uvedeným si vystačíme, ale nejsou to jediné metody pro získávání záznamů. Uveďme ještě několik dalších metod, které mohou být v některých případech výhodnější.

Dost častými dotazy jsou ty, od nichž čekáme jednořádkový výsledek. Pro takové existují zvláštní metody selectrow_array a selectrow_arrayref. Ty jsou kombinací tří metod.

Zahrnují prepare, execute a fetchrow_array resp. fetchrow_arrayref. Na základě jediného argumentu - SQL dotazu - vrátí záznam. Tím tyto metody celý mechanizmus podstatně zjednodušují.

```
my($s1, $s2) = $dbh->selectrow_array("SELECT kod, rating FROM tabulka LIMIT 1");
print "Záznam: $s1, $s2\n";
```

Metoda fetchrow_hashref dělá podobnou činnost jako již uvedená metoda fetchrow_arrayref(jejím aliasem je metoda fetch). Jak již z názvu plyne, rozdíl mezi nimi je v tom, že zde se data načítají do hashe. Následující kód vytiskne totéž co výše uvedený a jeho význam je intuitivní.

```
while(my $p_zaznam = $q->fetchrow_hashref){
print "Záznam: ". $$p_zaznam{"kod"}.", ".$$p_zaznam{"id"}.", ".$$p_zaznam{"rating"}."\n";
}
```

Další způsob, který lze užít pro získávání dat je navázání sloupců. Spočívá v tom, že si pomocí metody bind_col (nebo bind_columns) svážeme každý sloupec s nějakou proměnnou a po zavolání fetchjsou v těchto proměnných data jednoho záznamu.

```
$sth->bind_col(1, \$id);
$sth->bind_col(2, \$kod);
$sth->bind_col(3, \$rating);

while(my $p_zaznam = $sth->fetch){
print "Záznam: $kod, $id, $rating\n";
}
```

Pomocí metody selectall_arrayref lze získat odkaz na pole záznamů. Potom můžeme všechna data z tabulky zobrazit tímto způsobem.

```
my $p_zaznamy = $sth->fetchall_arrayref;
for (@$p_zaznamy){
print "Záznam: @$_\n";
}
```

Na podobném mechanizmu funguje metoda fetchall_hashref. Je zde stejný rozdíl jako mezi fetchrow_arrayref a fetchrow_hashref. Jednotlivé sloupce nejsou v hashi uloženy pod číselnými indexy ale pod názvy sloupců.

Perl (80) - Databáze - závěrečné poznámky

Uvedeme několik poznámek o dalších možnostech při práci s databázemi a o tom, co by mohlo působit problémy a je na to třeba dávat pozor.

Rychlý výpis výsledků pro účely ladění

Pokud potřebujeme výsledky SELECT dotazu pouze zobrazit, aniž bychom je chtěli dále upravovat, je zde k dispozici metoda dump_results. Užívá se zejména pro testování dotazů.

Po jejím zavolání jsou vypsána na určený výstup získaná data v nějakém daném formátu. Volání metody dump_results vypadá následovně.

```
$pocet_radku = $dbh->dump_results($max_delka_hodnoty, $oddelovac_radku, $oddelovac_hodnot, $ovladac);
```

Žádný parametr však není povinný.

Ve zdrojovém kódu pak pro výpis stačí uvést například toto.

```
my $sth = $dbh->prepare("SELECT * FROM tabulka");
$sth->execute();
my $rows = $sth->dump_results(15, "\n", "\t");
```

Problémy s uvozením v dotazech

Nyní narážíme na potenciální problém, který nastane v okamžiku, kdy do podmínky dotazu budeme chtít zahrnout řetězec. Pro začátek se podívejme na tento řádek.

```
my $sth = $dbh->prepare("SELECT * FROM tabulka WHERE description='$popis'");
```

Zde musíme opravdu vědět, co za dotaz to vlastně databázi posíláme. Co když se uvnitř proměnné zahrnuté do dotazu vyskytuje nějaký apostrof? Co když v $popis bude například hodnota získaná z formuláře na webu od nějakého návštěvníka a bude mít hodnotu"neco' or vyhra='ano"? Proti takovému obsahu je nutno se bránit.

V proměnné v dotazu může být obecně cokoliv a proto knihovna DBI pro ošetření nebezpečných dat nabízí vlastní metodu. Metoda quote nahradí v zadaném řetězci podle kritérií použitého databázového systému citlivé znaky escape sekvencemi.

Problém tedy vyřešíme, když kód přepíšeme do nové podoby.

```
my $quoted = $dbh->quote($popis);
my $sth = $dbh->prepare("SELECT * FROM tabulka WHERE description=$quoted");
```

Metadata

DBI nabízí několik metod pro získávání dat, která popisují jiná data. V našem případě jde například o informace o tabulkách, sloupcích, klíčích apod.

Jen pro představu uveďme, jak se s takovými metodami pracuje. Ukážeme si metodu table, která po zavolání nad ovladačem databáze vrací tabulky v ní obsažené.

```
print "Tabulka: $_\n" for $dbh->tables;
```

Detailnější informace o tabulkách nabízí metoda table_info. Existují i další podobné metody s názvem většinou končícím na _info.

Více informací lze nalézt v dokumentaci.

Trasování

Pokud nám něco dělá při práci s databází neplechu a potřebujeme to vypátrat, může pomoci trasovací režim. Pomocí něj můžeme sledovat všechny operace provedené s DBI.

Trasovací režim se zapíná metodou trace nad ovladačem databáze nebo jiným objektem. Jako argument tato metoda přijímá trasovací mód (0 vypíná trasování, nebo 1-5 pro rozsah zobrazovaných informací vzestupně) a jméno souboru, kam se trasovací informace budou zapisovat.

```
$dbh->trace(2, "dbitrace.log");
```

Většina řádků v dbitrace.log začíná šipkou. Směr -> označuje to, co bylo metodám předáno, opačná znázorňuje to, co jimi bylo vráceno.

Přidávání záznamů

Použití prepare a execute je výhodné také při mnohanásobném volání příkazu INSERT. Jako příklad si můžeme uvést konverzi dat ze zdroje do následující tabulky.

```
create table tabulka (
id serial primary key,
sloupec1 varchar(255),
sloupec2 varchar(255),
sloupec3 varchar(255)
);
```

Ve zdroji budou řádky ve formátu

```
sloupec1:sloupec2:sloupec3
```

A následující program naplní tabulku daty ze zdroje. Nejdříve příkazem split rozdělíme řádek na jednotlivé hodnoty a výsledný seznam předáme metodě execute.

```
$sth = $dbh->prepare("INSERT INTO tabulka (sloupec1, sloupec2, sloupec3) VALUES (?, ?, ?)");
while(<DATA>){
chomp;
$sth->execute(split /:/);
}
```

Podpora transakcí

DBI podporuje transakční zpracování dotazů. Podmínkou k tomu samozřejmě je, aby transakce podporoval i databázový systém (to znamená, že funkčnost závisí na ovladači). Problémy tak mohou nastat například u starších verzí MySQL.

Jsou dvě možnosti jak aktivovat transakční zpracování. Buď vypnout atribut AutoCommit nebo zavolat metodu begin_work.

```
$dbh->begin_work;
```

Dále můžeme například vkládat záznamy a kontrolovat, zda všechny dopadly v pořádku. Pokud jsme nakonec spokojeni s výsledkem, transakci potvrdíme a všechny změny se tak promítnou. Potvrzení provedeme metodou commit a v případě neúspěchu akce odvoláme pomocí rollback. Obě metody jsou volány nad ovladačem databáze, nikoliv dotazu.
Správnost všech dotazů ověříme podmínkou, která bude logickou spojkou AND spojovat všechna volání metody execute. V případě, že nenulový počet volání selže (to jest selže alespoň jedno), podmínka bude mít hodnotu false. Toto lze provést například tímto způsobem.

```
my $commit = 1;
while(<DATA>){
    chomp;
$commit &&= $sth->execute(split /:/);
    }

    if($commit){
$dbh->commit;
    }else{
$dbh->rollback;
    }
```

DBI shell
Jako zajímavost na závěr stojí za zmínku databázový klient DBI::shell (příkaz dbish), který pracuje s databázemi stejně jako DBI. Více informací o něm je v dokumentaci.
Perl (81) - CGI - příprava webového serveru

CGI je protokol pro propojení webového serveru s aplikacemi. Nainstalujeme webový server Apache a uděláme nezbytná nastavení k tomu, abychom rozběhali CGI.

CGI znamená Common Gateway Interface. Je to rozhraní, který umožňuje spolupráci programu a webového serveru. Funguje to tak, že webový server pověří nějaký externí program, aby obsloužil klienta.
Podrobněji, tato technologie funguje funguje na následujícím principu.

1. Server obdrží požadavek od klienta
2. Server pošle požadavek na obsloužení programu
3. Program pošle výsledek požadavku serveru
4. Server pošle výsledek požadavku klientovi

Server tak vůbec nemusí zajímat, co dělá CGI skript. Stará se pouze o umístění programu a předání hodnot pomocí CGI rozhraní. Hodnoty lze programu předávat buď metodou POST nebo GET. Odpověď na požadavek pak CGI skript předává serveru skrz standardní výstup. Na začátku tohoto výstupu musí být uvedeny hlavičky.
Důležité také je, že CGI skript běží na straně serveru. To znamená, že server klientovi nezasílá program, ale pouze jeho výsledky. Vzhledem ke klientským skriptovacím jazykům to má své výhody a nevýhody a obě technologie se mohou výhodně doplňovat.
CGI pracuje metodou "on-the-fly" a umí díky tomu zajistit dynamičnost webu. To znamená, že umožňuje interaktivní komunikaci s klientem. Webový server díky CGI dokáže místo statické stránky zasílat stránku ovlivněnou požadavkem. Tato stránka je tedy v reálném čase vygenerována externím programem.
Typickou ukázkou použití CGI je obsluha formulářů, které na základě vstupních dat vyberou vyhovující záznamy z databáze. Programům, které tímto způsobem externě spolupracují s webovým serverem se říká CGI skripty. Jak se liší takový CGI skript od běžného programu? Teoreticky vůbec nijak. Nicméně je pravidlem, že CGI skript generuje dokument v určeném formátu. Mohou to tedy být například programy, které vytvářejí stránku v nějakém značkovacím jazyce. Nejčastější je v tomto směru jazyk, kterému rozumí webové prohlížeče - tedy HTML. Samozřejmě je ale možné generovat cokoliv - ať už stránky v jiném značkovacím jazyce, čistý text, obrázky (zde jsou typickou ukázkou human checky) nebo něco úplně jiného.
Každý nepřeložený CGI skript musí obsahovat cestu ke svému interpretu, protože CGI musí vědět, v čem má být program zpracován. Je tedy nutné psát na začátek skriptů řádek #!/cesta/k/interpretu.
Zajímavou otázkou je, jak server pozná, který soubor má obyčejně poslat a který ještě předtím zpracovat - tedy zda bude odpovědí na požadavek statický nebo dynamický dokument. Odpovědí je, že to záleží na nastavení serveru. Je třeba určit soubory, které mají být dynamicky zpracovávány. Kritériem je obvykle typ souborů (tedy například můžeme rozlišovat dle přípony, běžně se používá .cgi případně .pl) nebo umístění ve speciálním adresáři s CGI skripty (obvykle s názvem cgi-bin).
Použití CGI zažívalo vrchol na konci minulého století. Dnes se masově prosazují spíše technologie jako je PHP. Rozdíl mezi aplikacemi napsanými v PHP a CGI je v tom, že webový server PHP skripty zpracovává přímo (samozřejmě s výjimkou případu, kdy použijeme PHP jako CGI), narozdíl od CGI skriptů, které, jak již bylo několikrát zmíněno, zpracovává samostatný proces. Nicméně i v dnešní době se s CGI můžeme mnohde setkat. Použití CGI je výhodné zejména tehdy, pokud je třeba provést časově náročné operace.
Ovšem nejen PHP, ale i Perl může být na webovém serveru zpracováván přímo. O tom bude dále v seriálu ještě podrobně pojednáno.
Apache
K rozběhání CGI budeme potřebovat nějaký webový server - použijeme Apache. Pokud Apache ve svém systému nemáte, nainstalujte si apache2 balíček, který by měl být dostupný ve vaší linuxové distribuci nebo si stáhněte aktuální verzi a nainstalujte klasicky pomocí příkazů

```
./configure
make
su -c 'make install'
```

Pokud bude server nainstalován tímto způsobem, mělo by být CGI již normálně nastaveno. Nebude-li vám z nějakých důvodů CGI chodit, je třeba editovat konfigurační soubor Apache.
Konfigurace Apache pro CGI
Pokud nemáte historickou verzi Apache nebo jiná specifika, pak by se měl konfigurační soubor jmenovat /usr/local/apache2/conf/httpd.conf. Pro podporu CGI skriptů přidáme následující řádky.

```
AddHandler cgi-script .cgi

ScriptAlias /cgi-bin /usr/local/apache2/cgi-bin

<Directory "/usr/local/apache2/cgi-bin">
```

AllowOverride None
Options ExecCGI
Order allow,deny
Allow from all
</Directory>
První řádek říká, že všechny soubory s příponou .cgi má server považovat za CGI skripty. Další řádky mapují skutečný adresář /usr/local/apache2/cgi-bin jako virtuální adresář /cgi-bin pro Apache. Do tohoto adresáře můžeme ukládat CGI skripty, jež budeme chtít přes Apache spouštět. Statické dokumenty jsou implicitně ukládány v /usr/local/apache2/htdocs.
To samozřejmě můžeme změnit. Chceme-li například, aby Apache hledal CGI programy v /home/www/cgi-bin a statické dokumenty v /home/www/htdocs, což je pro leckoho pohodlnější, stačí předchozí zápis změnit a upravit direktivu DocumentRoot.
DocumentRoot "/home/www/htdocs"

AddHandler cgi-script .cgi

ScriptAlias /cgi-bin /home/www/cgi-bin

<Directory "/home/www/cgi-bin">
AllowOverride None
Options ExecCGI
Order allow,deny
Allow from all
</Directory>
Ještě poznamenejme, že změny provedené v konfiguračním souboru se projeví až při dalším spuštění serveru.
Soubory Apache
Ty nejdůležitější soubory, které Apache využívá najdeme většinou v adresáři /usr/local/apache2, pokud jsme při instalaci volbou --prefix nenastavili jinou cestu. Dále budeme v seriálu pro zjednodušení předpokládat, že náš Apache je právě v /usr/local/apache2.
Výše zmíněný adresář obsahuje několik podadresářů. Za všechny jmenujme alespoň některé. V adresáři bin nalezneme spustitelné soubory. Nás z nich bude dnes zajímat pouze soubor apachectl, kterým nastartujeme či vypneme server.
/usr/local/apache2/bin/apachectl start
/usr/local/apache2/bin/apachectl stop
Do adresáře s názvem cgi-bin se obvykle umísťují samotné CGI skripty. To nás bude samozřejmě velmi zajímat. Z názvu cgi-bin je patrné, že zde budou výhradně binární soubory. Nicméně skutečnost je většinou jiná.
Dále musíme zmínit adresář conf, kde jsou umístěny konfigurační soubory Apache. Bude nás zajímat zejména soubor httpd.conf, ve kterém jsou umístěna nejdůležitější nastavení serveru.
Adresář htdocs uchovává obsah webových stránek. Jsou zde všechny HTML soubory, obrázky, PHP skripty a podobně. Obsah tohoto adresáře stejně jako adresáře cgi-bin je často měněn, a proto, pokud nevyhovuje jejich umístění v /usr/local/apache2, není nic jednoduššího, než vytvořit si sem odkazy z vhodnějších adresářů.
Za zmínku stojí také adresář manual. Obsahuje několik howto týkající se serveru.
Přístup k CGI skriptům na serveru
Nyní máme nainstalován Apache - vše je tedy připraveno na první spuštění. Nastartujme Apache následujícím příkazem.
/usr/local/apache2/bin/apachectl start
Na adrese 127.0.0.1 nebo chcete-li localhost nyní máme přístupný webový server. Nejjednodušší cesta, jak se s ním spojit, je spustit webový prohlížeč a zadat tuto adresu. Pokud jste nic neměnili v kořenovém adresáři, tak se obvykle zobrazí stránka It works! - tedy soubor /usr/local/apache2/htdocs/index.html.
Právě jsme zobrazili (z pohledu serveru) statický dokument. CGI skript test.cgiuložený v /usr/local/apache2/cgi-bin spustíme zadáním http://localhost/cgi-bin/test.cgi. Důležitý je fakt, že CGI skript můžeme spustit pouze tehdy, máme-li na to práva. Každému CGI skriptu tedy přiřadíme právo pro spouštění, stejně jako adresáři, ve kterém je uložen.
Perl (82) - CGI - první skripty

Napíšeme první CGI skript a vysvětlíme jeho strukturu.

Na úvod poznamenejme, že CGI skripty mohou být napsané prakticky v jakémkoliv programovacím jazyce, který je na serveru (nebo počítači z něj dostupném) interpretovatelný. Lze použít například Perl, Python, Bash, C a máte-li nějaký vlastní jazyk, tak ten také. Právě s Perlem bylo CGI v minulosti spojováno velmi často.
Archiv CPAN nabízí několik modulů usnadňujících tvorbu CGI skriptů. O několik dílů později se jednomu takovému s názvem CGI budeme věnovat. Nyní však budeme postupovat bez podpůrných modulů, protože tak lze snáze problematiku pochopit.
První CGI skript
V každém nekompilovaném CGI skriptu musíme uvést několik věcí.

- pomocí čeho zpracovat skript (například pomocí Perlu - informace pro webový server)
- co je výsledkem našeho skriptu (HTML soubor, text, obrázek, atd. - informace pro nějaký klientův prohlížeč)
- samotný skript

Úvodní řádek určuje cestu k interpretu a vypadá následovně.
#!/cesta/k/interpretu
Dále je třeba pro webový prohlížeč na straně klienta uvést hlavičky dokumentu dle příslušného HTTP standardu. Zde uvedeme bez podrobnějšího vysvětlování jen to, co je nezbytně nutné.
Zopakujme, že nyní jsme v druhém bodu a uvedeme tedy, jaké povahy je výsledek interpretování. Je nutné specifikovat MIME typ zprávy. Zde záleží na tom, co budeme chtít, aby náš CGI skript generoval.
MIME je typ dat. Mezi základní kategorie datových typů patří application, audio, example, image, message, model, multipart, text a video. Každý z nich pak může nabývat různých podtypů. Pro čistý text použijeme MIME text/plain, pro text formátovaný v HTML text/html apod. Více o MIME typech lze nalézt na www.iana.org.
MIME typ uvedeme v hlavičce s názvem Content-type. Bude-li výsledkem CGI skriptu HTML dokument, pak jako druhý řádek CGI skriptu napíšeme toto.

```
print "Content-type: text/html\n\n";
```

Protože HTTP protokol vyžaduje prázdný řádek mezi hlavičkou a tělem zprávy, je mezi nimi sekvence "\n\n". Tím jsme dokončili část s hlavičkami a nyní tedy můžeme začít psát vlastní tělo programu.

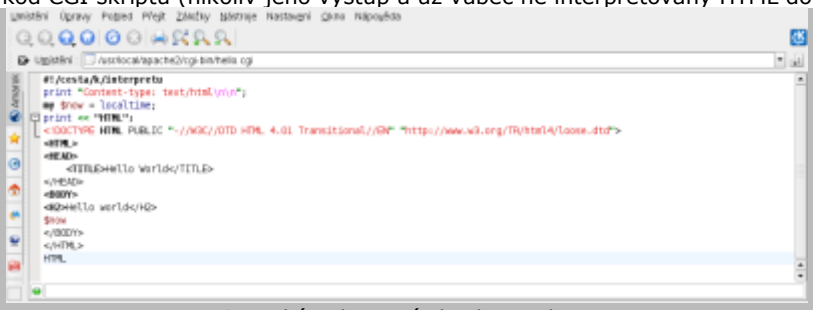
Protože jsme uvedli jako MIME typ text/html, bude náš skript generovat HTML dokument - měli bychom tedy dodržet HTML strukturu (přesněji řečeno, webový prohlížeč to předpokládá a bude se podle toho chovat). Abychom přidali stránce "dynamičnost", zobrazíme aktuální čas.

```
my $now = localtime;
print << "HTML";
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<H2>Hello world</H2>
$now
</BODY>
</HTML>
HTML
```

Nyní tento program spustíme nejprve v příkazovém řádku.

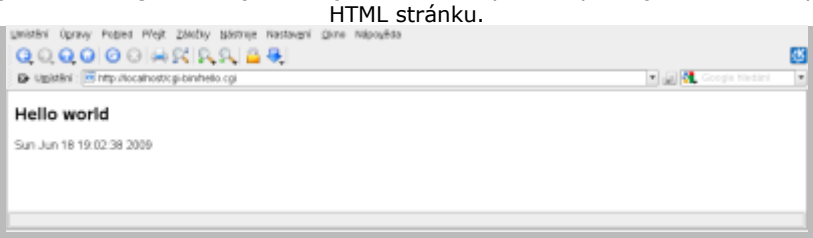
```
$ perl /usr/local/apache2/cgi-bin/hello.cgi
Content-type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<H2>Hello world</H2>
Sun Jun 18 19:02:38 2009
</BODY>
</HTML>
$
```

V podstatě se stalo přesně to, co bychom očekávali. Vygeneroval se nám HTML dokument s nějakou úvodní hlavičkou navíc. Nyní můžeme zkusit spustit skript v prohlížeči. Pokud zadáme jako umístění /usr/local/apache2/cgi-bin/hello.cgi, zobrazí se zdrojový kód CGI skriptu (nikoliv jeho výstup a už vůbec ne interpretovaný HTML dokument).



Statické zobrazení obsahu souboru

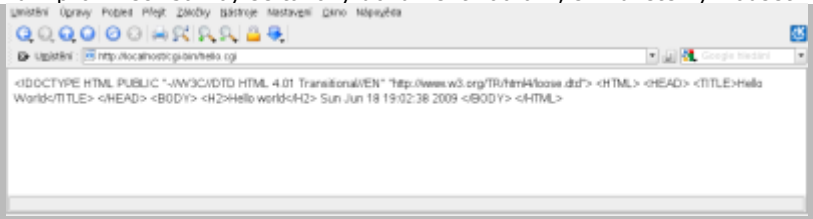
Nyní spustíme CGI skript prostřednictvím webového serveru. Nastartujeme tedy webový server a do prohlížeče zadáme příkaz http://localhost/cgi-bin/hello.cgi. Pokud jde vše, jak má, měli bychom spatřit jednoduchou dynamicky vygenerovanou HTML stránku.



Dynamicky vygenerovaný HTML dokument

Tento CGI program je velmi jednoduchý. Hlavním důvodem je, že od uživatele nepřijímá žádná data a zpracovává pouze informaci serveru o aktuálním čase.

Pokud bychom zadali do hlavičky Content-type místo text/html například text/plain, práce pro server by zůstala stejná. Lišila by se však interpretace výsledku v prohlížeči. Jak by se takový dokument zobrazil, si můžete vyzkoušet nebo se podívat na obrázek.



Dynamicky vygenerovaný textový dokument

Pro představu doplňme, že hlaviček je mnohem více, než jen povinná výše používaná hlavička Content-type. A dokonce oproti předchozímu ani hlavička Content-type nemusí být uvedena. V takovém případě však je nutné uvést hlavičku Location, jejímž parametrem je odkaz na umístění. Ostatní hlavičky jsou volitelné. Zde máme příklad Location hlavičky, která přesměruje na www.linuxsoft.cz.

```
print "Location: http://www.linuxsoft.cz\n\n";
```

Ladění CGI skriptů

Než začneme psát další CGI skripty, je užitečné mít na vědomí, že CGI skript je stále perlový skript a že ho je možné stále spustit v textovém režimu. Zdůrazňujeme to proto, že WWW prohlížeč nám neukáže standardní chybový výstup a v případě chyby bychom mohli být překvapeni, proč se nic neděje.

Absence chybového výstupu je značný hendikep, neboť hledat chybu bez jakýchkoliv hlášení je mnohdy téměř nemožný úkol. Dobrou a jednoduchou metodou pro ladění je spuštění CGI skriptu příkazem perl a sledovat výstup v konzoli. Jinou možností je kontrola log souborů apache. V /usr/local/apache2/logs/error_log bychom měli najít chybový výstup našich CGI skriptů.

Proměnné prostředí

Webový server obvykle nastavuje pro CGI skript některé speciální proměnné prostředí. Odtud můžeme čerpat zejména informace o serveru, ale jsou zde i položky, které charakterizují požadavek uživatele. Proměnné prostředí nám jsou v CGI programu přístupné z hashe %ENV. Tyto proměnné prostředí uvádí následující tabulka.

Je nutno podotknout, že záleží pouze na serveru, které proměnné budou dostupné. Také se mohou názvy proměnných v detailech lišit. Informace o serveru obvykle začínají prefixem SERVER_. Data o uživateli, který zaslal požadavek, jsou zase dostupná přes REMOTE_.

Proměnná	Hodnota
REQUEST_METHOD	metoda vstupu dat
QUERY_STRING	řetězec vstupních dat předaných metodou GET
CONTENT_TYPE	typ dat od uživatele
CONTENT_LENGTH	maximální délka vstupu pro metodu POST
GATEWAY_INTERFACE	verze CGI ve formátu CGI/*verze*
AUTH_TYPE	způsob autentifikace uživatele
REQUEST_URI	URL bez doménového jména
SCRIPT_FILENAME	cesta ke skriptu od adresáře /
SCRIPT_NAME	cesta ke skriptu z pohledu serveru
REMOTE_ADDR	IP ppočítače uživatele
REMOTE_HOST	doménové jméno počítače uživatele
REMOTE_PORT	port uživatele
SERVER_ADDR	IP serveru
SERVER_ADMIN	kontakt na administrátora serveru
SERVER_NAME	doménové jméno serveru
SERVER_PORT	port serveru
SERVER_PROTOCOL	protokol a verze
SERVER_SOFTWARE	jméno a verze serveru a modulů

Nastavují se také proměnné podle přijatých hlaviček. Jsou tak dostupné například HTTP_ACCEPT, HTTP_ACCEPT_CHARSET, HTTP_ACCEPT_ENCODING, HTTP_ACCEPT_LANGUAGE, HTTP_CONNECTION, HTTP_COOKIE, HTTP_HOST, HTTP_USER_AGENT.

Pro konkrétnější představu o tom, jakých hodnot jednotlivé proměnné nabývají si předveďme jednoduchý CGI program, který vypíše obsahy proměnných prostředí.

```
#!/usr/bin/perl
print "Content-type: text/plain\n\n";
foreach $p (sort keys %ENV) {
$h = $ENV{$p};
print "$p=$h\n";
}
```

CGI a bezpečnost

Při psaní webových aplikací si je dobré uvědomit, že každá neošetřená hodnota, která přijde od uživatele, je nakažená. Vždy předtím, než začneme pracovat s přijatými daty, je musíme vyléčit. Aby bylo ošetřování vynuceno, doporučuje se skripty spouštět s parametrem -T.

O nakaženém režimu již v rámci seriálu vyšel zvláštní díl.

Perl (83) - CGI - získávání dat od uživatele

Konečně se dostáváme ke kapitole, která popisuje získávání dat od klienta. Vstup je jedno z důležitých témat webového programování.

Co to vstup dat do CGI skriptu vlastně je? Nejprve uveďme, jakou má vstup formu. Můžeme říct, že to je datová struktura podobná hashi. Uživatel pošle serveru dvojice klíč - hodnota. Tyto informace jsou pak s předány CGI skriptu během volání.

Jsou dvě metody, jak může uživatel předávat serveru vstup. Buď pomocí formulářů (metoda POST) nebo je může zakomponovat do URL adresy požadovaného dokumentu. Oběma se budeme postupně zabývat.

Možná je vždy pouze jedna metoda vstupu. Nelze tedy zároveň přijímat zároveň jak formulářová data, tak data z URL. Abychom mohli ve sporných případech určit, který vstup je aktuální, získá CGI skript od serveru proměnnou prostředí REQUEST_HETHOD, ke které v Perlu přistupujeme pomocí $ENV{"REQUEST_METHOD"}. Metodu vstupu indikuje obsah této proměnné, což je buď řetězec POST nebo GET.

Vstup z formulářů

Data získaná z formulářů čteme ze standardního vstupu. Konkrétně, máme-li například formulář se dvěma položkami jmenoa heslo, přečteme ze standardního vstupu řetězec, do kterého budou zakódované položky jmeno a heslo i s hodnotami, které uživatel do formuláře zadal.

Formát vstupního řetězce má obecně tvar klíč1=hodnota1&klíč2=hodnota2&...&klíč*n*=hodnota*n*. To znamená, že pro situaci s položkami jmeno a heslo by vypadal získaný řetězec takto: jmeno=jiri&heslo=m0je_hEslo.

Pokud má nějaký parametr vstupního řetězce jako hodnotu seznam, je pro všechny hodnoty tohoto seznamu vytvořena dvojice klíč-hodnota se stejným klíčem. Příkladem může být řetězec check=a&check=b&check=c.

Vstup z URL

Druhou zmiňovanou možností vstupu je získání řetězce zakomponovaného do URL adresy dokumentu. Předání metodou GET spočívá v tom, že za umístění dokumentu zadáme otazník a dále dvojice ve formátu klíč=hodnota oddělené znakem & (ampérsand).

To znamená, že pokud zadáme do WWW prohlížeče jako umístění adresu http://localhost/cgi-bin/skript.cgi?jmeno=jiri&heslo=m0je_hEslo (mimochodem, nešifrované heslo v příkazovém řádku prohlížeče není příliš bezpečné), dostane CGI skript parametry jmeno a heslo metodou GET.

Získaný řetězec parametrů - tedy vše, co je v URL za otazníkem - se nám objeví v proměnné prostředí QUERY_STRING. V CGI skriptu tak můžeme k těmto datům přistupovat z proměnné $ENV{"QUERY_STRING"}.

Dále již postupujeme stejně jako u formulářového vstupu.

Zpracování získaných řetězců

Teď už by neměl být problém pomocí nám dobře známých nástrojů Perlu získat z řetězce hash parametrů. Máme zde dvě drobně překážky.

1. Rozdělením řetězců spojených znakem & získáme řetězce ve formátu klíč=hodnota. Tyto řetězce na pozici rovnítka opět rozdělíme a získáme tak hash parametrů.
2. Aby bylo možné přenášet v URL i písmena s háčky a čárkami a také mezery, probíhá již u klienta (ve WWW prohlížeči) konverze těchto znaků. Zkusíme-li zadat do formuláře písmena s háčky, CGI skript je dostane zakódované. Místo slova zpět tak dostaneme zp%ECt. Musíme tak ještě před definitivním uložením do hashe nahradit sekvenci %.. za správný znak. Mezery jsou nahrazovány znakem +.

```
my %hash;
for (split "&", $retezec){
my($k, $h) = split "=";
$h =~ tr/+/ /;
$h =~ s/%(..)/chr hex $1/ge;
$hash{$k} = $h;
}
```

Nyní máme v proměnné %hash požadované páry parametr-hodnota, se kterými můžeme pohodlně pracovat.

Poznámka - Úplně jsme vynechali parametry se seznamovou hodnotou. Prozatím se jim nebudeme věnovat.

Konkrétní příklad obsluhy formulářů

Jedno z nejčastějších užití CGI je zpracovávání dat, které uživatel zadal do formuláře. Na základě těchto dat mohou být například přidány záznamy do databáze nebo můžeme naopak z databáze požadovaná data získat.

Již víme, jak získat formulářová data. Máme tedy všechny potřebné znalosti k tomu, abychom si uvedli první ukázku programu, jež bude vstupu od uživatele využívat.

Vytvoříme si přihlašovací skript, který na se základě do formuláře zadaného uživatelského jména a hesla otáže databáze, zda takový uživatel se zadaným heslem existuje. Pokud zjistí, že tento uživatel skutečně existuje, zobrazíme z databáze zprávu. V opačném případě přesměrujeme na stránku, která vygeneruje chybu.

Ze všeho nejdříve bude třeba navrhnout datovou základnu. Otevřeme tedy řádkového databázového klienta, vytvoříme databázi s názvem projekt a zde umístíme tabulku users s následující strukturou. vložíme také jeden testovací záznam.

```
create table users (
id int not null primary key auto_increment,
username varchar(60) not null,
md5passwd varchar(32) not null,
zprava varchar(255) not null
);
insert into users (user, md5passwd, zprava) values ('pokus', 'fde27fdce626407e9ad040b7bb017882', 'TAJNA ZPRAVA');
```

Toto je zdrojový text pro databázový systém MySQL. Chcete-li použít například Postgres, pak místo druhého řádku napište id serial primary key,.

Další poznámka bude směřovat k uložení hesla. Do databáze neukládáme čisté heslo (pokus), ale jeho md5-hashovanou verzi. Použití hashovacího algoritmu nám zaručí, že i kdyby se potenciální útočník dostal do databáze, získá pouze data, ale nikoliv hesla. My heslo nepotřebujeme, protože budeme porovnávat pouze výsledek funkce md5.

Je třeba dodat, v roce 2004 byly nalezeny první kolize algoritmu md5 (to jest více řetězců se stejným hashem) a nyní je již možné provádět inverzní operaci k hashování na běžném počítači do několika sekund. Pro bezpečnější uložení hesel lze použít některý z rodiny algoritmů SHA-2, u kterých ještě žádná kolize objevena nebyla.

Nyní musíme vytvořit dva soubory. Prvním z nich bude obyčejný HTML formulář, který bude data odesílat CGI skriptu, což bude druhý soubor.

Soubor s formulářem je čisté HTML bez jakýchkoliv dynamických prvků. Dále budeme pracovat s níže uvedeným formulářem. Uložíme ho například jako formular.html do kořenového adresáře Apache. Fyzicky tedy bude ležet v /usr/local/apache2/htdocs a do prohlížeče budeme zadávat http://localhost/formular.html.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Log in</title>
</head>
<body>

<form action="/cgi-bin/login.cgi" method='POST'>
Uživatelské jméno: <input type='text' name='username'><br>
Heslo: <input type='text' name='heslo'><br>
```

```
<input type='submit' name='akce' value='Log in'>
</form>

</body>
</html>
```

Dále vytvoříme samotný CGI skript, který bude formulář obsluhovat. Jak vidíme z výše uvedeného HTML kódu, jako umístění CGI skriptu jsme zvolili /usr/local/apache2/cgi-bin/login.cgi.

Úvod skriptu login.cgi je obvyklý. Protože již víme, že budeme používat databázi, importujeme hned i modul DBI. Dále vytvoříme hash parametrů a budeme načítat data.

```
#!/usr/bin/perl
use strict;
use DBI;

#do proměnné %hash uložíme formulářová data
my %hash;
for (split "&", <STDIN>){
my($k, $h) = split "=";
$h =~ tr/+/ /;
$h =~ s/%(..)/chr hex $1/ge;
$hash{$k} = $h;
}
```

Nyní se musíme dotázat databáze na správnost uživatelem zadaných údajů. Podle tohoto kritéria se bude odvíjet tělo generovaného dokumentu. Musíme ošetřit případy selhání pokusu o připojení k databázi a nekorektní zadání údajů. Podle toho, kterou větví se bude program ubírat, uložíme do proměnné $telo tělo dokumentu. Na závěr celý dokument vytiskneme.

Nejdříve se tedy musíme pokusit navázat spojení s databází. Je-li toto spojení neúspěšné, přesměrujeme na CGI skript k tomu určený.

```
my $telo;
my $dbh = DBI->connect("dbi:mysql:dbname=projekt", "user", "");
if(!$dbh){
print "Location: error.cgi?cislo_chyby=1\n\n";
}else{
#sem se program dostane, bude-li navázání spojení k databázi úspěšné
}
$dbh->disconnect;
```

Nyní již předpokládáme, že jsme se úspěšně spojili s databází. Dále se musíme zamyslet nad dotazem, který jí položíme. Potřebujeme zjistit hashované heslo a zprávu. Budeme předpokládat, že existuje vždy jeden uživatel stejného jména nehledě na velikost písmen.

```
my $q = $dbh->prepare("SELECT md5passwd, zprava FROM users WHERE UPPER(username)=UPPER(?)");
$q->execute($hash{"username"});
```

Aktuální otázka teď zní, zda vůbec zadaný uživatel existuje. To zjistíme podle toho, jestli nám databáze na dotaz pošle nějaké výsledky. Pokud ne, uživatel zadal neplatné uživatelské jméno.

```
if(my($md5passwd, $zprava) = $q->fetchrow_array){
#sem se program dostane, pokud zadané uživatelské jméno je v databázi
}else{
$telo = "<h1>Uživatel neexistuje<h1>"
}
```

Zbývá ná zkontrolovat poslední věc, kterou je heslo. Tady nastává drobný problém, nebo□ v databázi máme heslo hashované pomocí md5 a heslo zadané je nešifrované. Jak je porovnat?

V podstatě jedinou možností, kterou máme, je zašifrovat zadané heslo a porovnat vzniklé md5 hashe. Perl má tu výhodu, že existuje archiv modulů CPAN, kde si snadno dohledáme modul Digest::MD5. Pro hexadecimální podobu md5 šifry si do CGI skriptu importujeme funkci md5_hex. Kdyby někdo chtěl použít zmíněný bezpečnější algoritmus, použije modul Digest::SHA2.

```
use Digest::MD5 qw(md5_hex);
```

Teď již není nic jednoduššího, než porovnat oba md5 hashe a v případě shody uložit zprávu o úspěchu nebo v opačném případě o neúspěchu.

```
if(md5_hex($hash{"heslo"}) eq $md5passwd){
$telo = "<h1>Úspěch!<h1>\n$zprava";
}else{
$telo = "<h1>Nesedí heslo<h1>";
}
```

Na závěr vytiskneme HTML dokument.

```
print << "HTML";
Content-type: text/html; Charset="iso-8859-2"

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>Výsledek</TITLE>
</HEAD>
<BODY>
<font color="#800000">$telo</font>
</BODY>
</HTML>
HTML
```

Je dobré si uvědomit, jak je to s ošetřováním nakažených dat. Zde k problémům nedošlo.

Pokud si chcete tuto miniaplikaci také vyzkoušet, stahujte zde.

Příklad pro vstup metdodou GET

Abychom si mohli ukázat v praxi, jak pracovat s metodou GET, nechali jsme si v předchozí ukázce CGI skriptu přesměrování v případě chyby databáze. Je-li databáze nedostupná, volá se error.cgi?cislo_chyby=1. Zde je ukázka poněkud jednoduché, leč názorné implementace skriptu error.cgi.

```
#!/usr/bin/perl
use strict;

my $cislo_chyby = $ENV{"QUERY_STRING"} =~ /cislo_chyby=(\d*)/;
my $chyba;

if   ($cislo_chyby == 1){$chyba = "Chyba při práci s databází";}
   elsif($cislo_chyby == 2){$chyba = "Jiná chyba";}
  else{                    $chyba = "Neidentifikovaná chyba";}

print << "HTML";
Content-type: text/html; Charset="iso-8859-2"

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>Výsledek</TITLE>
</HEAD>
<BODY>
<h1>$chyba</h1>
</BODY>
</HTML>
HTML
```

Perl (84) - CGI - usnadnění tvorby skriptů pomocí modulu CGI

Perl má k dispozici několik modulů, které poskytují prostředky pro usnadnění často používaných operací v CGI skriptech. Nejznámějším z této kategorie je modul CGI.

CGI modul v mnohém zjednodušuje programátorovi práci. Zejména pomáhá se zpracováváním parametrů, zpřístupňuje proměnné prostředí jako metody, usnadňuje práci s cookies. Dále možňuje zapisovat HTML elementy pomocí volání funkcí. Tyto funkce vracejí kód formátovaný do jazyka HTML, který pak můžeme například tisknout funkcí print.

Modul CGI poskytuje jak funkcionální tak objektově-orientované rozhraní. Záleží na nás, co je nám bližší. Abychom se s modulem CGI seznámili blíže, přepíšeme si příklad z 82. dílu s použitím obou těchto stylů zvlášť.

Funkcionálně-orientované rozhraní modulu CGI

U funkcionálního použití modulu voláme ke generování HTML, získávání parametrů, práci s cookies apod. k tomu určené funkce. Tyto funkce potřebujeme importovat do našeho jmenného prostoru.

Modul CGI obsahuje několik kategorií těchto funkcí. Jsou popsány v následující tabulce.

Skupina	Význam
:all	importují se všechny funkce, které modul CGI obsahuje
:standard	importuje standardní vlastnosti, :html2, :html3, :html4, :form a :cgi
:cgi	import všech obsluhujících metod jako například param, cookie
:html, :html2, :html3, :html4, :form	import příslušných HTML elementů

Toto sice nejsou veškeré dostupné třídy funkcí, nicméně ty ostatní jsou pro naše účely nevýznamné. Pro toho, kdo se chce možnostmi webového programování zabývat více do hloubky, může být v tomto směru poučné nahlédnout do zdrojového kódu modulu CGI, kde najdeme všechny třídy i se všemi funkcemi, které do nich patří.

Ve valné většině případů si vystačíme s importem kategorie :standard.

Přistupme nyní již ke konkrétní ukázce použití funkcionálního rozhraní modulu CGI.

```
#!/usr/bin/perl -T
use strict;
use CGI qw(:standard);

my $now = localtime;
print header("text/html");
print start_html("Hello World");
print h2("Hello World");
print $now;
print end_html;
```

HTML prohlížeč zobrazí to samé, co zobrazil náš první příklad v minulém dílu. Kód je však o poznání kratší a o něco jednodušší.

Pro lepší srovnání uveďme, jak vypadal náš původní kód.

```
my $now = localtime;
print << "HTML";
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<H2>Hello world</H2>
$now
</BODY>
```

</HTML>
HTML
Funkce header vrací povinnou hlavičku, ve které je MIME typ dokumentu. Další funkce start_html a end_htmlvytvářejí hlavu a patu HTML dokumentu. Importují příslušný DTD soubor a nastavují titulek stránky. Mezi nimi je tělo dokumentu.
Objektově-orientované rozhraní modulu CGI
Použití této části modulu je o něco náročnější na psaní kódu. Musíme totiž vytvořit objekt a následně nad ním volat metody namísto volání obyčejných funkcí, jako je tomu u funkcionálního programování. Avšak na druhou stranu nemusíme vybírat kategorie funkcí, které chceme importovat.
Ukázka kódu, která dělá pro srovnání opět totéž, je zde.

```
#!/usr/bin/perl -T
use strict;
use CGI;

my $now = localtime;

my $cgi = new CGI;
print $cgi->header("text/html");
print $cgi->start_html("Hello World");
print $cgi->h2("Hello World");
print $now;
print $cgi->end_html;
```

Vstup od uživatele a funkce param
Zdlouhavost získávání parametrů ze vstupu přímo vybízela k vytvoření funkce, která by tuto činnost zjednodušila. Jednou z nejdůležitějších úloh modulu CGI je právě obsluha vstupu dat.
Funkce param dělá dvě věci. Tou první je, že, je-li volána bez argumentu, vrací seznam parametrů vstupu bez hodnot. Tou druhou je vracení samotných hodnot. Abychom získali hodnotu, je třeba funkci předat navíc i název parametru. Hodnota může být pro určité typy fomulářů i seznam - potom je vracen seznam těchto hodnot.
Následující CGI skript zobrazí všechny parametry i s hodnotami, nehledě na typ vstupu. Funkce param řeší i parametry s hodnotou v seznamovém kontextu, takže je třeba ošetřit zobrazení seznamu.

```
#!/usr/bin/perl
use strict;
use CGI qw(:standard);

print "Content-type: text/plain\n\n";
for (param){
print "$_: ", join(", ", param($_)), "\n";
}
```

Voláním param s více parametry má za následek přidání parametru. První parametr se stane klíčem, další parametry jsou jeho hodnotami. Funkce append vybranému parametru přidá hodnotu. Funkce Delete (velké D proto, protože delete již existuje v základní distribuci Perlu) (resp. Delete_all) maže příslušné parametry.
Pro hledání chyb lze využít funkci Dump, která vrací seznam parametrů s hodnotami formátovaný v HTML.
Potřebujeme-li všechny přijaté parametry importovat do vybraného balíku, použijeme funkci import_names. Její užití vypadá následovně.

```
import_names("P");
print $P::jmenoparametru;
```

Struktura volání funkcí
Funkce nebo metody (dále pouze funkce) mohou obvykle přijímat žádný, jeden nebo více argumentů. Tyto argumenty jsou přijímany většinou ve formě hashe. Hashové klíče začínají vždy pomlčkou. Například funkci header můžeme volat tímto způsobem.

```
header(-type => "text/html", -charset=>"utf-8");
```

Nicméně právě header je možné volat i skalárně, jak je vidět na příkladu výše.
Funkce, kterým není předán žádný parametr vytvoří jednorázový HTML tag ve tvaru <element />. Příkladem takové funkce je hr.
V průběhu seriálu o CGI se však několikrát setkáme i s funkcemi, které mohou jako argument přijímat složitější datové struktury.
Prvním takovým příkladem mohou být HTML elementy, které přijímají nějaký parametr. Například chceme-li získat pomocí funkce h1 kód <h1 align="center">Hello World</h1>, předáme jako první argument odkaz na hash s příslušnými hodnotami.
Volání takové funkce tedy bude vypadat následovně.

```
h1({-align => "center"}, "Hello World");
```

Analogické je to i s ostatními funkcemi, které vytváří prvky HTML dokumentu.
Struktura volání funkcí, kterou zatím známe, opomíjí jednu velmi důležitou skutečnost. Neřeší vnořování elementů, což je i v jednoduchých HTML dokumentech nezbytné. Pro tento účel existují speciální funkce ve tvaru start_*element* a end_*element*, které generují pouze příslušnou polovinu elementu.
Tyto funkce nejsou importovány ani třídou :standard, ani třídou :all, ani žádnou jinou, takže je musíme zavést zvlášť. Připomeňme, že zde můžeme pro import funkcí start_*element* a end_*element* užít žolíkový znakhvězdička. Například pro import funkcí start_h1 a end_h1 dohromady můžeme napsat *h1.

```
use CGI qw(*h1);
```

Vnoření pak může vypadat třeba takto.

```
use CGI qw(*h1 i);
print start_h1, "Hello ", i("Perl"), end_h1;
```

*Poznámka - pokud importujete alespoň jednu z funkcí start_*element *a end_*element*, importují se automaticky obě.*
Perl (85) - CGI - generování dokumentu modulem CGI

Dnes postupně projdeme nejdůležitější funkce, jež nabízí modul CGI a podrobněji si rozebereme jejich možné parametry.

Již víme, jak pomocí modulu CGI vytvořit jednoduchý HTML dokument. Uvedli jsme si základní použití fukcí, které si dnes rozebereme důkladněji.

Hlavička dokumentu

Hlavička rozhoduje o tom, jak se bude dále ubírat generování dokumentu. Buď se vygeneruje dokument, nebo, v případě přesměrování, se zavolá nějaká jiná stránka.

Poznámka - Obyčejně prohlížeč hlavičky nezobrazuje. Pokud bychom je chtěli kontrolovat, lze spouštět skript s textovým výstupem z příkazového řádku nebo využít například lynx.

```
$ lynx -mime_header http://localhost/cgi-bin/skript.cgi
```

Určení typu dokumentu

Hlavička standardně obsahuje několik informací, z nichž nejdůležitější je datový typ dokumentu. Často se zde setkáme s informacemi o nastavení jazyka, kódování, datu vypršení platnosti stránky apod.

Nejjednodušší volání funkce header je bez parametrů. V takovém případě se do výsledné stránky vygeneruje řetězec Content-Type: text/html.

header však navíc volitelně přijímá hash s určenou strukturou. Ta je dána seznamem následujících klíčů.

Hlavička	Význam
-type	MIME typ generovaného dokumentu
-status	stavový kód vrácený webovým serverem
-charset	kódování
-attachment	je-li jako -type application/octet-stream, stránka bude otevřena jako příloha s daným názvem
-expires	datum a čas nebo změna oproti okamžiku, kdy dokument ztratí platnost
-target	podpora rámců
-cookie	podpora cookies (jim bude speciálně věnován příští díl)

Pomocí header je ale možné vytvořit libovolnou hlavičku. I přesto, že uvedeme klíč, jejž funkce headernerozpoznává, se vytvoří hlavička. Jejím jménem je automaticky klíč (bez úvodní pomlčky a s nahrazenými podtržítky) a hodnotou předaná hodnota prvku.

Přesměrování

Pokud nechceme generovat dokument, ale pouze přesměrovat jinam, použijeme funkci redirect, která funguje podobně jako header. Předáváme ji hash s hodnotami uvedenými v tabulce.

Hlavička	Význam
-location	celá URL adresa stránky, kam se bude přesměrovávat
-status	stavový kód vrácený webovým serverem (měl by mít standardně formát 3XX, což znamená "přesměrování požadavku")
-cookie	podpora cookies (jim bude speciálně věnován příští díl)

Hlavička HTML

Zatímco hlavičky dokumentu v předcházejícím oddílu mohly platit obecně, dále budeme předpokládat, že naše CGI skripty generují HTML dokumenty.

HTML potřebuje také svoji vlastní hlavičku. Ta obsahuje vše od určení DTD souboru pomocí <!DOCTYPE> až po element <BODY>. Dále následuje obsah dokumentu a na závěr je třeba vytisknout HTML patu, která zahrnuje elementy </BODY> a <HTML>.

Hlavičku generuje na základě předaných parametrů funkce start_html. Parametrem funkce start_html je hash, jehož rozpoznávané prvky jsou uvedené v tabulce.

Klíč	Význam
-title	titulek stránky
-encoding	kódování, implicitně je nastaveno ISO-8859-1
-author	kontakt na autora
-base	určuje adresář, vzhledem ke kterému se berou relativní odkazy
-dtd	určení DTD souboru
-lang	jazyk
-script	případné Javascript soubory
-noscript	obsah elementu NOSCRIPT
-meta	případné další meta řádky, přidávají se parametry name a content
-head	případný další obsah HTML hlavičky
-style	umístění kaskádového stylu

Uvedeme-li prvek, který není funkcí start_html rozpoznán, je z něj vytvořen parametr elementu <BODY>.

Jako příklad HTML hlavičky si uveďme následující kód.

```
my $meta = {"keywords" => "linux software", "description" => "archiv softwaru pro linux"};
print start_html(
-title=>"Linux Software",
-author=>'autor@server.cz',
-lang=>"cs",
-encoding=>"ISO-8859-2",
-meta=>$meta,
-style=>"/styles/style.css",
-bgcolor=>"green"
```

```
);
```
Ten vytvoří HTML hlavičku tak, jak ji vidíme v dalším výpisu.
```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="cs" xml:lang="cs">
<head>
<title>Homepage</title>
<link rev="made" href="mailto:autor%40server.cz" />
<meta name="keywords" content="linux software" />
<meta name="description" content="archiv softwaru pro linux" />
<link rel="stylesheet" type="text/css" href="/styles/style.css" />
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-2" />
</head>
<body bgcolor="green">
```
HTML obsah

Modul CGI obsahuje značné množství funkcí. Bylo by zbytečné zde uvádět kompletní přehled, neboť pro většinu elementů, které budeme potřebovat, s největší pravděpodobností platí, že jsou implementovány jako funkce se stejnými názvy jako elementy. Později si představíme pouze práci s formuláři, neboť ta je pro náš záměr klíčová. V tomto okamžiku si ukážeme pouze stručný úsek textu ze zdrojového kódu.
```
print h1("Vítejte");
print hr;
print start_p, "můžete mi napsat ";
print a({-href=>'mailto:muj@mail.cz'}, "email");
print end_p;
```
K němu snad jediný komentář. Výsledkem je řetězec na jediném řádku. To není u jazyka HTML podstatné, protože nebere bílá místa navíc v potaz. Máte-li zájem o přehledný výsledný HTML kód, je třeba na místech zalomení používat \n. Znak \n se promítne jako nový řádek do zdrojového kódu, ale nezmění výslednou podobu HTML dokumentu. Pro nový řádek v HTML se použije funkce br.

Autoloading

Pokud by vám zde i přes velké množství nějaký element scházel (například tehdy, máte-li svůj vlastní DTD soubor), je zde možnost autoloadingu. Lze tak volat metodu s prakticky libovolným názvem.

K tomu stačí zavést do programu modul s přepínačem -autoload.
```
use CGI qw(-autoload);
```
Pokud použijeme autoloading, pak musí mít každá funkce závorky i v případě, že jí nepředáváme žádný parametr.

Generování formulářů

Formuláře jsou základním HTML prvkem pro uživatelský vstup a následně tedy kritériem dynamičnosti. Z tohoto důvodu jim nyní budeme věnovat větší prostor.

HTML podporuje několik formulářových prvků, kterými uživatel ovlivňuje vstup. Jsou to tyto prvky.

- skrytý text
- jednořádkový text
- víceřádkový text
- heslo
- soubor
- zaškrtávací seznam
- přepínací seznam
- rozbalovací seznam
- tlačítka

Pro všechny tyto prvky existují v modulu CGI příslušné funkce, které je generují.

Nicméně používat CGI pro generování formuláře, pokud se na základě nějakých vstupních dat jeho struktura nemění, není teoreticky vůbec nutné. Nutné je CGI pouze pro zpracování získaných hodnot. Chcete-li formuláře vytvářet staticky, můžete klidně následující část přeskočit.

Všechny prvky jednoho formuláře jsou uvnitř elementu FORM. Tento element určuje mimo jiné metodu odesílání dat a umístění, kam se budou data spolu s požadavkem posílat. Základní struktura každého formuláře tak bude vypadat takto.
```
use CGI qw(*form);
print start_form(-method=>"post", -action=>"/cgi-bin/form.cgi");
#...
print end_form;
```
Uvnitř budou jednotlivé prvky formuláře, které jsou vytvářeny konkrétními funkcemi. Každá z nich přijímá jako hodnotu minimálně jméno a implicitní hodnotu - tedy klíče -name (který je jako jediný vždy povinný) a -value v předávaném hashi. Právě podle -name každý formulářový prvek vrací hodnotu zadanou uživatelem. Pokud jako -name použijeme slovo password, pak funkce param na základě identifikátoru password vrátí heslo zadané uživatelem.

Tlačítko

V HTML je několik typů tlačítek. My se budeme zabývat pouze tím nejdůležitějším - tlačítkem pro odeslání dat. K jeho vytvoření slouží funkce submit.
```
print submit(-name=>"akce", -value=>"Odešli data!");
```
Zadání textu a hesla

Funkce textfield resp. password_field vytvářejí prvky pro zadání jednořádkového textu resp. hesla. Dále je k dispozici funkce textarea pro textové pole.
```
print textfield(-name=>"url, -value=>"http://, -size=>50);
```
Seznamy

Funkce popup_menu vytváří rozbalovací seznam.
```
print popup_menu(-name=>"list, -value=>["textfield","password_field","textarea"], -default=>"textarea");
```

Zaškrtávací pole

Máme 2 funkce, které vytváří zaškrtávací pole. Pro skupinu polí se stejným jménem zde je checkbox_group a samostatné pole použijeme checkbox.

Zde je ukázka jednoduchého checkboxu. Je-li položka zaškrtnuta, automaticky se vrátí hodnota on.

```
print checkbox(-name=>"checkbox_name", -checked=>1, -label=>"I agre");
```

Skupinu zaškrtávacích políček se stejným jménem lze formátovat do daného počtu sloupců nebo řádků. Pomocí -colheaders nebo -rowheaders lze navíc dát jednotlivým sloupcům titulek.

```
print checkbox_group(-name=>"mesto", -values=>[qw(Praha Ostrava Brno Plzeň Liberec)], -columns=>2);
```

Přepínač

Přepínací pole se vytváří pomocí funkce radio_group. Funguje podobně jako skupina checkboxů s tím rozdílem, že nelze označit libovolný počet polí. Syntaxe je také totožná.

Skrytá hodnota

Velice často užívaný prvek hidden slouží pro předávání nějaké proměnné hodnoty, kterou však nezadává uživatel. Jeho typickým užitím je například uchování id uživatele, který mění údaje ve svém profilu. Sám uživatel o id nic neví a přesto bychom bez něj nevěděli, koho údaje změnit.

```
print hidden(-name=>"user_id", -default=>215);
```

Upload souboru

Na závěr uvedeme, jak lze získávat od uživatele soubory. K tomu je určena funkce filefield, jejíž volání může vypadat například takto.

```
print filefield(-name=>"jméno_souboru");
```

Zajímavější je obsluha takového formuláře. Na to použijeme funkci upload a získanou proměnnou dále používáme jako ovladač. Tento kód vytiskne obsah textového souboru na výstup.

```
$fh = upload("jméno_souboru");
while (<$fh>) {
print;
}
```

Perl (86) - CGI - cookies

Cookies je mechanizmus, který umožňuje ukládat data na počítači klienta. Typickými úlohami pro cookies je počítání počtu návštěv, rozlišování uživatelů, pamatování obsahu formulářů nebo nákupní košíky v internetových obchodech.

HTTP Cookie je nějaký textový soubor, který je součástí odpovědi webového serveru. Tato data si prohlížeč automaticky uloží. Jakmile někdy v budoucnu klient posílá tomuto webovému serveru další požadavek, pošle mu spolu s ním i uložená cookies. Vzhledem k tomu, že cookies jsou součástí hlavičky HTTP požadavku, si je představíme zde.

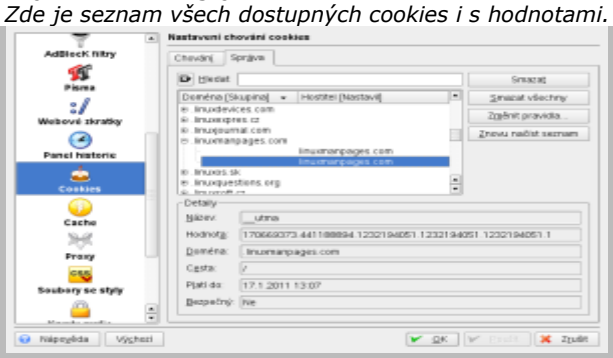
To, že cookies jsou uloženy na straně klienta má pro autora webových stránek kladné i záporné vlastnosti. Cookies nám umožňují ukládat a zpětně využívat data při opakovaných návštěvách z téhož počítače, což může do jisté míry automatizovat činnost klienta. Toho se využívá konkrétně například při vyplňování formulářů, při ukládání nastavení apod. Mezi negativní důsledky patří to, že se nelze na nic spolehnout. Klient si může s cookies volně manipulovat - může je mazat nebo libovolně upravovat.

Struktura cookie

Cookie je fyzicky šestice datových položek. Jsou to následující.

1. Název cookie - z jednoho serveru můžeme uchovávat několik cookies, které se budou lišit v názvu.
2. Hodnota
3. Doména - jméno webového hostitele, ze kterého sem byla cookie uložena
4. Cesta - určuje, na který adresář domény budou cookies odesílány
5. Čas vypršení - čas, kdy cookie pozbyde platnosti
6. Secure - klient neodešle cookie, nebude-li to bezpečné

Poznámka - Pro názornější představu, jak cookies fungují, můžeme nahlédnout do svého www prohlížeče na nastavení cookies. Zde je seznam všech dostupných cookies i s hodnotami.



Náhled na cookies v prohlížeči

Ukládání cookies a modul CGI

Jednou z HTTP hlaviček je i Set-Cookie. Tu nastavujeme klíčem -cookie, předanému funkci header, jež generuje hlavičky. Hodnotou Set-Cookie je řetězec formátovaný tak, aby obsáhl všechny datové položky cookies. Pro zformátování zde máme funkci cookie, která je součástí importované třídy :cgi a tedy i :standard.

Funkce cookie přijímá jako parametr opět hash, který může obsahovat prvky -name, -value, -domain, -path, -expires a -secure. Hodnotami těchto prvků jsou příslušné výše uvedené datové položky.

Hodnotou položky -expires může být buď datum ve formátu DD-MMM-YYYY hh:mm:ss, například 21-Feb-2009 09:11:55, nebo now pro platnost do konce relace. Dále je možné užít hodnotu ve formátu +<číslo><jednotka>. Taková cookie vyprší za daný počet jednotek. Dostupné jednotky jsou v tabulce.

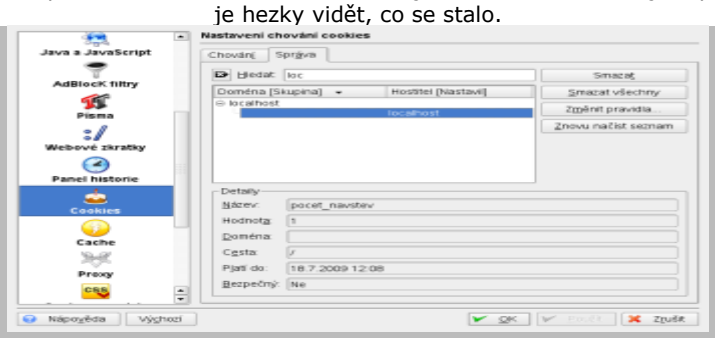
Jednotka	Význam
s	sekunda

m	minuta
h	hodina
d	den
M	měsíc
y	rok

Uveďme si příklad uložení cookie. Nejprve vytvoříme cookie stejnojmenou funkcí a následně odešleme hlavičky.

```
#!/usr/bin/perl
use strict;
use CGI qw(:standard);

my $cookie = cookie(
-name=>"pocet_navstev",
-value=>1,
-expires=>"+10d",
);
print header(-type=>"text/plain", -cookie=>$cookie);

#následuje samotný obsah dokumentu
```

Nyní se podívejte ve vašem oblíbeném www prohlížeči na správu cookies. Vytvořila se zde nová položka. Na adrese localhost máme proměnnou pocet_navstev s hodnotou 1. Zde je screenshot zobrazující správu cookies v Konqueroru, kde je hezky vidět, co se stalo.



Naše první cookie

Je-li zapotřebí uložit více než jedinou cookie, lze jako argument prvku -cookiepředat odkaz na seznam.

```
print header(-cookie=>[$cookie1,$cookie2]);
```

Získávání uložených cookies a modul CGI

Voláním funkce cookie s jménem cookie jako parametrem získáme nazpět hodnotu.

```
$cookie = cookie("cookie");
```

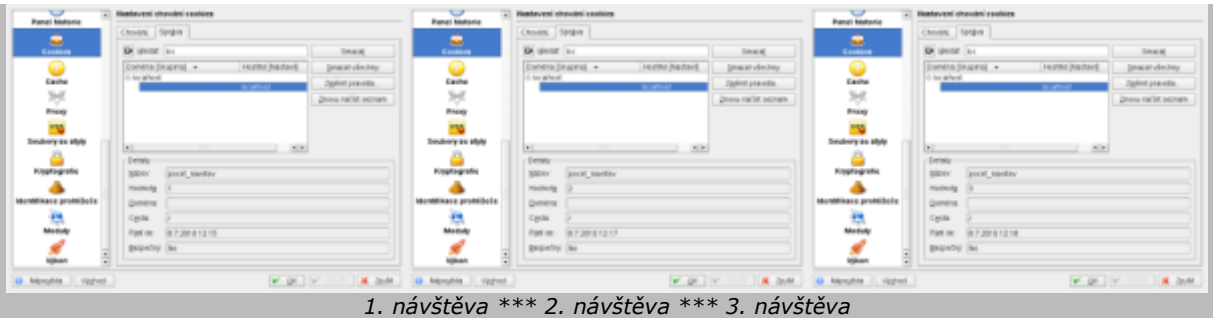
V seznamovém kontextu vrací funkce cookie seznam jmen všech cookies.

Příklad - počítadlo návštěv

Na závěr si napíšeme jednoduché počítadlo návštěv. Pokaždé, když návštěvník zobrazí naši stránku, dostane informaci, kolikkrát tu byl.

```
#!/usr/bin/perl
use strict;
use CGI qw(:standard);

my $pocet_navstev = cookie("pocet_navstev") or 0;
$pocet_navstev++;

my $cookie = cookie(
-name=>"pocet_navstev",
-value=>$pocet_navstev,
-expires=>"+1y",
);

print header(-cookie=>$cookie);

print start_html();
if($pocet_navstev == 1){
print h1("Vítejte! Jste zde poprvé!");
}else{
print h1("Jste tu již po $pocet_navstev.");
}
print end_html();
```

Nyní tento skript několikrát spustíme a budeme monitorovat stav cookies. Zde vidíme, jak se hodnota naší cookie mění.



*1. návštěva *** 2. návštěva *** 3. návštěva*

Perl (87) - CGI - příklad aplikace

Na závěr série o CGI napíšeme jednoduchou nástěnku s možností přidávání příspěvků, na čemž shrneme základní poznatky a ukážeme, jak může vypadat CGI aplikace.

Abychom si zrekapitulovali předchozích 6 dílů, napíšeme si v dnešním dílu malou CGI aplikaci. Tato aplikace bude plnit úlohu nástěnky, kam bude možno pomocí formuláře přidávat další položky.
CGI skript tedy bude muset vygenerovat formulář a zobrazit dosud vložené položky. Po odeslání formuláře se stránka aktualizuje.
Ještě dodejme, že každá položka na nástěnce bude obsahovat tři údaje:

- datum a čas vložení
- autora položky
- text položky

Realizace

Datová reprezentace

Začneme jako obvykle způsobem uložení dat. Pro tento účel bude vhodným prostředkem databáze. Vytvoříme tedy tabulku s následující strukturou. Uvedeme příkaz pro databázový systém MySQL.

```
create table nastenka (
id int not null primary key auto_increment,
jmeno varchar(255),
email varchar(255),
text blob,
datetime datetime
);
```

Sloupce jmeno, text a email jsou závislé na uživatelském vstupu. Datum a čas a id položky si bude hlídat sám skript.
Protože používáme databázi, budeme si muset osvěžit práci s modulem DBI, který byl představen v trojdílné minisérii (1, 2, 3).

CGI skript

Nyní budeme řešit už jen samotný CGI skript. Předtím, než začneme psát, si musíme ujasnit několik věcí.

- HTML stránka se bude členěna na začátek, tělo a konec. Začátek a konec stránky přitom bude definován v externím souboru. To je jeden z nejjednodušších způsobů, jak můžeme zajistit jednotnou tvář stránek (pokud bychom jich měli více).
- Než začneme, je také třeba zvolit si strategii odesílání dat klientovi. Lze buď odesílat po částech (postupně generovaný kód budeme okamžitě tisknout na standardní výstup) nebo odeslat vše najednou na konci provádění skriptu (generovaný kód budeme místo tisku ukládat). Obě varianty mají své pro a proti. My v naší aplikaci zvolíme druhou variantu.
  - Použijeme neobjektové rozhraní modulu CGI.

Dále si musíme uvědomit, co vše bude na výsledné HTML stránce zobrazeno. Uděláme následující kroky.

- vygenerovat formulář
- zobrazit položky seřazené podle času vložení
- byl-li formulář odeslán, přidat do databáze položku
- vytvořit obecný HTML začátek a konec stránky

Protože jsme se rozhodli použít strategii odeslání celé stránky najednou, vytvoříme si tři proměnné, které budou obsahovat začátek a konec stránky a samotné tělo HTML dokumentu.
Víme, že začátek a konec stránky nebude v pravomoci našeho CGI skriptu, ale nějakého vnějšího modulu. Proto do něj můžeme ihned přiřadit. Obsah těla budeme do proměnné dodávat postupně.

```
my $zacatek = Funkce::zacatek();
my $konec = Funkce::konec();
my $telo = "";
```

Vytvoření formuláře

První krok zahrnuje pouze vytvoření statického HTML kódu, který vytvoří formulář. Můžeme tedy napsat toto.

```
$telo .= << 'FORMULAR';
<form method="post" action="/cgi-bin/projekt/nastenka.cgi">
Jméno: <input type="text" name="jmeno"  size="30" /><br />
Email: <input type="text" name="email"  size="30" /><br />
<textarea name="text" rows="7" cols="40"></textarea><br />
<input type="submit" name="akce" value="OK" />
</form>
FORMULAR
```

Také však lze použít funkce pro generování formulářů z CGI modulu. Generování HTML kódu tak můžeme s použitím CGI modulu přepsat do následující podoby.

```
$telo .= start_form(-method=>"post");
$telo .=  "Jméno: " . textfield(-name=>"jmeno", -size=>30) . br;
```

```
$telo .=  "Email: " . textfield(-name=>"email", -size=>30) . br;
$telo .=  textarea(-name=>"text", -cols=>40, -rows=>7) . br;
$telo .=  submit(-name=>"akce", -value=>"OK");
$telo .=  end_form;
```

Zobrazení položek nástěnky

Pod formulářem budou zobrazeny jednotlivé položky nástěnky. Dalším naším úkolem bude získat je z databáze a zobrazit.
Nejprve tedy vytvoříme databázové spojení. K tomu musíme znát údaje potřebné ke spojení.

```
my $dbname = "project";
my $dbuser = "user";
my $dbpasswd = "";
my $dbh;

if(!($dbh = DBI->connect("dbi:mysql:dbname=$dbname", $dbuser, $dbpasswd))){
$telo .= h1("Nastal problém.");
}

#zde získáme data a přidáme je do proměnné $telo, případně zpracujeme data

$dbh->disconnect;
```

Získání dat bude poměrně jednoduchá záležitost. Nejprve pošleme na databázi dotaz.

```
my $q = $dbh->prepare("SELECT jmeno, email, text, datetime FROM nastenka ORDER BY datetime DESC");
$q->execute;
```

Následně přijmeme odpověď. Zformátujeme jednotlivé položky do nějaké podoby a postupně všechny přidáme k tělu dokumentu.

```
while(my($jmeno, $email, $text, $datetime) = $q->fetchrow_array){
$telo .=  hr;
$telo .= i($datetime) . br;
$telo .= a({-href=>"mailto:$email"}, $jmeno) . br;
$telo .= $text;
}
```

Zpracování přijatých dat

Nyní nám v základním souboru ještě zbývá zaktivovat formulář. Pokud nyní odešleme data, tak se vůbec nic nestane. Je třeba, abychom v případě obdržení dat tato data uložili.
To, zda nám data přišla, poznáme podle obsahu parametrů. Tlačítko submit definuje parametr akce. Jeho existencí můžeme podmínit vykonání kódu na přidání záznamu.
Kód pro přidání položky musíme umístit před dotaz na databázi, aby byl vrácen i vložený záznam. Zároveň musí být za otevřením spojení.
Abychom zabránili vkládání prázdných záznamů, vynutíme si od uživatele zadání alespoň položek jméno a text. Pokud některá z nich zadána nebude, vypíšeme místo vložení chybovou hlášku.

```
if(param("akce")){
my $jmeno = param("jmeno");
my $email = param("email");
my $text = param("text");
if(!$jmeno or !$text){
$telo .= h1("Jméno a text jsou povinné.");
}else{
#zadání v pořádku, zde se pokusíme vložit záznam do databáze
}
}
```

Je-li vše v pořádku, zakódujeme případné nevhodné znaky z uživatelského vstupu funkcí quote a vygenerujeme aktuální čas.

```
my $quoted_jmeno = $dbh->quote($jmeno);
my $quoted_email = $dbh->quote($email);
my $quoted_text = $dbh->quote($text);
my @d = localtime;
my $datetime = 1900+$d[5]."-$d[4]-$d[3] $d[2]:$d[1]:$d[0]";
```

Všechna data máme, poslední překážkou je poslání dat databázi.

```
if($dbh->do("INSERT INTO nastenka (jmeno, email, text, datetime)
VALUES ($quoted_jmeno, $quoted_email, $quoted_text, '$datetime')")){
$telo .= h1("Vloženo");
}else{
$telo .= h1("Nepodařilo se vložit.");
}
```

Odeslání stránky

Tělo dokumentu máme hotové. Vytiskneme tedy výsledek stránky na standartní výstup. To učiníme v bloku END, neboť příkazy uvnitř něj jsou volány vždy jako poslední.

```
END {
print $zacatek;
print $telo;
print $konec;
}
```

Začátek a konec stránky z externího modulu

Zbývá nám definovat funkce zacatek a konec. Rozhodli jsme se, že to bude realizováno v externím modulu, abychom jich mohli využít v příbuzných stránkách.
Do funkce zacatek zahrnmeme hlavičky dokumentu (MIME typ), HTML hlavičku a případný začátek HTML kódu za elementem BODY. Ve funkci konec bude pata dokumentu.

```
package Funkce;
```

```
use CGI qw(:standard);
use strict;

sub zacatek {
my $hlavicka = header;
$hlavicka .= start_html;
return $hlavicka;
}

sub konec {
$konec = end_html;
return $konec;
}
1;
```

Nyní nebude problém do HTML kódu všech stránek, které používají modul Funkce zahrnout reklamní banner, jméno autora či dělat jakékoliv jiné úpravy. V takovém případě stačí funkce zacatek a konecpříslušně upravit.

```
sub zacatek {
my $hlavicka = header;
$hlavicka .= start_html;
$hlavicka .= img({-src=>"/images/banner.png"});
return $hlavicka;
}

sub konec {
my $konec = hr;
$konec .= p({-align=>"center"}, "Author: xxx");
$konec = end_html;
return $konec;
}
```

Zdrojové kódy nástěnky jsou zde: Funkce.pm, nastenka.cgi.

Perl (88) - CGI - závěr

Dnes několika poznámkami dokončíme sérii o CGI. Zajímat nás budou zejména šablony.

Přepínače modulu CGI

Při načítání modulu CGI lze předat příkazu use jako další parametr pragmy. Pragma je něco jako přepínač, protože mění chování modulu. Pragma vždy začíná pomlčkou.

Pragma -any dokáže vytvářet HTML elementy, které nelze generovat modulem CGI definovanou funkcí. Tato pragma má vliv pouze v objektově-orientovaném přístupu. Po zavedení s -any můžeme psát toto.

```
use CGI qw(-any);
my $cgi = new CGI;
print $cgi->kontakt({-id=>1,-email=>'jiri@email.cz',-jmeno=>"jiri"});
```

Zmiňme si ještě pragmu -debug, která zapíná při spuštění v příkazovém řádku režim ladění. Ostatní lze nalézt v dokumentaci.

Získání URL ve skriptu

Funkce url vrací jméno současného skriptu. V závislosti na parametrech funkce urlobsahuje příslušné informace. Bez parametrů lze získat plnou adresu, s parametrem -absolute=1 resp. relative=>1 pouze adresu absolutní (vzhledem ke kořenovému adresáři) nebo relativní (vzhledem k aktuálnímu adresáři). Přidáme-li -query=>1, dostaneme i parametry předané metodou GET přes URL.

Šablony

V archivu CPAN je několik modulů, které se zabývají oddělením programu od dat. My si na ukázku představíme jeden z nich - modul HTML::Template.

HTML::Template umožňuje separovat HTML a programový kód v CGI skriptech. Funguje to tak, že vytvoříme HTML šablonu a programový kód v separovaných souborech. Program po spuštění načte HTML šablonu a do ní dosadí vygenerované hodnoty. HTML šablona je obyčejný HTML kód, ve kterém však je navíc několik speciálních elementů. Tyto elementy zastupují hodnoty, kterými budou později nahrazeny nebo řídí čtení kódu. Každý z nových elementů má prefix TMPL_.

Základní použití

Vytvoříme tedy nějaký CGI program s externím HTML kódem. Nejprve si vytvoříme šablonu a poté ji využijeme. Podívejme se na následující HTML kód a uložme ho do souboru hello.template.

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
Právě je <TMPL_VAR NAME="cas">.<br>
Používám <TMPL_VAR NAME="verze_cgi">.
</BODY>
</HTML>
```

Máme zde dva nové tagy <TMPL_VAR NAME="cas"> a <TMPL_VAR NAME="verze_cgi">. Poté, co bude šablona načtena naším CGI skriptem, budou elementy TMPL_VAR nahrazeny hodnotou identifikovanou přes hodnotu parametru NAME.

Nyní musíme vytvořit samotný CGI skript, který podle šablony vytvoří výslednou HTML stránku. HTML::Template má objektově-orientované rozhraní, takže ze všeho nejdříve zavoláme konstruktor importovaného modulu, kterému předáme jako parametr jméno souboru se šablonou.

```
use HTML::Template;
my $template = HTML::Template->new(filename => "hello.template");
```

Nyní nahradíme elementy TMPL_VAR příslušnými hodnotami. K tomu se používá metoda param.

```
my $now = localtime;
my $verze_cgi = $ENV{GATEWAY_INTERFACE};
$template->param(cas => $now);
$template->param(verze_cgi => $verze_cgi);
```

Na závěr vytiskneme HTTP hlavičku a pomocí metody output i HTML kód vygenerovaný podle šablony.

```
print "Content-Type: text/html\n\n";
print $template->output;
```

Výpis seznamu hodnot

HTML::Template umožňuje automatické vypisování více hodnot pomocí cyklů. Ideální nástroj to je pro výpis tabulky. Opět si nejprve napíšeme ukázkovou šablonu. Význam následujícího kódu je opět intuitivní.

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<HTML>
<HEAD>
<TITLE>Tabulka</TITLE>
</HEAD>
<BODY>
<TABLE>
<TR>
<TD>Jméno</TD>
<TD>Příjmení</TD>
<TD>Skóre</TD>
</TR>
<TMPL_LOOP NAME="loop">
<TR>
<TD><TMPL_VAR NAME="jmeno"></TD>
<TD><TMPL_VAR NAME="prijmeni"></TD>
<TD><TMPL_VAR NAME="skore"></TD>
</TR>
</TMPL_LOOP>
</TABLE>
</BODY>
</HTML>
```

A nyní implementujeme CGI soubor. Ten bude vypadat velmi podobně jako v prvním případě, pouze metodě param předáváme složitější datovou strukturu.

```
#!/usr/bin/perl
use strict;
use HTML::Template;

my $p_data = [
{jmeno=> "Jan",   prijmeni => "Novák",    skore=>23},
{jmeno=> "Josef", prijmeni => "Zápotocký", skore=>42}];

my $template = HTML::Template->new(filename => "moje.template");

$template->param(loop => $p_data);

print "Content-Type: text/html\n\n";
print $template->output;
```

Podmínky

HTML::Template podporuje podmíněné provádění kódu. To znamená, že lze zobrazit část HTML kódu pouze za určitých okolností. Následující kód ukazuje, jak takovou podmínku napsat. Pokud metodou param definujeme proměnnou vporadku s pravdivou hodnotou, vyhodnocuje se první blok, v opačném případě druhý.

```
<TMPL_IF NAME="vporadku">
V pořádku
<TMPL_ELSE>
Chyba
</TMPL_IF>
```

HTML::Template má řadu dalších možností. Pro zájemce je zde dokumentace.

Závěr

Použití CGI ve své podstatě není nic nového. Programování vypadá stále stejně a prostředky také. Modul CGI je pouze "něčím navíc". Důležitým znakem CGI je, že výsledkem není výsledná podoba dokumentu, ale pouze další zdrojový kód. Ať už pro HTML stránku, PDF soubor, obrázek nebo cokoliv jiného.

Část seriálu věnující se CGI nyní končí. Avšak část věnující se webu nikoliv. Vše zakončíme technologií Mason, která dokáže bez kompromisů zastoupit PHP.

Perl (89) - Mason - snadné psaní webů

Chcete psát weby v Perlu stejně jednoduše jako v PHP? Použijte Mason.

Slovo mason můžeme do češtiny přeložit jako kameník. Tento název o možnostech Masonu již mnohé vypovídá. mod_perl totiž ve spolupráci s Masonem vytváří výkonný systém, umožňující kombinování kódu Perlu přímo se značkovacím jazykem - tedy HTML kódem.

Tuto metodu lze přirovnat k PHP. Přestože lze PHP provozovat jako CGI nebo na příkazovém řádku, v naprosté většině případů funguje právě na výše uvedeném principu.

Výhodou mod_perl oproti PHP je dostupnost všech nástrojů Perlu. Narozdíl od PHP tak lze vytvářet jmenné prostory a máme zde daleko větší možnosti znovupoužitelného kódu. Můžeme využít archivu CPAN, jehož obdoba pro PHP ani žádný jiný jazyk neexistuje. Naopak nevýhodou je rozšířenost.

Instalace

Abychom mohli Mason používat, je třeba nainstalovat mod_perl. Masonem bez mod_perlu se zabývat nebudeme. Stáhneme tedy nejnovější verzi. Dostupná je na domovské stránce modulu perl.apache.org/download/index.html.

Stažený tar.gz soubor rozbalíme a nainstalujeme sekvencí těchto tří příkazů.

```
$ perl Makefile.PL MP_APXS=/usr/local/apache2/bin/apxs
$ make && make test
$ su -c 'make install'
```

Pokud nenastal problém, následuje instalace samotného modulu HTML::Mason. Spusťme CPAN shell a nainstalujme ho příkazem install. Může to chvilku trvat, protože se pravděpodobně bude instalovat i větší množství závislostí.

```
$ perl -MCPAN -eshell
cpan> install HTML::Mason
```

Konfigurace Apache

Poslední věcí pro fungování Masonu je konfigurace Apache. Je třeba upravit soubor /usr/local/apache2/conf/httpd.conf. Nejprve přidáme řádek, který aktivuje mod_perl.

```
LoadModule perl_module modules/mod_perl.so
```

Dále nastavíme, aby všechny soubory s příponou html byly brány jako Mason soubory.

```
<LocationMatch "\.html$">
SetHandler perl-script
PerlHandler HTML::Mason::ApacheHandler
</LocationMatch>
```

Aby se projevily změny, restartujeme Apache.

```
/usr/local/apache2/bin/apachectl restart
```

Pokud se neobjevila žádná chyba, měl by být Mason aktivní.

Jak to funguje

Aplikace v Masonu se skládá z komponent. Komponenta je jeden soubor, ve kterém se mísí text s řídícími prvky Masonu. Webová stránka může být výsledkem kombinace více komponent. Jedna může rozhodovat o menu, jiná o zápatí apod.

Základní syntaxe

Ke vkládání perlového nebo speciálního kódu definuje Mason několik vlastních značek. Jsou uvedeny v tabulce.

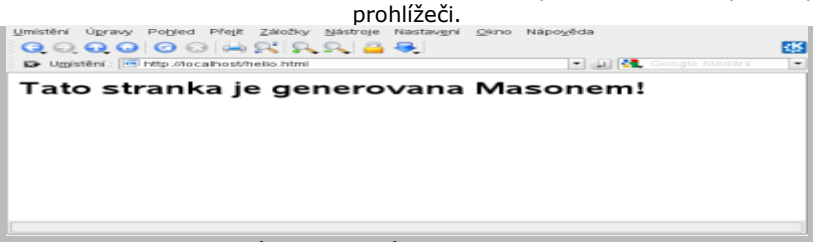
Tag	Význam
<% ... %>	obsahuje výraz, ten se vyhodnotí a výsledek se pošle na výstup
% ...	řádek kódu Perlu
<%*blok*> ... </%*blok*>	*blok* se nahrazuje klíčovým slovem, na něm závisí význam obsahu tagu

Uveďme si vůbec první příklad využívající technologie mod_perl. Zde je zdrojový kód.

```
% my $cim = "Masonem";
<h1>Tato stranka je generovana <% $cim %>!</h1>
```

Deklarace pomocí my je nutná, neboť Mason strict režim implicitně zapíná.

Uložíme si kód do souboru hello.html, umístěného do adresáře /usr/local/apache2/htdocs. Nyní skript spustíme přes server v prohlížeči.



Výsledek prvního Mason skriptu

Není těžké uhodnout, jak tento program pracuje. Úvodní řádek je uvozen procentem, což znamená, že obsahuje kód Perlu. Druhý řádek je normálně odesílán na výstup, ale ještě předtím se vyhodnotí výraz mezi <% a %>.

Blok perlového kódu bychom také mohli označit <%perl> ... </%perl>. Tento tag použijeme zejména tam, kde bude třeba označit více než několik řádků. Samozřejmě, že lze oba způsoby označení zaměňovat.

Escapování hodnot

Uvnitř <% ... %> lze uvést takzvaný přepínač. Ten specifikuje, zda a pokud ano, tak jak, se má escapovat výsledek výrazu. Jsou dvě možnosti escapování: HTML nebo URL.

Abychom pochopili význam escapování nebezpečných znaků, zkusme spustit tento kód.

```
% my $v = "<OK>";
Stav: <% $v %>
```

Hodnota $v se na výsledné stránce nepromítne. Interpret HTML si myslí, že <OK> je element jazyka.

Nyní nahradíme znaky <> příslušnými HTML escape sekvencemi. A to tak, že uvedeme přepínač h. Přepínače se uvádějí na konec výrazu za znak |. Pro URL escapování bychom zvolili přepínač u.

```
% my $v = "<OK>";
Stav: <% $v | h %>
```

Perl (90) - Mason - speciální bloky

Již víme, jak lze do Mason souborů vkládat kód Perlu pomocí speciálně označených pomocí bloků. Nyní se podíváme na některé další bloky, které přinášejí nové možnosti.

Komponenty Masonu jsou obyčejné HTML soubory. Obyčejné až na to, že obsahují jisté speciální úseky kódu, které jsou ještě před odesláním klientovi webového serveru zpracovány Masonem. Nyní se podrobněji podíváme na to, čím se tyto úseky vyznačují a jaký mají význam.

Inicializační blok

Blok <%init> ... </%init> je pro program něčím podobným jako konstruktor pro třídu nebo blok BEGIN pro perlový program. Jeho obsah je totiž vykonán (přesněji řečeno až na <%once> a <%shared>, ale to nás zatím nezajímá) ze všeho nejdříve. Nastavují se zde proměnné, které budou platné kdekoliv dále.

```
<% $file %>

<%init>
my $file = "/data.backup";
</%init>
```

Pokud v bloku <%init> děláme činnost, která musí být před ukončením programu zastavena, například otevírání databáze, můžeme toto zastavení provést v bloku <%cleanup>. Ten je volán po ukončení běhu programu.

Import komponent

Dalším uvedeným tagem je <& ... &>, který umožnuje import komponenty uvnitř jiné. To je praktické například tehdy, když na více stránek chceme umístit stejný prvek.

Základní použití může vypadat takto. V prvním souboru máme základní HTML strukturu dokumentu.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Moje www stránka</title>
</head>
<body>
<& obsah &>
</body>
<html>
```

Import probíhá na řádku <& obsah &>. Ten bude nahrazen obsahem souboru obsah.

Součástí importu komponenty mohou být takzvané argumenty. Ty uvádíme také mezi <& a &>. Import komponenty se dvěma argumenty může vypadat takto.

```
<& /cesta/komponenta, argument=>"hodnota", dalsi_argument=>"dalsi_hodnota" &>
```

Místo pevně zadaného názvu souboru, jež má být importován, lze zadat také výraz. Jeho výsledkem musí být jméno existující komponenty.

Díky této vlastnosti bloku <& ... &> zde nastává nejednoznačnost: jak odlišit výraz od názvu komponenty? Je třeba alespoň tušit, že existují jakási parsovací pravidla, pro něž je určující první uvedený znak. Pokud je alfanumerickým znakem, podtržítkem, tečkou nebo lomítkem, jde o pevně zadaný název souboru. V opačném případě jde o výraz. Je to sice vzácný jev, ale teoreticky se může stát, že výraz bude začínat jedním z výše uvedených znaků. V takovém případě je nejjednodušším řešením výrazu předřadit znaménko + nebo ho uzavřít do závorek.

Argumenty

Komponenta může přijímat argumenty a to hned několika způsoby. Již jsme si ukázali předávání argumentů při importu komponenty. Lze je také předávat jako součást požadavků POST a GET.

K zpracování argumentů uvnitř komponenty slouží blok <%args> ... </%args>. Do něj umístíme proměnné, které mají být hodnotami argumentů inicializovány.

Rozlišujeme povinné a nepovinné argumenty. Těm nepovinným nastavujeme defaultní hodnotu. Argument bez nastavené implicitní hodnoty ji bude vyžadovat.

Každý argument umístíme na jeden řádek a pomocí operátoru => mu případně definujeme implicitní hodnotu, která může být vyjádřena i výrazem.

```
<%args>
$scalar
$default  => 20
$default2 => $scalar-1000
@pole
%hash => (jmeno=>"anonym", heslo=>"gu1E-2")
</%args>
```

Definovali jsme 5 argumentů, z nichž jsou @pole a $scalar povinné. Ostatním nastavujeme implicitní hodnotu.

Zdánlivě se může kód uvnitř <%args> ... </%args> může zdát jako kód Perlu, ale tak tomu není. Nepoužívají se středníky a prostředky jazyka jsou také maximálně omezené.

Zmatek občas může nastat tehdy, pokud v bloku <%args> inicializujeme více než jednu proměnnou stejného názvu - tedy například $arg, @arg a %arg. Blok <%args>bude vypadat takto.

```
<%args>
$arg
@arg
%arg
</%args>
```

Jak se nám naplní příslušné proměnné? K tomu, abychom to zjistili, využijeme modul Data::Dumper.

Předáme-li skalární argument, program vyvolá chybu. $arg bude obsahovat příslušnou hodnotu, totéž první prvek pole @arg, ale u konverze na hash nastane chyba. Není zde možné konvertovat skalár na hash.

Nyní zavoláme komponentu se seznamovým argumentem: komponenta?arg=7&arg=9&arg=11. $arg bude ukazatelem na pole, které obsahuje prvky 7, 9 a 11 - tedy vlastně na pole @arg. Hash ale nelze vytvořit z lichého počtu argumentů.

Předáme-li sudý počet argumentů, bude to stejné jako v minulém případě, ale navíc se v pořádku naplní klíče a prvky hashe. Skalár, pole i hash budou zpřístupňovat vždy tatéž data, pouze vždy trochu jinak.

Další možností, jak získat předané argumenty, je proměnná %ARGS. Sem se zkopírují všechny předané hodnoty ve formátu klíč=>hodnota. Mnohy je výhodnější užít tento způsob získávání argumentů než blok <%args>. Přispívá k tomu i fakt, že název proměnné nemůže být libovolným řetězcem, ale hashový klíč ano.

Další bloky

Toto byly ty úplně nejzákladnější bloky, jež jsou pro používání Masonu bezpodmínečně nezbytné. Ale zdaleka to nejsou všechny. Pojďme si alespoň stručně představit ještě některé další.

Do bloku <%text> lze umístit libovlný text, který nemá být nijak formátován.

Blok <%doc> je určen pro generování dokumentace a překladač ho ignoruje.
Blok <filter> ... </%filter> se spouští mezi během a odesláním stránky. V něm ještě naposledy můžeme upravit výslednou stránku. Generovaný výstup máme v bloku < %filter> umístěn ve výchozí proměnné. Tu můžeme modifikovat. Tento příklad cenzuruje ze stránky slovo obsah.

```
<%filter>
s/obsah//;
</%filter>

<& header &>
<h1>Vítejte<h1>
<p>Toto je obsah stránky<p>
<& footer &>
```

Později se setkáme ještě s několika dalšími bloky.

Perl (91) - Mason - handlery

Jedním ze základních principů Masonu jsou autohandlery a dhandlery. To jsou speciální komponenty, které mezi komponentami zavádějí jistou formu dědičnosti a jsou pružným mechanizmem v navrhování webových aplikací.

Dhandlery

Dhandler je zkratka ze sousloví default handler. Přeložit bychom ji mohli jako implicitní obsluhovač. Dhandler je speciální komponenta s názvem dhandler.
Funkce dhandleru spočívá ve zpracovávání dat za komponenty, které neexistují. Jinými slovy, když voláme nějakou neexistující komponentu, místo ní se jejího úkolu zhostí komponenta dhandler.
Jestliže tedy Mason nenajde komponentu s jistým názvem, podívá se po komponentě s názvem dhandler v tomtéž adresáři (*pokud název dhandler nevyhovuje, lze ho změnit pomocí dhandler_name*). Pokud existuje, je použita místo požadované komponenty. Pokud ani dhandler ve stejném adresáři neexistuje, hledá se dhandler v rodičovském adresáři. A toto hledání postupuje rekurzivně stále výše až se nakonec dostáváme do kořenového adresáře.
Uveďme si konkrétní příklad hledání komponenty. Pokud Mason dostane požadavek na komponentu /data/2007/06/31/show, zavolá první existující komponnetu z těchto.

1. /data/2007/06/31/show
2. /data/2007/06/31/dhandler
3. /data/2007/06/dhandler
4. /data/2007/dhandler
5. /data/dhandler
6. /dhandler

Každá komponenta smí odmítnout požadavek Masonu. Pokud uvnitř komponenty voláme $m->decline, hledá se okamžitě bezprostředně nadřazený dhandler.

Dědičnost

Autohandlery poskytují další možnosti v oblasti znovupoužitelného kódu. Podobně jako v objektově-orientovaném programování od sebe mohou komponenty dědit.
Rodičovskou komponentu aktuální komponenty označujeme v bloku <%flags> ... </%flags> pomocí parametru inherit.

```
<%flags>
inherit => "rodicovska_komponenta"
</%flags>
```

Blok %flags obsahuje páry ve formátu *klíč-hodnota*. Obsahuje příznaky, podle kterých se k němu pak Mason chová.
Z programátorova hlediska to funguje tak, že rodičovská komponenta by měla obsahovat tento kód.

```
$m->call_next;
```

Za $m->call_next se dosadí kód potomka.
Pokud je třeba, aby měla sada stránek stejné hlavičky a patičky, bude nejlepším řešením, když je zdědí od rodičovské komponenty. Je to nepochybně pružnější mechanizmus než vkládat v každé stránce hlavičku a patičku pomocí <& ... &>. Jak to provést v praxi si ukážeme v oddílu Autohandlery.

Autohandlery

Pokud v komponentě parametr inherit nepoužijeme, přiřadí jí Mason implicitního rodiče. Tím je komponenta s názvem autohandler (opět lze změnit pomocí parametru autohandler_name). Pokud není dědičnost nastavena a autohandler v aktuálním adresáři neexistuje, hledá Mason autohandler postupně v rodičovských adresářích, stejně jako se hledá dhandler.
Zároveň autohandler dědí vždy od autohandleru v rodičovském adresáři.
call_next totiž funguje tak, že zavolá komponentu, od které autohandler dědí. Místo $m->call_next; je tedy dosazen kód potomka.

Užití dědičnosti

Pojďme se podívat, jak to v praxi funguje. Vytvoříme soubor /autohandler, který bude obsahovat HTML hlavičku a patičku.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Moje WWW</title>
</head>
<body bgcolor="yellow">
% $m->call_next;
</body>
</html>
```

Dále vytvoříme soubor /dedicnost/autohandler, který bude obsahovat menu specifické pro adresář /dedicnost.

```
<a href="...">menu1</a> | <a href="...">menu2</a> | <a href="...">menu3</a><br>
% $m->call_next;
```

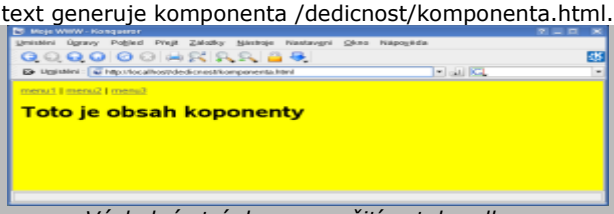
A nakonec vytvoříme komponentu /dedicnost/komponenta.html, která bude obsahem stránky.

```
<h1>Toto je obsah koponenty</h1>
```

/dedicnost/komponenta.html dědí od /dedicnost/autohandler a zároveň /dedicnost/autohandler dědí od /autohandler. Nejprve se tedy volá /autohandler. Když Mason narazí na $m->call_next, zavolá potomka - tedy /dedicnost/autohandler. Ten se chová stejně a když tedy narazí na $m->call_nextvolá požadovanou komponentu /dedicnost/komponenta.html. Výsledkem volání komponenty /dedicnost/komponenta.html je tedy následující HTML kód.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Moje WWW</title>
</head>
<body bgcolor="yellow">
<a href="...">menu1</a> | <a href="...">menu2</a> | <a href="...">menu3</a><br>
<h1>Toto je obsah koponenty</h1></body>
</html>
```

V prohlížeči spatříme toto. Žluté pozadí je generováno pomocí komponenty /autohandler, menu pomocí /dedicnost/autohandler a text generuje komponenta /dedicnost/komponenta.html.



Výsledná stránka po použití autohandleru

Autohandler může dělat řadu jiných užitečných věcí. Může třeba otevřít databázi a získat z ní data, která budou v požadované komponentě potřebná.

Metody $m->base_comp, $m->current_comp a $m->request_comp vracejí objekty s informacemi o základní, aktuální a požadované komponentě.

Perl (93) - Catalyst - MVC framework pro Perl

V několika následujících dílech se budeme věnovat Catalystu, což je MVC framework, který si za krátkou dobu vydobyl velkou popularitu. Catalyst nabízí obrovskou paletu různých nástrojů. Naším cílem nebude kompletně ho popsat, ale spíše seznámit se s jeho filozofií.

Obvyklý způsob navržení webové aplikace funguje takovým postupem, že každá stránka je generována nějakým skriptem. Tento skript nejprve získá data, dále s nimi něco udělá a na závěr vygeneruje HTML výstup. Takový postup bezesporu funguje. My si však nyní představíme jiný přístup.

MVC architektura

Model-View-Controller (MVC), pocházející původně ze Smalltalku, je způsob navržení aplikace, který ji rozděluje z hlediska programátora na tři části:

- rozhraní pro uživatele
- data
- řadič

Systém funguje tak, že uživatel vyvolá nějakou událost a ta je předána řadiči, který na základě ní udělá změny v datech. Podle toho, jak se změnila data, nyní může zareagovat i rozhraní.

Přitom platí, že modifikace jedné části by neměla mít vliv na ostatní. Tedy například to, kde jsou uložená data, by nemělo mít vliv na vzhled aplikace.

K vytvoření aplikace je obecně nutné vytvořit tři komponenty, které reflektují právě popsanou architekturu.

- View - tato komponenta rozhoduje o tom, jak se zobrazí aplikace uživateli
- Model - reprezentace dat; může jím být v podstatě cokoliv
- Controller - spolupracuje s modelem i uživatelským rozhraním; na základě požadavků uživatele (komponenta View) mění nebo získává data (komponenta Model)

Existuje celá řada frameworků implementujících architekturu MVC. Pro představu lze navštívit například wikipedii.

Catalyst

Catalyst je flexibilní web framework využívající architektury MVC inspirovaný zejména konkurenčním Ruby on Rails a Maypolem, který je určený pro rychlé vytváření velkých dobře udržovatelných aplikací. První verze Catalystu vznikla v roce 2005 a od té doby proběhl bouřlivým vývojem.

Catalyst lze používat na všech webových serverech s CGI nebo FastCGI. Pro testovací účely je navíc v distribuci Catalistu zahrnut i vlastní velmi jednoduchý HTTP server, který budeme používat při vývoji.

Další příjemnou vlastností je to, že bude potřeba psát minimum kódu. To ostatně uvidíme v následujících dílech.

Instalace

Je možné, že je ve vaší distribuci Catalyst již nainstalován, avšak nebývá to pravidlem. Některé distribuce ho dodávají jako balíček libcatalyst-perl.

Pokud Catalyst nainstalovaný nemáte, lze tak učinit standardním postupem pomocí perlového modulu CPAN.

```
$ cpan -i Catalyst::Runtime Catalyst::Devel
```

Obvykle se instaluje velké množství závislostí, takže celý proces může trvat i desítky minut. Pokud nastal nějaký problém, můžete se zkusit inspirovat v dokumentaci.

Vytvoření kostry aplikace

Catalyst je dodáván se shellovým skriptem catalyst.pl. Tento příkaz vytváří kostru každé webové aplikace, na kterou budeme chtít Catalyst použít. Zkusme spustit následující příkaz.

```
$ catalyst.pl MojeAplikace
```

Můžeme pozorovat, že se vytvořil adresář MojeAplikace a uvnitř něj řada souborů.

```
created "MojeAplikace"
created "MojeAplikace/script"
created "MojeAplikace/lib"
```

created "MojeAplikace/root"
created "MojeAplikace/root/static"
created "MojeAplikace/root/static/images"
created "MojeAplikace/t"
created "MojeAplikace/lib/MojeAplikace"
created "MojeAplikace/lib/MojeAplikace/Model"
created "MojeAplikace/lib/MojeAplikace/View"
created "MojeAplikace/lib/MojeAplikace/Controller"
created "MojeAplikace/mojeaplikace.conf"
created "MojeAplikace/lib/MojeAplikace.pm"
created "MojeAplikace/lib/MojeAplikace/Controller/Root.pm"
created "MojeAplikace/README"
created "MojeAplikace/Changes"
created "MojeAplikace/t/01app.t"
created "MojeAplikace/t/02pod.t"
created "MojeAplikace/t/03podcoverage.t"
created "MojeAplikace/root/static/images/catalyst_logo.png"
created "MojeAplikace/root/static/images/btn_120x50_built.png"
created "MojeAplikace/root/static/images/btn_120x50_built_shadow.png"
created "MojeAplikace/root/static/images/btn_120x50_powered.png"
created "MojeAplikace/root/static/images/btn_120x50_powered_shadow.png"
created "MojeAplikace/root/static/images/btn_88x31_built.png"
created "MojeAplikace/root/static/images/btn_88x31_built_shadow.png"
created "MojeAplikace/root/static/images/btn_88x31_powered.png"
created "MojeAplikace/root/static/images/btn_88x31_powered_shadow.png"
created "MojeAplikace/root/favicon.ico"
created "MojeAplikace/Makefile.PL"
created "MojeAplikace/script/mojeaplikace_cgi.pl"
created "MojeAplikace/script/mojeaplikace_fastcgi.pl"
created "MojeAplikace/script/mojeaplikace_server.pl"
created "MojeAplikace/script/mojeaplikace_test.pl"
created "MojeAplikace/script/mojeaplikace_create.pl"
Change to application directory and Run "perl Makefile.PL" to make sure your install is complete

Uveďme ještě, že obsahovalo-li by jméno aplikace čtyřtečky, nahradily by se lomítkem a vznikla by adresářová struktura.

Význam vytvořených souborů

Přímo v adresáři MojeAplikace je soubor mojeaplikace.conf, což je klasický konfigurační soubor pro aplikaci používající Apache syntaxi. Někdy zde místo něj je soubor se stejnou funkcí s koncovkou .plnebo .yml, kde mají konfigurační příkazy jiný tvar. Soubor Makefile.PL slouží k vygenerování Makefile pro build naší aplikace.

Dále zde jsou soubory jako README a Changes, které mají standardní význam.

Také je zde několik důležitých adresářů. Sem patří i lib, kde jsou uloženy moduly pro Controller, View a Model. Již zde existuje soubor MojeAplikace.pm, což je základní modul naší aplikace; dále adresář MojeAplikace, který obsahuje adresáře Controller, Model a View. Zde bude mít aplikace uložené komponenty všech tří typů, které později budeme vytvářet. Jak vidíme, je zde již Controller Root.pm, který se stará o úvodní stránku aplikace.

V adresáři root je skladiště šablon. Podadresář static se používá většinou k uchovávání obrázků, souborů ke stažení nebo CSS. Adresář t slouží k ukládání testů. Na začátku jsou zde vytvořené 3 testy: 01app.t pro test, zda se aplikace zkompiluje a 02pod.t a 03podcoverage.t pro účely dokumentace.

Nyní nás bude velmi zajímat adresář script obsahující několik perlových skriptů s různými účely.

Spuštění testovacího serveru a další skripty

Soubor script/mojeaplikace_server.pl je jednoduchý HTTP server, který můžeme používat při vývoji aplikace. Tento server spustíme standardním způsobem. Na obrazovce by se měl objevit výstup podobný následujícímu.

```
$ perl mojeaplikace_server.pl
[debug] Debug messages enabled
[debug] Statistics enabled
[debug] Loaded plugins:
.----------------------------------------------------------------------------.
| Catalyst::Plugin::ConfigLoader  0.27                                       |
| Catalyst::Plugin::Static::Simple  0.22                                     |
'----------------------------------------------------------------------------'

[debug] Loaded dispatcher "Catalyst::Dispatcher"
[debug] Loaded engine "Catalyst::Engine::HTTP"
[debug] Found home "/home/test/MojeAplikace"
[debug] Loaded Config "/home/test/MojeAplikace/mojeaplikace.conf"
[debug] Loaded components:
.-----------------------------------------------------------------+----------.
| Class                                                           | Type     |
+-----------------------------------------------------------------+----------+
| MojeAplikace::Controller::Root                                  | instance |
'-----------------------------------------------------------------+----------'

[debug] Loaded Private actions:
.----------------------+-------------------------------------+--------------.
| Private              | Class                               | Method       |
+----------------------+-------------------------------------+--------------+
| /default             | MojeAplikace::Controller::Root      | default      |
| /end                 | MojeAplikace::Controller::Root      | end          |
```

```
 | /index              | MojeAplikace::Controller::Root      | index       |
 '---------------------+-------------------------------------+-------------'

                    [debug] Loaded Path actions:
 .-------------------------------------+-------------------------------------.
        | Path                         | Private                             |
 +-------------------------------------+-------------------------------------+
        | /                            | /index                              |
        | /                            | /default                            |
 '-------------------------------------+-------------------------------------'

[info] MojeAplikace powered by Catalyst 5.80013
You can connect to your server at http://hostname:3000
```

Tím jsme spustili server. Pokud uděláme změny v adresáři lib, bude třeba webový server restartovat. Pokud však spustíme server s parametry -d -r, bude se o změny starat již spuštěný server sám.

```
$ perl mojeaplikace_server.pl -d -r
```

Dále jsou zde skripty mojeaplikace_fastcgi.pl a mojeaplikace_cgi.pl pro spuštění na jiném serveru.
Skript mojeaplikace_test.pl tiskne zdrojový kód požadované stránky bez spuštění serveru a dá se tedy použít na testování. Například pro výpis stránky /nejaka/cesta bychom použili následující příkaz.

```
$ perl mojeaplikace_test.pl /nejaka/cesta
```

Na konci výstupu vidíme zdrojový kód Page not found, což je správně, protože jsme zatím nic nevytvořili. Zkusme však otestovat kořenový adresář. Zde by se měl zobrazit zdrojový kód přednastavené webové stránky.

```
$ perl mojeaplikace_test.pl /
```

Úvodní stránka

Podívejme se ještě na výstup, který jsme dostali po spuštění serveru. Na konci je uvedena adresa http://hostname:3000, na níž se nám spustil webový server. Můžeme ji tedy zadat do webového prohlížeče a měli bychom spatřit úvodní stránku.

Úvodní obrazovka po spuštění webového serveru vygenerovaná Catalystem

Napíšeme první catalystovou aplikaci, což bude obnášet vytvoření několika komponent.

Vytvoříme si nyní jednoduchou aplikaci, na které si demonstrujeme základní postupy, které se při vývoji aplikací v Catalystu používají. Nejprve napíšeme aplikaci, která pouze zobrazí text. Na ní bude patrný způsob provázání jednotlivých souborů ve vytvořené adresářové struktuře.

Z teorie již víme, že je třeba vytvořit View, Controller a Model. Naše nová aplikace zatím žádná data potřebovat nebude, takže se zde obejdeme bez Modelu.

Základ aplikace

Na začátku tedy stejně jako minule vytvoříme adresářovou strukturu, která bude základem naší aplikace.

```
$ catalyst.pl Hello
```

Vytvoření View

Abychom nyní mohli ovlivňovat výslednou aplikaci, musíme přidat nějaké komponenty. V našem případě bude určitě nezbytný View (to bude výstup pro uživatele, tedy v důsledku jednoduchá stránka) a také Controller, který bude všechno řídit. Podotkněme pro pořádek, že každý View dědí od Catalyst::View, stejně tak každý Model od Catalyst::Model a každý Controller od Catalyst::Controller.

Vytvořme nyní naši první komponentu. Vytvoříme View s názvem TT. V adresáři script tedy spustíme následující příkaz.

```
$ perl hello_create.pl view TT TT
```

Jak již bylo zmíněno, skript hello_create.pl budeme používat k vytváření nových komponent. Při vytváření komponenty je třeba zvolit její typ jako první argument - zde vytváříme komponentu typu View a předáváme tedy argument view. Dalším parametrem je TT, což je jméno komponenty (má vliv na umístění v adresářové struktuře; naše komponenta bude na adrese lib/Hello/View/TT.pm). Poslední argument TT specifikuje, kterým modulem se bude vytvoření View řídit (to jest od kterého modulu bude dědit) - v našem případě jde o Catalyst::View::TT. Zkratka TT zde znamená Template Toolkit. Později si ukážeme i jiné typy View komponent a co to vlastně znamená "řídit se modulem".

Ještě poznamenejme, že místo použití skriptu hello_create.pl bychom mohli přímo vytvořit modul lib/Hello/View/TT.pm, do kterého bychom vložili následující obsah.

```
package Hello::View::TT;

use strict;
use warnings;
use base "Catalyst::View::TT";

1;
```

HTML kód

View je tímto hotov. Vytvořme nyní stránku, kterou uživatel ve skutečnosti uvidí. V adresáři root vytvoříme soubor hello.tt, kam vepíšeme (například) HTML kód. Takto může výsledný soubor vypadat.

```
<html>
```

<head><title>Toto je první stránka v Catalystu</title></head>

<body>
<h1>Ahoj světe!</h1>
Tady je něco napsáno
</body>

</html>

Zde jsme použili pouze čisté HTML, ale v následujícím dílu se dozvíme, že je to ve skutečnosti šablona, která má HTML teprve vygenerovat.

Vytvoření Controlleru

Nyní je potřeba nějak Catalystu sdělit, co má za jakých podmínek zobrazit. Vytvoříme tedy Controller a uvnitř něj uvedeme Catalystu všechny potřebné informace. Stejně jako posledně využijeme již hotový skript hello_create.pl.

```
$ perl hello_create.pl controller Hello
```

Tímto jsme vytvořili v adresáři lib/Hello/Controller náš první Controller s názvem Hello. Otevřeme soubor Hello.pm a podíváme se na jeho strukturu. Budeme ho muset pozměnit. Před editací se ale podíváme, jak vlastně na začátku vypadá.

```
package Hello::Controller::Hello;

use strict;
use warnings;
use parent "Catalyst::Controller";

=head1 NAME

Hello::Controller::Hello - Catalyst Controller

=head1 DESCRIPTION

Catalyst Controller.

=head1 METHODS

=cut


=head2 index

=cut

sub index :Path :Args(0) {
my ( $self, $c ) = @_;

$c->response->body('Matched Hello::Controller::Hello in Hello.');
}


=head1 AUTHOR

Jmeno Prijmeni
```

Typy akcí v Controlleru

Jak vidíme, je zde obsaženo několik use příkazů, nějaká dokumentace a především podprogram index. K čemu slouží, to se dozvíme později. My ho nyní nebudeme potřebovat a napíšeme si místo něj vlastní.

Na tomto místě je nejdůležitější zodpovědět si otázku, co to vlastně podprogram v Controlleru je. Tedy, odpověď zní, že každý podprogram v Controlleru je akcí, která se vykoná po zavolání jisté URL uživatelem aplikace.

Pojďme si neprve vysvětlit, jaké důsledky má pojmenování tohoto podprogramu a atributy.

Chceme-li, aby se podprogram aplikoval při volání /nejakynazev (to jest http://localhost:3000/nejakynazev), pojmenujeme ho přesně podle toho (tedy bude se v tomto případě jmenovat nejakynazev) a přidáme atribut :Global.

Použijeme-li atribut :Local, výsledná cesta se odvodí od názvu balíčku v Controlleru. Je-li název balíčku například tvaru Hello::Controller::Hello::Zeme::Argentina a podprogram je tvaru

```
sub info : Local {...}
```

pak je třeba do prohlížeče zadat adresu http://localhost:3000/zeme/argentina/info.

Dále je možné uvést atribut :Path("*cesta/k/necemu*"), kterým přímo určujeme cestu. V takovém případě nezáleží na názvu podprogramu. Cestu lze uvést v lokálním nebo globálním kontextu - to Controller pozná podle toho, zda cesta začíná lomítkem.

Taktéž se používají atributy :Regex("*regulární_výraz*") resp. :LocalRegex("*regulární_výraz*") pro specifikaci umístění pomocí regulárních výrazů. To je užitečné zejména ve spojení s tím, že lze zachytávat požadované části URL do závorek a získávat je později pomocí pole $c->req->captures. Uveďme příklad. Zadáme-li atribut :Regex("^id(\d+)/typ([ABCD])$"), vyhoví například cesta http://localhost:3000/id954/typB a $c->req->captures->[0] bude obsahovat 954 a $c->req->captures->[1] hodnotu B. Zanedlouho se na objekt $c podíváme hlouběji, a bude tak jasné, co tento zápis vlastně znamená.

Uveďme ještě, že pomocí atributu :Args určujeme, zda mají či nemají být přijaty argumenty. Pokud :Args neuvedeme, chová se Controller benevolentně. Pokud však uvedeme :Args(0), žádné argumenty vzaty nebudou a cesta musí vyhovět přesně - v opačném případě by stránka nebyla nalezena. Zadáme-li :Args(1), vyhoví pouze cesta s argumenty nebo alespoň s koncovým lomítkem.

Existují i další akce. Nyní se však pro zájemce opět odkážeme na dokumentaci, kde jsou navíc všechny atributy vysvětleny na příkladech.

Řešení kolizí

V případě, že více akcí vyhoví zadání URL, má přednost ta nejkonkrétněji určená. To se počítá podle počtu lomítek. Dále platí, že akce určené pomocí regulárních výrazů se zkouší až nakonec. Pokud dle předchozích podmínek nelze rozhodnout, vyhraje ta akce, která je nalezena první. Která to bude, to nemusí být vůbec zřejmé a proto je dobré se sporných případů snažit vyvarovat.

Příklad akce

Pro naše účely si vytvoříme podprogram nazvaný hello s atributem :Global, takže bude výsledná stránka dostupná na URL http://localhost:3000/hello. Podprogram hello tedy bude vypadat takto.

```
sub hello : Global {
my ($self, $c) = @_;
...
}
```

$self je objekt Hello::Controller::Hello a nebudeme ho vůbec potřebovat. Pomocí $cbudeme celou aplikaci ovládat. Například pomocí něj specifikujeme šablonu pro aktuální stránku, což právě potřebujeme. Pomocí metody stash přiřadíme položce template novou hodnotu. V našem případě vezmeme v adresáři root soubor hello.tt a výsledný podprogram tedy bude vypadat takto.

```
sub hello : Global {
my ($self, $c) = @_;
$c->stash->{template} = "hello.tt";
}
```

Objekt $c

Na okraj uveďme, že mimo stash lze použít nad objektem $c i několik dalších požadavků. Uveďme je zde však znovu pro přehlednost v jedné tabulce. U každé metody uvádíme i příklad použití. Více informací o tomto objektu lze nalézt v dokumentaci.

Požadavek	Význam	Příklady volání
stash	předání dat do View	$c->stash->{data} = "nazdar"
res	nastavení kódu HTTP statusu	$c->res->status(404); $c->res->redirect("http://www.linuxsoft.cz")
req	informace o požadavku - cookie, uploadované soubory, hlavičky, parametry	$c->req->headers->content_type
config	konfigurace	$c->config->{root}
log	tisk logovacích zpráv	$c->log->debug("Knihovna nactena"); $c->log->info("Zbyva 15 sekund")

Speciální akce

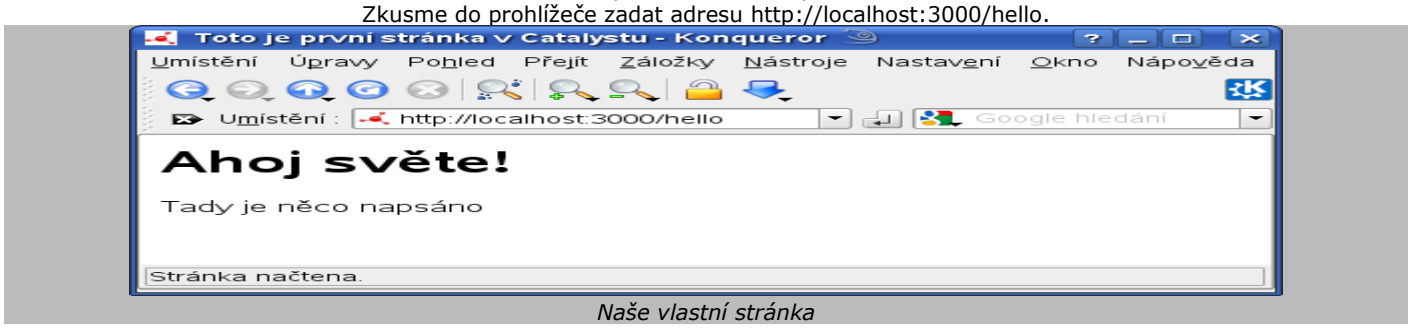
Existuje pět speciálních akcí mající svůj vlastní přednastavený význam. Jsou jimi default, begin, end, index a auto. Metoda default se volá v případě, že zadání URL nevyhoví žádná jiná akce. To je například způsob, jak vytvořit stránku známou jako The page not found. Podobný význam má index, ale ten narozdíl od default nepřijímá argumenty a má přednost.

Výsledná aplikace

Spusťme nyní server. V adresáři script spustíme následující příkaz.

```
$ perl hello_server.pl
```

Zkusme do prohlížeče zadat adresu http://localhost:3000/hello.



Naše vlastní stránka

Perl (95) - Catalyst - šablony

Blíže se seznámíme s několika různými syntaxemi, které lze používat v šablonách.

Podívejme se ještě jednou na soubor root/hello.tt, který jsme vytvořili v minulém dílu. Nazvali jsme ho šablonou, ale zatím obsahoval jen čistý HTML kód.

Ve skutečnosti může šablona vypadat takřka jakkoliv, protože si sami můžeme zvolit pravidla pro to, jak bude fungovat. O těchto pravidlech rozhoduje pomocný modul, kterým se daný View řídí.

My jsme použili v aplikaci Hello pomocný modul Template Toolkit (přesněji šlo o Catalyst::View::TT) a proto se primárně podíváme na tuto syntaxi. V tomto konkrétním případě platí, že šablona obsahuje speciální bloky kódu, které se před odesláním klientovi vyhodnotí a vygenerují místo sebe nějaký další HTML kód. Bloky jsou ohraničené závorkami [% ... %].

Zůstaňme ještě chvíli u Template Toolkit a představme si nejdůležitější syntaxi uvnitř speciálních bloků.

Vložení kódu

Nejjednodušší je vložení nějakých dat zvenku dovnitř šablony. K tomu používáme zápis buď [% text %] nebo [% text | html %]. Lze si to představovat tak, že text je proměnná, za kterou se dosadí její aktuální obsah. Zadáme-li modifikátor html sloužící pro filtrování výstupu, budou HTML znaky se speciálním významem překonvertovány na jejich zástupce uvozené ampersandem.

Naši šablonu můžeme upravit například takto.

```
<html>

<head><title>Toto je první stránka v Catalystu</title></head>

<body>
<h1>Ahoj světe!</h1>
```

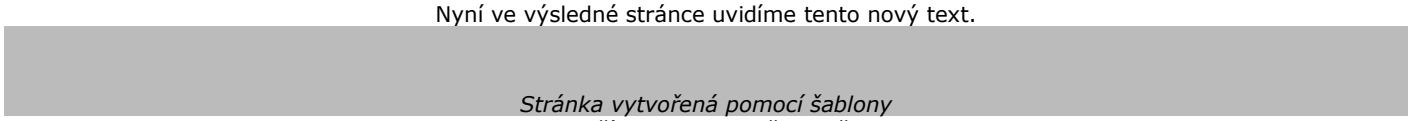
Tady je něco napsáno: **[% neco | html %]**
</body>

</html>

Tato data dodáme do výsledné podoby stránky v naší metodě hello. Využijeme opět objekt $c a pomocí stash přiřadíme našemu neco hodnotu. Procedura hello v našem Controlleru se změní do následující podoby.

```
sub hello : Global {
my ($self, $c) = @_;
$c->stash->{template} = "hello.tt";
$c->stash->{neco} = "toto jsou data získaná někde venku";
}
```

Nyní ve výsledné stránce uvidíme tento nový text.



Stránka vytvořená pomocí šablony

Další konstrukce v šabloně

Podívejme se ještě na složitější konstrukce, které můžeme použít uvnitř bloku [% ... %]. Lze používat nejběžnější řídící struktury jako IF, WHILE, FOREACH a podobně. Zde můžeme vidět příklad struktury IF, která ošetřuje, zda je neco definováno.

```
[% IF neco %]
<p>[% neco %]</p>
[% END %]
```

Dále bývá velmi užitečné použití FOREACH. Syntaxe je zde intuitivní. Konstrukce tohoto typu se používají při spolupráci s databází. S dolováním dat z databáze pomocí Catalystu se ještě později setkáme.

```
[% FOREACH polozka IN zbozi -%]
<tr>
<td>[% polozka.id %]</td>
<td>[% polozka.nazev %]</td>
<td>[% polozka.cena %]</td>
</tr>
[% END -%]
```

Komentář, který nebude vidět ve výsledném zdrojovém kódu pro HTML stránku můžeme napsat pomocí mřížky.

```
[% #toto je komentář %]
```

Pomocí META lze nastavit například titulek

```
[% META title = "titulek" -%]
```

Chceme-li vložit do šablony jinou šablonu, použijeme příkaz INCLUDE. Příkladem může být vkládání HTML hlavičky.

```
[% INCLUDE hlavicka titulek="Moje WWW" %]
```

V souboru hlavicka.tt pak může být tento obsah.

```
<html>
<head><title>[% titulek %]</title></head>
<body>
```

Zpracování argumentů

Upravíme-li podprogram hello vhodným způsobem, budeme moci pracovat s argumenty. Ale co to vlastně argumentu v Catalyst aplikaci jsou?

Argumenty jsou součástí URL adresy a zadávají se tak, že vypadají jako jména adresářů. Jsou oddělené lomítky a uživatel vůbec nepozná, že jde o argumenty. Voláme-li například URL http://localhost:3000/hello/arg1/arg2/arg3, ve skutečnosti se provádí naše procedura hello a té jsou předány navíc tři argumenty arg1, arg2, arg3. Catalyst je v tomto směru inteligentní, takže si poradí s různými cestami a atributy. Samozřejmě je třeba si dát o to větší pozor na kolize.

Modifikujme náš podprogram hello tak, abychom předali do šablony argumenty.

```
sub hello : Global {
my ($self, $c, @args) = @_;
$c->stash->{template} = "hello.tt";
$c->stash->{neco}     = "@args";
}
```

Připomeňme, že proměnná typu pole se do řetězce implicitně konvertuje tak, že se rozdělí na prvky oddělené mezerami. Šabloně tedy posíláme řetězec "arg1 arg2 arg3".

TTSite

View typu TTSite vytváří přednastavenou šablonu webu, obsahující hlavičky, patičky, CSS styly apod., kterou lze později různými způsoby konfigurovat. View TTSite vytvoříme standardně následujícím příkazem.

```
$ catalyst.pl Site
$ cd Site
$ perl script/site_create.pl view TT TTSite
```

Struktura adresáře root se nyní při použití TTSite lehce liší. root obsahuje tři podadresáře.

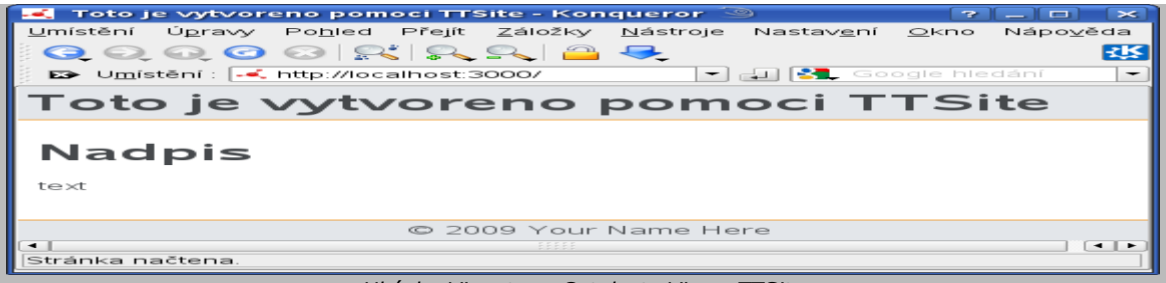
- static - pro uchovávání něměnných souborů, například obrázků
- src - zde jsou šablony
- lib - konfigurační soubory, hlavičky, patičky atd.

Jako ukázku TTSite vytvoříme opět jen úvodní stránku. Nejprve tedy založíme šablonu v souboru root/src/index.tt2.

```
[% META title = "Toto je vytvoreno pomoci TTSite" %]
<h1>Nadpis</h1>
<p>text</p>
```

Nyní ještě zeditujeme soubor lib/Site/View/TT.pm. Nakonfigurujeme příponu šablon na .tt2 tak, že k argumentům metody config přidáme řádek TEMPLATE_EXTENSION=>".tt2".

Dále vložíme metodu sub index : Private {} do Controlleru lib/Site/Controller/Root.pm. To je vše, co je potřeba k vytvoření jednoduché stránky. Spustíme-li server, uvidíme v prohlížeči následující stránku.



Ukázka View typu Catalyst::View::TTSite

Kdybychom chtěli vzhled aplikace upravit, můžeme zabrousit do adresáře root/lib a zde lze udělat různé úpravy v nastavení. Taktéž lze editovat kaskádové styly v CSS šabloně root/src/ttsite.css.

View a Mason

Poznamenejme, že lze vytvořit také masonovský View. Masonu jsme se již obsáhle věnovali a proto si jen náznakem ukažme, jak postupovat.

Na úvod opět vytvoříme komponentu, ta bude tentokrát typu Mason.

```
$ perl script/projekt_create.pl view Mason Mason
```

Pak lze v šablonách využívat masonovské syntaxe. Argumenty odeslané do šablony pomocí $c->stash získáme pomocí bloku <%args>.

Zde je příklad jednoduché masonovské šablony, která zobrazuje přijatý argument.

```
<%args>
$argument
</%args>
```

Od Controlleru jsme ziskali toto: **<%** $argument **%>**

Více o použití Masonu v Catalystu lze nalézt v dokumentaci.

View a další možnosti

Existuje řada dalších pomocníků usnadňujících vytváření View od vytváření PNG obrázků, generování PDF, uživatelská rozhraní, PHP kódu až po posílání emailů. K nalezení jsou v archivu CPAN a stojí za to si alespoň pro představu projít seznam modulů, protože lze narazit na opravdu zajímavé projekty.

Perl (96) - Catalyst - spolupráce s databází

V posledním dílu o catalystu si představíme databázový systém SQLite a naučíme se řešit základní úkoly při práci s databází.

V Catalystu máme zabudováno několik způsobů, jak se vypořádat s databázemi. Budeme zde používat modul DBIx::Class, který poskytuje pohodlné objektově-orientované rozhraní pro přístup k datům. Velkou výhodou je zde to, že nemusíme používat SQL a vyhneme se tak nekonzistencím mezi jednotlivými databázovými systémy.

Práci s databázemi si ukážeme na systému SQLite, což relační databázový systém od Richarda Hippa. Každá databáze je v SQLite uložena v samostatném .dbm souboru, který bude součástí aplikace.

Instalace

Nejprve tedy nainstalujeme SQLite a podporu databázím. Ve většině distribucí by měl být dostupný balíček sqlite nebo sqlite3. Pokud není, stáhneme ho například z domovské stránky projektu www.sqlite.org. Nainstalujeme ho a dále nainstalujeme i následující dva moduly.

```
$ cpan DBIx::Class Catalyst::Model::DBIC::Schema
```

Práce se SQLite

Práce se systémem SQLite je velmi intuitivní a není potřeba se k ní téměř nic nového učit. SQLite editor databáze spustíme příkazem sqlite3 (případně sqlite) a jako parametr uvedeme soubor, kde je databáze uložena (resp. kam ji uložit). Spustí se interpret, do kterého můžeme zadávat příkazy. Za prvé existují příkazy začínající tečkou, jimiž lze vykonávat činnosti jiné než zadávání SQL dotazů. Například .help vytiskne seznam tečkovaných příkazů.

```
$ sqlite3 db
SQLite version 3.5.7
Enter ".help" for instructions
sqlite> .help
.bail ON|OFF           Stop after hitting an error.  Default OFF
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in an SQL text format
.echo ON|OFF           Turn command echo on or off
.exit                  Exit this program
.explain ON|OFF        Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF      Turn display of headers on or off
.help                  Show this message
.import FILE TABLE     Import data from FILE into TABLE
.indices TABLE         Show names of all indices on TABLE
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                         csv      Comma-separated values
                         column   Left-aligned columns.  (See .width)
                         html     HTML \<table> code
                         insert   SQL insert statements for TABLE
                         line     One value per line
                         list     Values delimited by .separator string
                         tabs     Tab-separated values
                         tcl      TCL list elements
.nullvalue STRING      Print STRING in place of NULL values
.output FILENAME       Send output to FILENAME
```

```
.output stdout         Send output to the screen
.prompt MAIN CONTINUE  Replace the standard prompts
.quit                  Exit this program
.read FILENAME         Execute SQL in FILENAME
.schema ?TABLE?        Show the CREATE statements
.separator STRING      Change separator used by output mode and .import
.show                  Show the current values for various settings
.tables ?PATTERN?      List names of tables matching a LIKE pattern
.timeout MS            Try opening locked tables for MS milliseconds
.timer ON|OFF          Turn the CPU timer measurement on or off
.width NUM NUM ...     Set column widths for "column" mode
sqlite> .q
$
```

Dále lze zadávat SQL dotazy, jejichž tvar se většinou příliš neliší od tvaru, který používají velké databázové systémy. Následující posloupnost příkazů vytvoří naši první SQLite databázi. Vytvoříme tabulku druhů zboží.

```
$ sqlite3 db
SQLite version 3.5.7
Enter ".help" for instructions
sqlite> create table zbozi (id int primary key, nazev varchar(255), cena float);
sqlite> insert into zbozi (id, nazev, cena) values (null, "klavesnice", 100);
sqlite> insert into zbozi (id, nazev, cena) values (null, "mys", 200);
sqlite> select * from zbozi;
1|klavesnice|100.0
2|mys|200.0
```

Zkusme dále vytvořit tabulku, která bude obsahovat sériové číslo pro každý jednotlivý kus zboží. Jeden druh zboží může mít více takových kusů.

```
sqlite> create table kusy (id int primary key, druh_zbozi int, seriove_cislo varchar(100), stav int);
sqlite> insert into zbozi (id, druh_zbozi, seriove_cislo, stav) values (null, 2, "abc-123456789", 1);
```

Pokud již máme SQL příkazy připraveny v souboru db.sql, stačí pro import zadat pouze následující příkaz.

```
$ sqlite3 db < db.sql
```

Přístup k databázi

Nejprve si opět vytvoříme kostru aplikace pomocí příkazu catalyst.pl.

```
$ catalyst.pl Databaze
```

Abychom mohli přistupovat k databázi v Catalystu, je potřeba vytvořit datový Model. To uděláme spuštěním následujícího příkazu v adresáři script.

```
$ perl databaze_create.pl model MojeDatabaze DBIC::Schema Databaze::Schema::MojeDatabaze dbi:SQLite:db
```

Prvním argumentem je jako obvykle jméno komponenty. V důsledku to znamená, že komponenta bude žít v souboru lib/Databaze/Model/MojeDatabaze.pm. Dále DBIC::Schema je typ modelu a Databaze::Schema::MojeDatabazeuchovává strukturu tabulek uvnitř databáze specifikované posledním argumentem.

Struktura tabulek

Následkem zadání tohoto příkazu se vytvoří mimo jiné také soubor lib/Databaze/Schema/MojeDatabaze/Zbozi.pm a lib/Databaze/Schema/MojeDatabaze/Kusy.pm. Název Zbozi.pm je odvozen od jména tabulky v importované databázi. Pro každou tabulku se totiž vytvoří samostatný modul, který bude uchovávat její strukturu.

Jako ukázku toho, jak je zde zachycená struktura tabulek si vypišme například soubor Zbozi.pm.

```
package Databaze::Schema::MojeDatabaze::Result::Zbozi;

use strict;
use warnings;

use base "DBIx::Class";

__PACKAGE__->load_components("InflateColumn::DateTime", "Core");
__PACKAGE__->table("zbozi");
__PACKAGE__->add_columns(
    "id",
    {
  data_type => "INTEGER",
  default_value => undef,
    is_nullable => 0,
     size => undef,
    },
   "nazev",
    {
  data_type => "VARCHAR",
  default_value => undef,
    is_nullable => 0,
     size => 255,
    },
   "cena",
    {
  data_type => "FLOAT",
  default_value => undef,
    is_nullable => 0,
     size => undef,
    },
```

```
);
__PACKAGE__->set_primary_key("id");

# Created by DBIx::Class::Schema::Loader
# DO NOT MODIFY THIS OR ANYTHING ABOVE!

# You can replace this text with custom content, and it will be preserved on regeneration

1;
```

Soubor Kusy.pm vypadá analogicky.

V těchto souborech lze přímo určovat vztahy mezi tabulkami. K tomu existují funkce has_many a belongs_to. Například každý druh zboží ma obecně nějaký počet kusů na skladě. Tento vztah určíme přidáním následujícího řádku do Zbozi.pm.

```
__PACKAGE__->has_many(kusy => "Databaze::Schema::MojeDatabaze::Kusy", "druh_zbozi", {cascading_delete => 1});
```

Naopak do Kusy.pm bychom přidali následující kód, protože každý kus náleží k nějakému druhu zboží.

```
__PACKAGE__->belongs_to(zbozi => "Databaze::Schema::MojeDatabaze::Zbozi");
```

Zajímavého efektu lze také dosáhnout přidáváním nových podprogramů. Například chceme-li zformátovat jeden nebo několik sloupců tabulky a vytvořit něco jako imaginární sloupec, lze přidat následující kód.

```
sub cena_s_menou {
my($self) = @_;
return $self->cena." CZK";
}
```

Zobrazení dat

Nyní si ukážeme, jak se dají zobrazit data. Vytvoříme tedy soubor root/db.tt, což bude naše nová šablona. Dovnitř vepíšeme následující kód.

```
<html>

<head></head>

<body>
<h1>Výpis databáze</h1>
<table>
[% WHILE (radek = radky.next) %]
<tr><td>[% radek.id | html %]</td><td>[% radek.nazev | html %]</td><td>[% radek.cena | html %]</td></tr>
[% END %]
</table>
</body>

</html>
```

Nyní je potřeba pomocí Controlleru propojit model a tuto šablonu. Pro přidanou databázi tedy vytvoříme nový Controller s názvem Databaze.

```
$ perl databaze_create.pl controller Databaze
```

Naším cílem je, aby se po zadání http://localhost:3000/zbozi zobrazila naše šablona, do které se doplní data z naší databáze. Vytvoříme tedy metodu zbozi, která se nám o to postará.

```
sub zbozi :Global {
my($self, $c) = @_;
$c->stash->{template} = "db.tt";
$c->stash->{radky} = $c->model("MojeDatabaze");
}
```

Poslední řádek v podprogramu nás zajímá nejvíce, protože ten naplní proměnnou radky, ze které čerpáme v šabloně data. Metoda model nám umožňuje přistupovat k modelu pomocí jeho názvu.

Mazání

Upravme nyní šablonu db.tt tak, že přidáme možnost smazání záznamu. Můžeme to udělat například přidáním nového sloupce do tabulky.

```
<html>

<head></head>

<body>
<h1>Výpis databáze</h1>
<table>
[% WHILE (radek = radky.next) %]
<tr><td>[% radek.id | html %]</td><td>[% radek.nazev | html %]</td><td>[% radek.cena | html %]
</td><td><a href="[% Catalyst.uri_for("/zbozi/smazat/$radek.id") | html %]">smazat</a></td></tr>
[% END %]
</table>
</body>

</html>
```

Všimněme si, že pro odkaz používáme k tomu určenou funkci Catalyst.uri_for.

Nyní bude potřeba napsat metodu smazat, která nám akci smazání záznamu provede. Jak je vidět ze zápisu cesty /zbozi/smazat/$radek.id, předáváme této metodě jako parametr ID mazaného záznamu. Například, zadáme-li do prohlížeče URL http://localhost/zbozi/smazat/154, budeme chtít, aby se smazal záznam s id 154.

V komponentě Zbozi.pm tedy vytvoříme novou proceduru, která se o smazání postará.

```
sub smazat : Local {
my ($self, $c, $id) = @_;
```

```
my $polozka : Stashed = $c->model("MojeDatabaze::zbozi")->find({id => $id});
if($polozka){
$polozka->delete;
$c->stash->{vysledek} = "Smazano";
}
$c->forward("seznam_zbozi");
}
```

Vytváření a editace položek a automatické generování formulářů

Abychom mohli měnit upravovat data v databázi, budeme je muset od uživatele nějak získávat. Standardní cestou pro to jsou formuláře. Catalyst podporuje automatické vytváření formulářů pomocí modulu Catalyst::Controller::Formbuilder. Ten automaticky vygeneruje HTML, zkontroluje uživatelský vstup - a to jak na straně klienta pomocí Javascriptu, tak na straně serveru.

V adresáři root vytvoříme adresář forms a v něm zbozi/edit.fb. Do něj zapíšeme následující text.

```
name: zbozi_edit
method: post
fields
nazev:
label: Nazev zbozi
type: text
size: 100
required: 1
cena
label: Cena
type: text
size: 6
required: 1
```

Dále vytvoříme soubor /root/src/zbozi/edit.tt2 - toto bude samotná šablona, která použije k vygenerování HTML soubor edit.fb.

```
[% META title = "Editace zbozi" %]
[% form.render %]
```

Nyní tedy máme formulář a ještě bude potřeba vytvořit příslušnou akci. Do Controlleru přidáme následující řádky.

```
use base qw(Catalyst::Controller::FormBuilder Catalyst::Controller::BindLex);

sub edit : Local Form {
my ($self, $c, $id) = @_;
my $zbozi = $c->model("MojeDatabaze::zbozi")->find_or_new({id => $id});
if ($c->form->submitted and $c->form->validate) {
$zbozi->nazev($c->form->field("nazev"));
$zbozi->cena($c->form->field("cena"));
$zbozi->update_or_insert;
$c->stash->{vysledek} = $zbozi->nazev.": uspesne editovano!";
$c->forward("seznam_zbozi");
}
}
```

Perl (97) - Curses - tvorba textových uživatelských rozhraní

Dnes se budeme zabývat tvorbou uživatelského rozhraní pomocí knihovny Curses.

Curses je knihovna pro správu a řízení textového výstupu. Obvykle můžeme na výstup v terminálu pouze přidávat řádky a případně editovat poslední. Curses toto omezení odstraňuje a umožňuje nám libovolně měnit obsah po celé ploše terminálu.

Mezi základní vlastnosti knihovny Curses patří vytváření oken, používání různých stylů písma nebo práce s myší. Lze tedy vytvářet použitelná uživatelská rozhraní v textovém režimu.

Curses se asi nejčastěji používá v kombinaci s jazykem C. Nicméně díky modulu Curses, který je třeba získat z CPAN, lze tuto knihovnu používat i v kombinaci s Perlem.

V několika následujících dílech si tedy základy práce s Curses představíme. Nejprve budeme pracovat pouze s čistým Curses a v další části se soustředíme na knihovny Curses::UI, které obsahují několik již vytvořených widgetů, s kterými lze pracovat daleko rychleji.

Úvod do syntaxe

Na úvod si řekneme o několika nezbytných funkcích, jež modul Curses poskytuje. Ještě předtím upozorněme, že jména funkcí používaných v Perlu se občas liší od názvů funkcí v Curses pro C. Velmi často je například v názvy funkce vynechané podtržítko.

Funkce initscr inicializuje Curses režim. Tuto funkci budeme psát na začátek všech Curses programů. Obvykle má za následek vyprázdnění obrazovky.

Naopak poslední funkcí, kterou budeme v rámci Curses volat je endwin, která ukončuje Curses mód - dochází k takovým činnostem jako uvolnění paměti a návratu na původní obrazovku apod.

Pokud funkci endwin na konci programu neuvedeme, z terminálu se stane rozsypaný čaj. Proto se doporučuje uzavřít endwin do bloku END, čímž získáme jistotu, že bude proveden i v případě selhání programu.

```
END {
endwin;
}
```

Dále zde je funkce printw, která tiskne předaný řetězec na výstup. Funkcí na tisk textu existuje více, patří mezi ně také addstr.

Na základě těchto informací napíšeme program, který pouze vytiskne text na výstup.

```
use Curses;

initscr;
END{endwin;}
printw("***Hello World***");
```

Po spuštění se zdá, jako by se nic nestalo. Je to tak proto, že program začal, nato vytiskl text a okamžitě se ukončil. Musíme ho proto něčím zdržet. Nejlepším řešením bude použití funkce getch, která čte klávesu ze vstupu. Přepišme tedy náš program tak, aby s ukončením počkal na stisk klávesy.

```
use Curses;

initscr;
END{endwin;}
printw("***Hello World***");
getch;
```

Základní vstup a výstup

Jak již bylo uvedeno výše, funkce getch vrací klávesu, která byla stisknuta.

```
initscr;
END{endwin;}

printw("Stiskni nějakou klávesu: ");
my $key = getch;
printw("\nStiskl jsi tuto klávesu: <$key>");

getch;
```

Dále zde máme funkcí getstr pro čtení celých řádků. Jejím parametrem je proměnná, kam se má řetězec uložit.

```
initscr;
END{endwin;}

addstr("Zadej řádek: ");
getstr($text);
printw("Napsal jsi: <$text>");

getch;
```

Funkce echo resp. noecho povolují resp. zakazují zobrazování vstupních kláves. Implicitně je nastaveno echo. noecho usnadňuje kontrolu nad výstupem a je dobré ho používat zejména u čtení funkčních kláves.

Detekce kláves

Funkce keypad(*okno*, *true/false*) umožňuje číst funkční klávesy, kurzorové klávesy apod.

Poslední zmiňované funkce si můžeme předvést na programu, který načítá a vypisuje znak tak dlouho, dokud není stisknuta klávesa F3, která má kód 267. Když neznáme kód, stačí si ho nechat vypsat v následujícím programu.

```
initscr;
END{endwin;}

noecho;
keypad(stdscr, 1);

while(($k = getch) != 267){
printw("$k\n");
}
```

Abychom si nemuseli dohledávat kódy, lze použít v našem případě funkci KEY_F.KEY_F(3) vrátí hodnotu 267. Totéž jako předchozí lze tedy přepsat následovně.

```
initscr;
END{endwin;}

noecho;
keypad(stdscr, 1);

while(($k = getch) != KEY_F(3)){
printw("$k\n");
}
```

Doba čekání na vstup

Upozorněme ještě na funkci timeout. Ta nastavuje, jak dlouho bude program čekat na vstup. Pokud nedojde k odeslání příslušných dat na vstup programu do určené doby (nastavené v milisekundách), už se žádný vstup vyžadovat nebude. Například, chceme-li, aby uživatel zadal na vstup nějaký znak do 5 sekund, stačí před načtení vstupu uvést tento příkaz.

```
timeout(5000);
```

Pokud čas vyprší, do proměnné se uloží -1.

Styly písma

Písmu lze nastavit několik atributů. Funkce attron(*ATRIBUT*) zapíná určitou vlastnost a attroff(*ATRIBUT*) ji vypíná. Použijeme-li attrset (v dokumentaci má název attr_set), nastaví se uvedené vlastnosti oddělené bitovým NEBO (a ostatní vlastnosti se vypnou). V tabulce jsou uvedeny nejběžnější vlastnosti písma.

Konstanta	Význam
A_NORMAL	nastaví běžný text
A_STANDOUT	nastavuje zvýraznění; na většině terminálů totožné s A_REVERSE
A_REVERSE	prohodí barvu písma s pozadím
A_UNDERLINE	podtržené
A_BLINK	blikání textu
A_BOLD	buď tučné písmo nebo zesvětlení
A_INVIS	neviditelné písmo

COLOR_PAIR(*id*)	nastaví barevnou kombinaci číslo *id*; definicí barevných kombinací se budeme zabývat později

Předvedeme si názorně použití těchto tří funkcí a uvedených atributů. Spusťme následující program.

```
initscr;
END{endwin;}

printw("Normalni text\n");
attron(A_BOLD);
printw("Tucny text\n");
attron(A_UNDERLINE);
printw("Tucny a podtrzeny text\n");
attron(A_BLINK);
printw("Tucny, podtrzeny a blikajici text\n");
attron(A_STANDOUT);
printw("Jeste navic zvyrazneny text\n");
attroff(A_BLINK);
printw("Vypiname blikani\n");
attrset(A_BLINK | A_BOLD);
printw("Nyni je text pouze blikajici a tucny\n");

getch;
```

Perl (98) - Curses - pozicování a okna

Hlavním cílem dílu je ovládnout prostředky pro vytváření oken, které jsou analogií oken z grafických uživatelských rozhraní.

Již víme, že při použití Curses máme k dispozici celou plochu terminálu. Abychom ji mohli začít využívat, je třeba pochopit systém souřadnic.

Kurzor

Každý znak na obrazovce má přiřazeny dvě souřadnice. Té první říkejme například *y*-ová a té druhé *x*-ová.

- *y*-ová souřadnice: reprezentuje číslo řádku od shora; horní řádek má jako tuto souřadnici hodnotu 0
- *x*-ová souřadnice: reprezentuje číslo slouce zleva; nejlevější sloupec má jako tuto souřadnici hodnotu 0

Kurzor lze přesunovat funkcí move, které uvedeme jako parametr nové souřadnice.

V souvislosti s pozicí kurzoru jistě velmi často užijeme funkci getmaxyx, která zjišťuje pozici nejvzdálenějšího místa obrazovky od počátku souřadného systému - tedy souřadnice pravého dolního rohu. Parametrem jsou proměnné, do kterých se uloží souřadnice tohoto místa.

Existují další dvě funkce pro získávání nějakých souřadnic. Uvedeme-li getyx, získáme aktuální pozici kurzoru. Dále pomocí getbegyx získáme počáteční pozici kurzoru v okně - tedy pozici levého horního rohu okna. V okně stdscr získáme samozřejmě [0;0]. Více práci s okny rozvedeme v dalším oddílu.

Jiná funkce pro práci s kurzorem, addstr, dokáže přesunout kurzor, avšak to je jen vedlejší úkol k tisku textu.

Příklad na pozicování

Nyní máme dost informací na to, abychom pochopili tuto ukázku.

```
initscr;
END{endwin;}

printw("Text, zacinajici na pozici [0;0]\n");
move(3, 8);
printw("Text, zacinajici na pozici [3;8]\n");

$text = "Text uprostred";
getmaxyx($y, $x);
move($y/2, ($x-length($text))/2);
printw($text);

addstr(15, 2, "Text, zacinajici na pozici [15;2]");

getch;
```

Zvýraznění aktuální pozice kurzoru

Kurzor je obvykle na pozici, která je nějak zvýrazněná - většinou na styl A_STANDOUT. Pomocí funkce curs_set to lze změnit. Pokud nebudeme chtít zvýraznění pozice kurzoru, použijme následujícího volání.

```
curs_set(0);
```

Funkce inch vrací znak, který je na obrazovce již vytištěn. Parametrem inch jsou souřadnice.

Barvy

U většiny dnešních terminálů lze pracovat minimálně se základními barvami. To, zda to umí konkrétně váš terminál lze zjistit pomocí funkce has_colors. Pokud tomu tak skutečně je, zavolejme funkci start_color, která aktivuje barevný režim.

Abychom mohli tisknout barevný text, je třeba ještě definovat barevné kombinace. K tomu je k dispozici funkce init_pair. Jejím parametrem je ID barevné kombinace, které si můžeme zvolit; a poté dvě konstanty pro barvu textu a barvu pozadí.

Konstanty jsou ve tvaru COLOR_*BARVA* - tedy konkrétně jsou to následující hodnoty:

- COLOR_BLACK
- COLOR_RED
- COLOR_GREEN
- COLOR_YELLOW
- COLOR_BLUE
- COLOR_MAGENTA
- COLOR_CYAN

- COLOR_WHITE

Pokud váš terminál umožňuje měnit definici barev, lze tak činit pomocí init_color(COLOR_*BARVA*, *r*, *g*, *b*).
Nyní můžeme nastavovat barevnou kombinaci pomocí attron, kterému předáme parametr COLOR_PAIR(*id*).
Následuje jednoduchá ukázka demonstrující použití barev.

```
initscr;
END{endwin;}

start_color;

init_pair(1, COLOR_RED, COLOR_GREEN);
init_pair(2, COLOR_BLUE, COLOR_YELLOW);

printw("Normalni text\n");
attron(COLOR_PAIR(1));
printw("Cerveny text na zelenem pozadi\n");
attron(COLOR_PAIR(2));
printw("Modry text na zlutem pozadi\n");

getch;
```

Práce s okny

Práce s okny je možná vůbec nejdůležitější schopnost knihovny, která je potřeba v každém uživatelském rozhraní.
Ve skutečnosti se veškerá činnost děje v oknech. I my jsme dosud pracovali v okně, aniž bychom si to uvědomili. Existuje totiž standartní okno stdscr, které pokrývá celý rozsah terminálu.
Jako důkaz, že standartní okno stdscr skutečně existuje si můžeme zkusit následující příkaz. Jeho smysl bude vysvětlen vzápětí.

```
box(stdscr, 0, 0);
```

Můžeme však tvořit okna vlastní s různými rozměry a s ohraničením.
Okno se vytváří funkcí newwin. Jejími parametry jsou výška v řádcích, šířka v sloupcích a souřadnice levého horního rohu. Ještě před newwin je třeba obnovit obrazovku pomocí refresh.
Nyní již okno existuje a můžeme s ním manipulovat. Abychom však okno zobrazili, je třeba volat funkci box, jejímiž parametry jsou okno a styl vodorovných a horizontálních čar. U obou stylů většinou použijeme 0 pro normální ohraničení.
Místo box můžeme zvolit též obecnější border, pomocí níž lze též měnit ohraničení okna. Přijímá celkem 9 parametrů. Prvním je okno a pak následuje postupně ohraničení levé hrany okna, pravé hrany, horní hrany, spodní hrany, levého horního rohu, pravého horního rohu, pravého dolního rohu a levého dolního rohu.
Chceme-li do okna vepsat nějaký text, postupujeme úplně stejně jako dosud. Pouze příslušným funkcím uvedeme okno, kam se má tisknout. To se uvádí nejčastěji jako první parametr.
To samé platí o většině ostatních funkcí. Často použijeme zejména getbegyx a getmaxyx.
Pomocí jedné z funkcí derwin a subwin je možné vytvořit okno v okně.
Na závěr práce s oknem bychom měli uvolnit zabranou paměť. K tomu zavoláme delwin. Je třeba alespoň tušit, že text v okně ani ohraničení nezmizí, protože nejsou na okně po svém vzniku již závislé. To znamená, že když okno smažeme, tak zde jeho obsah zůstane a musíme se ho zbavit zvlášť. Nelepším řešením odstranění rámečku je volání border, kterému uvedeme jako znaky ohraničení mezery.
Ukažme si, jak tedy vytváření oken funguje. Kód proložíme komentáři, aby bylo jasně vidět, co se kdy děje.

```
initscr;
END{endwin;}

start_color;
init_pair(1, COLOR_RED, COLOR_GREEN);
attron(COLOR_PAIR(1));

refresh;

#vytvorime nove okno
$w = newwin(5, 30, 10, 10);

#ohranicime okno klasicky
box($w, 0, 0);

#vytiskneme text doprostred okna
$text_v_okne = "JSEM V OKNE";
getmaxyx($w, $w_y, $w_x);
addstr($w, 2, ($w_x - length($text_v_okne))/2, $text_v_okne);
refresh($w);

#cekame na libovolnou klavesu
getch;

#jakmile ji uzivatel stiskne, zmenime ohraniceni
border($w, ".", ".", ".", ".", ".", ".", ".", ".", );
refresh($w);

getch;

delwin($w);
```

Přechod mezi Curses módem a shell módem

Po funkci endwin se režim přepne zpět do shellu, kde můžeme například vyvolávat systémové příkazy.
Funkcí reset_prog_mode se opět vrátíme do Curses režimu.

```
initscr;
END{endwin;}


printw("Zadej prikaz: ");
getstr($cmd);
endwin();

system($cmd);
system("sleep 5");

reset_prog_mode();

printw("\nPokracujeme");

getch;
```

Perl (99) - Curses - měření rychlosti psaní

Znalosti nabyté v předchozích dvou dílech nyní použijeme k napsání programu, který bude testovat rychlost psaní.

Aplikace, která je cílem dnešního dílu, bude obsahovat několik textů a uživatel se bude snažit náhodně vybraný text opsat v co nejkratším čase.

Program bude fungovat tak, že budeme mít na úvod na obrazovce nějaký text. Po každém uživatelem správně opsaném znaku z tohoto textu se kurzor posune na následující pozici. Tak tomu bude až do té doby, než uživatel opíše celý text. Jeho čas budeme stopovat a na závěr vypíšeme statistiku počtu úhozů za sekundu. Pokud uživatel udělá chybu (stiskne jinou klávesu, než by měl), nebude na ni program reagovat.

Poznámka - Lepším řešením místo ignorování špatně opsaných znaků by z hlediska uživatele možná bylo počítání chyb. Avšak to bychom museli nějak definovat počet chyb, nebo-li metriku, která dvěma textům přiřadí jejich vzdálenost. Příkladem vhodné metriky pro takovýto účel je Levenshteinova metrika. Mohli bychom tedy použít například modul Text::LevenshteinXS.

A abychom vyzkoušeli vše, co jsme se naučili, tak na úvod programu navíc vytiskneme doprostřed obrazovky uvítací okno.

Realizace

Abychom mohli napsat tento program, musíme nějak vyřešit problém, jak stopovat čas. Nejsnažším řešením bude použití modulu - například Time::Stopwatch, který je k dispozici na CPANu.

```
use Time::Stopwatch;
```

Dále třeba definovat několik textů, z nichž bude později jeden vybrán.

```
my @texty = ("text 1", "dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text, dlouhy text", "a jeste jeden text");
```

Začátek programu bude stejný jako vždy. Po volání initscr ještě zneviditelníme kurzor a zavoláme noecho. Výstup si budeme řídit sami.

```
initscr;
END{endwin;}
curs_set(0);
noecho;
```

Prvním úkolem bylo vytvořit uvítací okno uprostřed obrazovky. K tomu budeme potřebovat pozice horního levého rohu. Musíme tedy získat rozměry obrazovky.

```
getmaxyx(stdscr, $y, $x);
```

A také je třeba určit rozměry okna. Protože zde uvedeme jméno programu a verzi zvolíme výšku 7 řádků.

```
my $vyska_okna=7;
my $sirka_okna=30;
```

Dále již můžeme okno vytvořit

```
refresh;
my $w = newwin($vyska_okna, $sirka_okna, ($y-$vyska_okna)/2, ($x-$sirka_okna)/2);
box($w, 0, 0);
```

Nyní doprostřed druhého a čtvrtého řádku v okně vložíme text.

```
getmaxyx($w, $w_y, $w_x);
addstr($w, 2, ($w_x - length($intro))/2, $intro);
addstr($w, 4, ($w_x - length($verze))/2, $verze);
refresh($w);
```

Úvodní okno je hotové. Počkáme tedy na stisk klávesy. Nastavíme timeout tak, že pokud by do dvou sekund nepřišel, budeme pokračovat dále.

```
timeout(2000);
getch;
notimeout(stdscr, 1);
delwin($w);
```

Protože se ale zrušením okna na obrazovce nic nesmaže, bude třeba ho odstranit. Momentálně nemáme zobrazeno nic jiného než toto okno, tedy můžeme použít funkci clear.

```
clear;
```

Nyní zobrazíme náhodně vylosovaný text a výzvu k zadání klávesy ENTER. Jakmile se tak stane, začneme stopovat čas. Výzvu zobrazíme na úvodní řádek a text začneme zobrazovat až na řádek čtvrtý.

```
addstr("Stisknete ENTER pro pokracovani");
```

Nyní vylosujeme text. Pro jistotu z něj odstraníme znaky nového řádku, aby nám nedělělaly neplechu.

```
$text = $texty[int(rand($#texty+1))];
$text =~ tr/\n/ /;
```

A teď nastane problém, pokud máme text delší než jeden řádek. Protože chceme, aby uživatel opisoval vždy řádek pod vzorovým textem, budeme muset tisknout řádek po částech tak, aby mezi nimi byly dvouřádkové mezery.

Řádek tedy budeme postupně rozdělovat pomocí substr a v cyklu tisknout. Vždy před tiskem se budeme muset posunout funkcí move o tři řádky dolů. Nastavíme proměnnou $prvni_radek na 3 a od tohoto řádku začneme postupně vypisovat text.

```
for(my($radek,$pozice)=($prvni_radek,0); $pozice<length($text); $pozice+=$x, $radek+=3){
    move($radek, 0);
    addstr(substr($text, $pozice, $x));
}
```

Jakmile dostaneme od uživatele znak nového řádku, můžeme pokračovat.

```
while(($key=getch()) ne "\n"){}
```

Musíme udělat několik věcí:

- vymazat výzvu k zadání ENTERu
- nastavit kurzor na pozici pod prvním řádkem vzorového textu
- začít stopovat čas

Smazání provedeme tak, že text přepíšeme mezerami. Potřebujeme tedy znát délku textu. Budeme muset přepsat výzvu, abychom ji mohli určit.

```
my $s = "Stisknete ENTER pro start";
addstr($s);
```

Dále napíšeme funkci delete_string, která bude mazat daný počet znaků od určité pozice. Protože by funkce neměla zasahovat do okolí, přesuneme v jejím závěru kurzor opět na původní souřadnice.

```
sub delete_string {
    my($y, $x, $kolik) = @_;
    my($puv_x, $puv_y);
    getyx($puv_y, $puv_x);
    move($y, $x);
    addstr(" " x $kolik);
    move($puv_y, $puv_x);
}
```

Pro smazání výzvy tedy použijeme tento příkaz.

```
delete_string(0, 0, length($s));
```

Druhým úkolem bylo nastavit kurzor na příslušnou pozici.

```
move($pozice_y, $pozice_x);
curs_set(1);
```

Odteď začneme měřit čas. Time::Stopwatch využívá mechanizmus tie, takže stačí navázat proměnnou $timer a při jejím čtení získáme vždy uplynulý čas.

```
tie my $timer, "Time::Stopwatch";
```

V cyklu budeme načítat znak po znaku text od uživatele.

```
while($key = getch){
    #...
}
```

Pokud dostaneme správný znak, posuneme se na pozici následující. Abychom mohli správnost znaku testovat, musíme pomocí substr tento znak vyseparovat ze zadání textu. Také budeme muset udržovat proměnnou uchovávající pozici v textu. Napíše-li uživatel správný znak, musíme nastavit $pozice_y a $pozice_x na nové hodnoty a přesunout na tuto pozici kurzor.

```
if($key eq substr($text, $pozice_v_textu, 1)){
    addstr($key);
    $pozice_v_textu++;
    $pozice_y = $prvni_radek+1+3*int($pozice_v_textu/$x);
    $pozice_x = $pozice_v_textu % $x;
    move($pozice_y, $pozice_x);
}
```

Pokud se uživatel splete, znak vymažeme a vrátíme kurzor.

```
else{
    move($pozice_y, $pozice_x);
    addstr(" ");
    move($pozice_y, $pozice_x);
}
```

To, zda již byl opsán celý text, poznáme podle stavu proměnné $pozice_v_textu. Shoduje-li se její obsah s délkou řetězce, pak bylo dosaženo cíle. Můžeme tedy směle tisknout statistiku.

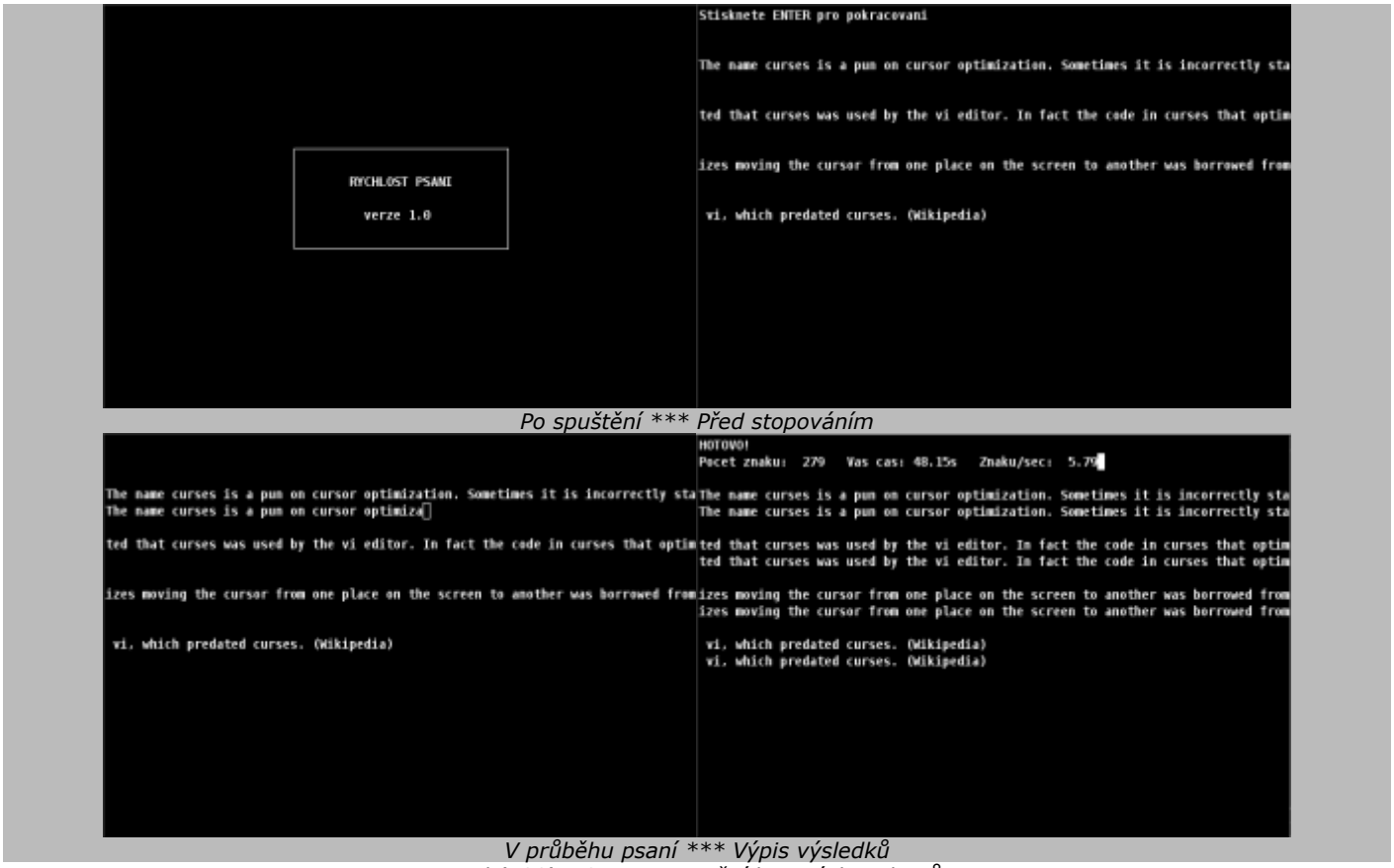
```
if(length($text) == $pozice_v_textu){
    move(0, 0);
    addstr("HOTOVO!");
    move(1, 0);
    addstr(sprintf("Pocet znaku: %4d   Vas cas: %5.2fs   Znaku/s: %5.2f",
    length($text), $timer, length($text)/$timer));
    last;
}
```

Na závěr počkáme na ENTER a program ukončíme.

```
while(getch() ne "\n"){;}
```

Výsledek

Jak vypadá náš program po spuštění, zachycují následující obrázky. Zdrojový kód je možné stáhnout.



*Po spuštění *** Před stopováním*

*V průběhu psaní *** Výpis výsledků*

Perl (100) - Curses - použití hotových widgetů

Modul Curses::UI je objektově-orientovanou nadstavbou pro Curses a umí vytvářet textová uživatelská rozhraní rychleji a lépe, než jsme toho byli schopni dosud.

Curses::UI je objektově orientovaný modul, který poskytuje rozhraní pro práci s Curses. Psaní aplikací pomocí Curses::UI je daleko rychlejší. To především díky tomu, že zde již máme předprogramované komponenty (widgety), ze kterých se programy skládají.

Modul je třeba nainstalovat obvyklým způsobem.

```
$ cpan Curses::UI
```

Vytvoření objektu

Pro inicializaci Curses::UI módu je třeba začít těmito dvěma řádky. Vytvoříme objekt typu Curses::UI.

```
use Curses::UI;
$cui = new Curses::UI;
```

Odteď již můžeme používat nadefinované widgety. Ještě předtím si ale zmiňme alespoň dva parametry konstruktoru. Prvním je -color_support => 1, který zapíná podporu barev. Pak je zde parametr -language (hodnota pro češtinu je czech - to znamená, že se použije překlad v modulu Curses::UI::Language::czech), avšak ten není zmíněn v dokumentaci.

```
my $cui = new Curses::UI(-color_support => 1);
```

Abychom předali řízení programu objektu, zavoláme na konci metodu mainloop.

```
$cui->mainloop;
```

Dnes si představíme několik widgetů z Curses::UI a příště je použijeme k napsání textového editoru.

Dialogy

Začněme zobrazováním dialogových oken. Díky metodě dialog je zobrazování velmi jednoduché a rychlé - stejně tak jako u všech dalších widgetů, které budou následovat. Existuje několik druhů dialogových oken.

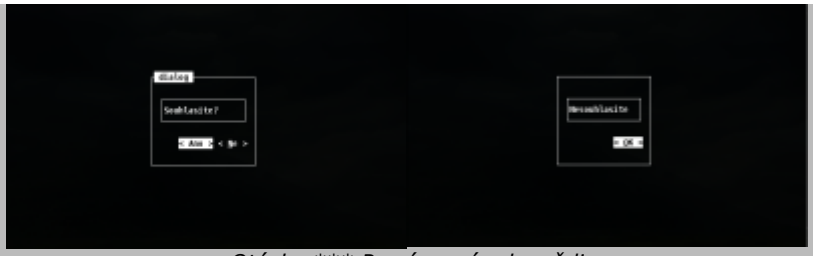
Nejjednodušším použitím je informativní dialog, kterému předáme pouze řetězec.

Chceme-li dialog s otázkou, je třeba definovat několik parametrů. Předně to je -message, což je samotná otázka. Dále jsou to tlačítka a jejich hodnoty. Jako tlačítka lze použít hodnoty yes, no a cancel. Texty tlačítek ale nakonec budou zobrazeny ve vybraném jazyce. Návratovou hodnotou metody dialog je jedna z hodnot parametru -values podle toho, které tlačítko bylo stisknuto.

Tento příklad ilustruje práci s dialogy.

```
use Curses::UI;
$cui = new Curses::UI;

$souhlas = $cui->dialog(
    -title   => "dialog",
    -message => "Souhlasite?",
    -buttons => ["yes", "no"],
    -values  => [1, 0],
    );

$cui->dialog($souhlas ? "Souhlasite" : "Nesouhlasite");
```

Vytvořili jsme dva dialogy. Na obrázku můžeme vidět, jak bude v takovém případě vypadat program.



*Otázka *** Po záporné odpovědi*

Existuje ještě další typ dialogu, který se používá u hlášek pro varování o chybách. Zde ovšem už používáme metodu error.

```
$cui->error("Neco je spatne");
```

Menu

Nezbytnou součástí většiny složitějších aplikací je menu. Pro něj zde máme definován widget Curses::UI::Menubar. Umí všechny základní vlastnosti menu včetně vytváření submenu.

Pro přidání widgetu se se používá metoda add, která má řadu parametrů. Prvními dvěmi hodnotami jsou ID a jméno widgetu - tedy v případě menu je to Menubar.

Dále můžeme přidat parametry -fg a -bg pro barvu textu a pozadí.

Nejdůležitějším parametrem však je -menu, kterému předáváme složitou datovou strukturu. Nebudeme si ji blíže rozebírat, protože by to mělo být zřejmé z následujícího příkladu.

```
my @menu = (
{
-label => "Menu1", -submenu => [
{-label => "Polozka", -value => "1"},
{-label => "Submenu", -submenu =>
[
{-label => "Polozka", -value => "2"},
{-label => "Polozka", -value => "3"},
]
},
{-label => "Konec", -value => \&exit_dialog},
]
},
{
-label => "Menu2", -submenu =>[
{-label => "Polozka", -value => "4"},
{-label => "Polozka", -value => "5"},
]
},
);
```

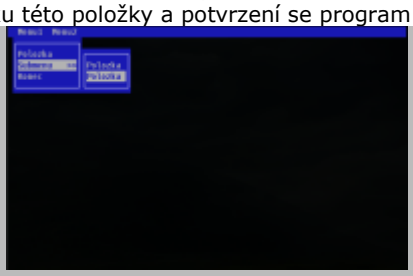
Nyní přidáme menu widget do aplikace

```
use Curses::UI;
my $cui = new Curses::UI(-color_support => 1, -language => "czech");
my $menu = $cui->add(
"menu",
"Menubar",
-menu =>\@menu,
-fg => "white",
-bg => "blue"
);
$cui->mainloop;
```

Jedna z položek vykonává po vybrání podprogram exit_dialog, který musíme dopsat.

```
sub exit_dialog {
exit if $cui->dialog(
-message   => "Konec?",
-title     => "konec",
-buttons   => ["yes", "no"],
);
}
```

Po stisku této položky a potvrzení se program ukončí.



Menu

Vazby a klávesové zkratky

Funkcí set_binding můžeme navázat k podprogramům klávesové zkratky. set_binding má tento tvar volání.

```
$cui->set_binding(\&podprogram, klávesová_zkratka);
```

Konkrétní příkaz pro svázání zkratky Ctrl-o s voláním podprogramu open_file může vypadat třeba takto.

```
$cui->set_binding(\&open_file, "\\cO");
```

Velice často se jako úvodní parametr používají anonymní podprogramy. Následující příkaz sváže stisk tabulátoru s přesunem kurzoru na menu.

```
$cui->set_binding(sub {$menu->focus();}, "\t");
```

Metodu focus, která přesune kurzor do příslušného widgetu, lze díky dědičnosti volat i nad většinou ostatních widgetů, které si dnes budeme představovat nebo které naleznete v dokumentaci k Curses::UI.

Okna

Mimo menu musí být všechny labely v okně. Pro vytváření dalších widgetů je tedy třeba nejprve vytvořit jim okno.

```
my $okno = $cui->add("okno1", "Window");
```

A poté tomuto oknu přiřadíme widget.

```
my $widget = $okno->add(...);
```

Label

Pro nápis kamkoliv na obrazovku slouží widget Label, kterému stačí předat text a souřadnice v aktuálním okně.

```
my $label = $okno->add(
"label", "Label",
-text  => "LABEL",
-x     => 10,
-y     => 5,
-bold  => 1,
);
$label->draw;
```

Texteditor

Dalším widgetem, se kterým se seznámíme, a který budeme ještě potřebovat, je Texteditor, v němž může uživatel editovat text.

```
my $texteditor = $dalsi_okno->add(
"text", "TextEditor",
-border => 1
);
```

Prohlížeč adresářové struktury

Aby mohl uživatel vybírat soubory z adresářové struktury, máme zde widget filebrowser. Ten ještě dále můžeme dělit na openfilebrowser a savefilebrowser. Parametrem widgetu filebrowser je cesta, ve které má výběr začínat.

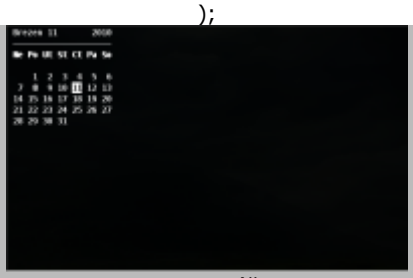
```
my $file = $cui->filebrowser(
-path => "/"
);

if(defined $file){
$cui->dialog("Vybral jste soubor $file");
}else{
$cui->error("Nevybral jste zadny soubor");
}
```

*Výběr souborů *** Vybraný soubor*

Kalendář

Pro zobrazení kalendáře zde je widget Calendar. Parametr -date specifikuje úvodní datum, na které má být kalendář nastaven.

```
my $kalendar = $okno->add(
"mylabel", "Calendar",
-date    => "2010-3-8"
);
```



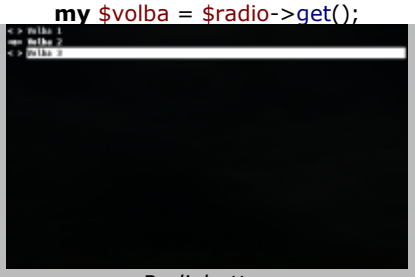
Kalendář

Radiobuttony

Ještě uveďme použití widgetu Radiobuttonbox. Pomocí nich může uživatel vybírat jednu z definovaných voleb. K podobným účelům lze užít též Listbox.

```
my $radio = $okno->add(
"radio", "Radiobuttonbox",
-values    => [1, 2, 3],
-labels    => {
1 => "Volba 1",
2 => "Volba 2",
3 => "Volba 3"
}
```

```
);

my $volba = $radio->get();
```

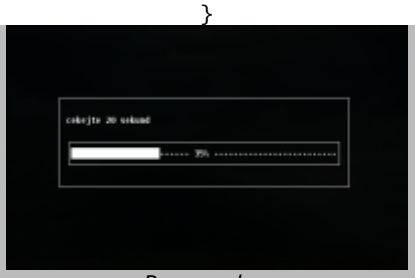


Radiobutton

Progressbar

Na závěr si uveďmě widget Progressbar. Ten obvykle uvádí, kolik procent něčeho je hotovo. Příklad progressbaru, který 20 sekund čeká, je zde.

```
$cui->progress(
-max => 20,
-message => "cekejte 20 sekund",
);

for (0..20) {
$cui->setprogress($_);
sleep 1;
}
```



Progressbar

Perl (101) - Curses - jednoduchý textový editor

Dnes pomocí knihovny Curses napíšeme textový editor se základními funkcemi, který bude plně použitelný.

V tomto závěrečném dílu o knihovně Curses::UI si napíšeme textový editor. Požadavky na něj sice nebudou příliš náročné, protože půjde o obyčejný editor, který poslouží pouze k rychlé editaci textových souborů, ale měl by být schopen základních operací.

Požadavky

Náš textový editor by měl umět několik základních operací jako načíst vybraný soubor a uložit aktuální data do souboru. Tedy pokud dostaneme z příkazového řádku soubor jako parametr, měli bychom tento soubor přímo otevřít.

Realizace

Nejprve vytvoříme menu, potom widget TextEditor a pak vše ostatní.

```
my $cui = new Curses::UI(
-color_support => 1,
-language => "cs"
);
```

Menu se bude skládat ze dvou hlavních nabídek. V první nabídce budou tři položky: načtení souboru, uložení souboru a položka pro ukončení programu. Druhá nabídka bude obsahovat jedinou položku O aplikaci. Vytvoříme tedy odpovídající datovou strukturu.

```
my @menu = (
{
-label => "Soubor", -submenu =>
[
{-label => "Nacist  Ctrl-O", -value => \&open_file},
{-label => "Ulozit  Ctrl-S", -value => \&save},
{-label => "Konec   Ctrl-X", -value => \&exit_dialog}
]
},
{
-label => "Napoveda", -submenu =>
[
{-label => "O aplikaci", -value => \&about},
]
},
);
```

Tuto strukturu použijeme k vytvoření menu.

```
my $menu = $cui->add(
"menu",
"Menubar",
-menu =>\@menu,
-fg => "white",
```

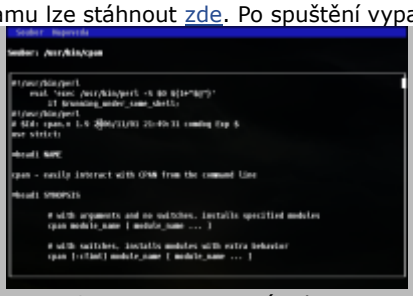
```
-bg => "blue"
);
```
Dále vytvoříme hlavní okno.
```
my $win = $cui->add(
"win",
"Window",
-y => 2,
-x => 0,
-border => 0
);
```
Do něj můžeme vkládat další widgety. Vytvoříme tedy pomocí klíčového slova TextEditor textové pole, ve kterém bude uživatel moci editovat text.
```
my $texteditor = $win->add(
"text",
"TextEditor",
-hscrollbar => 1,
-vscrollbar => 1,
-border=>1
);
```
Do tohoto okna budeme chtít hned přesunout kurzor.
```
$texteditor->focus();
```
To jsou všechny widgety, které potřebujeme. Zbývá nám už jen napsat podprogramy save a open_file.
Podprogram exit_dialog bude mít za úkol zeptat se uživatele po stisku tlačítka na ukončení programu, zda chce skutečně editor zavřít. Pokud uživatel ztvrdí rozhodnutí, program ukončíme.
```
sub exit_dialog {
my $return = $cui->dialog(
-message   => "Opravdu chcete ukoncit program?",
-title     => "Jste si jisty?",
-buttons   => ["yes", "no"],
);
exit(0) if $return;
}
```
V podprogramu openfile bude třeba vybrat soubor, který má být otevřen. K tomu je určena metoda loadfilebrowser. Jakmile tento soubor dostaneme, zobrazíme ho v textovém poli.
```
sub open_file {
my $soubor = shift;
$soubor = $cui->loadfilebrowser(
-path => "/home"
) unless -r $soubor;

if(-r $soubor){
local $/ = undef;
open DATA, $soubor;
my $data = <DATA>;
$texteditor->text($data);
close DATA;
}else{
$cui->error("Nevybral jste soubor");
}
return $soubor;
}
```
Ještě je třeba vyřešit ukládání souborů. Tady uplatníme metodu savefilebrowser. Do vybraného souboru pak standardním způsobem zapíšeme obsah textového pole.
```
sub save {
my $soubor = $cui->savefilebrowser(
-path => "/home"
);

if(defined $soubor){
local $/ = undef;
open CIL, ">$soubor";
print CIL $texteditor->get();
close CIL;
}else{
$cui->error("Nevybral jste soubor");
}
}
```
Na závěr můžeme dopsat metodu about například takto.
```
sub about {
$cui->dialog("Textovy editor, verze 1.0");
}
```
Ještě bychom chtěli, aby se v případě volání programu s argumentem zobrazil v textovém poli přímo obsah tohoto souboru. K tomu využijeme naši funkci open_file, které předáme parametr.
```
open_file($ARGV[0]) if -r $ARGV[0];
```
Dále nastavíme pro všechny činnosti v menu klávesové zkratky.
```
$cui->set_binding(\&open_file, "\cO");
```

```
$cui->set_binding(\&save, "\cS");
$cui->set_binding(\&exit_dialog, "\cX");
$cui->set_binding(sub {$menu->focus();}, "\t");
```

Na závěr ještě můžeme upravit chování programu tak, že v dialogu na umístění souboru (tedy dialog s otázkou kam chceme ukládat nebo který soubor chceme otevřít) otevřeme jako implicitní adresář ten, ze kterého pocházel původně otevřený soubor. To můžeme udělat tak, že zavedeme nějakou globální proměnnou, ve které budeme tuto informaci uchovávat a při každém otevření souboru aktualizovat.

Závěr

Zdrojový kód programu lze stáhnout zde. Po spuštění vypadá náš editor takto.



Curses::UI textový editor

Perl (102) - Rozšiřování Perlu pomocí XS

Přestože lze v Perlu lze napsat "téměř vše", lze občas najít nějakou úlohu, na kterou by byl jiný jazyk vhodnější. Nemusí to být proto, že by v Perlu daná věc napsat nešla, ale třeba kvůli optimalizaci rychlosti běhu programu. Tento díl ukáže, jak jednoduše integrovat jazyk C do Perlu.

Myšlenka spojení zdánlivě nespojitelného není složitá. Takový výsledný program se obecně skládá z několika souborů. První soubor je psán v jazyce C a obsahuje obvykle nějaké knihovny, které potřebujeme použít v našem původním perlovém programu. V perlovém souboru přistupujeme k C souboru jako k modulu. Aby to mohlo fungovat, je třeba vytvořit něco, co spojí oba tyto soubory dohromady - v literatuře se pro to vžil název vazebný kód (glue code).

Vazebný kód se musí postarat o několik věcí:

- Alokace paměti
- Převod datových typů - U standardních datových typů není problém, a tak je třeba řešit především uživatelsky definované datové typy. To vyžaduje od programátora napsat speciální funkce a jde o poměrně pokročilou záležitost.
- Dodržení syntaxe Perlu při volání (například pole jako návratová hodnota)

Příklad - s editací XS souboru

Napíšeme si jako ukázku jednoduchý modul Hello, který bude obsahovat funkci hello. Tato funkce pouze vypíše nějaký text na výstup.

Nejprve provedeme následující příkaz.

```
$ h2xs -nHello
Defaulting to backwards compatibility with perl 5.10.0
If you intend this module to be compatible with earlier perl versions, please
specify a minimum perl version with the -b option.

Writing Hello/ppport.h
Writing Hello/lib/Hello.pm
Writing Hello/Hello.xs
Writing Hello/fallback/const-c.inc
Writing Hello/fallback/const-xs.inc
Writing Hello/Makefile.PL
Writing Hello/README
Writing Hello/t/Hello.t
Writing Hello/Changes
Writing Hello/MANIFEST
$
```

Z výstupu je vidět, že se vytvoří adresář Hello, který obsahuje několik souborů. Nás teď bude nejvíce ze všech zajímat soubor Hello/Hello.xs. To je totiž soubor jazyka XS a budeme ho nyní editovat. V něm vytvoříme naši funkci hello.

Soubor Hello/Hello.xs by měl na začátku obsahovat kód podobnému tomuto.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"

#include "const-c.inc"

MODULE = Hello          PACKAGE = Hello

INCLUDE: const-xs.inc
```

Tento soubor upravíme a na konec přidáme kód naší funkce hello. Kód v XS souboru vypadá trochu jinak než jazyk C - je třeba zde použít klíčové slovo CODE, které určuje kód funkce. Nyní tedy máme v souboru Hello/Hello.xs následující.

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"
```

```
#include "const-c.inc"

MODULE = Hello          PACKAGE = Hello

INCLUDE: const-xs.inc

void hello();
CODE:
printf("Tento text tiskne jazyk C\n");
```

XS soubor začíná deklaracemi jazyka C. Zde je uvedeno, které hlavičkové soubory chceme vložit. Následuje řádek tvaru

```
MODULE = Hello              PACKAGE = Hello
```

Po něm následují XS funkce, které budou přeloženy pomocí nástroje xsubpp do nějakého kódu jazyka C.
Dále budeme chtít vygenerovat z XS souboru vazebný kód. Proto nás nyní bude zajímat soubor Makefile.PL. Provedeme následující příkazy.

```
$ cd Hello
$ perl Makefile.PL
$ make
```

Teď můžeme pozorovat, že v adresáři přibylo několik souborů. Mezi nimi je i soubor Hello.c, který obsahuje onen vazebný kód.
Na závěr nainstalujeme modul.

```
# make install
```

Nyní bychom měli mít v systému nový modul Hello. Přesvědčme se o tom. Vytvoříme soubor hello.pl s následujícím obsahem.

```
use Hello;
Hello::hello();
```

Spustíme program a měli bychom vidět tento výsledek.

```
$ perl hello.pl
Tento text tiskne jazyk C
$
```

Jak vzniká vazebný kód - nástroj xsubpp
xsubpp je nástroj, který konvertuje XS kód do kódu jazyka C a obvykle je spouštěn automaticky pomocí makefile. Funguje tak, že vytvoří soubor Modul.c a v něm funkce tvaru Modul_xs_funkce (například v našem příkladu je to Hello_xs_hello v souboru Hello.c).
Příklad - bez editace XS souboru
Předchozí příklad byl trochu podvod, protože ve skutečnosti jsme z žádného C programu nevycházeli. To nyní napravíme. Vytvoříme program, který bude dělat to samé, ale provedeme malé zjednodušení.
Jak je z názvu patrné, nástroj h2xs funguje tak, že převede rozhraní .h souboru do jazyka XS. Je tedy možné zavolat h2xs v následujícím tvaru.

```
$ h2xs -nHello2 hello2.h
```

Zde hello2.h obsahuje deklaraci funkce hello2.

```
void hello2(void);
```

V takovém případě se v XS souboru už objeví následující řádky:

```
void
hello();
```

Dále vytvoříme v adresáři Hello2 soubor hello2.c, který bude obsahovat funkci hello2. To znamená, že bude vypadat takto:

```
#include <stdio.h>

void hello2(void){
printf("Toto je uz opravdu ciste C");
}
```

Nyní upravíme Makefile.PL, abychom slinkovali vše, co je potřeba. Uvnitř něj se volá funkce WriteMakefile, což by mělo vypadat zhruba takto.

```
WriteMakefile(
                NAME              => 'Hello2',
        VERSION_FROM      => 'lib/Hello2.pm', # finds $VERSION
         PREREQ_PM        => {}, # e.g., Module::Name => 1.1
    ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
    (ABSTRACT_FROM  => 'lib/Hello2.pm', # retrieve abstract from module
               AUTHOR         => 'Jiri Vaclavik <jv@jv.cz>') : ()),
                LIBS              => [''], # e.g., '-lm'
              DEFINE            => '', # e.g., '-DHAVE_SOMETHING'
             INC               => '-I.', # e.g., '-I. -I/usr/include/other'
            # Un-comment this if you add C files to link with later:
           # OBJECT            => '$(O_FILES)', # link all the C files too
);
```

V tomto volání upravíme řádek s OBJECT. Přepišme celý příkaz znovu. V nové podobě bude vypadat takto.

```
WriteMakefile(
                NAME              => 'Hello2',
        VERSION_FROM      => 'lib/Hello2.pm', # finds $VERSION
         PREREQ_PM        => {}, # e.g., Module::Name => 1.1
    ($] >= 5.005 ?     ## Add these new keywords supported since 5.005
    (ABSTRACT_FROM  => 'lib/Hello2.pm', # retrieve abstract from module
               AUTHOR         => 'Jiri Vaclavik <jv@jv.cz>') : ()),
                LIBS              => [''], # e.g., '-lm'
              DEFINE            => '', # e.g., '-DHAVE_SOMETHING'
             INC               => '-I.', # e.g., '-I. -I/usr/include/other'
            # Un-comment this if you add C files to link with later:
```

```
    OBJECT          => 'hello2.o Hello2.o', # link all the C files too
);
```

Nyní již postupujeme stejně jako posledně, to jest následujícími příkazy.

```
$ perl Makefile.PL
$ make
# make install
```

Nyní bychom měli mít nainstalován modul Hello2 s funkcí hello2, která opět vypíše nějaký text.

Detekce prvočísel - ukázka zahrnutí parametrů

Ukážeme ještě jedno trochu smysluplnější použití spojení jazyků C a Perl. V C implementujeme funkci, která o daném čísle zjistí, zda je nebo není prvočíslo. Vytvoříme tedy nejdříve v jazyce C funkci je_prvocislo. V souboru prvocislo.h bude hlavička.

```
int je_prvocislo(int cislo);
```

A v souboru prvocislo.c samotná implementace. Zde je jednoduchý algoritmus, který zjistí, zda je předané číslo prvočíslem.

```
#include <stdio.h>

int je_prvocislo(int cislo){
  int delitel, prvocislo=1;
for(delitel=2; prvocislo!=0; delitel++){
  if (delitel<cislo){
  if (cislo%delitel!=0)
    prvocislo=1;
      else
    prvocislo=0;
     }else
    break;
     }
if (prvocislo==0) return 0;
  else return 1;
  }
```

A úplně stejně jako v předchozím příkladu vytvoříme modul. Zavoláme tedy příkaz h2xs.

```
$ h2xs -x -n Prvocislo prvocislo.h
```

Pro zajímavost můžeme nahlédnout do souboru Prvocislo.xs. Zde se nám na konci souboru objevil následující obsah.

```
int
je_prvocislo(cislo)
    int     cislo
```

V závorce jsou obvykle parametry bez datového typu, který je určen posléze pro každý argument na vlastním řádku.

Dále postupujeme opět standardní cestou. Editujeme soubor Makefile.PL a volání funkce WriteMakefile přepíšeme do následující podoby.

```
WriteMakefile(
                NAME            => 'Prvocislo',
        VERSION_FROM    => 'lib/Prvocislo.pm', # finds $VERSION
          PREREQ_PM     => {}, # e.g., Module::Name => 1.1
    ($] >= 5.005 ?    ## Add these new keywords supported since 5.005
  (ABSTRACT_FROM  => 'lib/Prvocislo.pm', # retrieve abstract from module
                AUTHOR      => 'Jiri Vaclavik <jv@jv.cz>') : ()),
                LIBS            => [''], # e.g., '-lm'
              DEFINE            => '', # e.g., '-DHAVE_SOMETHING'
              INC               => '-I.', # e.g., '-I. -I/usr/include/other'
              # Un-comment this if you add C files to link with later:
    OBJECT          => 'prvocislo.o Prvocislo.o', # link all the C files too
);
```

Následuje trojice obligátních příkazů.

```
$ perl Makefile.PL
$ make
# make install
```

Nyní můžeme vytvořit perl program prvocislo.pl využívající modul Prvocislo.

```
#!/usr/bin/env perl
use Prvocislo;
print $_.(Prvocislo::je_prvocislo($_)?" je ":" neni ")."prvocislo\n" for (1..10);
```

Po jeho spuštění bychom měli spatřit následující výstup.

```
$ perl prvocislo.pl
1 je prvocislo
2 je prvocislo
3 je prvocislo
4 neni prvocislo
5 je prvocislo
6 neni prvocislo
7 je prvocislo
8 neni prvocislo
9 neni prvocislo
10 neni prvocislo
$
```

Perl (103) - Rozšiřování Perlu pomocí SWIG

Alternativou k minule představenému nástroji XS je SWIG. Uvidíme jinou možnost, jak vytvářet moduly v jazyce C.

SWIG má tu výhodu, že podporuje širší paletu skriptovacích jazyků (Perl, Tcl, Python, PHP, Ruby, Java, C# atd. - více na www.swig.org) - nejen Perl, jak je tomu v případě XS. My se však budeme zabývat pouze rozšířením Perlu. Nevýhodou je zejména menší rozšířenost, protože SWIG se narozdíl od XS nedodává se základní distribucí Perlu, ale je třeba ho doinstalovat. Využití obou nástrojů se příliš neliší a je tedy možné si vybrat.

SWIG byl stvořen v červenci roku 1995 Davem Beazleyem v americkém Los Alamos. Poslední verze se datuje na polovinu minulého roku.

Instalace

Již bylo řečeno, že SWIG obvykle mít nainstalovaný nebudeme. Ve většině velkých linuxových distribucí by měl existovat balíček swig, který stačí nainstalovat. Pokud ho nemáte, lze nejnovější verzi tohoto nástroje najít na www.swig.org.

Příklad - implementace prvočísel

Ukážeme si příklad, který bude dělat to samé, jako v minulém dílu a srovnáme oba postupy. Připomeňme si zde, že máme v jazyce C napsané dva soubory. První se jmenuje prvocislo.h a obsahuje hlavičku funkce prvocislo.

```
int je_prvocislo(int cislo);
```

V souboru prvocislo.c je tato funkce implementována.

```
#include <stdio.h>

int je_prvocislo(int cislo){
int delitel, prvocislo=1;
for(delitel=2; prvocislo!=0; delitel++){
if (delitel<cislo){
if (cislo%delitel!=0)
prvocislo=1;
else
prvocislo=0;
}else
break;
}
if (prvocislo==0) return 0;
else return 1;
}
```

Nástroj XS vyžadoval vytvoření nějakého souboru ve speciálním XS jazyce. SWIG od programátora zase vyžaduje vytvořit nějaký kód v jazyce svém. Tento soubor se nazývá rozhraní (interface) a skládá se z příkazů začínajících znakem %. Je-li na začátku řádku znak %, znamená to, že tento řádek bude zpracováván přímo nástrojem SWIG.

Podívejme se nejprve na náš SWIG soubor. Nazveme ho Prvocislo.i a bude obsahovat následující řádky.

```
%module Prvocislo
%{
#include "prvocislo.h"
%}
int je_prvocislo(int cislo);
```

Mezi %{ a %} je úsek kódu v jazyce C. Vše ostatní jsou příkazy pro SWIG. Na posledním řádku jsou deklarace použitých exportovaných funkcí a datových struktur. Syntaxe je stejná jako u deklarací v .h souboru a tedy deklarace můžeme nahradit SWIG příkazem %include. Soubor Prvocislo.i tedy může vypadat i takto.

```
%module Prvocislo
%{
#include "prvocislo.h"
%}
%include prvocislo.h
```

Nyní potřebujeme vytvořit pro soubor prvocislo.h vazebný kód. K tomu použijeme příkaz swig. Je třeba specifikovat skriptovací jazyk, pro který chceme vygenerovat modul. V našem případě je to Perl5. Zadejme tedy následující příkaz.

```
$ swig -perl5 Prvocislo.i
```

Nyní v aktuálním adresáři vzniklo několik souborů. Soubor Prvocislo.pm je samotná knihovna. Dále vznikl soubor Prvocislo_wrap.c, který pro každou exportovanou funkci funkce obsahuje funkci _wrap_funkce a stará se o zpětný překlad.

Nyní nám zbývá vytvořit z .c souborů knihovny. Toho docílíme překladem. Můžeme použít příkaz gcc.

```
$ gcc prvocislo.c Prvocislo_wrap.c -I /usr/lib/perl5/5.10.0/i586-linux-thread-multi/CORE
$ ld -shared prvocislo.o Prvocislo_wrap.o -o prvocislo.so
```

Lepší ale je vygenerovat makefile a postupovat klasicky. Makefile si necháme vytvořit modulem ExtUtils::MakeMaker. Vytvoříme tedy soubor Makefile.PL, který bude mít následující obsah.

```
use ExtUtils::MakeMaker;
WriteMakefile(
"NAME"   => "Prv2",
"LIBS"   => ["-L /usr/local/lib"],
"OBJECT" => "Prvocislo_wrap.o prvocislo.o"
);
```

Poté nám již zbývá pouze nainstalovat modul.

```
$ perl Makefile.PL
$ make
# make install
```

Tento modul funguje stejně jako modul vytvořený v minulém dílu a tak nám opět bude fungovat následující program.

```
#!/usr/bin/env perl
use Prvocislo;
print $_.(Prvocislo::je_prvocislo($_)?" je ":" neni ")."prvocislo\n" for (1..10);
```

Perl (104) - Testování rychlosti

Naučíme se odhalovat pomalé úseky kódu pomocí metod pro měření rychlosti a porovnávání různých programových úseků.

Výkonostní testování, nebo-li benchmarking, zjišťuje, jak rychlý je náš kód. Užijeme ho zejména při optimalizaci kódu. Pokud potřebujeme, aby byl kód co nejvýkonnější a máme více možností, jak celý problém implementovat, vyzkoušíme je všechny. Poté se rozhodneme pomocí výkonostních testů, kterou možnost aplikujeme.

Výkonostní testy nemají většinou smysl, když je daná část kódu používaná v malém rozsahu. Pokud nejde o vyloženě špatný algoritmus, ušetříme tím jen zlomky sekundy a ty obvykle příliš nehrají roli. Pokud ale tyto zlomky vynásobíme 100000krát, už může být ušetřený čas patrný.

Měření rychlosti běhu programu

Obecně bychom mohli vytořit program pro stopování doby běhu úseků kódu velice jednoduše. Zde máme jednoduchý výkonnostní test, který funguje stejně jako stopky. Poprvé stopneme, když děj začíná. Po jeho konci stopneme podruhé.

```
use Time::HiRes;
my($stop1, $stop2);
my @pole;

$stop1 = Time::HiRes::time();

for(my $i=0; $i<1_000_000; $i++){
$pole[$i] = $i**2;
}

$stop2 = Time::HiRes::time();

print "Výsledek: ", $stop2-$stop1, "\n";
```

Měření rychlosti běhu programu pomocí modulu Benchmark

Výkonostní testy jsou v Perlu ale obvykle záležitostí modulu Benchmark. Ten již máte s největší pravděpodobností v systému předinstalován.

Výše uvedený kód bychom mohli přepsat za pomoci modulu Benchmark. Tím získáme i podprobnější informace.

```
use Benchmark;
my($stop1, $stop2);
my @pole;

$stop1 = new Benchmark;

for(my $i=0; $i<1_000_000; $i++){
$pole[$i] = $i**2;
}

$stop2 = new Benchmark;

print "Cas: ", timestr(timediff($stop2, $stop1)), "\n";
```

Přehled funkcí v Benchmark

Benchmark zpřístupňuje mimo již uvedených timediff a timestr několik dalších funkcí, pomocí nichž lze měřit čas běhu a porovnávat je. Mimo zmiňovaných budeme používat nejčastěji také následující.

Funkce	Popis
timeit	změří dobu provádění kódu
timethis	spustí několikrát úsek kódu a změří dobu provádění
timethese	spustí několikrát několik úseků kódu a změří doby provádění
countit	změří, kolikrát proběhl úsek kódu ve specifikovaném čase
cmpthese	tiskne výsledky porovnání několika úseků kódu v tabulce (touto funkcí již jsme se zabývali při měření rychlostí regulárních výrazů)

Nejjednodušší funkcí ze všech je timeit. Na základě počtu opakování a kódu provede výkonostní test a vrátí objekt typu Benchmark. Tedy v podstatě to, co jsme dělali v obou příkladech.

Kód je třeba uvést tak, jako kdyby měl být předán funkci eval. Musíme zabránit, aby byl vyhodnocen dříve, než bude předán funkci timeit, neboť by tak byl celý test znehodnocen. Použijeme tedy apostrofy.

```
$i=0;
my $o = timeit(1_000_000, '$pole[$i] = $i**2; $i++');
print timestr($o);
```

Můžeme si ukázat, jak vypadá výstup po volání timeit. Nejcennějším údajem bude většinou údaj *počet_běhů*/s.

```
4 wallclock secs ( 4.37 usr + 0.19 sys = 4.56 CPU) @ 1096491.23/s (n=5000000)
```

Podobně funguje také další funkce, timethis. Nejzřetelnější rozdíly oproti timeit jsou v tom, že timethis výsledky přímo tiskne a že lze zadat místo počtu cyklů záporné číslo. To znamená čas násobený -1, po který bude testování běžet.

```
my $i=0;
timethis(5_000_000, '$pole[$i] = $i**2; $i++');
```

Doba běhu části kódu je ale v podstatě nicneříkající. Smysl dostává až tehdy, když ji vztáhneme k nějakému srovnatelnému údaji. Proto ve většině případů oceníme spíše funkce k porovnávání. Jednou z nich je funkce timethese, která použije timethis na několik různých úseků.

Porovnání dvou programů

Další funkcí na porovnávání více úseků je cmpthese, kterou jsme siž představili v 25. dílu.

Zkusíme si porovnat rychlosti následujících dvou podprogramů.

```
sub bubblesort {
my @a = @_;
foreach $i (reverse 0..$#a) {
foreach (0..$i-1) {
($a[$_],$a[$_+1]) = ($a[$_+1],$a[$_]) if ($a[$_] > $a[$_+1]);
```

```
    }
    }
    return @a;
    }

    sub quicksort {
    @_ or return();
    my $p = shift;
    return (quicksort(grep $_ < $p, @_), $p, quicksort(grep $_ >= $p, @_));
    }
```

Vygenerujeme tedy náhodnou posloupnost čísel a ty se potom pokusíme seřadit.

```
my @cisla;
for(my $i=0; $i<100; $i++){
$cisla[$i] = int rand 100;
}
```

Pokud chceme zobrazit co nejvíce získaných dat, je výhodné použít cmpthese i timethese najednou.

```
use Benchmark qw(cmpthese timethese);
$o = timethese(-5, {
"bubble sort" => sub{bubblesort(@cisla)},
"quick sort" => sub{quicksort(@cisla)},
"perl sort" => sub{sort {$a<=>$b} @cisla;},
});
cmpthese($o);
```

Nyní spustíme program. Výsledek by měl vypadat přibližně takto. Nelze sice objektivně srovnávat výsledky vestavěného příkazu sort s výše uvedenými podprogramy, ale výsledek je jistě zajímavý.

```
Benchmark: running bubble sort, perl sort, quick sort for at least 5 CPU seconds...
bubble sort:  5 wallclock secs ( 5.21 usr +  0.01 sys =  5.22 CPU) @ 166.86/s (n=871)
perl sort:  4 wallclock secs ( 5.45 usr +  0.00 sys =  5.45 CPU) @ 3065623.49/s (n=16707648)
quick sort:  6 wallclock secs ( 5.32 usr +  0.00 sys =  5.32 CPU) @ 2454.14/s (n=13056)
                Rate bubble sort  quick sort   perl sort
bubble sort    167/s         --       -93%       -100%
quick sort    2454/s      1371%         --       -100%
perl sort  3065623/s   1837162%    124817%         --
```

Perl (105) - Testování programových jednotek

Při komplexnějších úlohách bývá zvykem testovat funkčnost jednotlivých komponent (například podprogramů). Předvedeme si jeden z nástrojů, jak na to.

Testování programových jednotek (unit testing) znamená ověřování správného chodu jednotlivých úseků programu. Je to tedy metoda hledání chyb.

Testovat lze libovolný úsek programu, u kterého to má smysl. Často se například testují podprogramy nebo metody. Záměrem je právě testovaný úsek maximálně izolovat od zbytku programu. Měl by být na okolí co možná nejnezávislejší.

Unit testing a Perl

Podívejme se na úvod na nejjednodušší možný test.

Příklad testu

Zde je zdrojový kód.

```
print "1..2\n";
print 1 == 1 ? "ok 1\n" : "not ok 1\n";
print 1 == 2 ? "ok 2\n" : "not ok 2\n";
```

Prvním řádkem avizujeme, že uděláme dva testy, které na dalších dvou řádcích následují. Tento zápis je obvyklou konvencí.

Spuštěním takového programu získáme následující výstup.

```
1..2
ok 1
not ok 2
```

První test dopadl podle očekávání úspěšně, druhý nikoliv.

Testy pomocí modulu Test::More

V Perlu se dá pro usnadnění testování použít řada modulů. Podíváme se zde speciálně na modul Test::More. Naučíme se používat některé jeho funkce.

Modul Test::More zavádíme obvykle s parametrem, kterým říkáme, kolik testů budeme provádět. Zde je příklad.

```
use Test::More tests => 10;
```

Pokud zadáme jiný počet, než nakonec provedeme, program si na konci postěžuje. Modul lze také zavádět s parametrem no_plan. To znamená, že počet testů neznáme. V takovém případě se budou normálně provádět testy, na které program narazí, ale ztratíme tím některé možností.

```
use Test::More "no_plan";
```

Lepší než no_plan však často je lepší dodat počet testů na konci. Pak má program následující schéma.

```
use Test::More tests => 2;

# testy

done_testing(pocet_testu);
```

Náš první příklad bychom mohli přehledněji přepsat pomocí exportované funkce ok do následující podoby.

```
use Test::More tests => 2;
ok(1 == 1);
ok(1 == 2);
```

Spustíme-li tento program, provede se totéž jako prve, ale navíc zde přibyde nějaký závěrečný komentář o tom, kolik testů našlo chyby.

```
1..2
```

```
ok 1
not ok 2
#   Failed test at more.pl line 3.
# Looks like you failed 1 test of 2.
```

Příklad testování modulu

Vybereme si nějaký modul a ukážeme pomocí něj několik věcí, které můžeme testovat. Zvolíme například modul Math::Complex, který zavádí objekty typu komplexní číslo a provádí s nimi základní operace.

Nejprve uveďme, že funkci ok lze předat jako argument komentář, který se má objevit na výstupu.

Nyní zkusme udělat několik jednoduchých testů.

```
use Test::More tests=>4;
use Math::Complex;

$z1 = Math::Complex->make(1, 2);
$z2 = Math::Complex->make(3, 4);
$z3 = Math::Complex->make(4, 6);
$z4 = Math::Complex->make(-8, 10);

ok(defined $z1, "metoda make chodi. Vytvoreno: '$z1'");
ok($z1->isa("Math::Complex"), "metoda make vraci objekt typu Math::Complex");
ok($z1 + $z2 == $z3, "scitani funguje");
ok($z1 * $z2 == $z4, "nasobeni funguje");
```

Poslední test je ukázkou toho, že když test selže, nutně to neznamená, že je chyba v modulu. V tomto případě je chybně napsaný test a proto selhal. Horší však je, když test neselže, protože je chybně napsaný. Poučením je, že je potřeba vždy dávat pozor.

Porovnávání řetězců - vylepšení funkce ok pomocí is

Funkce is dělá téměř totéž jako ok, ale má jinou strukturu volání a vypisuje více informací. Místo výrazu jí předáme dva argumenty, které se mají porovnat pomocí operátoru eq (to jest skutečná hodnota a očekávaná hodnota). Zatímco ok se může rozhodovat pouze na základě toho, zda je argument pravdivý či nikoliv, funkce is obě porovnávané hodnoty navíc zná.

Upravme tedy testy pomocí is. První dva necháme stejné, protože zde není nic k porovnávání. Avšak na další dva is aplikovat lze.

```
ok(defined $z1, "metoda make chodi. Vytvoreno: '$z1'");
ok($z1->isa("Math::Complex"), "metoda make vraci objekt typu Math::Complex");
is($z1 + $z2, $z3, "scitani funguje");
is($z1 * $z2, $z4, "nasobeni funguje");
```

Výstup u čtvrtého testu se změnil. Test selhal a dostáváme navíc informaci o tom, jak se liší obě hodnoty.

```
1..4
ok 1 - metoda make chodi. Vytvoreno: '1+2i'
ok 2 - metoda make vraci objekt typu Math::Complex
ok 3 - scitani funguje
not ok 4 - nasobeni funguje
#   Failed test 'nasobeni funguje'
#   at komplex2.pl line 12.
#          got: '-5+10i'
#     expected: '-8+10i'
# Looks like you failed 1 test of 4.
```

Existuje také funkce isnt (pro puntičkáře též varianta s apostrofem isn't), což je negace is. Lze ji užít například následovně k testu neprázdnosti proměnné nebo rozdílnosti dvou proměnných.

```
isnt($data, "", "mame data");
isnt($a, $b, "a, b se lisi");
```

Může se hodit též funkce is_deeply, která pomocí porovnává do hloubky datové struktury.

Porovnávání obecně - vylepšení funkce ok pomocí cmp_ok

Podobně funguje funkce cmp_ok. Zde je ale navíc potřeba zadat explicitně operátor, protože máme univerzálnější použití.

```
cmp_ok($promenna, "==", $ocekavano, "plati rovnost");
cmp_ok($p, "&&", $q, "plati p && q");
```

Regulární výrazy - vylepšení funkce ok pomocí like

like zjistí, zda předaná hodnota vyhovuje regulárnímu výrazu.

```
like($cislo, "/^\w+$/", "cislo je korektne zadane");
```

Lze použít i unlike. Postup je analogický jako v předchozích dvou případech

Testování existence metod

Funkce can_ok otestuje, zda v nějakém modulu existují dané metody. Zde je příklad použití.

```
can_ok("Modul", qw(metoda1 metoda2 metoda3));
```

Jiné platformy a přeskakování testů

Pokud některé testy nebudeme moci provést na určitém operačním systému (například nebude obsahovat některé k tomu nezbytné nástroje nebo budeme mít přednastavené jiné hodnoty), můžeme je přeskakovat.

Existuje funkce skip, která přeskočí požadovaný počet testů. Pokud budeme skutečně chtít podmiňovat vykonání testů operačním systémem, pravděpodobně sáhneme po proměnné $^O (alternativně $OSNAME). Její hodnoty jsou například linux, MSWin32, MacOs. Chceme-li rozlišit mezi různými verzemi operačního systému, musíme už využít nějakého externího modulu. To se může stát zejména s různými verzemi Windows, které se mohou výrazně lišit. V takovém případě užijeme například funkci GetOSName v modulu Win32.

```
SKIP: {
skip("Nelze testovat na operacnich systemech Windows.", 1) if $^O eq "MSWin32";

is(...);
}
```

Subtesty

Jen na okraj poznamenejme, že lze dělat též subtesty.

```
use Test::More tests => 2;
```

```
ok(1);
subtest "subtest" => sub {
    plan tests => 1;
    ok(1);
};
```

Tento program vyprodukuje následující výstup.

```
1..2
ok 1
    1..1
    ok 1
ok 2 - subtest
```

Perl (106) - Debugování pomocí komentářů

Do komentářů budeme zapisovat příkazy se speciální syntaxí, které nám vypíší ladící informace. Jde o zajímavou alternativu k již známým nástrojům.

Komentáře v programu se dají využít i k jiným účelům než jen k popisování toho, co se na kterém řádku děje. Modul Smart::Comments stanovuje pravidla, jak využít komentáře k debugování. Funguje tak, že komentáře mají speciální syntaxi a nejsou jen přeskakovány, nýbrž zvlášť vyhodnoceny.

Nejprve nainstalujeme modul Smart::Comments.

```
$ cpan Smart::Comments
```

Aktivace režimu Smart::Comments

Abychom mohli využívat výhod, které modul Smart::Comments skýtá, stačí na začátek programu přidat následující řádek.

```
use Smart::Comments;
```

Jakmile program odladíme, budeme pravděpodobně chtít režim vypnout. Zde je nejlepším řešením právě uvedený řádek pouze zakomentovat.

Syntaxe speciálních komentářů

Všechny komentáře, které mají být dále nějakým způsobem vyhodnocovány mají následující tvar.

```
### příkaz
#### příkaz
##### příkaz
```

Tyto příkazy mají za následek vypsání požadovaných dat na STDERR.

Křížků na začátku komentáře může být libovolně mnoho, nejméně však tři. Jejich počet dělí příkazy do skupin, které lze podle potřeby aktivovat nebo deaktivovat.

To, které skupiny příkazů budou právě aktivní se určuje podle seznamu parametrů předanému příkazu use. Například tento příkaz ovlivní komentáře tak, že aktivuje pouze příkazy uvedené po třech nebo čtyřech křížcích.

```
use Smart::Comments "###", "####";
```

Ostatní (například ty, které začínají pěti křížky) interpretovány nebudou.

Výpis textu

Nejjednodušší úkon, který nám provede Smart::Comments je výpis textu. Jako text se chápe každý příkaz za křížky, který končí třemi tečkami. Zkusme spustit následující kód.

```
use Smart::Comments;
...
### Pripravuji testovani...
...
```

Na výstupu na STDERR uvidíme následující.

```
### Pripravuji testovani...
```

Na tomto místě je zajímavé se znovu zamyslet nad tím, že komentář skutečně vypisuje data. Takové užití je možná neočekávané, avšak účelné, zejména, jak uvidíme dále, ve spojitosti s dalšími druhy příkazů.

V textu můžeme použít i speciální proměnné.

- <now> - vypíše místo sebe datum a čas
- <here> - vypíše místo sebe jméno souboru a řádek

Zkusme tedy do programu zapracovat následující řádek.

```
### [<now>] <here> -- Pripravuji testovani...
```

Ve výstupu bychom měli vidět, že se skutečně proměnné nahradily.

```
### [Wed Apr 14 02:12:44 2010] "pokus.pl", line 8 -- Pripravuji testovani...
```

Výpis proměnných

Další věc, kterou lze se Smart::Comments dělat, je výpis aktuální hodnoty nějakého výrazu. Napišme tedy za křížky nějaký výraz. Příklad úseku zdrojového kódu je zde.

```
use Smart::Comments;
$u = 7;
### 5 * 6
### $u
```

Tento kód vytiskne po spuštění programu na STDERR následující.

```
### 5 * 6: 30
### $u: 7
```

Výrazy s popisky

Příkaz tvaru *popisek : výraz* tiskne k hodnotě výrazu vlastní popisek. Uveďme příklad.

```
$pi=$namereny_obvod_kruhu/$namereny_prumer_kruhu;
### Ocekavano: 3.14159265
### Nase aproximace: $pi
```

Výsledkem po spuštění bude tento výstup.

```
### Ocekavano: '3.14159265'
### Nase aproximace: '3.14245681'
```

Kontrola výrazů

Klíčové slovo require (resp. jeho synonyma ensure, insist, assert) vyhodnocuje logický výraz. Očekáváno je, že jeho výsledkem bude pravda. Pokud tomu tak není, bude volána funkce die.

Vložme tedy někam do zdrojového kódu tento řádek.

```
### require: $rok > 2008
```

Je-li v proměnné $rok skutečně hodnota větší než 2008, vše je v pořádku a nedostaneme ani žádný výstup. V opačném případě ale už vynadáno dostaneme.

```
### $rok > 2008 was not true at pokus.pl line 9.
###     $rok was: 2001
```

Ukažme si ještě složitější příklad. Změníme logický výraz.

```
### require: $rok > 2008 or ($rok == 2008 and $mesic > 6)
```

Jak si s ním Smart::Comments poradí? V případě nepravdy nám vypíše hodnoty všech proměnných, které byly ve výrazu použité.

```
### $rok > 2008 or ($rok=2008  and $mesic > 6) was not true at comm.pl line 10.
###     $rok was: 2008
###     $mesic was: 5
```

Zajímavé je, že se zde samo rozpozná, které proměnné ve výrazu figurují. Kdybychom nebyli v kometáři, museli bychom s obdobným problémem postupovat daleko složitěji (stačí se podívat například na rozdíl mezi funkcemi ok a is z modulu Test::More), protože takový výraz by se vyhodnotil, aniž bychom s ním stihli cokoliv udělat.

Použijeme-li místo require (případně jeho synonym) klíčové slovo check, confirmnebo verify, bude v případě neúspěchu místo die volána funkce warn. Nezastavíme tedy program, ale jen způsobíme varování.

Animovaný progresbar

Perl obsahuje několik druhů cyklů. V případě, že právě používáme for, foreach, whilenebo until, můžeme k němu vytvořit animovaný progresbar, nebo-li měnící se komentář.

Pokud neprovádíme nějaké složité výpočty nebo neprovádíme velké množství iterací, cykly většinou zaberou jen zlomky sekundy. V takovém případě bude jakékoliv animování nepostřehnutelné.

Trvá-li však výpočet dlouho, můžeme progresbar vyzkoušet. Vložme do programu nějaký cyklus a za ním uveďme komentář následujícího tvaru.

```
for (1..1_000_000) {      ### pracuji...     hotovo
    1+1;
}
```

Říká se, že milion nic zabije slona. Pravdivost tohoto tvrzení potvrzuje i předchozí kód. Na běžném počítači jeho provedení několik sekund trvá. Díky tomu progressbar snadno zaznamenáme. Uvidíme řádek, na kterém budou přibývat s postupujícími iteracemi tečky. Po pár set tisících iterací bude progressbar vypadat takto.

```
pracuji.....                hotovo
```

Komu se nelíbí tečky, může použít jiný symbol. Zde je několik možností.

```
### pracuji:::     hotovo
### pracuji===     hotovo
### pracuji|||     hotovo
### pracuji[]      hotovo
### pracuji{}      hotovo
### pracuji===>    hotovo
### pracuji [===>   ] hotovo
```

Předposlední komentář na konec progressbaru přidává symbol >. Poslední navíc umísťuje celý progressbar do závorek, kde je o něco patrnější.

Možností je ještě více. Koho zajímají, může buď experimentovat nebo se podívat do dokumentace.

Ještě zajímavější to můžeme udělat, když přidáme počet hotových procent. Změníme náš kód na následující.

```
for (1..1_000_000) {      ### pracuji [===>[%]   ] hotovo
    1+1;
}
```

Pak bude výstup zhruba v polovině iterací vypadat následovně.

```
pracuji [.......[63%]       ] hotovo
```

U cyklů, kde neznáme předem počet iterací (například while) se místo údaje o procentech zobrazí počet hotových iterací.

Časový progresbar

U cyklu for progressbar mimo jiné odhaduje, jak dlouho bude ještě trvat. Měří se zde čas a počítá se průměrná délka iterace. Vyzkoušejme následující úsek.

```
for (1..15) {     ### prubeh [===[%]    ]
    sleep 2;
}
```

Po chvíli uvidíme, kolik sekund nám zbývá do konce cyklu.

```
prubeh [====[35%]              ]  (about 20 seconds remaining)
```

Perl (107) - Moose - moderní objektový systém

Standardní objektový systém Perlu 5 dnes je již poměrně zastaralý a proto by mohlo být zajímavé poohlédnout se po alternatovách. Tou nejznámější je Moose. Článek je úvodem do studia tohoto tématu a přináší základní myšlenky práce s Moose.

Perl je jazyk, umožňující objektově-orientovaný způsob programování již od verze 5.000 z roku 1994. I proto má velmi daleko k dnešním objektově-orientovaným jazykům. Cílem Moose (v angličtině označuje moose losa nebo někdy jeden z jeho druhů) je přiblížení Perlu moderním objektovým jazykům. Moose je postaven na základech Class::MOP.

Moose samotný neumí o tolik více než čistý Perl. Za uživatele však odvede spoustu práce a výsledný kód je kratší a přehlednější. V této minisérii se například vůbec nesetkáme s funkcí bless, budeme psát podstatně méně metod než dosud, budeme srozumitelněji deklarovat atributy a neuvidíme žádné odkazy.

Před studiem Moose je vhodné se alespoň zběžně podívat, jak funguje standardní objektový systém v Perlu. Bude pak snažší pochopit práci s Moose.

Moose je řešením objektového systému pro Perl 5. V připravovaném Perlu 6 se počítá s novým modelem, kde budou pro OOP nová klíčová slova.

Instalace

Jediná věc, která je potřebná před používáním Moose je stažení a nainstalování modulu z CPANu. To tradičně zajistí následující příkaz.

```
$ cpan Moose
```

Moose a Perl

Moose exportuje funkce, které můžeme používat podobně jako klíčová slova jazyka (a tak je také budeme dále nazývat). Za chvíli se s některými z nich seznámíme.

Vytvoření objektu

Zopakujme si nejprve, jak jsme vytvářeli objekty v čistém Perlu. Pokaždé jsme definovali nějaký balíček a v něm jsme vytvořili konstruktor - tedy metodu s názvem new. Kód začátku objektového modulu mohl vypadat takto.

```
package PoStaru;

sub new {
my $trida = shift;
my $objekt = {};
bless $objekt, "PoStaru";
return $objekt;
}
```

Co když teď použijeme Moose? Zde podobnou věc uděláme snadno.

```
package PomociMoose;
use Moose;
```

Je vidět, že kódu je v této fázi u Moose mnohem méně. Konkrétně stačí dva řádky. Moose si již vytvoří metodu new sám (přesněji řečeno ji zdědí z Moose::Object, ze které nyní budou dědit všechny naše třídy). Vzápětí uvidíme, jak se s ní pracuje. Poznamenejme také, že Moose navíc automaticky zapne režimy warnings a strict.

Atributy a jednoduché metody

V čistém Perlu jsou atributy většinou uloženy v hashi. To je jednoduché a poměrně efektivní. Problém později bývá v tom, že je obvykle třeba definovat řadu metod, které parametry například pouze vracejí nebo přenastavují. Samotný hash to neobstará, a tak je na řadě objektový systém.

Tady by se většina uživatelů asi zeptala, zda to nejde jednodušeji a neexistuje něco jako automatické vytvoření takových metod. Standardní objektový systém jejich automatické vytvoření neumožňuje. Avšak s použitím externích modulů to samozřejmě možné je.

Správa atributů pomocí modulu Class::Accessor

Můžeme kupříkladu využít modulu Class::Accessor, který nám snadno vytvoří potřebné metody a atributy. Podívejme se na jednoduchý příklad. Stačí nastavit Class::Accessor jako rodiče naší třídy a pomocí metody mk_accessors vytvořit atributy a metody, které se použijí k nastavení nebo získání hodnot. Díky tomu můžeme definovat všechny metody a atributy jediným řádkem.

```
package MojeTrida;
use base qw(Class::Accessor);

MojeTrida->mk_accessors(qw(nazev cena info));

$sluzba = MojeTrida->new();

$sluzba->nazev("strihani vlasu");
$sluzba->cena(100);
$sluzba->info("bla bla bla");

print $sluzba->get("nazev");
print $sluzba->get(qw(nazev cena info));
```

Co vše Class::Accessor dokáže, se lze dočíst v dokumentaci. Je toho dost. Stejně jako Class::Accessor ale dokáže s atributy pracovat Moose.

Atributy pomocí Moose, typová kontrola

Nyní se už budeme věnovat systému Moose. Podívejme se nejprve, jak pracuje s atributy ten. Atribut se vytváří pomocí klíčového slova has, kterému předáme dvě hodnoty. Těmi jsou požadované jméno atributu a hash s danou strukturou. Touto strukturou se budeme ještě dále zabývat.

Zde je pro začátek jednoduchý příklad, který vytvoří třídu a tři atributy.

```
package MojeTrida;
use Moose;

has "nazev" => (
is => "rw",
isa => "Str"
);

has "cena" => (
is => "rw",
isa => "Int"
);

has "info" => (
is => "rw",
isa => "Str"
);
```

Hash parametrů může obsahovat řadu různých položek. My jsme zatím použili pouze dvě. is určuje, zda má být umožněno do atributu zapisovat (tj. read-write; v takovém případě se nastavuje hodnota rw) nebo nikoliv (tj. read-only; píše se jako ro).

Pomocí isa se zapíná typovou kontrolu. Lze použít například hodnoty Str, Num, Bool, HashRef. Pokud hodnotou atributu má být objekt, můžeme zde vepsat i jméno třídy. To bychom využili například pro atribut uchovávající datum a čas posledního přístupu k naší stránce.

```
has "posledni_pristup" => (
    is => "rw",
    isa => "DateTime"
);
```

Pro další možnosti okolo typů jako třeba co všechno za typy lze zadávat, jak definovat nové typy nebo subtypy pomocí regulárních výrazů, je užitečné nahlédnout do dokumentace.

Je dobré si uvědomit, že has je obyčejná funkce. Proto s ni podle toho můžeme zacházet. Předchozí kód bychom mohli přepsat nějak takto.

```
for (qw(nazev cena info)) {
    has $_ => (is => "rw", isa => zjisti_typ($_));
}
```

Také je možné nastavit více atributů pomocí jednoho volání has. Zde je opět příklad.

```
has ["nazev", "info"] => (is => "rw", isa => "Str");
```

Perl (108) - Moose - základní vlastnosti

Moose poskytuje automatické vytváření některých druhů metod. Díky tomu mohou být naše programy podstatně přehlednější. Také nahlédneme, jak se v Moose řeší dědičnost, delegování a k čemu jsou tzv. modifikátory.

Posledně jsme zabývali vytvářením nových atributů. Moose atributy definuje pomocí klíčového slova has. Modifikacemi ve volání lze provádět různé činnosti jako například vytváření nových metod.

V klasickém OOP jsme si museli každou metodu napsat sami, ačkoliv byla sebetriviálnější. V aplikacích jsme například často potřebovali napsat metodu, která pouze vrací nějaký atribut. To je úkol, o kterém by se asi nezasvěcení domnívali, že ho řešit nemusí, protože ho zvládne objektový systém.

My jsme byli v klasickém OOP schopni přistupovat k atributům přímo, avšak tento postup jsme zavrhli. Pokud jsme problém chtěli řešit pomocí metod, museli jsme si je napsat sami.

Existují moduly, které tyto rutinní činnosti udělají za nás (a jak dnes uvidíme dále, nemusí jít pouze o čtení atributů). Moose mezi ně samozřejmě patří také. Představme si tedy metody, které Moose dokáže automaticky vygenerovat.

Metody pro čtení a zápis atributů

Již jsme si řekli, že jedním ze základních problémů v téměř každé třídě je to, jak atributům nastavovat hodnoty a jak je číst, aniž bychom museli psát příslušné metody.

Zde je jedna možnost. Využijeme již předdefinovaných metod, které mají stejný název jako atribut.

```
$a = MojeTrida->new;
$a->nazev("ahoj");
print $a->nazev; #tiskne ahoj
```

O nic jsme se v souvislosti s těmito metodami nemuseli starat.

Totéž lze zařídit také pomocí metod reader a writer. Pomocí nich si můžeme pro metody určit vlastní jména. Příklad změny definice atributu nazev tak, abychom získali metody get_nazev pro přečtení hodnoty a set_nazev pro nastavení hodnoty je zde.

```
has "nazev" => (
    is => "rw",
    isa => "Str",
    reader => "get_nazev",
    writer => "set_nazev"
);
```

Nyní můžeme vytvořit objekt a pracovat s ním.

```
$objekt = MojeTrida->new;
$objekt->set_nazev("ahoj");
print $a->get_nazev; #tiskne ahoj
```

Další automaticky vytvářené metody

Pomocí přidání položky default=>"hodnota" můžeme nastavit implicitní hodnotu atributu. To znamená, že nastavíme atribut na nějakou výchozí hodnotu a ta bude platná, dokud ji uživatel nezmění.

```
has "nazev" => (
    ...
    default=>"hodnota"
);
```

Místo implicitní hodnoty lze nastavit také volání anonymního podprogramu (nebo dokonce i neanonymního, avšak to už musíme použít místo default položku builder). Pak bude hodnotou atributu vrácená hodnota. Taktéž můžeme pomocí lazy=>1 udělat to, aby se implicitní hodnota hodnota nastavila až před prvním čtením.

Pokud tedy vytvoříme atribut s implicitní hodnotou, a pak vytvoříme objekt, můžeme vidět, že jeho daný atribut je již nastaven.

```
my $objekt = MojeTrida->new();
print $objekt->get_nazev; #tiskne "hodnota"
```

Metodou určenou pomocí clearer smažeme atribut. Naopak pomocí predicatemůžeme nastavit detektor atributu. Uveďme si ještě jednou, jak by vypadalo volání has.

```
has "nazev" => (
    ...
    clearer=>"unset_nazev",
    predicate=>"je_nazev"
);
```

Pomocí require=>1 lze nastavit atribut jako povinný.

Lze vytvořit trigger pomocí trigger=>\&spoustec. Trigger spoustec je podprogram, který se provede po každém přenastavení hodnoty atributu.

Ručně vytvářené metody

Ostatní metody lze vytvářet stejně jako v klasickém OOP pomocí klíčového slova sub.

Dědičnost

Při práci s Moose je lepší nepoužívat pragmu base. Zavádí se zde nově klíčové slovo extends, které použijeme v třídách upravujících nějakou starší třídu (tj. nastavuje nadtřídu). Použití je intuitivní.

```
package Clovek;
use Moose;
extends "Organizmus";
```

Pak můžeme upravovat atributy.

```
has "+pocet_nohou" => (
default => 2,
lazy    => 1
);
```

Modifikátory metod

Kód, který se spustí v souvislosti s voláním metody, se nazývá modifikátor. Je několik druhů modifikátorů. Nejprve si představme modifikátory before, after, a around.

Například následující modifikátor zálohuje data před voláním metody uprav_data.

```
before "uprav_data" => sub {
zalohuj_data();
};
```

V případě, že použijeme více modifikátorů, volají se v následujícím pořadí.

- before
- around
- samotná metoda
- around
- after

Volání dědící metody - obrácené dědění

Dále existuje modifikátor augment a funkce inner, které umožňují něco jako obrácené dědění.Funguje to tak, že v nadtřídě voláme jistou metodu podtřídy, která je v ní definována pro tento účel (děje se tak pomocí augment). Pokud nadtřída vytvoří nějakou metodu, pak ji tedy můžeme pomocí augment modifikovat. Podívejme se na jednoduchý příklad.

```
package HTMLSablona;
sub generator {
return "<html><head></head><body>".inner()."</body></html>";
}

package MujWeb;
extends "HTMLSablona";
augment "generator" => sub {
return "<h1>Ahoj!</h1>";
}
```

Překrývání metod

Pomocí modifikátoru override lze překrýt hierarchicky vyšší metodu. Pomocí funkce super lze uvnitř override zavolat metodu z nadtřídy, od které se dědí (funguje podobně jako SUPER).

Podívejme se na příklad. Překryjeme metodu specificke_vlastnosti z třídy Clovek. Ta vrací nějaký řetězec (například "dvě ruce, dvě nohy, schopnost řeči"). Na jeho konec přidáme specifické vlastnosti pro nějakou blíže určenou rasu. Pomocí funkce super zde získáme výstup hierarchivcky vyšší metody. Výsledkem bude zkombinování těchto řetězců.

```
package MongoloidniTypCloveka;
extends "Clovek";

override "specificke_vlastnosti" => sub {
my $self = shift;
return super().", tmave vlasy, sikme oci".$self->poznavaci_znameni;
};

#zde mohou byt definovany nove atributy
```

Delegace

Delegování je něco jako vytváření virtuálních metod. Pomocí delegování lze například zpřehlednit některá volání.

Při definici atributu lze použít v hashi klíč handles. Tamu můžeme předat anonymní pole nebo hash. Zde je příklad použití, díky kterému budeme moci psát $clovek->krestni místo $clovek->jmeno->krestni.

```
has "jmeno" => (
  is      => "rw",
 isa      => "Jmeno",
handles => [qw(krestni prijmeni)],
);
```

Zde je zajímavější příklad, převzatý z dokumentace, ve kterém se pracuje i s parametry. Místo $r->request->header("UserAgent", "MyClient") voláme jen $r>set_user_agent("MyClient").

```
has request => (
  is      => "ro"
 isa      => "HTTP::Request",
  handles => {
set_user_agent => [ header => "UserAgent" ]
}
)
```

Perl (109) - Moose - role

Role jsou dalším přiblížením objektově orientovaného programování reálnému světu.

Pro porozumnění rolím si na začátku představme nějakého konkrétního člověka. Ten může být zároveň zaměstnancem stavební firmy, sběratelem mincí, hráčem lakrosového klubu nebo návštěvníkem divadla. To můžeme nazvat jeho rolemi. Je dobré si uvědomit, že zaměstnanec, sběratel, hráč i návštěvník jsou ve skutečnosti jediná osoba. Přitom kontakt s okolím tohoto člověka se liší v závislosti na aktuální roli. Na základě této přirozené myšlenky fungují role v objektově orientovaném programování.
Role je fyzicky soubor metod (resp. balík, který obsahuje tyto metody), které obvykle reprezentují nějaké vlastnosti. Je to něco podobného jako třída, avšak v několika věcech se tyto objekty liší.
Jedna role může spojovat několik navzájem jen vzdáleně souvisejících tříd tak, že sdílejí nějakou část chování. Všechny metody, které jsou uvnitř role definovány potom mohou být "používány" konkrétními (a samozřejmě předem určenými) třídami a chovat se jakoby byly metodami nebo atributy oněch tříd (a tedy jsou mimo jiné také děděny).
Abychom se vyhnuli omylům a zavedli jasnou terminologii, budeme pro popsané spojení role a třídy používat termín "třída pohlcuje roli" nebo "role je pohlcena třídou" (v anglicky psaných textech se vyskytuje "roles are consumed by class").
Role verzus třídy
Je tedy role třídou? Už jsme řekli, že nikoliv. Rozdíl mezi rolí a třídou je v tom, že nevytváříme instance od role. Dalším rozdílem je, že od role nedědíme. Děděny mohou být pouze její vlastnosti prostřednictvím třídy, která ji pohltila (jinými slovy podtřída třídy, která pohltila nějakou roli, získá tuto roli také).
První příklad
Co mají společného třídy Okno a Dalnice? Třeba to, že se dají opravit. Mají-li tedy obě nějakou metodu oprava, můžeme na ni navázat akce, které se provedou v souvislosti s opravou.
Předpokládejme, že každá opravitelná věc bude mít atribut potrebuje_opravu uchovávající pravdu nebo nepravdu. Proto ho naše role zadefinuje. Dále zadefinujeme nějakou metodu, která danou věc opraví. Tu nazveme oprava. Zde je kód takové role.

```
package LzeOpravit;
use Moose::Role;

has "potrebuje_opravu" => (
is=>"rw",
isa => "Bool"
);

sub oprava {
my $self=shift;
print "opravuji nejakou obecnou vec...\n";
$self->potrebuje_opravu(0);
}
```

Roli bychom tedy měli alespoň v náznaku napsanou. Jak ji pohltit třídou? Pro tento účel existuje klíčové slovo with, které použijeme v třídách Okno a Dalnice. Pro jednoduchost nebudeme zavádět žádné nové parametry a naše třída bude tedy mít pouze tři řádky.

```
package Okno;
use Moose;
with "LzeOpravit";
```

Podobně bychom mohli vytvořit například třídu Dalnice.
Podívejme se nyní na to, jak se třída Okno používá. Příslušné objekty mají vlastní atribut a lze je opravovat.

```
my $stresni_okno = Okno->new("potrebuje_opravu"=>1);
print $stresni_okno->potrebuje_opravu; #tiskne 1
print $stresni_okno->oprava;          #tiskne "opravuji nejakou obecnou vec..."
print $stresni_okno->potrebuje_opravu; #tiskne 0
```

Detekce role
Uveďme ještě, že pomocí metody does volané nad objektem můžeme zjistit, zda třída pohlcuje danou roli. Je to analogie metody isa pro dědičnost.

```
print $stresni_okno->does("LzeOpravit"); #tiskne 1
```

Požadavky rolí na třídy
Další vlastností rolí je, že mohou po třídách, které je pohlcují, vyžadovat definici některých konkétních metod nebo atributů. Protože oprava dané věci je většinou naprosto konkrétní věc, bude naše role chtít, aby si třída sama definovala metodu oprava (oprava okna a dálnice jsou dvě docela odlišné věci). K tomu použijeme klíčové slovo require. V důsledku můžeme z naší role metodu oprava odstranit. Nově tedy bude vypadat takto.

```
package LzeOpravit;
use Moose::Role;

requires "oprava";

has "potrebuje_opravu" => (
is=>"rw",
isa => "Bool"
);
```

Nyní budeme muset metodu oprava vložit do všech tříd pohlcujících naši roli. V opačném případě bychom byli svědky chybového hlášení.

```
'LzeOpravit' requires the method 'oprava' to be implemented by 'Okno'
```

Metodu oprava můžeme upravit na míru pro okno. Třída Okno bude nově vypadat takto.

```
package Okno;
use Moose;
with "LzeOpravit";

sub oprava {
my $self=shift;
print "opravuji okno...\n";
$self->potrebuje_opravu(0);
}
```

Stejně tak můžeme definovat další třídy.

```
package Dalnice;
use Moose;
with "LzeOpravit";

sub oprava {
my $self=shift;
print "opravuji dalnici...\n";
$self->potrebuje_opravu(0);
}
```

Nyní lze vesele vytvářet objekty typu Okno nebo Dalnice a opravovat.

Modifikátory

Pomocí modifikátorů můžeme provádět různé akce v souvislosti s voláním nějaké metody. Podívejme se na roli, která vykoná během opravy nějaké vedlejší činnosti.

```
package LzeOpravit;
use Moose::Role;
requires "oprava";
has potrebuje_opravu => {isa => "Bool"};

before "oprava" => sub {
my $self=shift;
$self->vycisli_naklady_na_opravu();
}

after "oprava" => sub {
my $self=shift;
$self->uklid_naradi();
}
```

Problémy s kolizemi při pohlcování více rolí

Funkci with lze předat i seznam názvů rolí. V případě, že má více rolí stejnou metodu, dojde ke kolizi. Ty lze řešit tak, že metody vhodně přejmenujeme. To zajistíme pomocí volání with ve speciálním tvaru.

```
with "PrvniRole" => {-alias => {"kolidujici_metoda" => "kolidujici_metoda1", -excludes => "kolidujici_metoda"}},
"DruhaRole" => {-alias => {"kolidujici_metoda" => "kolidujici_metoda2", -excludes => "kolidujici_metoda"}};
```

Konkrétněji, pokud máme člověka hokejistu a zároveň lukostřelce, nastane nám kolize u metody vystrel. Hokejista vystřelí puk a lukostřelec šíp. Pokud pohlcujeme obě role, není volání funkce vystrel jednoznačné. Ve třídě Člověk, která pohlcuje role Hokejista a Lukostřelec tedy přímo ve with podle výše uvedeného návodu metody přejmenujeme. Parametrem -alias tedy vytvoříme kopie k oběma funkcím vystrel a zároveň je smažeme parametrem -excludes.

```
with "Hokejista"   => {-alias => {"vystrel" => "vystrel_puk"}, -excludes => "vystrel"},
"Lukostřelec" => {-alias => {"vystrel" => "vystrel_sip"}, -excludes => "vystrel"};
```

Více o rolích a věcech okolo se lze dočíst v dokumentaci.

V Moose máme speciální aplikační rozhraní, díky němuž jsme schopni v jistém ohledu modifikovat samotný objektový systém.

Pomocí rozhraní meta API lze nastavovat základní vlastnosti objektového systému a ovlivňovat věci jako chování metod, atributů a tříd.

Získání metadat

Základní metodou pro práci s vlastnostmi objektového systému je meta, která zpřístupňuje metadata dané třídy. Zde je příklad užití. Takto lze získat seznam metod nebo atributů s vlastnostmi.

```
$meta = MojeTrida->meta();

for $atribut ($meta->get_all_attributes){
print "Atribut: ", $atribut->name(), " ";
print $atribut->type_constraint->name if $atribut->has_type_constraint;
print "\n";
}

for $method ($meta->get_all_methods){
print $method->name," ";
}
```

Poznamenejme, že namísto MojeTrida na prvním řádku můžeme použít slovo __PACKAGE__, které vyjadřuje jméno aktuálního balíku.

```
$meta = __PACKAGE__->meta;
```

Podobně lze získat metodou linearized_isa seznam rodičovských tříd a pomocí subclasses podtřídy.

Modifikace tříd pomocí metadat

Pro zajímavost uveďme, že třídy lze upravovat přímo pomocí změny metadat. Ukažme si, jak lze přidávat metody a atributy. Máme zde k dispozici metody add_method a add_attribute, kterým předáme příslušné parametry.

```
$meta = __PACKAGE__->meta;
$meta->add_method("nova_metoda" => sub{udelej_neco(@_)});
$meta->add_attribute(
name => "novy_atribut",
is   => "rw",
isa  => "Int"
);
```

Toto chování lze vypnout pomocí metody make_immutable (případně zapnout pomocí make_mutable). V takovém případě bude při pokusu o změnu třídy pomocí metadat vyvolána výjimka.

```
$meta->make_immutable;
```
Fyzická reprezentace atributů

Atribut v Moose je ve skutečnosti objekt typu Moose::Meta::Attribute. To zásadně mění náš pohled na ně. K těmto objektům lze samozřejmě přistupovat. K tomu slouží metoda get_attribute("jméno atributu"). Například objekt reprezentující atribut nazev získáme následovně.

```
$nazev = $meta->get_attribute("nazev")
```

Tento objekt má k dispozici několik metod. Pro příklad uvedeme metodu, která vrátí název datového typu.

```
print $nazev->type_constraint;
```

Abychom lépe nahlédli na strukturu, uvědomme si, že místo předchozích volání lze typ získat jedním příkazem.

```
print MojeTrida->meta->get_attribute("nazev")->type_constraint;
```

Modifikace fyzické reprezentace atributů, metaatributy

Celá situace je o to zajímavější, že strukturu třídy reprezentující atribut lze měnit. Můžeme například přidávat atributy k atributům. Jinými slovy chceme převzít řízení toho, jaký vliv na chování bude mít hash předávaný při vytváření atributu. Ukážeme si, jak k atributu přidat popisek. Tedy chceme v řídícím hashi vytvořit nový klíč popisek. Názorněji, požadujeme, aby fungoval následující kód.

```
package MojeTrida::Main;
use Moose;

has "atribut_s_popiskem" => (
        is       => "rw",
        isa      => "Str",
    popisek   => "Toto je meta-popisek atributu atribut_s_popiskem",
    );
```

Je dobré si zde uvědomit, že zde opravdu má smysl chtít vytvářet atributy. Popisek je totiž vlastností našeho atributu a proto by nemělo smysl ho uvádět kamkoliv jinam (například do metod naší třídy).

Této funkcionality se dosáhne tak, že definujeme vlastní metatřídu. Každý atribut je objekt typu Moose::Meta::Attribute. My ji potřebujeme přetížit. To znamená, že vytvoříme třídu, kterou nazveme MojeTrida::Meta::Attribute::Popisek, která bude dědit od Moose::Meta::Attribute. V této nové třídě definujeme nový atribut. Zároveň bude třeba někde uvést, že se daný atribut má řídit naší třídou MojeTrida::Meta::Attribute::Popisek (nikoliv standardní třídou Moose::Meta::Attribute).

Poslední požadavek (nastavení metatřídy) se vyřeší přidáním klíče metaclass. Definici našeho atributu tedy lehce upravíme tím, že přidáme jeden řádek.

```
has "atribut_s_popiskem" => (
metaclass => "MojeTrida::Meta::Attribute::Popisek",
        is       => "rw",
        isa      => "Str",
    popisek   => "Toto je meta-popisek atributu atribut_s_popiskem",
    );
```

Nyní již Moose ví, že chceme přetěžovat. Zbývá tedy pouze vytvořit třídu MojeTrida::Meta::Attribute::Popisek, která zdědí vše od Moose::Meta::Attribute. Vložíme sem navíc nový atribut popisek. Pomocí predicate=>je_popisek navíc přidáme detektor, protože bude časem potřeba.

```
package MojeTrida::Meta::Attribute::Popisek;
use Moose;
extends "Moose::Meta::Attribute";

has "popisek" => (
        is       => "rw",
        isa      => "Str",
    predicate => "je_popisek"
    );
```

Nyní můžeme standardně pracovat s objekty typu MojeTrida::Main. Podívejme se pro zajímavost, jak získáme text popisku.

```
package main;
print MojeTrida::Main->meta->get_attribute("atribut_s_popiskem")->popisek;
```

Lze sem samozřejmě přistupovat i pomocí objektu.

```
$a = MojeTrida::Main->new(atribut_s_popiskem=>"hodnota");
print $a->meta->get_attribute("atribut_s_popiskem")->popisek;
```

Pro přehlednost napíšeme v MojeTrida::Main ještě metodu metainfo, která nám vytiskne informace o všech atributech a jejich popiscích. Odlišíme tak hodnotu atributu a popisek. Metoda metainfo může vypadat takto.

```
sub metainfo {
my $self = shift;
my $info = "";

for my $a ($self->meta->get_attribute_list){
my $atribut = $self->meta->get_attribute($a);
        $info .= $atribut->name. ":\n";
    $info .= "  popisek: ".$atribut->popisek."\n"
if $atribut->isa("MojeTrida::Meta::Attribute::Popisek") and $atribut->je_popisek;
    $info .= "  hodnota: ". $self->{$atribut->get_read_method}. "\n";
    }

    return $info;
    }
```

V našem příkladu bychom po zavolání print $a->metainfo; obdrželi následující výstup.

```
atribut_s_popiskem:
popisek: Toto je meta-popisek atributu atribut_s_popiskem
hodnota: hodnota
```

Více popisků pro atribut

Budeme-li chtít pro každý atribut vytvořit více různých atributů, pak se stává naznačený postup nečitelný - lze totiž sice podobně jako pro popisek vytvářet další a další podtřídy, avšak po jisté době se každý zamyslí, zda-li je kombinování nesourodých tříd opravdu nezbytné.

Řešení nabízejí role. Klíčové slovo has totiž přijímá jako parametr mimo jiné i traits=>[qw(role1 role2 ...)]. Trait je speciální role, která se použije na objekt a od běžných rolí se jinak ničím neliší.

Modifikujme tedy předchozí příklad tak, aby dělal to samé, ale využil přitom rolí.

Nejprve definujme parametry pro atribut_s_popiskem. Zde vložíme klíč traits, do kterého nyní zadáme pouze jednoprvkový seznam (ačkoliv bychom zde mohli použít více rolí).

```
has "atribut_s_popiskem" => (
traits    => ["MojeTrida::Meta::Attribute::Trait::Popisek"],
is        => "rw",
isa       => "Str",
popisek   => "Toto je meta-popisek atributu atribut_s_popiskem",
);
```

Podtřídu MojeTrida::Meta::Attribute::Popisek nahradíme naši novou rolí MojeTrida::Meta::Attribute::Trait::Popisek. Celkem bude vypadat následovně.

```
package MojeTrida::Meta::Attribute::Trait::Popisek;
use Moose::Role;

has "popisek" => (
is        => "rw",
isa       => "Str",
predicate => "je_popisek"
);
```

V metodě metainfo pouze zaměníme volání metody isa na volání does. To proto, že metoda isa je vztah pro podtřídu, avšak nyní máme roli a tak použijeme does.

Vše ostatní může zůstat stejné. Nyní tedy pro přidání dalších atributů stačí do traits přidat MojeTrida::Meta::Attribute::Trait::Cokoliv a tuto roli implementovat podobně jako jsme to dělali u popisku.

Rozšíření Moose - MooseX

Jednou z aplikací meta API je též tzv. MooseX. Díky MooseX si každý může napsat vlastní Moose rozšíření. Řada rozšíření je již k dispozici na CPAN. Návod a možnosti pro rozšiřování je v dokumentaci.

Perl (111) - Pokročilá práce se seznamy

Seznámíme se s několika zajímavými funkcemi, které jsou k dispozici v externích modulech. Zejména se zaměříme na efektivnější práci při operování se seznamem jako celkem, složitější cyklení a na permutování.

Když jsme se v minulosti zabývali seznamy, nevyčerpali jsme plně všechny nástroje, které jsou nám k dispozici. Tento díl se to pokusí napravit. Vedle standardních funkcí ze základní distribuce Perlu budeme používat ještě navíc modul Algorithm::Loops. Ten tedy stáhneme z archivu CPAN a nainstalujeme.

Skalární operace provedená na pole po složkách

Ačkoliv jsme velkou část často používaných funkcí ze základní distribuce Perlu představili v začátku toho seriálu, stále jsme se neseznámili s veledůležitou funkcí map. Ta dokáže vzít pole, na každý jeho prvek aplikovat nějakou funkci a vytvořit tak pole nové.

Typickou úlohou pro map je konverze všech prvků seznamu řetězců z malých písmen na velké. Žádná běžná funkce, která by toto řešila neexistuje. Avšak máme zde funkci uc, která je schopna provést změnu velikosti písmen ve skalárním kontextu. Pomocí map ji provedeme po složkách.

```
@male=qw(abc def ghi);
print @velke = map(uc, @male);
```

Na operaci lze samozřejmě použít i funkci napsanou uživatelem. Příkladem nechť je následující kód, kde funkce map pro každý prvek $_ z pole @cisla vrátí výraz, který vznikne aplikací uvedené funkce na proměnnou $_ (tedy prvek pole @cisla). Do pole @dvojnasobky tedy v našem případě uložíme hodnoty 2, 4, 6, 8, 10, 12, 14, 16, 18, 20.

```
@cisla=1..10;
sub zdvojnasob {2*$_};
@dvojnasobky=map(zdvojnasob, @cisla);
```

Jako první argument funkce map lze zadat i samotný výraz z definice funkce zdvojnasob. Následující kód tak docílí téhož, aniž bychom definovali nový podprogram.

```
@cisla=1..10;
@dvojnasobky=map(2*$_, @cisla);
```

Poznamenejme, že v tomto výrazu nemusí $_ vůbec figurovat. Pro vytvoření pole pravdivých hodnot o stejné délce jako původní bychom tedy mohli použít i jeden z následujících příkazů.

```
print map(1, @puvodni);
print map(defined, @puvodni);
```

Chceme-li zvolit jinou datovou strukturu, do které se mají data ukládat, můžeme udělat jednoduchý trik a nevracet jen skalární hodnotu. Máme-li nějakou množinu klíčů, ke kterým nalezneme hodnoty, budeme je pravděpodobně chtít nějak rozumně uložit - tedy do hashe. V prvním argumentu funkce map tedy pro každý klíč nalezneme hodnotu pomocí nějaké funkce a vrátíme jak tento klíč, tak i získanou hodnotu. Návod dává tento příkaz.

```
%hash = map{$_ => najdi_hodnotu($_)} @klice;
```

Poznamenejme, že varianta map používající složené závorky může občas dělat problémy. To proto, že složené závorky mohou indikovat blok i anonymní hash. To, jak se vyhnout problémům je obsáhleji diskutováno v dokumentaci.

map však není schopen řešit úplně všechno tak, jak bychom možná na první pohled očekávali. Chceme-li aplikovat na seznam operace tr/// nebo s///, pak je vhodné použít funkci Filter. Funkce map by zde fungovala jinak - zapsala by do nového pole vždy pravdivou nebo nepravdivou hodnotu podle toho, zda došlo k nahrazení. Tak se děje proto, že se fakticky vrací hodnota výrazu ($neco =~ tr///), nikoliv hodnota $neco.

Funkce Filter je k dispozici v modulu Algorithm::Loops. Je proto třeba tento modul zavést. Filter se používá podobně jako map. Nikoliv však stejně. Například vyžaduje použití složených závorek.

Příkladem je nahrazení všech slov kurs slovem kurz v prvcích pole @texty.

```
use Algorithm::Loops qw(Filter);
@zmenene = Filter{s/kurs/kurz/} @texty;
```

Jiným příkladem je modifikace čtění ze vstupu tak, aby byly ignorovány bílá místa na konci řádků.

```
use Algorithm::Loops qw(Filter);
@radky = Filter {s/\n+$//} <VSTUP>;
```

Analogie map pro celá pole

Funkce MapCar aplikuje danou funkci po složkách na pole polí. Používá se například na práci s maticemi. MapCar příjímá jako parametr anonymní podprogram a seznam odkazů na pole. Tento podprogram je aplikován na pole, která vzniknou seskupením prvních, druhých atd. prvků v předaných seznamech.

```
print MapCar sub{"@_\n"}, [1, 2, 3], [4, 5, 6], [7, 8];
```

Tento příklad vytiskne na výstup následující matici.

```
1 4 7
2 5 8
3 6
```

Existují další varianty příkazu MapCar. Pro bližší informace nahlédněte do dokumentace. Zde pro představu z dalšího uvedeme jen jeden příklad. Transponovanou matici získáme následujícím příkazem.

```
$transpozice = MapCarU {[@_]} @$matice;
```

Vnořené cykly

Co když potřebujeme vytvořit kód, který bude vypadat nějak takto?

```
for $i (...){
...
for $j (...){
...
for $k (...){
...
for $l (...){
...
}
}
}
}
```

Funkce NestedLoops řeší problém libovolně hlubokého vnoření cyklů do sebe. Uvedenou situaci tak lze vyřešit poměrně elegantně.

NestedLoops přijímá tři parametry. Není ale třeba uvádět všechny. Prvním parametrem je odkaz na pole, které obsahuje pro každý cyklus jeden prvek (od vnějšího cyklu po vnitřní). Zde je určeno, přes které hodnoty budeme iterovat. Druhým parametrem je odkaz na hash přepínačů, pomocí kterých se nastavují pokročilejší vlastnosti. Na závěr se udává odkaz na podprogram, který se má provést.

Podívejme se na příklad, kde máme dva cykly. Vnější jde od 1 do 2 a vnitřní od 3 do 4. Budeme uvnitř nich tisknout aktuální hodnotu počítadel obou těchto cyklů.

```
NestedLoops(
[([1..2], [3..4])],
sub {print "@_\n"}
);
```

V hashi uvedeném ve druhém parametru můžeme použít zápis OnlyWhen => &test. Takto bude podprogram vykonán pouze tehdy, jsou-li splněny příslušné podmínky (test vrátí pravdivou hodnotu). Podívejme se na toto volání.

```
NestedLoops(
[([1..2]) x 3],
{OnlyWhen => \&test},
\&udelej_neco
);
```

Bude mít stejný efekt, jako bychom napsali tento kód.

```
for $i (1..2){
udelej_neco($i) if test($i);
for $j (1..2){
udelej_neco($i, $j) if test($i, $j);
for $k (1..2){
udelej_neco($i, $j, $k) if test($i, $j, $k);
}
}
}
```

Permutace

V modulu Algorithm::Loops je též k dispozici několik funkcí pro práci s permutacemi. Permutace pole je obecně nějaké přeuspořádání prvků v poli. Pokud se hodnoty v poli opakují, pak algoritmus, který nalezne všechny možné permutace není vůbec triviální. Pomocí funkcí NextPermute a NextPermuteNum lze toto všechno zařídit velmi pohodlně.

Funkce NextPermute (resp. NextPermuteNum pro čísla - rozdíl je v tom, že se shodnost prvků seznamů vyhodnocuje pomocí operátoru == nebo eq) se obvykle používá v podmínce cyklu. Stará se o to, abychom v každé iteraci dostali novou permutaci našeho seznamu. Jakmile projdeme všechny permutace, cyklus skončí, neboť NextPermute vrátí nepravdivou hodnotu.

Chceme nyní vypsat všechny permutace prvků 2, 3, 3, 1. Podívejme se na následující příklad.

```
use Algorithm::Loops qw(NextPermuteNum);
@seznam = (2, 3, 3, 1);
do{
print @seznam,"\n";
}while(NextPermuteNum(@seznam))
```

Výstupem jsou následující řádky.

```
2331
3123
```

```
3132
3213
3231
3312
3321
```

Na první pohled vidíme, že něco není správně, protože dokonce žádný z řádků nezačína jedničkou. To je proto, že funkce NextPermuteNum požaduje vzestupně setříděný seznam. To může být zdrojem chyb. Příklad totiž musíme upravit.

```
use Algorithm::Loops qw(NextPermuteNum);
@seznam = sort {$a<=>$b} (2, 3, 3, 1);
do{
print @seznam,"\n";
}while(NextPermuteNum(@seznam))
```

Jestliže se některá hodnota vyskytuje v seznamu vícekrát, pak teoreticky můžeme dostat stejnou permutaci vícekrát. Funkce NextPermuteNum ji ale vrátí vždy jen jednou. Můžeme se o tom přesvědčit v následujícím příkladu.

```
@seznam = (1, 1, 1, 1);
do{
print @seznam,"\n";
}while(NextPermuteNum(@seznam))
```

Seznam má 4 prvky a tedy máme 4!=4*3*2*1=24 permutací. Avšak jakákoliv permutace aplikovaná na náš seznam dá výsledek 1111. Proto tento kód produkuje jen jediný řádek.

Existuje též modul Algorithm::Permute, který umí o něco více a řeší problematiku permutací pomocí objektového rozhraní.

Perl (112) - Práce s PDF

Podívejme se na několik základních postupů, jak lze vytvářet a modifikovat PDF dokumenty.

PDF patří dnes mezi nejrozšířenější formáty pro distribuci textových dokumentů. Jednou z vlastností PDF je to, že je obtížné ho jakkoliv modifikovat. Co kdybychom chtěli rozeslat dopisy stejné až na záhlaví, které bychom vygenerovali pro každý dokument individuálně podle nějaké databáze? Určitě můžeme použít TeX. Ale moc jiných nástrojů nemáme. Podívejme se, jak bychom mohli postupovat při řešení podobných problémů v Perlu. Jako úvod do studia probereme některé základní operace pro manipulaci s PDF.

Pro manipulaci s PDF soubory je třeba využít některého z dostupných modulů. Za všechny můžeme jmenovat například PDF::Reuse, PDF::API2 a Text::PDF.

PDF::Reuse je velmi rychlý modul a ho výhodně použijeme zejména tam, kde je potřeba vyprodukovat větší množství podobných PDF dokumentů - typicky například sadu dopisů, které se liší adresou v hlavičce.

Vytváření PDF dokumentů

Funkcí prFile definujeme výstupní soubor a pomocí prEnd práci ukončíme. Mezi tyto dva příkazy pak budeme zapisovat další kód, který určí, co bude na výsledné stránce. Tedy pro začátek vytvoříme program, jež po spuštění vygeneruje prázdný pdf soubor. Kód bude vypadat následovně.

```
use PDF::Reuse;

prFile("blank.pdf");
prEnd();
```

Text

Dalším úkolem bude vypisování nějakého obsahu do dokumentu. Funkcí prText lze do výsledného souboru zapsat text. prText přijímá mimo řetězce také souřadnice - tím jsou myšleny vzdálenosti od levého a od dolního okraje stránky. Takto vypadá kód, který vygeneruje PDF soubor s nějakým textem.

```
prFile("text.pdf");
prText(50,800,"Text");
prEnd();
```

S textem můžeme provádět různé akce. Například rotovat.

```
prText(200, 200, " Rotace o 0","", 0);
prText(200, 200, " Rotace o 60","", 60);
prText(200, 200, " Rotace o 120","", 120);
prText(200, 200, " Rotace o 180","", 180);
prText(200, 200, " Rotace o 240","", 240);
prText(200, 200, " Rotace o 300","", 300);
```

Pomocí funkce prPage můžeme zalomit stránku a pokračovat na nové. Potom padesátistránkový dokument s očíslováním stran vytvoříme takto.

```
use strict;
use PDF::Reuse;

prFile("text.pdf");

for $_ (1..50) {
prText(500, 800, $_);

prText(50,740,"text 1");
prText(50,720,"text 2");
prText(50,700,"text 3");
prText(50,680,"text 4");

prPage();
}

prEnd();
```

Pro formátování textu existují následující funkce. prFont nastavuje písmo a jako hodnoty lze užít například Times-Roman, Courier, Helvetica. Velikost písma lze měnit funkcí prFontSize.

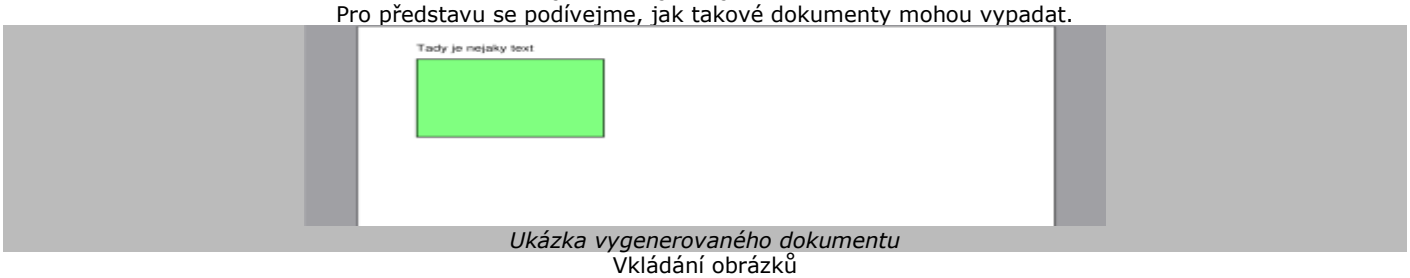
Grafika

Dále lze použít prAdd pro "nízkoúrovňové" instrukce. Těmi se nebudeme dále zabývat, jen si uvedeme příklad, který vykreslí zelený obdélník.

```
prAdd("50 700 160 100 re\\n0 1 0 rg\\nb\\n");
```

K umístění nějakého objektu je třeba zadat souřadnice.

Pro představu se podívejme, jak takové dokumenty mohou vypadat.



Ukázka vygenerovaného dokumentu

Vkládání obrázků

Pomocí prJpeg, prImage atd. lze vkládat již hotové obrázky. Obvykle potřebujeme znát jeho rozměry. Ty můžeme zjistit například pomocí modulu Image::Info. Zde je příklad, který vytvoří vysledek.pdf, který obsahuje předem určený obrázek.

```
$jpg = "foto.jpg";
$sirka = 400;
$vyska = 300;

prFile("vysledek.pdf");
$int = prJpeg($jpg, $sirka, $vyska);
prAdd("q\\n$sirka 0 0 $vyska 10 10 cm\\n/$int Do\\nQ\\n");
prEnd();
```

Spojování PDF dokumentů

Podívejme se na trochu jinou úlohu. Máme-li několik PDF dokumentů, ze kterých potřebujeme udělat jediný, nebo pokud chceme udělat z jednoho souboru několikastránkový výřez, pak zde máme funkci prDoc. Ta do výstupního souboru vytiskne příslušný rozsah z vybraného dokumentu. Ukažme si, jak bychom vytvořili dokument, sestávající se z prvních pěti stránek souboru text.pdf, kompletního obsahu blank.pdf a stránky 7 opět z text.pdf.

```
prFile("vysledek.pdf");

prDoc("text.pdf",1,5);
prDoc("blank.pdf");
prDoc("text.pdf",7);

prEnd();
```

Modifikace PDF dokumentů

Hlavním využitím modulu PDF::Reuse je hromadná práce s již existujícími dokumenty. Díky němu můžeme relativně jednoduše modifikovat PDF soubory a provádět takové věci jako je přidání čísla stránky nebo hlaviček dopisů.

Pojďme se podívat na tu druhou úlohu. Nechť máme v souboru adresy kontakty v následujícím formátu.

jméno[TAB]adresa[TAB]město[TAB]psč

A dále máme dopis.pdf který chceme každému z nich poslat. Naším úkolem bude vytvořit sadu dopisů, ke kterým navíc vzhledem k dopis.pdf přidáme pro každého člověka vygenerovanou hlavičku s adresou.

Vše spočívá v použití funkce prForm, které předáme PDF soubor, jež bude šablonou (pozadím) pro vytvářené dokumenty. Implementace našeho problému tak bude vypadat následovně.

```
prFile("vysledek.pdf");
prForm("dopis.pdf");

my $fr;
open($fr, "adresy") or die;

while (<$fr>)  {
my($jmeno, $adresa, $mesto, $psc) = split /\\t/;
prText(50, 800, $jmeno);
prText(50, 785, $adresa);
prText(50, 770, "$psc $mesto");
prPage()
    }
prEnd();
close $fr;
```

Perl (113) - Práce s archivy

Budeme se věnovat práci s archivy, zejména pak rozbalování a zabalování. Zajímat nás budou nejrozšířenější formáty tar, zip a rar.

Podívejme na to, jak manipulovat s TAR, ZIP a RAR v Perlu.

Budeme používat zejména modul Archive::Tar, jelikož tento typ archivu je v linuxovém světě asi nejrozšířenější.

Archive::Tar je sice jako modul pomalý a občas náročný na paměť, ale díky implementaci pomocí perlu je snadno přenositelný.

Vytvoření objektu pro manipulaci s archivy

Pro jakoukoliv manipulaci s archivy je třeba vždy vytvořit Tar objekt. To se dělá standardní cestou metodou new.

```
$tar = Archive::Tar->new("priklad.tar");
```

Je-li uvedeno jméno archivu jako parametr, pak se z něj provedením tohoto příkazu se do operační paměti načte seznam souborů. S těmito soubory tak můžeme dále pracovat.

Pro vymazání seznamu souborů z paměti lze využít metodu clear a pro znovunačtení metody read.

Přidávání se provádí metodou add_files a přejmenování pomocí rename.

Kontrola obsahu archivu

Pomocí metody contains_file lze ověřit, zda archiv obsahuje zadaný soubor.

```
$tar->contains_file("index.html");
```

Chceme-li zkontrolovat celý obsah, pak oceníme spíše funkci list_files, která vrátí seznam názvů všech souborů.

Pokud se nechceme obtěžovat s vytvářením objektů, lze pro získání seznamu souborů v archivu archiv.tar zadat také následující.

```
Archive::Tar->list_archive("archiv.tar");
```

Rozbalování

Metoda read slouží k načtení komprimovaného souboru do paměti. Jejími argumenty jsou jméno komprimovaného souboru a jako třetí parametr odkaz na hash s pevně určenou strukturou, který ovlivní, kolik a jaké soubory mají být z vybaleny z archivu.

Návratová hodnota funkce read je pole obsahující názvy rozbalených souborů.

Po volání read máme obsah k rozbalení načten v paměti. Samotné rozbalení provede jedna z metod extract nebo extract_file.

Podívejme se na možné klíče hashe parametrů ve volání metody read.

Klíč	Význam
filter	pomocí regulárního výrazu určí, vybere jen některé soubory
extract	nastavíme-li hodnotu na 1, pak se automaticky zavolá metoda extract (a archiv bude rozbalen)
limit	omezí počet rozbalovaných souborů na uvedené množství

Chceme-li rozbalit soubor priklad.tar, pak tedy dle výše uvedeného napíšeme následující kód.

```
use Archive::Tar;
my $tar = Archive::Tar->new;
$tar->read("priklad.tar");
$tar->extract();
```

Uvedeme-li parametr extract s pravdivou hodnotou, pak lze program zredukovat o poslední řádek.

```
use Archive::Tar;
my $tar = Archive::Tar->new;
$tar->read("priklad.tar", "", {"extract"=>1});
```

Pro rozbalení pouze souborů s příponou html použijeme klíč filter.

```
$tar->read("priklad.tar", "", {"extract"=>1, "filter"=>'.*html\$'});
```

Pro vybalení jednoho souboru zde máme metodu extract_file. Zde můžeme jako druhý parametr zadat, kam se má vybalený soubor přesunout.

Pokud nám jde pouze o rozbalení běžného archivu, pak jistě oceníme metodu třídy extract_archive. Ta funguje následovně.

```
Archive::Tar->extract_archive("priklad.tar");
```

Důkladnější výběr rozbalovaných souborů

Soubor rozbalený pomocí Archive::Tar se stává objektem typu Archive::Tar::File. To nám přináší zejména možnost filtrovat soubory podle dalších kritérií.

Uveďme si některá kritéria v tabulce.

Kritérium	Význam
name	jméno souboru
size	velikost souboru v bajtech
type	typ souboru; Archive::Tar pro tento účel exportuje konstanty FILE, HARDLINK, SYMLINK, CHARDEV, BLOCKDEV, DIR, FIFO, SOCKET
mktime	čas poslední modifikace
mode	mód
uid, gid, uname, gname	uživatel, skupina vlastnící soubor

Pak lze provádět takové věci jako například rozbalení souborů, jejichž velikost nepřesahuje 1000 bajtů. To by se provedlo takto.

```
$tar->extract(grep {$_->size < 1000 } $tar->get_files);
```

Taktéž lze rozbalovat podle obsahu souboru. Existuje metoda get_content, jež vrací obsah souboru. Pro rozbalení pouze těch souborů, které obsahují slovo linuxsoft provedeme následující příkaz.

```
$tar->extract(grep {$_->get_content =~ /linuxsoft/i} $tar->get_files);
```

Existují další zajímavé metody pro objekty typu Archive::Tar::File, o nichž se lze dočíst v dokumentaci.

Zabalování

Do tvořeného archivu se vždy vloží soubory, které jsou aktuálně v paměti. Ovšem jak přidat další soubory? K tomu máme metodu add_files, které předáme seznam souborů.

Přidání souboru tar.pl do našeho virtuálního seznamu tak provedeme následujícím voláním.

```
$tar->add_files("tar.pl");
```

Archivy se zabalují metodou write. Musíme vždy uvést co zabalit a jak to zabalit.

Prvním argumentem určujeme soubor, kam se archiv zapíše. Druhým je pro gzip úroveň komprese od 1 nebo 9 nebo pro bzip2 hodnota COMPRESS_BZIP.

Pro vytvoření nového archivu ze souborů v paměti lze užít například následující příkaz.

```
$tar->write("novy_archiv.tgz", 9);
```

Rychlejší cesta pro vytvoření archivu vede přes metodu třídy. Aniž bychom vytvářeli objekt, můžeme volat metodu create_archive. Té předáme požadovaný název výsledného archivu, typ komprese a seznam souborů, které chceme zabalit. Na ukázku přiložme tento příkaz.

```
Archive::Tar->create_archive( "vysledek.tgz", COMPRESS_GZIP, "tar.pl", "formular.html");
```

Práce se ZIP archivy

Myšlenka u práce se ZIP archivy je podobná jako u TAR archivů a proto se v rychlosti podívejme na základní operace.

Opět je třeba na začátku načíst archiv, který budeme rozbalovat. To provedeme pomocí metody new.

```
use Archive::Zip;
my $zip = Archive::Zip->new("archiv.zip");
```

Poznamenejme, že i zde je k dispozici metoda read, kterou lze přidávat nové soubory z archivů.

Seznam souborů z archivu archiv.zip nyní máme k dispozici. Pro seznam jmen souborů z tohoto archivu stačí zavolat metodu memberNames.

```
print $zip->memberNames;
```

Rozbalení proběhne po zavolání metody extractMember. Jako parametr uvedeme seznam souborů, které se mají rozbalit.

```
$zip->extractMember("titulky.sub");
```

Se seznamem souborů můžeme volně manipulovat. Pro odstranění některého souboru použijeme metodu removeMember.

```
$zip->removeMember("nepotrebny_soubor");
```

Naopak pro přidání souboru využijeme metodu addFile. Do nově tvořeného archivu lze nový soubor přidat pod změněným jménem.

```
$zip->addFile("data", "nove_jmeno")
```

Máme-li seznam souborů připraven, můžeme je zabalit.

```
$zip->writeToFileNamed("vysledek.zip");
```

Možností Archive::Zip je více. Pokročilejší záležitosti lze nalézt v dokumentaci.

Práce s RAR archivy

Pro rozbalení RAR archivu lze použít následující sekvenci příkazů.

```
use Archive::Rar;
$rar = Archive::Rar->new(-archive => "archiv.rar");
$rar->List;
$rar->Extract;
```

Další moduly

Existuje univerzální modul Archive::Extract, který dokáže rozbalit soubory různých typů. Jeho použití je následující.

```
use Archive::Extract;
$extract = Archive::Extract->new(archive => "priklad.tar");
$extract->extract;
```

Naopak dalším modulem zaměřeným na vytváření archivů je Archive::Builder.

Mimo uvedené existuje řada dalších modulů tvaru Archive::.

Perl (114) - Tk - úvod

Dneškem začíná minisérie o programování jednoduchých grafických aplikací pomocí grafické knihovny Tk.

X Window System

Unixové systémy používají jako grafické rozhraní toolkit X Window System, který tvoří aplikační vrstvu mezi jádrem operačního systému a aplikacemi. Toolkit je sada nástrojů, která umožňuje vykreslit na monitoru grafické prvky, pohybovat s okny, interagovat s klávesnicí a myší apod.

Grafické uživatelské rozhraní

Grafické uživatelské rozhraní (Graphical User Interface, zkráceně GUI) je jeden z typu rozhraní počítače pro uživatele (pro srovnání mezi další bychom mohli zařadit příkazový řádek, textové uživatelské rozhraní, rozhraní pro ovládání hlasem atd.). GUI umožňuje uživatelům ovládat počítač pomocí speciálních prvků, což funguje tak, že uživatel pomocí klávesnice a myši používá grafické prvky uvnitř oken.

Základní grafický prvek, o kterém je zde řeč, se nazývá widget. Jednoduše řečeno je widget nějaká věc v okně programu (například tlačítko, úsek textu, obrázek, ikona atd.).

To, které widgety jsou programátorovi k dispozici závisí na knihovně, kterou používá. Základní prvky jako jsou tlačítka, formulářové prvky apod. jsou jistě ve všech rozšířených grafických knihovnách, ale některé méně používané se již vždy vyskytovat nemusí.

Existence widgetů je základním předpokladem pro tvorbu GUI. Widgety jsou již hotové součásti našeho programu a na nás už zbývá jen vhodně je poskládat a přiřadit jim požadované činnosti.

My si v seriálu stručně některé grafické knihovny představíme. Začneme knihovnou Tk.

Tk

Knihovna Tk je jedna z nejznámějších knihoven pro tvorbu grafických uživatelských rozhraní. Mezi její výhody patří otevřenost a multiplatformnost.

Tato knihovna byla vyvinutá Johnem Ousterhoutem společně s jazykem Tcl. Později se Tk začalo používat i v jiných jazycích včetně Perlu.

Instalace

Je velmi pravděpodobné, že knihovnu máte již v systému. Stejně tak bývá nainstalován i modul Tk, který ji zprostředkovává pro Perl.

Schéma psaní Tk aplikace

Psaní programu pomocí knihovny Tk probíhá v několika hlavních krocích.

Ze všeho nejdříve je třeba vytvořit hlavní okno. Těch může mít aplikace libovolné množství, ale musí být minimálně jedno.

V hlavním okně se dále vytoří widgety. Tyto widgety můžeme různými způsoby konfigurovat, nastavit jejich chování a umístit na požadované místo v okně. To je hlavní část aplikace.

Na závěr spustíme cyklus událostí. Od této chvíle již necháváme další průběh programu na uživateli. Na základě konfigurace se program sám postará o to, aby obsloužil uživatelovy požadavky.

První program v PerlTk

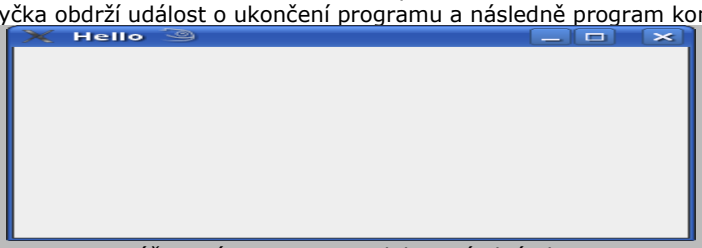
Vytváření widgetů v PerlTk je intuitivní a proto nebudeme syntaxi rozebírat do detailů. Tento miniseriál založíme spíše na příkladech, z nichž bude obvykle syntaxe patrná.

Uveďme nyní, jak vytvořit hlavní okno a spustit smyčku událostí. Tyto dvě věci budeme potřebovat v každém programu, který používá knihovnu Tk. Zde je nejjednodušší program v PerlTk.

```
use Tk;

$m = MainWindow->new();

#zde by mohlo probíhat vytváření a uspořádávání widgetů

MainLoop();
```

Předposlední příkaz vytvořil hlavní okno a poslední spustil smyčku událostí. Hlavní okno obsahuje mimo jiné dekoraci se třemi tlačítky - minimalizovat, maximalizovat a zavřít. Tento program nic nedělá - pouze se zobrazí okno a čeká se na uživatele, až ho zavře. Zavření okna, minimalizace, maximalizace a změna rozměrů okna pomocí šipek jsou jediné činnosti, které může uživatel

vykonat. O zachytávání událostí se stará zmiňovaná událostní smyčka. Jakmile klikneme na křížek v pravém horním rohu, smyčka obdrží událost o ukončení programu a následně program končí.



Náš první program v PerlTk - prázdné okno

Program s prvním widgetem

Podívejme se nyní na to, jak vkládat do hlavního okna widgety. Nejprve vytvoříme program, který v okně zobrazí nějaký text a zároveň nastavíme vlastní titulek stránky.

Nejprve nastavíme titulek. K tomu zde máme metodu title, které předáme text titulku.

```
$m->title("2. Tk program");
```

Poznámka - Nad hlavním oknem lze volat i několik dalších metod. Užitečná je například metoda geometry, která nastavuje počáteční rozměry okna. Předáváme jí řetězec tvaru šířkaxvýška; například 400x200.

Text vytvoříme pomocí metody Label volané nad objektem hlavního okna. Tato metoda přijímá několik parametrů; nás bude zajímat zejména parametr -text, kterým určujeme, jaký nápis se má zobrazit. Toto volání tedy bude vypadat takto.

```
$napis=$m->Label(-text=>"Ahoj! Toto je první program v Perl/Tk.");
```

Tím je náš vytvořen prvek. Nyní ho je třeba umístit. Kdybychom ho neumístili, ve výsledné aplikaci by se nezobrazil. K umisťování slouží nejčastěji metoda pack (o dalších se zmíníme později). Ta určuje mimo pozice v okně také rozměry.

```
$napis->pack();
```

Poznamenejme ještě, že řádky

```
$napis=$m->Label(-text=>"Ahoj! Toto je druhý program v Perl/Tk.");
$napis->pack();
```

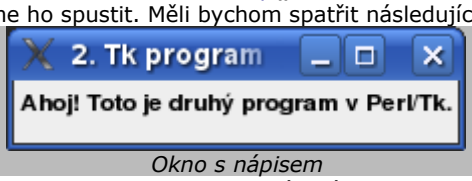
lze zapsat ekvivalentně také následujícím způsobem.

```
$napis=$m->Label(-text=>"Ahoj! Toto je druhý program v Perl/Tk.")->pack();
```

Náš program nyní vypadá takto.

```
use Tk;

$m = MainWindow->new();
$m->title("2. Tk program");
$napis=$m->Label(-text=>"Ahoj! Toto je druhý program v Perl/Tk.");
$napis->pack();
MainLoop();
```

Zkusíme ho spustit. Měli bychom spatřit následující okno.



Okno s nápisem

Program s událostí

Ve třetí ukázce uvidíme první jednoduché události. Opět zde budeme mít nějaký textový nápis. Dále vytvoříme tři tlačítka, kterým přiřadíme následující činnosti.

1. Tlačítko změní barvu textu a jeho pozadí na předdefinované barvy
2. Tlačítko změní barvu textu a jeho pozadí na barvy, které náhodně vybereme z předdefinované množiny barev
3. Tlačítko ukončí aplikaci

Tlačítko vytvoříme pomocí metody Button. Té můžeme předat text, který na něm má být a dále akci. Tlačítko tedy můžeme obecně vytvořit například takto.

```
$tlacitko = $m->Button(-text => "Potvrdit", -command=>\&potvrdit)

sub potvrdit {
#příkazy, které se mají vykonat po stisknutí tlačítka Potvrdit}
}
```

Uveďme ještě, že místo odkazu na podprogram budeme někdy používat přímo anonymní podprogram. Taktéž je možná třetí varianta nastavení, která spočívá v předání anonymního pole, jehož první prvek odkazuje na podprogram, který se má vykonat; a ostatní prvky jsou pak předány tomuto podprogramu jako argumenty.

V podprogramu potvrdit můžeme též nastavovat parametry ostatních widgetů. K tomu lze užít metodu configure zavolanou nad příslušným widgetem, které předáme nové parametry.

Zkusme tedy vytvořit naše první tlačítko. Toto tlačítko má za úkol změnit barvu textu. Zavoláme tedy metodu configure nad nápisem, tj. nad objektem $napis a předáme jí nové hodnoty pro parametry -background a -foreground. Jako barvy můžeme používat klasicky zápis tvaru #RRGGBB, my pro jednoduchost použijeme symbolické hodnoty.

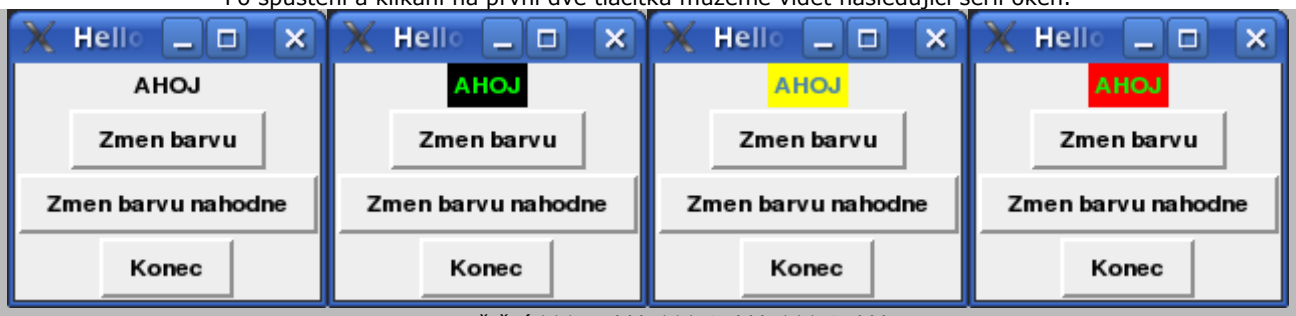
```
$m->Button(-text => "Zmen barvu", -command=>sub{$napis->configure(-background=>"black",
-foreground=>"green")})->pack();
```

Další tlačítka by se napsala analogicky. Takto vypadá náš nový program.

```
use Tk;

srand;
@barvy = qw(black yellow red green steelblue darkviolet);

$m = MainWindow->new();

$napis=$m->Label(-text=>"AHOJ")->pack();
```

```
$m->Button(-text => "Zmen barvu", -command=>sub{$napis->configure(-background=>"black",
-foreground=>"green")})->pack();
$m->Button(-text => "Zmen barvu nahodne", -command=>sub{$napis->configure(-background=>
$barvy[int(rand(@barvy))], -foreground=>$barvy[int(rand(@barvy))]})->pack();
$m->Button(-text => "Konec", -command=>sub{exit})->pack();

MainLoop();
```

Po spuštění a klikání na první dvě tlačítka můžeme vidět následující sérii oken.



*Po spuštění *** 1. klik *** 3. klik *** 3. klik*

Vhodné rozmístění prvků v okně je důležitým měřítkem každé aplikace.

Na rozmístění prvků v aplikaci závisí významně její přehlednost a vzhled. Proto je vhodné znát způsoby, kterými lze ovlivňovat umísťování prvků v okně. My jsme zde minule používali výhradně metodu pack, která si nějak sama poradila. U většiny aplikací však je potřeba, aby byly prvky rozložené tak, jak chce programátor.

Představíme si zde dvě metody rozmísťování prvků dostupných v knihovně Tk - skládání prvků do vyhrazeného místa pomocí pack rozebereme podrobněji a také se podíváme na umísťování do mřížky pomocí grid.

Uveďme na úvod, že přímí potomci jednoho rodiče musí používat vždy stejný způsob umísťování.

Skládání prvků pomocí pack

Umísťujeme-li prvek pomocí metody pack volané bez parametrů, znamená to, že se prvek má umístit na nějaké volné místo v oblasti rodičovského prvku. Pořadí skládání je stejné jako pořadí volání pack.

Metodě pack můžeme poslat argument side => "strana", kde strana nabývá hodnot left, right, top, bottom. Podívejme dva konkrétní příklady.

- Voláme-li pack(-side => "left"), vezme se šířka umísťovaného prvku a zabere se levá část oblasti rodičovského prvku této šířky. Prvek pak bude umístěn uprostřed takto vymezeného prostoru.
- Voláme-li pack(-side => "bottom"), vezme se výška umísťovaného prvku a zabere se dolní část oblasti rodičovského prvku této výšky. Prvek pak bude umístěn uprostřed takto vymezeného prostoru.

Také existuje skupina parametrů, která určuje, jak se má vymezený prostor vyplnit. Lze specifikovat parametr -fill, který určuje, zda se má prvek roztáhnout do celého vymezeného prostoru (hodnota both), do jeho části (pro šířku volíme hodnotu x, pro výšku y), či zda se roztahovat nemá (none). Parametr -anchor ukotví prvek do jedné ze světových stran (n, w, e, s, ne, nw, se, sw) nebo do středu (center). Pomocí -ipadx a -ipady lze určit tloušťku okolí, které má zůstat vždy prázdné.

Ještě existuje jeden zajímavý parametr metody pack, kterým je -expand. Ten řídí to, jakým způsobem si rozdělí prostor potomci aktuálního prvku. Pokud nastavíme -expand na hodnotu y, pak každý potomek dostane přidělen stejně velký prostor jako ostatní se stejně nastaveným parametrem -side. Místo -expandje však často výhodnější použít umísťování do mřížky.

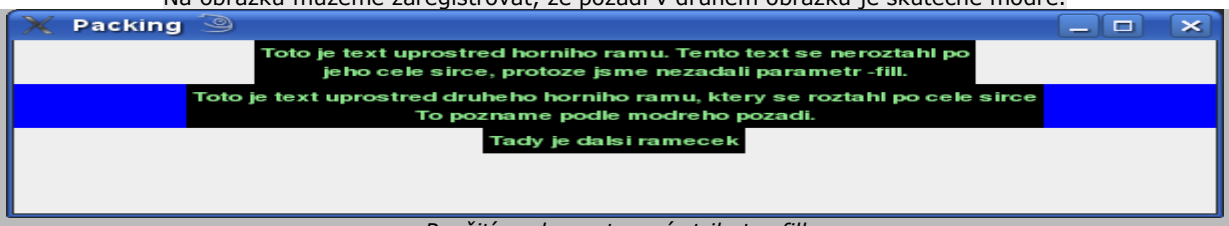
Ilustrujme si nyní, na obrázcích, jak vlastně metoda pack pracuje. Rozdělíme okno na tři pruhy pomocí rámců. Rámec vytvoříme pomocí prvku Frame. Těmto rámům nastavíme odlišné vlastnosti a budeme sledovat jejich chování. Do každého rámu vložíme nějaký text s pozadím, abychom viděli, jakou oblast pokrývá.

Nejprve demonstrujme, jak funguje atribut -fill. Ten dokáže popisek roztáhnout tak, že zabere veškeré volné místo. Vezměme tedy prostřední rám a přidejme mu parametr -fill=>"x".

```
use Tk;
use strict;

my $m = MainWindow->new;
$m->geometry("600x150");


my $ram1 = $m->Frame(-background => "green")->pack(-side => "top");
my $ram2 = $m->Frame(-background => "blue")->pack(-side => "top", -fill=>"x");
my $ram3 = $m->Frame(-background => "yellow")->pack(-side => "top");

$ram1->Label(
-text => "Toto je text uprostred horniho ramu. Tento text se neroztahl po\n
jeho cele sirce, protoze jsme nezadali parametr -fill.",
-background => "black",
-foreground => "lightgreen"
)->pack(-side => "top");

$ram2->Label(
-text => "Toto je text uprostred druheho horniho ramu, ktery se roztahl po cele sirce\n
To pozname podle modreho pozadi. Navic jsme ted pridali parametr\n
-expand, takze kdyz menime velikost okna, meni se tento ram, nikoliv zbyle misto.",
-background => "black",
-foreground => "lightgreen"
```

```
)->pack(-side => "top");

$ram3->Label(
-text => "Tady je dalsi ramecek",
-background => "black",
-foreground => "lightgreen"
)->pack(-side => "top");

MainLoop;
```

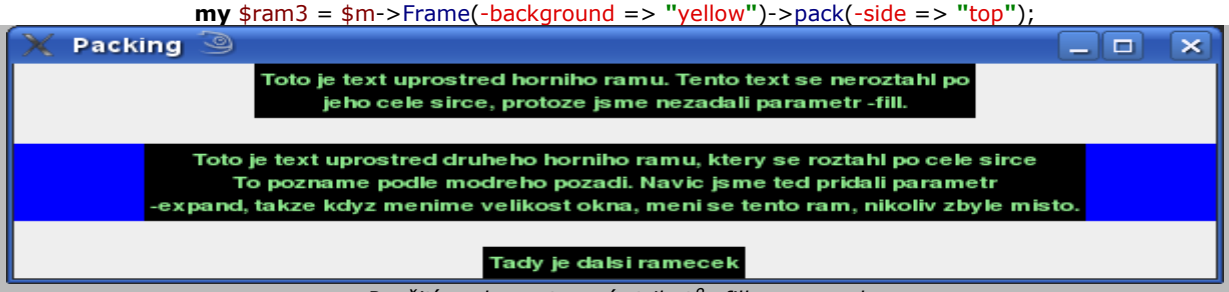
Na obrázku můžeme zaregistrovat, že pozadí v druhém obrázku je skutečně modré.



Použití pack, nastavení atributu -fill

Přidejme druhému rámu ještě parametr -expand=>"y". Nyní můžeme vidět, že místo volného místa při změně rozměrů okna měníme velikost tohoto rámu.

```
my $ram1 = $m->Frame(-background => "green")->pack(-side => "top");
my $ram2 = $m->Frame(-background => "blue")->pack(-side => "top", -fill=>"x", -expand=>"y");
my $ram3 = $m->Frame(-background => "yellow")->pack(-side => "top");
```

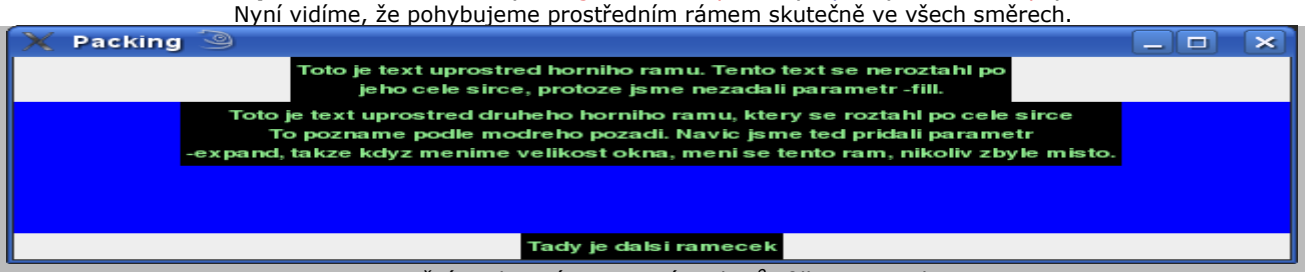


Použití pack, nastavení atributů -fill a -expand

Abychom to dokázali, potřebovali bychom, aby bylo vidět celé pozadí. Zkusme tedy roztáhnutí oběma směry.

```
my $ram1 = $m->Frame(-background => "green")->pack(-side => "top");
my $ram2 = $m->Frame(-background => "blue")->pack(-side => "top", -fill=>"both", -expand=>"y");
my $ram3 = $m->Frame(-background => "yellow")->pack(-side => "top");
```

Nyní vidíme, že pohybujeme prostředním rámem skutečně ve všech směrech.



Použití pack, jiné nastavení atributů -fill a -expand

Umísťování do mřížky

Umísťujeme-li prvky do mřížky, znamená to, že se vytvoří něco jako tabulka, do jejíchž buněk můžeme jednotlivé ukládat. Každá buňka a tedy přeneseně i každý prvek tak má v této tabulce své souřadnice. Protože prvky bývají různě velké, lze buňky navzájem spojovat.

Metoda grid přijímá argumenty -row a -column, kterými jsou určeny souřadnice prvku v tabulce. Středem tohoto souřadnicového systému je levá horní buňka s oběma parametry nulovými. Dále lze uvést parametry -rowspan a -columnspan pro spojení s následujícími řádky a sloupci. Stejně jako u pack lze uvést parametry -ipadx a -ipady. Parametr sticky uchycuje obsah buňky na nějakou nebo nějaké světové strany (lze uvést například hodnotu ne pro uchycení vpravo a nahoře, nsew pro roztažení prvku přes celou buňku nebo centerpro umístění do středu).

Příklad - kalkulačka

Pokusme se nyní pomocí mřížky vytvořit jednoduchou klávesnici kalkulačky.

```
$k1=$m->Button(-text=>"1")->grid(-row =>2, -column =>0);
$k2=$m->Button(-text=>"2")->grid(-row =>2, -column =>1);
$k3=$m->Button(-text=>"3")->grid(-row =>2, -column =>2);
$k4=$m->Button(-text=>"4")->grid(-row =>1, -column =>0);
$k5=$m->Button(-text=>"5")->grid(-row =>1, -column =>1);
$k6=$m->Button(-text=>"6")->grid(-row =>1, -column =>2);
$k7=$m->Button(-text=>"7")->grid(-row =>0, -column =>0);
$k8=$m->Button(-text=>"8")->grid(-row =>0, -column =>1);
$k9=$m->Button(-text=>"9")->grid(-row =>0, -column =>2);
$k0=$m->Button(-text=>"0")->grid(-row =>3, -column =>0, -columnspan =>2, -sticky =>"we");
$rovnitko=$m->Button(-text=>"=")->grid(-row =>3, -column =>2);
```

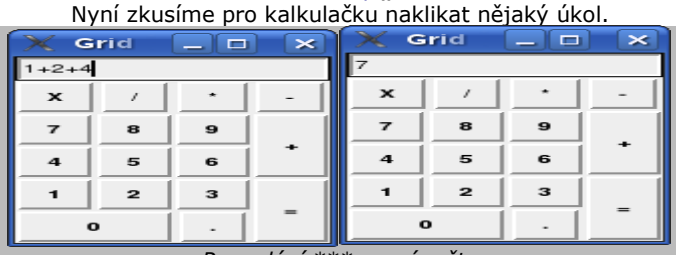
Výsledek je následující.

Ukázka použití mřížky

Ačkoliv ještě neznáme všechny potřebné věci, pro ukázku zkusíme tuto kalkulačku zprovoznit. Přidáme tedy tlačítka s operacemi a nějaký řádek, kde se bude objevovat aktuální výpočet čekající na zpracování.
Dále je třeba pro každé tlačítko vytvořit akci, která do řádku přidá požadovaný symbol. Tlačítko = bude mít za úkol výpočet vyhodnotit a ukázat výsledek.K vyhodnocení použijeme nebezpečnou funkci eval.

```
#!/usr/bin/perl -T
use Tk;

$m = MainWindow->new();
$entry=$m->Entry()->grid(-row =>0, -column =>0, -columnspan =>4, -sticky =>"we");
$k1=$m->Button(-text=>"1", -command=>sub{$entry->insert("end", "1")})->grid(-row =>4, -column =>0);
$k2=$m->Button(-text=>"2", -command=>sub{$entry->insert("end", "2")})->grid(-row =>4, -column =>1);
$k3=$m->Button(-text=>"3", -command=>sub{$entry->insert("end", "3")})->grid(-row =>4, -column =>2);
$k4=$m->Button(-text=>"4", -command=>sub{$entry->insert("end", "4")})->grid(-row =>3, -column =>0);
$k5=$m->Button(-text=>"5", -command=>sub{$entry->insert("end", "5")})->grid(-row =>3, -column =>1);
$k6=$m->Button(-text=>"6", -command=>sub{$entry->insert("end", "6")})->grid(-row =>3, -column =>2);
$k7=$m->Button(-text=>"7", -command=>sub{$entry->insert("end", "7")})->grid(-row =>2, -column =>0);
$k8=$m->Button(-text=>"8", -command=>sub{$entry->insert("end", "8")})->grid(-row =>2, -column =>1);
$k9=$m->Button(-text=>"9", -command=>sub{$entry->insert("end", "9")})->grid(-row =>2, -column =>2);
$k0=$m->Button(-text=>"0", -command=>sub{$entry->insert("end", "0")})->grid(-row =>5, -column =>0,
-columnspan =>2, -sticky =>"we");
$te=$m->Button(-text=>".", -command=>sub{$entry->insert("end", ".")})->grid(-row =>5, -column =>2);
$pl=$m->Button(-text=>"+", -command=>sub{$entry->insert("end", "+")})->grid(-row =>2, -column =>3,
-rowspan =>2, -sticky =>"ns");
$mi=$m->Button(-text=>"-", -command=>sub{$entry->insert("end", "-")})->grid(-row =>1, -column =>3);
$kr=$m->Button(-text=>"*", -command=>sub{$entry->insert("end", "*")})->grid(-row =>1, -column =>2);
$de=$m->Button(-text=>"/", -command=>sub{$entry->insert("end", "/")})->grid(-row =>1, -column =>1);

$ro=$m->Button(-text=>"=", -command=>sub{$vysledek=eval($entry->Get()); $entry->delete(0, "end");
$entry->insert("end", $vysledek)})->grid(-row =>4, -column =>3, -rowspan =>2, -sticky =>"ns");
$re=$m->Button(-text=>"X", -command=>sub{$entry->delete(0, "end");})->grid(-row =>1, -column =>0);

MainLoop();
```

Nyní zkusíme pro kalkulačku naklikat nějaký úkol.



*Po zadání *** po výpočtu*

Perl (116) - Tk - základní widgety

Představíme si několik nejdůležitějších widgetů z knihovny Tk, které můžeme využívat ve svých aplikacích.

Představme si nyní několik nejčastěji užívaných widgetů a jejich vlastnosti.
Obecné vlastnosti
Uveďme nejprve některé vlastnosti, které lze nastavit u většiny prvků.

Název	Význam
-width	šířka widgetu v pixelech
-height	výška widgetu v pixelech
-background(-bg)	barva pozadí
-foreground(-bg)	barva popředí
-relief	okraj; k dispozici raised, sunken, ridge, flat, groove
-font	styl písma
-anchor	umístění obsahu - zadává se center nebo n, e, s, w podle světových stran; dále je možné zadávat ne, se, nw, sw
-text	zobrazený text
-image, -bitmap	obrázek, který se má zobrazit ve widgetu

Barvy zapisujeme buď symbolicky nebo kódem. Na většině systémů jsou přednastaveny desítky až stovky symbolických názvů barev. Jejich seznam je obvykle v souboru /usr/share/X11/rgb.txt, kam můžete pro představu nahlédnout. Zápis kódem je standardní označení tvaru #RRGGBB, případně i #RGB, #RRRGGGBBB či #RRRRGGGGBBBB.

Jako formát stylu písma můžeme použít jeden ze zápisů výstupu příkazu
$ xlsfonts
Zde je konkrétní příklad.
-adobe-times-medium-r-normal--8-80-75-75-p-44-iso8859-2
Tento řetězec je ve formátu *-výrobce-rodina-tloušťka-šikmost-těsnost-styl-pixelů-rozlišení x-rozlišení y-šířka-kódování*. Pokud nechceme vyplňovat všechny vlastnosti, lze použít zastupné znaky * a ?.
U všech prvků také můžeme použít následující metody.

Název	Význam
configure("vlastnost"=>"nová hodnota")	změní vlastnosti widgetu
cget("vlastnost")	vrátí aktuální hodnotu zvolené vlastnosti

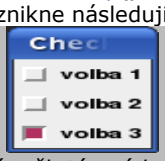
Popis některých widgetů
Tlačítko
Tlačítko patří mezi nejpoužívanější prvky a již jsme si ho představili. Uveďme ještě, že lze použít metodu flash pro bliknutí a invoke pro vyvolání metody přiřazené vlastnosti -command.
Zaškrtávací políčko
Dalším prvkem je zaškrtávací políčko (Checkbutton). Funguje tak, že do vlastnosti -variable uložíme odkaz na proměnnou. Je-li tlačítko zaškrtnuto, hodnota této proměnné se nastaví na pravdivou. Je-li naopak tlačítko nezaškrtnuto, hodnota je nepravdivá.
Uveďme zde vlastnosti pro Checkbutton.

Název	Význam
-command=>\&podprogram	při zaškrtnutí provede podprogram, jehož odkaz je atributu přiřazen (toto se provede až po aktualizaci proměnné vlivem zaškrtnutí)
-onvalue, -offvalue	přepíná tlačítko; možné hodnoty jsou implicitně 0, 1, lze nastavit pomocí -variable
-indicatoron	nastavuje, zda se má zobrazovat indikátor

Ukažme si jednoduchý příklad vykreslení zaškrtávacích tlačítek.

```
$m = MainWindow->new();
$t1=$m->Checkbutton(-text=>"volba 1", -variable =>\$v1)->pack();
$t2=$m->Checkbutton(-text=>"volba 2", -variable =>\$v2)->pack();
$t3=$m->Checkbutton(-text=>"volba 3", -variable =>\$v3)->pack();
$n=2;
$t3->configure(-onvalue=>2, -variable =>\$n);
MainLoop();
```

Tímto vznikne následující okno.



Použití zaškrtávacích políček
Přepínací políčko
Podobné jako předchozí tlačítko Chceckbutton je Radiobutton. Liší se tím, že v jeden okamžik může být zaškrtnuta maximálně jedna položka. Jsou typické tím, že se obvykle používá několik tlačítek Radiobutton najednou a uživatel si pomocí nich vybírá jednu možnost.
Parametr -value určuje hodnotu, která je uložena do proměnné v parametru -variable, je-li aktuální tlačítko aktivní. Nastavíme-li u několika tlačítek stejnou hodnotu do -variable, pak se automaticky synchronizují (a uživatel může vybrat vždy nejvýše jednu z těchto voleb).
Uveďme opět jednoduchý příklad.

```
$m = MainWindow->new();
$m->Label(-text=>"Souhlasis?")->pack();
$t1=$m->Radiobutton(-text=>"Ano", -variable =>\$v, -value=>1)->pack();
$t2=$m->Radiobutton(-text=>"Ne", -variable =>\$v, -value=>0)->pack();
MainLoop();
```

Po spuštění se zobrazí následující okno.

Použití přepínacích políček
Škála
Škála (Scale) je stupnice, kterou můžeme ovládat obsah nějaké proměnné. Funguje tak, že pohybem posuvníku na škále měníme současně hodnotu nějaké svázané proměnné.
Podívejme se na vlastnosti škály.

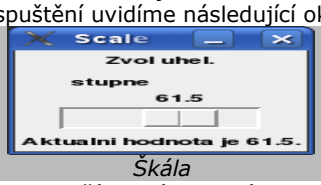
Název	Význam
-variable =>\$promenna	určuje, která proměnná bude se škálou svázána; s každým posuvem škály se bude její hodnota aktualizovat
-from, -to	nastavení rozsahu škály
-resolution	specifikuje velikost skoku, implicitně 1
-command=>\&podprogram	odkaz na podprogram, který se má vykonat po každé změně hodnoty na škále

-tickinterval	vzdálenost popisků
-label, -font	popisek

Taktéž lze využít metodu set pro nastavení hodnoty na škále.
Zde je jednoduchý příklad, který ilustruje možnosti škály. Po každé změně hodnoty na škále aktualizujeme Label, který nám bude ukazovat současnou hodnotu.

```
$m = MainWindow->new();
$m->Label(-text=>"Zvol uhel.")->pack();
$skala=$m->Scale(
-variable =>\$hodnota,
-orient=>"horizontal",
-from=>0,
-to=>90,
-resolution=>0.5,
-label=>"stupne",
-command=>\&zmena_uhlu
)->pack();
$status = $m->Label(-text=>"Aktualni hodnota je $hodnota")->pack();
MainLoop();

sub zmena_uhlu {
$status->configure(-text=>"Aktualni hodnota je $hodnota.")
}
```

Po spuštění uvidíme následující okno.



Škála

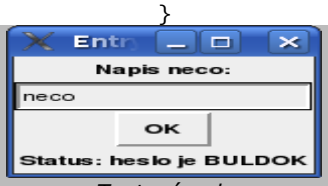
Jednořádkové textové pole

Jednořádkové textové pole (Entry) je základním prvkem pro textový vstup. Umožňuje uživateli napsat krátký text.
Lze využít vlastnost -show pro zadávání hesel (místo textu se zobrazují hvězdičky).
U textového pole lze využít několik metod.

Název	Význam
get	vrátí řetězec, který je aktuálně v poli
insert(pozice, řetězec)	na danou pozici vloží daný řetězec
selectionFrom, selectionTo, selection, selectionClear, selectionPresent	nastavuje resp. vrací výbraný text

Zde je opět jednoduchý příklad s ukázkou toho, jak získat text z formuláře.

```
$m = MainWindow->new();
$m->Label(-text=>"Napis neco:")->pack();
$entry=$m->Entry()->pack();
$m->Button(-text => "OK", -command=>\&over)->pack();
$status=$m->Label(-text=>"Status: pristup zakazan")->pack();
MainLoop();

sub over{
$status->configure(-text=>($entry->Get() eq "neco" ? "Status: heslo je
BULDOK" : "Status: pristup zakazan"))
}
```



Textové pole

Jednořádkové textové pole s popiskem

V různých formulářích se často používá textové pole s popiskem. Prvek LabEntry v sobě kombinuje prvky Label a Entry.

```
$m->->LabEntry(
-label => "Vloz text:",
-labelPack => [ -side => "left" ],
-textvariable => \$text
)->pack();
```

Obrázky

V Tk je každý obrázek samostatný objekt. Máme-li tedy nějaký soubor s obrázkem, nejprve vytvoříme objekt. To lze udělat jedním z následujících způsobů.

```
$obrazek=$m->Bitmap(-file=>"obr.xbm");
$obrazek=$m->Pixmap(-file=>"obr.xpm");
$obrazek=$m->Photo(-file=>"obr.gif");
```

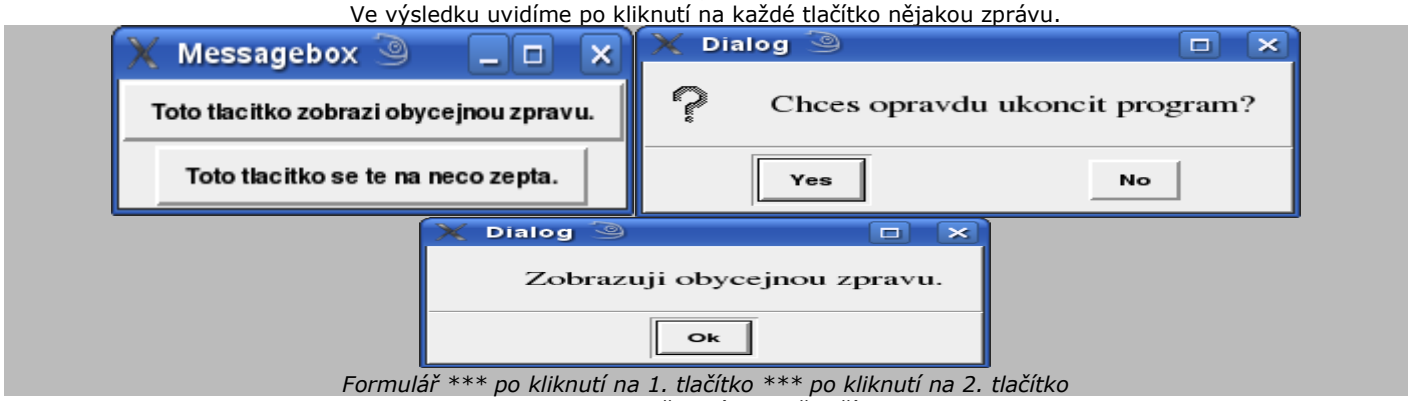
Nyní můžeme obrázek přiřazovat k atributům různých prvků (obvykle s názvem -image či -bitmap). Příklad uvidíme později.

Zpráva ve zvláštním okně

Nyní bude úkolem zobrazit po kliknutí na tlačítko nějakou zprávu. Máme k dispozici dva druhy takových zpráv - buď oznámení (zpráva typu ok) nebo zjišťovací otázku (zpráva typu yesno). Zprávu zobrazíme pomocí metody messageBox volané nad hlavním oknem. Musíme určit typ zprávy a obsah zprávy. Dále můžeme zobrazit ve zprávě ikonu (například otazník v případě otázky).

```
my $t1 = $m->Button(-text => "Toto tlacitko zobrazi obycejnou zpravu.",
-command => \&t1)->pack();
my $t2 = $m->Button(-text => "Toto tlacitko se te na neco zepta.",
-command => \&t2)->pack();

sub t1 {
$m->messageBox(-message => "Zobrazuji obycejnou zpravu.", -type => "ok");
}

sub t2 {
my $dotaz = $m->messageBox(-message => "Chces opravdu ukoncit program?",
-type => "yesno", -icon => "question");

$m->messageBox(-message => "Sbohem", -type => "ok") and exit if ($dotaz eq "Yes");
}
```

Ve výsledku uvidíme po kliknutí na každé tlačítko nějakou zprávu.



*Formulář *** po kliknutí na 1. tlačítko *** po kliknutí na 2. tlačítko*

Perl (117) - Tk - některé pokročilejší widgety

Navážeme na předchozí díl a podíváme se na pár dalších widgetů. Tentokrát půjde o widgety, které již mají složitější strukturu a větší množství vlastností.

Víceřádkové textové pole s posuvníkem

Víceřádkový vstup lze realizovat pomocí prvku Text. Umožňuje uživateli editovat více řádků textu. Lze nastavit vlastnosti state (pro vypnutí nastavíme na disabled) a dále height, width. Podívejme se nyní na metody. V několika metodách využíváme parametru pozice, což bývá řetězec, jež může mít několik různých tvarů. Zde je několik intuitivních příkladů, jak lze specifikovat pozici.

- *číslo_řádku.číslo_sloupce*
- insert (aktuální pozice)
- end (konec textu)
- +počet lines, -počet lines, +počet chars, -*počet* chars (relativní určení pozice)
- linestart, lineend, wordstart, wordend (opět relativní určení)

Název	Význam
get(*pozice1*, *pozice2*)	vrátí text, volitelně pouze na určeném rozsahu
insert(*pozice*, *text*)	vloží text na zadanou pozici
see(*pozice*)	přesune pohled na text tak, aby byla vidět daná pozice
search(*vzor*, *pozice1*, *pozice2*)	vyhledá text
search(*volby* --, *vzor*, *pozice1*, *pozice2*)	to samé, ale lze určit následující volby intuitivního významu: exact, backward, forward, regexp, nocase, count*proměnná_kam_se_uloží_délka_nalezeného_řetězce*
index	vrátí index tvaru *číslo_řádku.číslo_sloupce*
markSet, markUnset	záložky na pozice

Ukažme si příkazy, které nejprve vytvoří textové okno a pak vloží na nějakou pozici zadaný text.

```
$pole = $m->Text(-width=>100, -height=>4)->pack();
$pole->insert("3.50", "toto se vlozi na zadanou pozici");
$pole->insert("end", "toto se vlozi na konec textu");
```

Prvek Text nabízí tzv. tagování, což je možnost nastavení stylu písma. Je možné si nadefinovat styl o nějakém názvu a pak ho používat.

```
$text->tagConfigure("styl", -foreground=>"green", -background=>"black");
$text->insert("1.1", "tento text je napsan stylem 'styl'.", "styl");
$text->tagAdd("styl", "2.2", "4.4");
```

Je-li pole moc malé, nemá pole implicitně oproti jiným knihovnám žádný posuvník. Ten je potřeba dopsat ručně. Nejprve tedy vytvoříme posuvník pomocí Scrollbar a poté ho přiřadíme textovému poli pomocí -xscrollcommand nebo -yscrollcommand a nakonfigurujeme, aby se posunoval současně s textem.

```
$posuvnik = $m->Scrollbar();
$text = $m->Text(-yscrollcommand => ["set", $posuvnik]);
$posuvnik->configure(-command => ["yview", $text]);

$posuvnik->pack(-side=>"right", -fill => "y");
$text->pack(-side => "left");
```

Více oken

Je-li potřeba více než-li jedno okno, použijeme prvek Toplevel. K novému oknu se můžeme chovat stejně jako k hlavnímu oknu. Zde jsou příkazy, které vytvoří druhé okno a nastaví mu titulek.

```
$okno2 = $m->Toplevel;
$okno2->title("Toto je druhe okno");
```

Druhé okno se otevře v okamžiku, kdy dojde k vykonání těchto příkazů. Může se tak stát hned po spuštění programu nebo až po nějaké uživatelem vyvolané akci.

Menu

Existuje více metod, jak vytvořit menu. Některé si tu v následujících odstavcích představíme.

Menu přes rámečky

Vytvoření menu se skládá z několika kroků.

Nejprve je potřeba vyhradit pro menu lištu, kam pak umístíme jednotlivé nabídky. Lištu vytvoříme pomocí prvku Frame (který lze též použít pro uspořádávání většího množství prvků na stránce pomocí seskupení). Obvykle se lišta umísťuje nahoru a proto metodě pack sdělíme, že právě tam chceme menu umístit. To uděláme předáním argumentu -side => "top".

Nabídky vytvoříme pomocí Menubutton. Do každé nabídky dále přidáme příslušné položky, k čemuž existuje metoda command.

```
$lista=$m->Frame()->pack(-side => "top");
$nabidka_soubor = $lista->Menubutton(-text => "Soubor", -borderwidth=>1,
-relief=>"raised")->pack(-side=>"left", -padx=>1);
$nabidka_soubor->command(-label=>"Otevrit", -accelerator=>"Ctrl+O", -command=>\&otevrit);
$nabidka_soubor->command(-label=>"Ulozit", -accelerator=>"Ctrl+S", -command=>\&ulozit);
$nabidka_soubor->command(-label=>"Konec", -accelerator=>"Ctrl+Q", -command=>sub{exit});

$nabidka_oprogramu = $lista->Menubutton(-text => "O programu", -borderwidth=>1,
-relief=>"raised")->pack(-side=>"left", -padx=>1);
```

Výsledkem bude následující menu.

Menu

V Tk má menu ještě několik dalších možností. Přímo do něj lze přidávat přepínací a zaškrtávací položky. Mezi naše nabídky Soubor a O programu vklíníme ještě jednu, kde si je vyzkoušíme.

Metoda radiobutton vytvoří přepínací položku. Funguje stejně jako prvek Radiobutton a tedy je třeba specifikovat parametr -variable, kterým svážeme tyto položky s nějakou proměnnou. Analogicky funguje metoda checkbutton. Také se může hodit metoda separator, která vloží do vybrané nabídky oddělovač. Zkusme přidat do našeho příkladu následující kód.

```
$nabidka_upravy = $lista->Menubutton(-text => "Upravy", -borderwidth=>1,
-relief=>"raised")->pack(-side=>"left", -padx=>1);
$nabidka_upravy->radiobutton(-label=>"Zalamovat", -variable =>\$zalamovani);
$nabidka_upravy->radiobutton(-label=>"Nezalamovat", -variable =>\$zalamovani);
$nabidka_upravy->separator();
$nabidka_upravy->checkbutton(-label=>"Ukladat automaticky", -variable =>\$ukladat);
```

Získáme následující menu.

Složitější menu

Menu přes vlastnost hlavního okna

Nyní využijeme speciální vlastnosti hlavního okna prvku Menu. Nejprve tedy vytvoříme menu a poté pomocí configure nastavíme vlastnost -menu hlavního okna. Dále již můžeme vytvářet nabídky pomocí metody cascade a jednotlivé položky pomocí command.

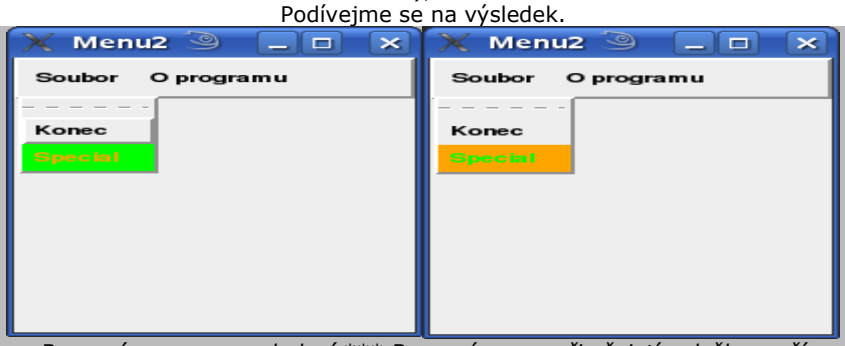
```
$menu = $m->Menu();
$m->configure(-menu => $menu);

$nabidka_soubor = $menu->cascade(-label=>"Soubor");
$nabidka_soubor->command(-label=>"Konec", -command=>sub{exit});

$nabidka_oprogramu = $menu->cascade(-label=>"O programu");
$nabidka_oprogramu->command(-label=>"Info", -command=>sub{$m->messageBox(
-message=>"Tento program vznikl v roce 2010 a jinak nic nedela", -type => "ok")});
```

Menu vytvořené tímto způsobem lze obarvovat. Metoda command má mimo jiné parametry -background a -activebackground (resp. -foreground a -activeforeground), které nastavují stálou barvu a barvu při přejetí myší. Zde je příklad takového barevného tlačítka.

```
$nabidka_soubor->command(
-label => "Special",
```

```
-activebackground => "green",
-activeforeground => "orange",
-background => "orange",
-foreground => "green",
-command => sub{$m->messageBox(-message => "Stiskl jsi barevne tlacitko!")}
);
```

Podívejme se na výsledek.



*Barevné menu po rozbalení *** Barevné menu při přejetí položky myší*

Grafická plocha

Prvek Canvas umožňuje vytvářet základní grafické objekty, to jest body, úsečky, kružnice, křivky, a podobně. Každý takovýto objekt je navíc skutečně objektem i z programátorského hlediska, a tak lze snadno konfigurovat jeho vlastnosti.

Obecně pro nakreslení nějakého objektu na grafickou plochu voláme metodu create. Tato metoda přijímá jako první argument typ objektu a ostatní argumenty již závisejí na něm. Jako typy lze

volit arc, bitmap, image, line, polygon, oval, rectangle, text, window. Také lze volat přímo metody createLine, createOval apod.

Každý vytvořený objekt má nějaké vlastnosti. Zde je několik vlastností, které mají všechny nakreslené objekty.

Vlastnost	Význam
-fill	barva vnitřku
-outline	barva okraje
-width	tloušťka okraje
-stipple, -outlinestipple	vykreslí objekt vzorem

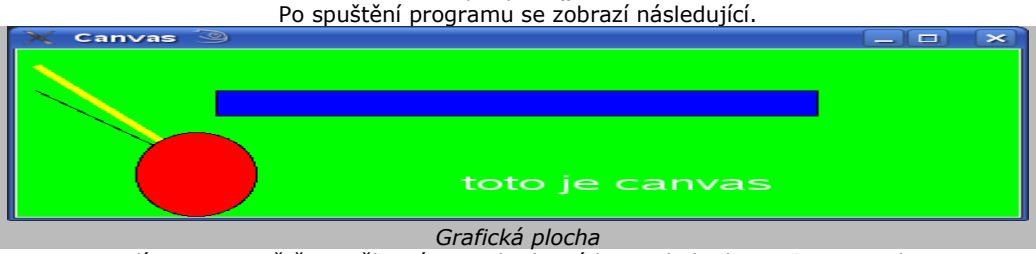
Uveďme ještě vlastnosti specifické pro konkrétní objekty.

Objekt	Parametr	Vlastnost	Význam
Oblouk	arc	-start	počáteční úhel
		-extent	rozpětí
		-style	pieslice, chord nebo arc
Bitmapa resp. obrázek	bitmap resp. picture	-anchor	strany pro ukotvení
		-bitmap resp. -image	soubor
Úsečka	line	-arrow	zobrazení šipky; možné jsou hodnoty none, first, last, both
		-arrowshape	hodnotou je odkaz na pole o 3 prvcích, které specifikují rozměry šipky
Text	text	-text	text, který se má zobrazit
		-justify	zarovnání; možnosti jsou left, right, center
		-anchor	ukotvení
Okno	window	-window	vloží jiný prvek do grafické plochy

Zde je ukázka umístění několika objektů do grafické plochy.

```
$p=$m->Canvas(-width=>500, -height=>200, -background => "green");
$p->create("line", 10, 50, 100, 150);
$p->createLine(10, 20, 100, 150, -width => 5, -fill => "yellow");
$p->createRectangle(100, 80, 400, 50, -fill => "blue");
$p->createOval(60, 100, 120, 200, -fill => "red");
$p->createText(300, 160, -text=>"toto je canvas", -fill=>"white",
-font=>"-*-*-*-*-*-*--25-*-*-*-*-*-*");
$p->pack();
```

Po spuštění programu se zobrazí následující.



Grafická plocha

Podívejme se ještě na některé metody, které lze nad objektem Canvas volat.

Metoda	Význam
create("objekt", x, y, ...), create("objekt", x1, y1, x2, y2, ...)	vytvoří objekt
coords(objekt, nová_pozice), move(objekt, nová_pozice)	přesune objekt na souřadnice nová_pozice resp. na relativní pozici
raise, lower	posune objekt nahoru, dolů
scale	upraví rozměry objektu
delete	smaže objekt
itemconfigure(objekt, parametry)	konfiguruje objekt
postscript	vygeneruje postscript soubor
find("*příkaz*")	vyhledá seznam vyhovujících objektů; například "closest x y", "below $objekt" atd.
bind	svázání s událostí

Perl (118) - Tk - čas a události

K bezproblémovému běhu každé aplikace, která má něco smysluplného umět, je potřeba, abychom uměli vyvolávat a zpracovávat události.

Práce s časem

Pro odložené nebo opakované vykonávání nějaké akce máme v Tk zabudované metody after a repeat. Metodou after vytvoříme jednorázový příkaz, který bude vykonán za určený čas. Toto je v jistém smyslu analogie s unixovým nástrojem at.
Podobně funguje metoda repeat, která vytváří trvalý příkaz. Zde se akce periodicky opakuje.
Vyzkoušíme si to a napíšeme si jednoduchou hru. Ta bude spočívat v tom, že uživatel musí co nejdéle bez přestávky klikat na tlačítko. Jakmile jednou nestihne kliknout včas, hra skončí a oznámíme mu, kolikkrát se mu podařilo kliknout. V programu využijeme ještě metodu cancel, která jednorázový příkaz ruší.

```
use Tk;
use strict;

my $m = MainWindow->new();
my $pocetkliku = 0;
my $casovac;

$m->Label(-text=>"Ukolem hry je klikat na tlacitko co nejdele.")->pack();
$m->Button(-text=>"Aktivita!", -command=>\&klik)->pack();
$m->Button(-text=>"Konec", -command=>sub{exit})->pack();

MainLoop();

sub klik {
$pocetkliku++;
$casovac->cancel if $pocetkliku!=1;
$casovac=$m->after(200, \&konec);
}

sub konec {
$m->messageBox(-message => "Klikl jsi ${pocetkliku}krat.", -type => "ok");
$pocetkliku = 0;
}
```

Události

S některými událostmi jsme se setkali již dříve. Šlo například o stisk tlačítka. Teorii událostí nyní podstatně rozvineme.
S událostmi lze manipulovat pomocí metody bind, kterou lze volat nad libovolným widgetem. Předáváme jí typ události a akci, která se má vykonat. Metoda bind nám umožňuje zachytávat takové události jako je například stisky a uvolnění kláves nebo tlačítek na myši, pohyb kurzorem myši.

Zápis událostí

Událost se zapisuje jako nějaký řetězec uzavřený mezi symboly <>. Přesněji řečeno má každá událost následující tvar.
<klávesa1-klávesa2-...-klávesan-typ-symbolické_jméno_klávesy>
Přitom klávesy můžeme dosadit například pro události z myši Button1, Button2, Double, Triple a pro události z klávesnice Control, Alt, Shift. Any zastupuje libovolnou klávesu.
Typ události nabývá jedné z následujících hodnot:

- ButtonPress - stisk tlačítka myši
- ButtonRelease - uvolnění tlačítka myši
- KeyPress - stisk klávesy
- KeyRelease - uvolnění klávesy
- Motion - pohyb kurzoru
- Enter - vstup kurzoru myši na nějakou oblast
- Leave - opuštění oblasti

Poslední položka, *symbolické_jméno_klávesy*, nabývá obvykle pro tisknutelné znaky tohoto znaku. Pro jiné klávesy existují speciální řetězce jako například Enter, BackSpace, F7 apod. Kompletní seznam možností je na tcl.activestate.com/man/tcl8.4/TkCmd/keysyms.htm.

Často používané události lze různými způsoby zkracovat zkracovat, takže můžeme vynechávat KeyPress a ButtonPress. Uveďme několik příkladů zápisu události.

- <KeyPress-q> - stisknuto tlačítko q
- <Alt-q> - stisknuta tlačítka Alt+q
- <BackSpace> - stisknut mezerník
- <Button2> - stisknuto tlačítko 2 na myši
- <Button2-Motion> - stisknuto tlačítko 2 na myši a ta se přitom pohybuje
- <1> - stisknuto první tlačítko myši
- <g> - stisknuta klávesa g
- <Any-KeyPress> - stisknuta libovolná klávesa

Již bylo řečeno, že příkaz bind sváže událost a akci. bind přijímá jako parametr buď událost nebo posloupnost událostí. Posloupnost se zapisuje jako několik událostí oddělených mezerou.

Zjištění přesné příčiny události

Již jsme zmínili, že událost <Any-KeyPress> znamená stisk libovolné klávesy. Otázkou však je, jak potom zjistíme, která klávesa vlastně byla stisknuta. Řešení nalezneme ve volání funkce Ev, která nám je schopna poskytnout libovolnou informaci o nastálé události. Tato funkce přijímá parametr ve formě řetězce, podle kterého vrací požadovanou informaci. Možné parametry jsou například následující.

parametr	Význam
x	x-souřadnice myši
y	y-souřadnice myši
k	název klávesy
t	čas události

```
use Tk;

$m = MainWindow->new;
$m->configure(-width=>400, -height=>400);

$p = $m->Canvas(-cursor=>"crosshair", -width=>400, -height=>400, -background=>"white")->pack;

for($i=10; $i<200; $i+=10){
$h=255/400*$i;
$p->createRectangle($i, $i, 400-$i, 400-$i, -outline=>sprintf("#%02x%02x%02x", $h, $h, $h));
}
$status=$m->Label(-text=>"Klikni nekam do okna.")->pack();

$p->Tk::bind( "<Button-1>", [sub{$status->configure(
-text=>"Klikl jsi na pozici [$_[1], $_[2]]")}, Ev("x"), Ev("y")]);
MainLoop;
```

Zajímavý je pro nás předposlední řádek. Jakmile klikneme levým tlačítkem někam do okna, vyvolá se událost <Button-1>. Protože jsme pomocí bind vytvořili k této události vazbu, dojde k provedení příslušné akce. Připomeňme, že, chceme-li akci předat parametry, musíme je vložit společně s odkazem na podprogram do pole.

Předchozí program vytvoří následující okno.

Zachycení události (kliknutí do prostoru okna) a výpis souřadnic kurzoru myši

Vyvolání více událostí zároveň

Podívejme se nyní na situaci, kdy uživatel vykonal dvojité kliknutí myši. Nastaly dvě události: <Button-1> a <Double-Button-1>. V takovém případě má přednost méně obecná událost, tedy v našem případě <Double-Button1>. To znamená, že po prvním kliknutí je okamžitě vyvolána událost <Button-1> a po druhém kliknutí už pouze <Double-Button-1>.

Perl (119) - Tk - CD man

Zúročíme znalosti z předchozích dílů a naprogramujeme si zjednodušenou hru CD man.

Na závěr minisérie o Tk využijeme naše dosavadní znalosti a napíšeme si hru.

O hře

CDman, PACman, Waka-waka, Pakuman; to jsou jen některé z názvů pro dnes již 30 let starou kultovní japonskou hru. Hra se hraje obvykle v bludišti. Hráč hraje za žluté kolečko a jeho cílem je projít všechny políčka, to jest sežrat na každém políčku tečku. Proti hráči hraje jeden nebo několik duchů, kteří se mu snaží ve žraní teček zabránit. Jakmile ho chytí, hra končí neúspěchem. Když se hráči podaří sežrat všechny tečky, hra končí.

Existuje obrovské množství variant této hry, mnohé přidávají speciální předměty, průchozí tunely nebo políčka se speciálními funkcemi. Informace o historii a možnostech různých variant hry lze nalézt na wikipedii.

Cíl

Naším cílem bude implementovat jednoduchou variantu CDmana. Vytvoříme tedy nějakou hrací plochu, na které bude žluté kolečko pro hráče a duch jako nepřítel, který se ho bude snažit dostihnout. V okně bude ještě informace o tom, kolik teček bylo již sežráno a dvě tlačítka - jedno pro start hry nebo pozastavení a druhé pro ukončení programu.

Implementace

Základní program se bude skládat ze čtyř příkazů. Nejprve vykreslíme obrazovku a aktivujeme klávesy tak, že je svážeme s událostmi. Na tyto obě činnosti si vytvoříme vlastní podprogramy.

```
my $m = MainWindow->new();
$m->title("CDman");

&vykresli_obrazovku();
&aktivuj_klavesy();
```

Vykreslení prvků je rutinní záležitost a proto není třeba se ji tolik zabývat. Nejprve vytvoříme grafickou plochu Canvas, do které vykreslíme mřížku pomocí opakovaného volání createLine. Dále vytvoříme podle požadavků dvě tlačítka a ještě přidáme stavový řádek, ve kterém bude vidět, kolik teček již bylo sežráno.

Tečky je třeba vykreslit a protože je budeme časem postupně mazat, ponecháme si jejich id, které jim přiřadila grafická plocha. Dále vytvoříme CD mana a ducha. CD man je v podstatě oblouk vyplněný žlutou barvou a proto můžeme použít metodu createArc. Ducha pomocí takto jednoduché grafiky nenakreslíme, ale musíme jej importovat z nějakého obrázku.

```
sub vykresli_obrazovku {
$canvas = $m->Canvas(-width=>$SLOUPCU*$TLOUSTKA, -height=>$RADKU*$TLOUSTKA, -relief=>"ridge",
-background=>"green", -border=>0)->pack();
$t1 = $m->Button(-text => "Start", -command => \&start)->pack(-side=>"left");
$l = $m->Label(-text => "Hotovo: 1/$CIL_HRY")->pack(-side=>"left");
$t2 = $m->Button(-text => "Konec", -command => sub{exit})->pack(-side=>"right");

#mrizka
foreach my $i (0..$RADKU){
$canvascreateLine(0, $i*$TLOUSTKA, $SLOUPCU*$TLOUSTKA, $i*$TLOUSTKA,-fill => "lightgreen");
}
foreach my $i (0..$SLOUPCU){
$canvascreateLine($i*$TLOUSTKA, 0, $i*$TLOUSTKA, $RADKU*$TLOUSTKA, -fill => "lightgreen");
}

#tecky
for(my $i=2;$i<=$SLOUPCU-1;$i++){
for(my $j=2;$j<=$RADKU-1;$j++){
$m[$i][$j]=$canvascreateArc(($i-0.6)*$TLOUSTKA, ($j-0.6)*$TLOUSTKA, ($i-0.4)*$TLOUSTKA,
($j-0.4)*$TLOUSTKA, -fill=>"red", -outline=>"red");
}
}

#cdman
$cdman=$canvascreateArc($TLOUSTKA*($x-1)+$SP, $TLOUSTKA*($y-1)+$SP, $x*$TLOUSTKA-$SP,
$y*$TLOUSTKA-$SP, -start=>30, -extent=>300, -fill=>"yellow");

#nepritel
my $obrazek = $m->Photo(-file=>"Ghost.gif");
$nepritel=$canvascreateImage($TLOUSTKA*($nepritelx-.5)+$SP, $TLOUSTKA*($nepritely-.5)+$SP,
-image=>$obrazek);

#okoli
$canvascreateRectangle($TLOUSTKA/2, $TLOUSTKA/2, ($SLOUPCU-.5)*$TLOUSTKA, $TLOUSTKA*($RADKU-.5),
-outline=>"darkgreen", -width=>$TLOUSTKA);
}
```

Použili jsme zde několik konstant, které je třeba nastavit na začátku programu. Zvolme je tedy například takto.

```
my $SLOUPCU=7;
my $RADKU=7;
my $TLOUSTKA=50;
my $KROK=0.1;
my $STARTX=2;
my $STARTY=2;
my $NEPRITELX=4;
my $NEPRITELY=5;
my $SP=2;
my $CIL_HRY=($SLOUPCU-2)*($RADKU-2);
```

Aktivace kláves spočívá ve volání metody bind. Vytvoříme 6 událostí - pro stisky šipek jako změnu směru CD mana a dále klávesy P pro pozastavení a s pro start hry.

```
sub aktivuj_klavesy{
$m->bind("<Left>", \&doleva);
$m->bind("<Right>", \&doprava);
$m->bind("<Up>", \&nahoru);
$m->bind("<Down>", \&dolu);
$m->bind("<p>", \&pauza);
$m->bind("<s>", \&start);
}
```

Nyní napišme čtyři metody použité pro změnu směru. Pokud uživatel stiskne nějakou kurzorovou klávesu, je třeba udělat dvě věci. Otočit CD mana příslušným směrem (tj. změnit vlastnosti oblouku) a dále si zapamatovat směr (to uděláme uložením do proměnné $smer).

```
sub doprava{
$canvas->itemconfigure($cdman, -start=>30, -extent=>300);
$smer="p";
}

sub doleva{
$canvas->itemconfigure($cdman, -start=>210, -extent=>300);
$smer="l";
}

sub nahoru{
$canvas->itemconfigure($cdman, -start=>120, -extent=>300);
$smer="n";
}

sub dolu{
$canvas->itemconfigure($cdman, -start=>300, -extent=>300);
$smer="d";
}
```

Dalšími podprogramy, na které jsme se odkazovali výše, jsou podprogramy pro obsluhu tlačítek.
Pokud uživatel klikne na start, spustíme časovač, který každý časový interval přednastavené délky obnoví obrazovku. Tím do aplikace dodáme pohyb, protože v podprogramu refresh můžeme periodicky manipulovat s objekty na grafické ploše.

```
sub start {
$t1->configure(-text=>"Pauza", -command=>\&pauza);
$timer=$canvas->->repeat($refresh, \&refresh);
}
```

Pokud uživatel pozastaví hru, volá se podprogram pauza. Ta zruší náš časovač, čímž hra ustrne.

```
sub pauza {
$t1->configure(-text=>"Start", -command=>\&start);
$timer->cancel;
}
```

Nyní je třeba dopsat hlavní část programu, kterou je podprogram refresh.

```
sub refresh{
#pokud je blízko CD mana nějaká tečka, tak jí sežereme
#zkontrolovat, zda nenastal konec hry
#posunout CD mana o zadaný směr vpřed, pokud tedy nenarazil do překážky
#posunout ducha směrem k CD manovi
}
```

Nejprve posuneme CD mana. Máme 4 možné směry. Každý z nich obsloužíme zvlášť. Cílem je posunout CD mana o daný krok, pokud již nenarazil do zdi. Řešením je například následující série podmínek.

```
if($smer eq "p" and $x+$KROK<$SLOUPCU-1){
$x+=$KROK;
$canvas->move($cdman, $KROK*$TLOUSTKA, 0);
}elsif($smer eq "l" and $x>2){
$x-=$KROK;
$canvas->move($cdman, -$KROK*$TLOUSTKA, 0);
}elsif($smer eq "n" and $y>2){
$y-=$KROK;
$canvas->move($cdman, 0, -$KROK*$TLOUSTKA);
}elsif($smer eq "d" and $y+$KROK<$RADKU-1){
$y+=$KROK;
$canvas->move($cdman, 0, $KROK*$TLOUSTKA);
}
```

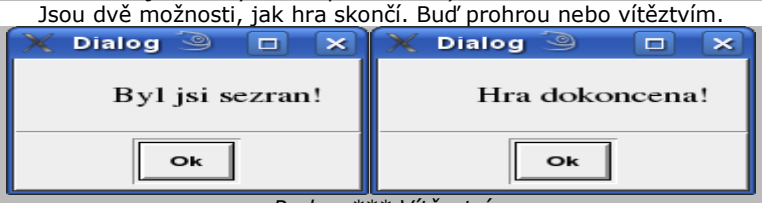
Pojďme nyní posunout ducha. Toho žádný uživatel neovládá, takže to musíme udělat sami. Porovnáme souřadnice CD mana a ducha a podle toho s ním pohneme.
Jsou (pro jednoduchost) dva možné směry, jak se může duch CD manovi přiblížit - buď vertikální nebo horizontální. My si jeden z nich vylosujeme.

```
if(int(rand(2))==1){
if($x>$nepritelx+.5){
$nepritelx+=$KROK;
$canvas->move($nepritel, $KROK*$TLOUSTKA*0.7, 0);
}else{
$nepritelx-=$KROK;
$canvas->move($nepritel, -$KROK*$TLOUSTKA*0.7, 0);
}
}else{
if($y>$nepritely+.5){
$nepritely+=$KROK;
$canvas->move($nepritel, 0, $KROK*$TLOUSTKA*0.7);
}else{
$nepritely-=$KROK;
$canvas->move($nepritel, 0, -$KROK*$TLOUSTKA*0.7);
}
```

```
}
```
Ještě jsme se nepodívali na to, zda je v aktuálním okamžiku CD man na nějaké tečce, kterou by mohl sežrat. To uděláme tak, že spočítáme vzdálenost CD mana vzhledem ke každé z teček (pro jednoduchost) a pokud bude nějaká hodně blízko, tak ji smažeme.
Dále zde potřebujeme počítat, kolik teček jsme již sežrali. Pokud tedy sežereme nějakou tečku, uchováme si její souřadnice a počet sežraných teček zvýšíme o jednu.
Jakmile docílíme maximálního počtu sežraných teček, můžeme již zde ukončit hru.
```
for(my $i=2;$i<=$SLOUPCU-1;$i++){
 for(my $j=2;$j<=$RADKU-1;$j++){
if(sqrt(($i-$x)**2+($j-$y)**2)<1/3 and not (grep $_ eq "$i-$j", @sezrano)){
 push(@sezrano, "$i-$j");
 $canvas->delete($m[$i][$j]);
 $sezrano++;
$l->configure(-text => "Hotovo: $sezrano/$CIL_HRY");
 &uspech if($sezrano==$CIL_HRY);
 }
 }
 }
```
Stejným způsobem jako u žraní teček zkontrolujeme, zda náhodou duch nesežere CD mana. Spočítáme tedy vzdálenost a pokud jsou oba objekty příliš blízko, ukončíme hru voláním podprogramu neuspech.
```
if(sqrt(($x-$nepritelx-.5)**2+($y-$nepritely-.5)**2)<1/2){
 &neuspech;
 }
```
Nyní nám již zbývá pouze implementace podprogramů neuspech a uspech. Vytvoříme tedy vyskakovací okno, které se po zavolání těchto podprogramů objeví. Oznámíme tak výsledek hry. Tímto hra končí.
```
sub uspech {
 $timer->cancel;
 $t1->configure(-state=>"disabled");
$m->messageBox(-message => "Hra dokoncena!", -type => "ok");
 }

sub neuspech {
 $timer->cancel;
 $t1->configure(-state=>"disabled");
$m->messageBox(-message => "Byl jsi sezran!", -type => "ok");
 $canvas->itemconfigure($cdman, -fill=>"black");
 }
```
Závěr
Celou hru si můžete stáhnout v souborech cdman.pl a Ghost.png.
Nyní si hru můžeme obvyklým způsobem spustit. Zde je několik screenshotů.

*Před zahájením hry *** V průběhu hry *** Po sežrání duchem*

Jsou dvě možnosti, jak hra skončí. Buď prohrou nebo vítězstvím.



*Prohra *** Vítězství*

Tato varianta CD mana je velice jednoduchá a byla psána jen jako demonstrace PerlTk. Existuje řada návrhů na vylepšení. Mohli bychom pokračovat vytvořením bludiště. To už není tak jednoduché a bylo by to poměrně náročné na psaní kódu - potřebovali bychom vytvořit několik dalších objektů na grafické ploše, uchovat (například ve dvojrozměrném poli) pozice zdí a kontrolovat, zda se CD man pohybuje pouze v chodbách.

Dále bychom mohli vytvořit nějaké speciální objekty, které by CD man mohl sežrat a získat nové (ačkoliv třeba časově omezené) schopnosti - například se v mnoha variantách vyskytují v rozích speciální tečky, které umožňují CD manovi po několik sekund prohodit role s duchy a sežrat je.

Perl (120) - Wx - základní práce s widgety

Druhým widget toolkitem v seriálu bude wxPerl. Tento díl má za úkol uvést čtenáře do problematiky a představit několik základních widgetů.

Wx (známé též jako "wxWidgets" nebo "Windows and X widgets"; do roku 2004 se hovořilo o wxWindows) je multiplatformní grafická knihovna základních widgetů, která je používána pro tvorbu grafických uživatelských rozhraní. Knihovna je původně napsána pro C++, avšak je možné ji využívat v řadě dalších jazyků. Perl je jedním z nich. Mutace knihovny pro Perl se nazývá wxPerl.

Představíme si pár základních ideí, které by měly být spíše motivací k dalšímu studiu, než-li obsáhlým návodem. Nevýhodou wxPerl je, že neexistuje pořádná dokumentace a v podstatě ani žádné tutoriály apod. K dispozici je pouze dokumentace Wx pro jazyk C++, na kterou si lze po čase zvyknout.

Instalace

Modul stáhneme z archivu CPAN a nainstalujeme následujícím příkazem.

```
$ cpan Wx
```

wxWidgets a Perl

Jak tedy používat wxWidgets? Podívejme se na pár základních pravidel.

- Každý program, který budeme ve wxPerl psát, bude obsahovat základní třídu. Tato třída bude dědit od Wx::App.

- Jakmile spustíme program, je automaticky volána metoda s názvem OnInit. Zde budeme řídit všechny globální události. Budeme zde také vytvářet okna.
- Každé okno má svoje ID, pomocí kterého se na něj budeme v budoucnu odkazovat.
- Jak to tak u mnoha grafických uživatelských rozhraní bývá, musíme ručně vytvořené widgety zobrazit.

První program

Na základě těchto pravidel napíšeme základní schéma programu ve wxWidgets. Vytvořme tedy neprve základní třídu. Nazveme ji MojeAplikace.

```
use Wx;
package MojeAplikace;
use base "Wx::App";
```

Do ní umístíme jedinou metodu OnInit, která nám pouze vytvoří okno. Okno vytoříme pomocí metody Wx::Frame->new. Tato metoda přijímá několik parametrů.

- ID rodičovského okna nebo undef
- ID nebo -1 pro náhodné ID
- titulek
- pozice - anonymní pole tvaru [x, y]
- velikost - anonymní pole tvaru [x, y]

Příkladem volání nechť je tento příkaz.

```
Wx::Frame->new(undef, -1, "Prvni aplikace", [-1000, -1000], [400, 300])
```

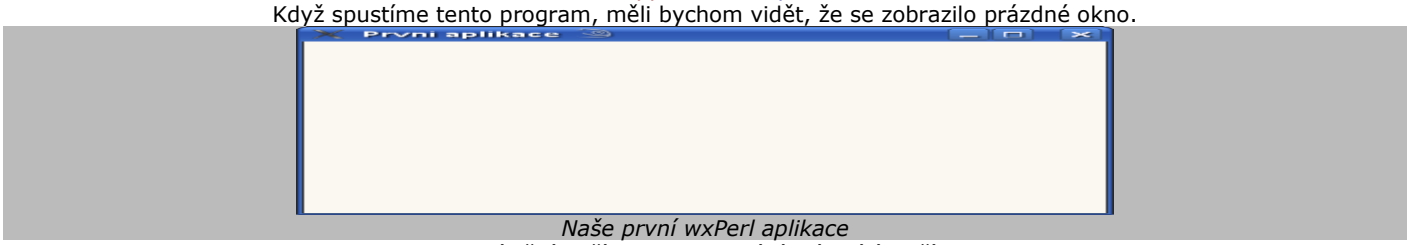
Pomocí metody Show také nesmíme zapomenout každé naše okno zobrazit. Uveďme nyní celou metodu OnInit.

```
sub OnInit {
my $okno = Wx::Frame->new(undef, -1, "Prvni aplikace", [-1000, -1000], [400, 300]);
$okno->Show(1);
}
```

Tímto je naše aplikace hotova. Zbývá ji použít v balíku main a vyzkoušet. Vytvoříme tedy objekt typu MojeAplikace a spustíme smyčku událostí metodou MainLoop.

```
package main;
my $app = MojeAplikace->new;
$app->MainLoop;
```

Když spustíme tento program, měli bychom vidět, že se zobrazilo prázdné okno.



Naše první wxPerl aplikace

Umístění tlačítka - upravování stávajících tříd

Náš předchozí příklad vylepšíme tak, aby se v okně objevilo tlačítko. To uděláme tak, že upravíme stávající třídu Wx::Frame (ta už je modulem předdefinovaná). Vytvoříme tedy novou třídu FrameSTlacitkem (ta bude dědit od zmíněné Wx::Frame), ve které napíšeme vlastní konstruktor.

```
package FrameSTlacitkem;
use base "Wx::Frame";
sub new {
my $r = shift;
my $self = $r->SUPER::new(@_);
#zde vytvoříme tlačítko
}
```

Všimněme si, že jako parametr metody SUPER::new předáváme @_. To znamená, že FrameSTlacitkembude přijímat stejné parametry jako původní Wx::Frame.

Tlačítko a další widgety se obvykle neumísťují přímo do okna typu Wx::Frame, ale do speciálního widgetu Wx::Panel. Proč, to teď není důležité. Vytvořme tedy nejprve nový panel.

```
my $panel = Wx::Panel->new($self,-1);
```

Dále vytvořme tlačítko. Umístíme ho do panelu. To znamená, že jako první argument vepíšeme právě vytvořený panel, který je reprezentován v proměnné $panel.

```
my $button = Wx::Button->new($panel, -1, "Zavrit", [10, 10], [-1, -1]);
```

Náš konstruktor tedy bude vypadat takto.

```
sub new {
my $r = shift;
my $self = $r->SUPER::new(@_);
my $panel = Wx::Panel->new($self,-1,);
my $button = Wx::Button->new($panel, -1, "Zavrit", [10, 10], [-1, -1]);
return $self;
}
```

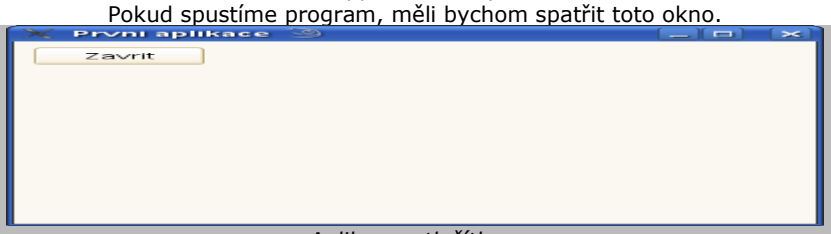
Nyní napíšeme základní třídu, která bude náš nový FrameSTlacitkem používat.

```
package MojeAplikace;
use base "Wx::App";

sub OnInit {
my $frame = FrameSTlacitkem->new(undef, -1, "Prvni aplikace", [-1000, -1000], [400, 300]);
$frame->Show(1);
}
```

A na závěr balík main, kde tuto třídu použijeme. Bude vypadat stejně jako posledně.

```
package main;
my $app = MojeAplikace->new;
$app->MainLoop;
```

Pokud spustíme program, měli bychom spatřit toto okno.



Aplikace s tlačítkem

Prostý text

Pro zobrazení prostého textu lze použít widget Wx::StaticText. Použití je následující.

```
Wx::StaticText->new($panel, -1, "Ahoj", [50, 50],);
```

Sešit

Ještě se pro lepší představu podívejme na jeden widget. Ukážeme si například třídu Wx::Notebook, což jsou klasické panely, které se používají kvůli přehlednosti nebo když na stránce není dost místa.

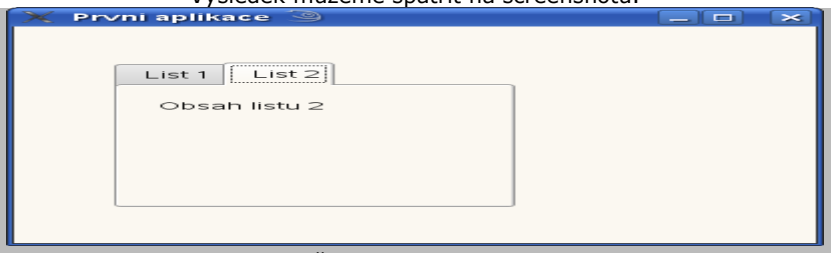
Vytvořme tedy v konstruktoru následující kód.

```
my $panel = Wx::Panel->new($self);
my $notebook = Wx::Notebook->new($panel, -1, [50, 50], [200, 200]);
```

Tím jsme vytvořili notebook. Přidáme do něj dva listy a do každého napíšeme nějaký text.

```
my $list1 = Wx::Panel->new($notebook);
Wx::StaticText->new($list1, -1, "Obsah listu 1", [20, 20]);
$notebook->AddPage($list1,"List 1"');

my $list2 = Wx::Panel->new($notebook);
Wx::StaticText->new($list2, -1, "Obsah listu 2", [20, 20]);
$notebook->AddPage($list2, "List 2");
```

Výsledek můžeme spatřit na screenshotu.



Sešit (widget Notebook)

Další widgety

Z předcházejícího by mělo být jasné, jak práce s widgety funguje. Proto již nebudeme představovat další, ale odkážeme na wxPerl wiki cheat sheet, kde je seznam widgetů a některé dokonce popsané.

Perl (121) - Wx - události

Druhá polovina tutoriálu o wxPerl obsahuje především návod, jak pracovat s událostmi. Jeho aplikace je ukázána na práci s menu.

Zásadní nevýhodou je, že nám tlačítko vytvořené v předchozí aplikaci nefunguje tak, jak má. Dále se podíváme, jak ho pomocí správy událostí zprovoznit.

wxPerl funguje tak, že nastavuje tzv. konektory pro řízení událostí. To znamená, že pro každý typ události zde máme funkci, která nám ji obslouží.

Upravme náš program tak, aby se po kliknutí na tlačítko aplikace uzavřela. Budeme tedy chtít volat funkci exit, jakmile uživatel tlačítko stiskne.

To obnáší tři jednoduché věci.

- zavést modul Wx::Event
- napojit událost (v našem případě stisk tlačítka) na někakou metodu, která se provede
- napsat tuto metodu

Nás bude nyní zajímat zejména funkce EVT_BUTTON pro obsluhu tlačítek. Proto ji importujeme do třídy FrameSTlacitkem.

```
use Wx::Event qw(EVT_BUTTON);
```

Uvnitř konstruktoru ji zavoláme a napojíme pomocí ní naše tlačítko $button na podprogram konec, který v zápětí napíšeme.

```
EVT_BUTTON($self, $button, \&konec);
```

Celkem jsme tedy modifikovali třídu FrameSTlacitkem do následující podoby. Vše ostatní zůstává stejné.

```
package FrameSTlacitkem;
use Wx::Event qw(EVT_BUTTON);
use base "Wx::Frame";

sub new {
my $ref = shift;
my $self = $ref->SUPER::new(@_);
my $panel = Wx::Panel->new($self,-1,);
my $button = Wx::Button->new($panel, -1, "Zavrit", [10, 10], [-1, -1]);
EVT_BUTTON( $self, $button, \&konec );
return $self;
}
```

```
sub konec {exit;}
```
Jakmile nyní spustíme aplikaci a stiskneme tlačítko, vyvolá se událost a protože ji máme pomocí EVT_BUTTON napojenou na funkci konec, program skončí.

Menu - praktická ukázka použití událostí

Ukažme si, jak v aplikaci vyrobit klasické menu. Budeme postupovat velice podobně jako doposud.

Balík main bude vypadat opět stejně.

```
#!/usr/bin/perl -w
use strict;
use Wx;

package main;
my $app = MojeAplikace->new;
$app->MainLoop;
```
Balík MojeAplikace se v podstatě také příliš lišit nebude.
```
package MojeAplikace;
use base "Wx::App";

sub OnInit {
my $frame = FrameSMenu->new(undef, -1, "Prvni aplikace", [-1000, -1000], [400, 300]);
$frame->Show(1);
}
```
Zbývá nám napsat konstruktor balíku FrameSMenu. Opět samozřejmě dědíme od Wx::Frame.
```
package FrameSMenu;
use base "Wx::Frame";

sub new {
my $ref = shift;
my $self = $ref->SUPER::new(@_);

#zde napíšeme menu

return $self;
}
```
A konečně můžeme psát to, proč to celé děláme. Vytvoření menu obnáší následující.

- Vytvoření menubaru (to je obvyklý pruh nabídek, který je při běhu aplikace stále vidět)
- Vytvoření jednotlivých nabídek a jejich vložení do menubaru
- Aktivování menubaru
- (Dále lze pro různé položky nabídek v menu vytvářet akce)

Začněme druhým krokem. Vytvoříme si pro ukázku dvě nabídky. Nabídku vytvoříme standardním Wx::Menu a poté do ní pomocí metody Append přidáváme položky a případně pomocí AppendSeparator vkládáme oddělovače.

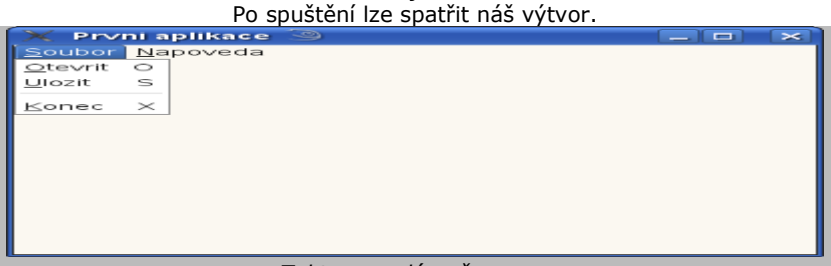
Dále je možné používat AppendCheckItem a AppendRadioItem pro checkbuttony a radiobuttony. Taktéž lze vytvářet submenu pomocí AppendSubMenu (to obnáší vytvoření nabídky pomocí Wx::Menu, pro kterou poté pomocí AppendSubMenu vytvoříme položku v menu).

Standardní nabídka Soubor může vypadat například nějak takto.
```
my $menu1 = Wx::Menu->new();
$menu1->Append(1, "&Otevrit\tC-o","Otevrit");
$menu1->Append(2, "&Ulozit\tC-s","Ulozit");
$menu1->AppendSeparator();
my $menu_konec = $menu1->Append(-1, "&Konec\tC-x","Konec");
```
Pro komplexnější představu si vytvořme ještě nabídku Nápověda.
```
my $menu2 = Wx::Menu->new();
$menu2->Append(4, "&O programu","O programu");
```
Tímto je druhý krok hotov. Ještě jsme nevytvořili menubar, což učiníme právě teď. Menubar je schován v třída Wx::MenuBar a metodou Append do něj vkládáme již existující nabídky. Použití je tedy v našem případě následující.
```
my $menubar = Wx::MenuBar->new();
$menubar->Append($menu1, "&Soubor");
$menubar->Append($menu2, "&Napoveda");
```
Zbývá nám aktivovat menubar. K tomu máme metodu SetMenuBar.
```
$self->SetMenuBar($menubar);
```
Nyní bychom samozřejmě měli jednotlivým položkám v menu přiřadit nějakou činnost. Ukažme si, jak bychom rozchodili položku Konec v nabídce Soubor.

Nejprve je třeba importovat potřebné akce z Wx::Event. To se udělá stejně jako jsme to dělali u akcí pro tlačítka. Importujme tedy EVT_MENU.
```
use Wx::Event qw(EVT_MENU);
```
A nyní již jen přímočaře voláme EVT_MENU a definujeme požadovanou akci.
```
EVT_MENU($self, $menu_konec, sub{$_[0]->Close(1)});
```
Celý balík FrameSMenu tedy vypadá takto.
```
package FrameSMenu;
use Wx::Event qw(EVT_MENU);
use base "Wx::Frame";

sub new {
my $ref = shift;
my $self = $ref->SUPER::new(@_);
```

```
my $menu1 = Wx::Menu->new();
$menu1->Append(1, "&Otevrit\tC-o","Otevrit");
$menu1->Append(2, "&Ulozit\tC-s","Ulozit");
$menu1->AppendSeparator();
my $menu_konec = $menu1->Append(-1, "&Konec\tC-x","Konec");

my $menu2 = Wx::Menu->new();
$menu2->Append(4, "&O programu","O programu");

my $menubar = Wx::MenuBar->new();
$menubar->Append($menu1, "&Soubor");
$menubar->Append($menu2, "&Napoveda");

$self->SetMenuBar($menubar);

EVT_MENU($self, $menu_konec, sub{$_"[0]->Close(1)});

return $self;
}
```

Po spuštění lze spatřit náš výtvor.



Takto vypadá naše menu

Informativní okno

Rozšiřme ještě předchozí příklad a zprovozněme položku O programu v druhé nabídce našeho menu. Chceme, aby po kliknutí na tuto položku vyskočilo nějaké okno se zprávou.

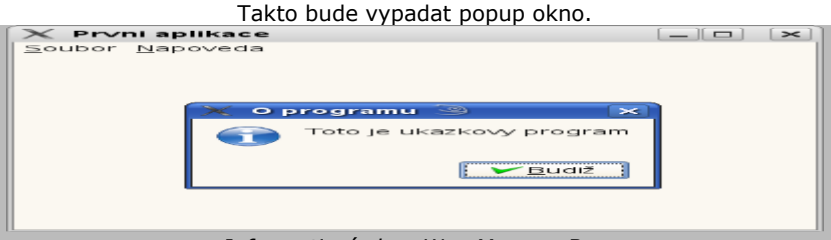
Nejprve tedy standardně vytvoříme v konstruktoru akci.

```
EVT_MENU($self, 4, \&oprogramu);
```

A následně již stačí jen vytvořit podprogram oprogramu. Zprávu zobrazíme intuitivním použitím widgetu Wx::MessageBox.

```
sub oprogramu {
my $self = shift;
Wx::MessageBox("Toto je ukazkovy program", "O programu", "", $self);
}
```

Takto bude vypadat popup okno.



Informativní okno Wx::MessageBox

Perl (122) - Gtk2 - úvod

Další GUI knihovnou, kterou lze používat v Perlu je Gtk2. Tento toolkit toho již dokáže opravdu mnoho a lze s ním psát plnohodnotné aplikace. Podívejme se, jak na to.

Sada knihoven GTK+ vznikla původně jako náhrada Motifu pro potřebu grafického editoru GIMP verze 1.0. Je výsledkem skupiny studentů XCF z univerzity v Berkeley z roku 1997. Z názvu Gimp také pochází zkratka GTK. Znamená totiž Gimp Tool Kit. Dnes patří GTK vedle Qt ke dvěma nejpoužívanějším toolkitům pro X11.

GTK+ je napsáno v jazyce C, avšak existují vazby na početnou skupinu známých jazyků. Pro zajímavost můžeme nahlédnout na oficiální stránku, kde je seznam podporovaných jazyků.

Gtk2 a Perl

Projekt podpory Gtk2 v Perlu má oficiální stranku na gtk2-perl.sourceforge.net. Najdeme zde dokumentaci (ta se mi ovšem nezdá příliš vhodná pro ty, kteří s Gtk2 začínají) a případně další informace.

Gtk2 by ve většině distribucí již mělo být připraveno k použití. Je třeba mít nainstalovaný modul Gtk2, což lze uskutečnit tradičním příkazem cpan.

```
# cpan Gtk2
```

Dokumentace

Kdo již Gtk2 zná (používal ji s jazykem C) a chce ho pouze používat v Perlu, může se podívat na změny mezi rozhraními. Používat původní dokumentaci zde totiž není příliš těžké. Stačí si zvyknout na několik pravidel. Velmi pěkně je udělaná také dokumentace PyGtk2, která má k perl-gtk2 ještě o něco blíže. Jak již bylo zmíněno, Existuje též oficiální POD dokumentace, avšak ta je užitečná pouze pro dohledávání syntaxe a vyžaduje již větší zkušenost Gtk2, neboť neobsahuje téměř žádné komentáře.

Gtk2 a čeština

Gtk2 bezproblémově podporuje kódování utf-8. Proto není problém vytvářet aplikace s českou diakritikou (nebo prakticky libovolnými národními znaky). Potřebujeme k tomu dvě věci.

- uložit zdrojový kód v kódování utf-8 (to u nejpoužívanějších textových editorů obvykle nabízí přímo dialog pro uložení)
- načíst modul utf8

```
use utf8;
```

TRUE a FALSE hodnoty

Je vhodné využívat předdefinované pravdivostní konstanty TRUE a FALSE, které exportujeme do programu následovně.

```
use Glib qw(TRUE FALSE);
```

První program

Již tradičně začínáme u grafických knihovem programem, který zobrazí prázdné okno. Držme se této konvence dále a napišme nejjednodušší možnou grafickou aplikaci.

Již víme, že budeme v každém Gtk2 programu používat modul Gtk2.

```
use Gtk2;
```

Ve všech Gtk2 aplikacích se používá ještě jeden příkaz - init. Ten se se pokusí najít nějaký displej, kde se aplikace zobrazí (Mimo X Server nebo jiné grafické prostředí init selže; pomocí init_check můžeme zkontrolovat, zda je aktuálně takové prostředí dostupné - pokud není, může program pokračovat jinak a použít například textové rozhraní).

```
Gtk2->init;
```

Avšak místo něj lze volat use se speciálním parametrem, jak budeme často činit v budoucnu. Tím zredukujeme většinu aplikací o jeden řádek.

```
use Gtk2 "-init";
```

Program budeme ukončovat vždy vstupem do smyčky událostí. Pokaždé, když se program dostane k této smyčce, se zastaví a čeká na události, které vzápětí obsluhuje. Jakmile je obslouží, čeká na další události. To se opakuje až do ukončení programu.

```
Gtk2->main;
```

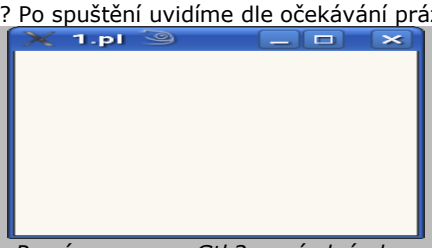
Zbývá vyřešit zobrazení okna. Okno je widget. Vytvoříme ho jako instanci Gtk2::Window. Konstruktoru předáme parametr typu Gtk2::WindowType, který může nabývat hodnot 'toplevel' nebo 'popup'. Každý widget nakonec zobrazíme pomocí metody show.

```
$okno = Gtk2::Window->new("toplevel");
$okno->show;
```

Dejme tedy dohromady všechny tyto poznatky. Náš úvodní Gtk2 program bude vypadat takto.

```
use Gtk2;

Gtk2->init;

$okno = Gtk2::Window->new("toplevel");
$okno->show;

Gtk2->main;
```

A výsledek? Po spuštění uvidíme dle očekávání prázdné okno.



První program v Gtk2 - prázdné okno

Možnosti oken

Metodou set_position nastavujeme pozici okna na displeji. Následující dva příklady budeme dále občas potřebovat.

```
$okno->set_position("center");
$okno->set_position("center-always");
```

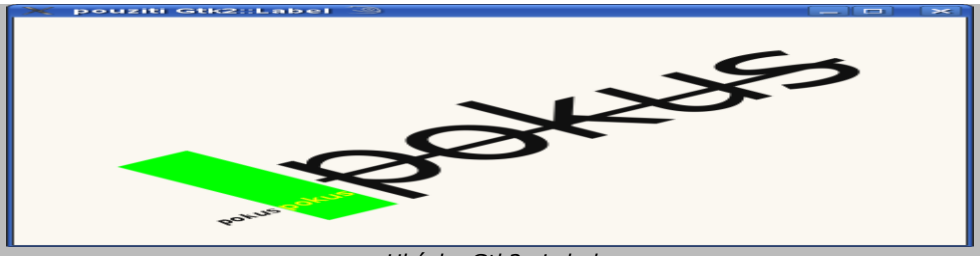
Přidejme text

Text se vytváří ve widgetu Gtk2::Label. Užitečné je, že text lze formátovat markup jazykem. Je to podstatně přímočařejší, než používat nějaké speciální metody. Pro úpravy textu jako je velikost písma, rodina písma, styl, barvy atd. lze používat značky podobné jako v HTML.

Důležité je, že každý widget musíme přiřadit do nějakého okna metodou add (později uvidíme, že lze přiřazovat i do jiných widgetů) a nechat ho zobrazit metodou show.

```
use Gtk2 "-init";

$okno = Gtk2::Window->new("toplevel");
$okno->set_border_width(20);

$label = Gtk2::Label->new();
$label->set_angle(44);
$label->set_markup("<b><i>pokus</i></b> <span background='green' foreground='Yellow'
size='large'><b>pokus</b></span> <s><span size='100000'>pokus</span></s>");

$okno->add($label);
$label->show;
$okno->show;

Gtk2->main;
```

Jak vidíme, styl, barvy, velikost stejně jako otočení a další parametry můžeme nastavovat velmi jednoduše.



Ukázka Gtk2::Label

Přidejme tlačítko

Tlačítko je reprezentováno widgetem Gtk2::Button. Standardně tedy vytvoříme tlačítko.

```
$tlacitko = Gtk2::Button->new("Odeslat");
```

Tlačítko bychom vždy měli přiřadit nějakému oknu, kde se má zobrazit. Funguje to stejně, jako jsme to dělali u zobrazení textu. Provedeme to tedy metodou add.

```
$okno->add($tlacitko);
```

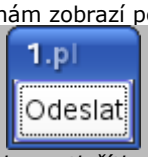
Samozřejmě nesmíme zapomenout na zobrazení tlačítka.

```
$tlacitko->show;
```

Jak tedy vypadá náš upravený program? Podívejme se na aktuální verzi zdrojového kódu.

```
use Gtk2;

Gtk2->init;

$okno = Gtk2::Window->new("toplevel");

$tlacitko = Gtk2::Button->new("Odeslat");

$okno->add($tlacitko);

$tlacitko->show;
$okno->show;

Gtk2->main;
```

Podívejme se, co se nám zobrazí po spuštění programu.

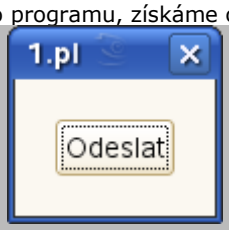


Okno s tlačítkem

Tlačítko je zde přes celé okno, což není příliš hezké. Přidejme si do okna rámeček.

```
$okno->set_border_width(20);
```

Když tento řádek připíšeme do programu, získáme okolo tlačítka kýžený rámeček.



Okno s tlačítkem

Deaktivace tlačítka

Metoda set_sensitive je přepínač pro aktivaci a deaktivaci tlačítek (nastavením na false zešednou a nepůjde na ně kliknout).

Perl (123) - Gtk2 - základní práce s obrázky

V druhém dílu série o Gtk2 budeme vkládat ikony do různých widgetů, avšak podíváme se též na obrázky obecně.

Tlačítko ze stocku

Obyčejné tlačítko již vytvořit umíme. Stock je množina speciálních standardizovaných tlačítek. Jsou to tedy šablony, pomocí kterých můžeme tvořit tlačítka do našich aplikací. To v tomto případě znamená, že se zobrazí v tlačítku ještě nějaká ikona z používaného prostředí (například červený křížek pro tlačítko Zrušit apod.).

Seznam šablon pro tlačítka nalezneme v poli Gtk2::Stock->list_ids. Podívejme se na pár úvodních hodnot.

gtk-zoom-out
gtk-zoom-in
gtk-zoom-fit
gtk-zoom-100
gtk-yes
gtk-unindent
gtk-undo
gtk-underline
gtk-undelete
gtk-strikethrough
...

Abychom přesně věděli, o čem je řeč, podívejme se, jak jednotlivá stock tlačítka vypadají.

Stock tlačítka

Tato stock tlačítka se vytvářejí pomocí speciálního konstruktoru new_from_stockwidgetu Gtk2::Button. Tomu předáváme vždy některý z řetězců z předchozího seznamu. Podívejme se, jak vytvoříme tlačítko Ano (na obrázku první sloupec, pátý řádek).

```
my $ano = Gtk2::Button->new_from_stock("gtk-yes");
```

Nyní s tímto tlačítkem již manipulujeme stejně jako s běžným tlačítkem.

Vlastní stock tlačítko

Poznamenejme, že do stocku lze definovat vlastní tlačítka. Ukažme si, jak se to provede. Základem je metoda add, kterou voláme nad třídou Gtk2::Stock.

```
Gtk2::Stock->add ({
stock_id => "moje-vlastni-tlacitko",
label    => "_Ahoj",
keyval   => $Gtk2::Gdk::Keysyms{A},
});
```

Nyní se v seznamu Gtk2::Stock->list_ids objeví též moje-vlastni-tlacitko a můžeme s ním pracovat stejně jako s ostatními. Jinými slovy, můžeme nyní tlačítkovému konstruktoru new_from_stock předávat řetězec moje-vlastni-tlacitko.

Avšak v tuto chvíli nemáme u tlačítka žádnou ikonu. Zde nestačí přiřadit soubor, ale je to trochu složitější. Pro přiřazení ikony ze souboru ./ikona.png je třeba vykonat následující posloupnost příkazů.

```
$icon_factory = Gtk2::IconFactory->new;
$icon_factory->add("moje-vlastni-tlacitko", Gtk2::IconSet->new_from_pixbuf(
Gtk2::Gdk::Pixbuf->new_from_file("./ikona.png")));
$icon_factory->add_default;
```

Titulek okna a rohová ikona

Přidejme někam do našeho programu následující řádky. Máme-li v aktuálním adresáři nějaký obrázek v souboru ikona.png, uvidíme, že kód ovlivní hlavní lištu okna.

```
$okno->set_title("okno s ikonou :-)");
$okno->set_icon_from_file("./ikona.png");
```

Ikonu bychom neměli zapomenout nastavit u žádné aplikace, která půjde k uživatelům, protože taková věc je schopna udělat dobrý dojem.

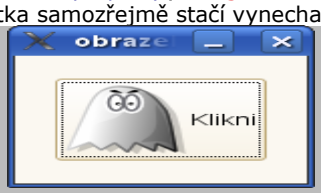
Titulek okna s ikonou

Tlačítko s vlastním obrázkem

Pokud si nevybereme některé z přednastavených tlačítek a nechceme oklikou vytvářet nové stock tlačítko (protože chceme tlačítko použít jen jednorázově), můžeme vytvořit vlastní. To je podstatně jednodušší a obnáší to pouze vytvoření objektu obrázku a přiřazení ho tlačítku.

```
$tlacitko = Gtk2::Button->new("Klikni");
$obrazek = Gtk2::Image->new_from_file("./Ghost.png");
$tlacitko->set_property("image"=>$obrazek);
```

Poznamenejme, že pro čistě obrázková tlačítka samozřejmě stačí vynechat textový popisek a použít samotný obrázek.



Tlačítko s obrázkem

Práce s obrázky - zobrazování, škálování, ořezávání

Nejprve se podívejme na úplně nejzákaldnější věc - zobrazíme v okně obrázek ze souboru v aktuálním adresáři. Vytvoříme tedy nejprve objekt $obrazek a ten (jakožto widget) standardně zobrazíme (například přidáme do okna). Zobrazování obrázků je základní funkcí grafické knihovny a proto to ani nemůže být něco náročného na psaní kódu.

```
$obrazek = Gtk2::Image->new_from_file("./obrazek.png");
$okno->add($obrazek);
```

Podívejme se ještě, jak z obrázku vytvořit buffer pixelů. Občas se to může hodit, jak uvidíme dále.

```
$raw = `cat ./obrazek.png`;
$pixbufloader = Gtk2::Gdk::PixbufLoader->new;
$pixbufloader->write($raw);
$pixbufloader->close;
$pixbuf = $pixbufloader->get_pixbuf;
```

Odtud můžeme samozřejmě jít zase zpátky a vytvořit znovu Gtk2::Image.

```
$obrazek = Gtk2::Image->new_from_pixbuf($pixbuf);
```

PixBuf můžeme škálovat. Pro změnu rozměrů obrázku na 500x500 metodou bilinear vyzkoušejme tento příkaz. Další metody škálování jsou v dokumentaci, avšak pro běžné účely vůbec nezáleží na tom, které škálování se provede.

```
$pixbuf = $pixbuf->scale_simple(500, 500, "bilinear") ;
```

Na závěr vyřízneme z obrázku kousek o velikosti 50x50, jehož horní levý roh má souřadnice 10, 10.

```
$pixbuf = $pixbuf->new_subpixbuf(10, 10, 50, 50);
```

Změna kurzoru

Ještě neumíme pracovat s událostmi, avšak předběhněme trochu a podívejme se, jak lze měnit kurzorovou ikonu. Signál enter se emituje po najetí myši na daný widget (hned zmiňme, že existuje též signál leave, který vyvolá tentokrát opuštění prostoru kurzorem myši). Zkusme po najetí do prostoru tlačítka změnit kurzor a nastavit ho na nějakou z hodnot typu Gtk2::Gdk::CursorType. Kurzor je objektem typu Gtk2::Gdk::Cursor a podle toho s ním také budeme zacházet. Nejprve ho vytvoříme a poté nastavíme.

```
$tlacitko = Gtk2::Button->new("1");
$tlacitko->signal_connect("enter" => sub {
$tlacitko->window->set_cursor(Gtk2::Gdk::Cursor->new("man"));
});
```

Perl (124) - Gtk2 - události a čas

S událostmi a časem již máme z předchozích grafických toolkitů nějaké zkušenosti. Jak k této oblasti ale přistupuje Gtk2?

Události

Představme si, že program právě vykonává Gtk2::main a čeká na události. Jakmile nastane událost (událostí se rozumí například stisk klávesy, pohyb myší do nějakého daného prostoru atd.), emituje se signál. Tyto signály mohou být emitovány různými widgety a existuje jich řada druhů, podle toho, jak byly vyvolány. S nejběžnějšími se v seriálu postupně seznámíme. Signály budeme odchytávat. Pokud signál neodchytneme, nic se nestane. Událost nezpůsobí žádnou akci. To se děje velmi často, neboť to prostě nemusí být potřeba. Pro uvědomění si zkuste například teď v internetovém prohlížeči stisknout klávesu ESC. Ve většině prohlížečů na tuto událost není definována reakce, tudíž se vůbec nic nestane. Navíc signálů se i v jednoduché aplikaci emituje obrovské množství.

Pokud ale uživatel například klikne na tlačítko nebo provede jinou událost, od které je nějaká návazná činnost očekávána, měli bychom se postarat o to, aby byla příslušná akce provedena. To je případ, který nás zajímá.

Signály se obvykle odchytávají voláním nějaké funkce, která se nám o to postará. Musíme jí však sdělit několik věcí, aby věděla, co přesně po ní chceme.

- určit widget, který emitoval signál
- určit signál, na který chceme reagovat
- jak chceme reagovat (tj. odkaz na nějakou funkci [tzv. callback funkci], která se provede)
- případně něco dalšího - přepínače, parametry atd.

Callback funkce je obyčejný podprogram. Může být buď anonymní nebo pojmenovaný. Ten pojmenovaný obvykle jako parametry dostane objekt reprezentující widget, který vyvolal signál, a případné parametry. Vypadá nějak takto.

```
sub akce {
my($self, $parametr) = @_;
#provedení akce
}
```

Signál obvykle propojujeme s akcí příkazem signal_connect, který dodržuje tuto strukturu.

```
$widget->signal_connect(typ_signalu => \&akce, $parametry);
```

Akce lze samozřejmě také modifikovat. Pro zrušení spojení signálu a akce lze použít metodu signal_handler_disconnect.

Příklad události - kliknutí na tlačítko

Upravíme náš program s tlačítkem tak, aby získalo funkcionalitu. Budeme chtít, aby se po stisknutí ukončila aplikace. Jinými slovy chceme, aby se po stisku tlačítka provedl podprogram konec, který vypadá takto.

```
sub konec {
my($widget) = @_;
Gtk2->main_quit;
}
```

Poznamenejme, že příkaz Gtk2->main_quit končí smyčku událostí a tedy v našem případě i celý program (ale kdybychom dále opět volali Gtk2->main, celá smyčka by se rozběhla nanovo).

Propojme tedy emitovaný signál s naším podprogramem. Použijme metodu signal_connectzavolanou nad tlačítkem. Po stisku tlačítka se vždy emituje signál clicked.

```
$tlacitko->signal_connect(clicked => \&konec);
```

Upravili jsme tedy celý program do následující podoby.

```
use Gtk2;
Gtk2->init;

sub konec {
my($widget) = @_;
Gtk2->main_quit;
}


$okno = Gtk2::Window->new("toplevel");
$okno->set_border_width(20);

$tlacitko = Gtk2::Button->new("Konec");
$tlacitko->signal_connect(clicked => \&konec);
$okno->add($tlacitko);
```

```
$tlacitko->show;
$okno->show;

Gtk2->main;
```

Proč se při zavření okna neukončí aplikace?

Gtk2 má tu vlastnost, že, zavřeme-li hlavní okno, aplikace běží dál. Na tom není nic špatného, ale ve většině případů budeme chtít aplikaci v takové situaci tak jako tak ukončit. Nyní to je třeba udělat například posláním SIGTERM, což není příliš elegantní řešení. Jak docílíme nápravy?

Existuje signál delete_event. Ten se emituje v reakci na událost zavření hlavního okna. Jednoduchým trikem zajistíme, aby se při emitaci delete_event celá aplikace ukončila.

```
$okno->signal_connect(delete_event => sub{Gtk2->main_quit});
```

Příklad - detekce stisknuté klávesy

Pomocí signálu key-press-event lze zachytávat klávesy z klávesnice. Zkusme tedy vytvořit akci, která proběhne po stisku libovolné klávesy. Budeme chtít vypsat nějakou informaci o stisknuté klávese.

```
$okno->signal_connect("key-press-event" => \&info);
```

Informaci budeme vypisovat pomocí popisku, který vytvoříme následovně.

```
$label = Gtk2::Label->new();
```

Jeho text pak nastavujeme pomocí set_markup.

```
$label->set_markup("Ahoj");
```

Podprogram info dostane jako parametr událost. Z ní můžeme získat id klávesy (metodou keyval). Máme k dispozici také hash %Gtk2::Gdk::Keysyms, který každé klávese přiřazuje nějaké id. Podívejme se pro zajímavost na část dat z tohoto hashe.

```
...
"downarrow" => 2302,
"Greek_chi" => 2039,
"ISO_Level2_Latch" => 65026,
"doubledagger" => 2802,
"guillemotleft" => 171,
"w" => 119,
"Pointer_Drag_Dflt" => 65268,
"threequarters" => 190,
"Hangul_Hanja" => 65332,
"cedilla" => 184,
"Armenian_e" => 16778599,
"MouseKeys_Enable" => 65142,
...
```

Z něj můžeme získat název stisknuté klávesy.

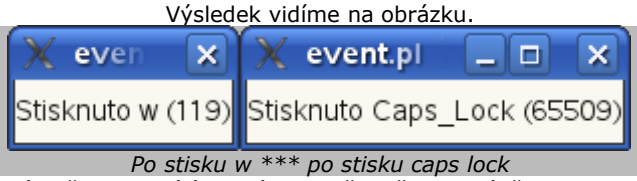
Podívejme se tedy na program, který nás informuje o událostech z klávesnice.

```
use Gtk2::Gdk::Keysyms;
use Gtk2 "-init";

sub info {
my($widget, $udalost)= @_;
my $id = $udalost->keyval;
for $k (keys %Gtk2::Gdk::Keysyms){
$label->set_markup("Stisknuto $k ($id)") if $Gtk2::Gdk::Keysyms{$k} == $id);
}
return 0;
}

$okno = Gtk2::Window->new("toplevel");
$okno->signal_connect("key-press-event" => \&info);
$label = Gtk2::Label->new();
$okno->add($label);
$okno->show_all;
Gtk2->main;
```

Uděláme ještě drobnou úpravu ve zdrojovém kódu, abychom se vyhnuli používání globálních proměnných. Při reakci na signál si necháme jako parametr předat label, který budeme modifikovat.

```
use Gtk2::Gdk::Keysyms;
use Gtk2 "-init";

sub info {
my($widget, $udalost, $label)= @_;
my $id = $udalost->keyval;
for $k (keys %Gtk2::Gdk::Keysyms){
$label->set_markup("Stisknuto $k ($id)") if $Gtk2::Gdk::Keysyms{$k} == $id);
}
return 0;
}

$okno = Gtk2::Window->new("toplevel");
$label = Gtk2::Label->new();
$okno->signal_connect("key-press-event" => \&info, $label);
$okno->add($label);
$okno->show_all;
Gtk2->main;
```

Výsledek vidíme na obrázku.



*Po stisku w *** po stisku caps lock*

Řekněme na tomto místě ne úplně související poznámku. Všimněme si závěru programu. Abychom nemuseli volat metodu show nad každým widgetem, lze použít místo všech těchto volání metodu show_all.

```
$okno->show_all;
```

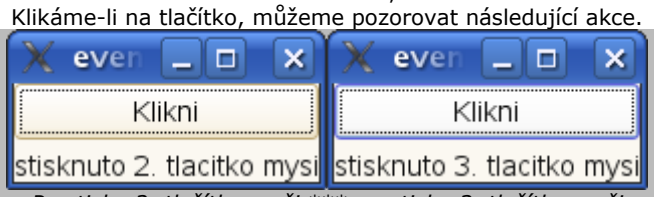
Příklad - tlačítka myši

Druhou důležitou skupinou signálů jsou ty, které byly emitovány událostmi myši. Signál button-release-event se vyvolá po kliknutí na daný widget.

Vytvořme si tlačítko, na které bude uživatel klikat. My budeme pro informaci vypisovat, které tlačítko myši uživatel stiskl. Všechny potřebné věci již umíme a tak se podívejme na zdrojový kód této aplikace. Dodejme jen, že nad objektem události (ten dostaneme v callback funkci jako parametr) voláme metodu button, která nám vrací id tlačítka myši.

```
use Gtk2 "-init";

$okno = Gtk2::Window->new ("toplevel");
$vbox = Gtk2::VBox->new(0, 5);
$tlacitko = Gtk2::Button->new("Klikni");
$label = Gtk2::Label->new();

$tlacitko->signal_connect("button-release-event" => sub {
$label->set_markup("stisknuto ".$_[1]->button().". tlacitko mysi");
});

$vbox->pack_start($tlacitko, 1, 1, 0);
$vbox->pack_start($label, 1, 1, 0);
$okno->add($vbox);
$okno->show_all;

Gtk2->main;
```

Klikáme-li na tlačítko, můžeme pozorovat následující akce.



*Po stisku 2. tlačítka myši *** po stisku 3. tlačítka myši*

Čas

Pomocí Glib::Timeout lze nastavovat akce, které se budou periodicky vykonávat. Metodou add přidáváme úlohu. Uvádíme zde interval opakování v milisekundách, podprogram, který se má provést, a případně parametr a prioritu. Dokud podprogram nevrátí false, bude se jeho vykonávání opakovat.

```
Glib::Timeout->add(1000, \&akce, $parametr);
```

Perl (125) - Gtk2 - vlastní widgety

Na příkladu blikajícího tlačítka se naučíme odvodit z nějakého existujícího standardního widgetu nový widget, který budeme moci používat v aplikacích.

V téměř každé aplikaci potřebujeme přiohnout některé standardní widgety tak, aby více odpovídaly našim představám. Toho dosahujeme změnami vlastností, nastavením metod atd. Pokud potřebujeme udělat jistý typ změn často, pak bývá praktické vytvořit si vlastní widget.

Spočívá to v tom, že vytvoříme objekt na základě nějakého již existujícího widgetu. Ten upravíme a případně vytvoříme nové atributy a metody. Vzhledem k tomu, že celá tato úprava je zabalena ve zvláštní třídě, chová se celý modul jako obyčejný widget.

Jako ukázku si vytvoříme jenoduché upravené tlačítko, ze kterého bude hezky vidět, jak widgety tvořit. Naše tlačítko bude obsahovat dva popisky a ty se budou v požadovaných intervalech střídat. Použijeme přitom dva různé přístupy.

Vytvořme tedy třídu MojeBlikajiciTlacitko, která bude upraveným standardním Gtk2::Button (jinými slovy, bude dědit z Gtk2::Button).

```
package MojeBlikajiciTlacitko;
use base "Gtk2::Button";
```

Třída MojeBlikajiciTlacitko bude obsahovat dvě metody: konstruktor a metodu pro bliknutí. Napišme nejprve konstruktor.

```
sub new {
my $trida = shift;

#zde vytvoříme vlastní objekt $self

return $self;
}
```

Vytvořme tedy nové tlačítko Gtk2::Button a pomocí bless z něj udělejme objekt.

```
my $self = bless Gtk2::Button->new, $trida;
```

Jako první dva parametry konstruktoru budeme očekávat dva popisky. Uložme tyto popisky do speciálních atributů.

```
$self->{text0} = $_[0];
$self->{text1} = $_[1];
```

Implicitně jeden z popisků nastavíme.

```
$self->set_label($self->{text0});
```
Chceme, aby tlačítko blikalo. Spusťme tedy pomocí časových funkcí pravidelně se opakující změnu popisku. Délka intervalu se bude nastavovat třetím parametrem konstruktoru. Změnu popisku nám zařídí funkce blik, kterou si v zápětí napíšeme.
```
Glib::Timeout->add($_[2], \&blik, $self);
```
Metoda blik pouze nastaví popisek na ten, který právě není aktivní.
```
sub blik {
my $self = shift;
$self->set_label($self->get_label eq $self->{text0} ? $self->{text1} : $self->{text0});
return 1;
}
```
Právě jsme vytvořili widget MojeBlikajiciTlacitko, které nyní již můžeme vesele používat. Následuje ukázkový program obsahující tento widget.
```
package main;

use Gtk2 -init;

my $okno = Gtk2::Window->new;
$okno->set_border_width(5);
$okno->signal_connect(delete_event => sub{Gtk2->main_quit;});

my $tlacitko = MojeBlikajiciTlacitko->new("Klikni", "sem!", 500);
$okno->add($tlacitko);

$okno->show_all;
Gtk2->main;
```
Jiný způsob přetěžování widgetů, emitace vlastních signálů

Existuje ještě metoda, jak přetěžovat widgety. Přimějeme naši třídu, aby se chovala jako Glib::Object. Bohužel to není tak jednoduché, aby stačilo pouze voláním bless prohlásit třídu za objekt Glib::Object, protože se musíme starat o věci jako jsou signály. Stručně se podívejme, jak se to tedy dělá.

Vytvořme opět widget MojeBlikajiciTlacitko.
```
package MojeBlikajiciTlacitko;
```
Pomocí Glib::Object::Subclass nastavíme všechny atributy a signály, které budeme chtít dále používat. Provádíme v zásadě podobnou činnost jako pragma base. Předáme tedy následující datové struktury příkazu use. Vždy musíme předat alespoň první parametr, který určuje, z čeho budeme dědit.
```
use Glib::Object::Subclass
    Gtk2::Button::,
    signals => {
        text_se_zmenil => {},
        show => \&new_show,
    },
    properties => [
        Glib::ParamSpec->string(
            "text0",
            "Text 0",
            "Prvni z textu, ktery bude problikavat",
            "",
            [qw(readable writable)]
        ),
        Glib::ParamSpec->string(
            "text1",
            "Text 1",
            "Druhy z textu, ktery bude problikavat",
            "",
            [qw(readable writable)]
        ),
        Glib::ParamSpec->uint(
            "interval",
            "Interval",
            "Casovy interval mezi bliknutimi",
            0,100000,
            1000,
            [qw(readable writable)]
        ),
    ]
    ;
```
U atributů je důležitý hlavně první parametr - to je jeho název. Pro význam dalších atributů a přidávání jiných datových typů než je číslo a řetězec se lze podívat do dokumentace na oddíl o Glib::ParamSpec.

Přetížili jsme metodu show. To proto, že je třeba nastavit úvodní hodnoty, jakmile se widget zobrazí.
```
sub new_show {
my $self = shift;
$self->set_label($self->{text0});
Glib::Timeout->add($self->{interval}, \&blik, $self);
$self->signal_chain_from_overridden;
}
```
Volání signal_chain_from_overridden žádá nadřazenou třídu o zavolání příslušné metody. V seriálu jsme obvykle používali pro podobné účely SUPER::show, avšak zde je třeba s dědičností zacházet trochu jinak.

Protože zde obvykle nevytváříme metodu new (ta je zděděna), je automaticky volána metoda INIT_INSTANCE. Ta se používá pro inicializaci objektů
Samozřejmě potřebujeme i metodu na bliknutí textu. Zde pouze přidáme emitaci signálu, který detekuje změnu textu na tlačítku. K tomu použijeme metodu signal_emit.

```
sub blik {
my $self = shift;
$self->set_label($self->get_label eq $self->{text0} ? $self->{text1} : $self->{text0});
$self->signal_emit("text_se_zmenil");
1;
}
```

A použití widgetu je stejné.

```
package main;

use Gtk2 -init;

my $okno = Gtk2::Window->new;
$okno->set_border_width(5);
$okno->signal_connect(delete_event => sub{Gtk2->main_quit;});

my $tlacitko = MojeBlikajiciTlacitko->new("Klikni", "sem!", 500);
$okno->add($tlacitko);

$okno->show_all;
Gtk2->main;
```

Vzhledem k tomu, že se vždy po změně textu tlačítka emituje signál, můžeme ho zpracovat. My pouze jako ukázku vypíšeme na standardní výstup hlášku o tom, že signál byl úspěšně zachycen.

```
$tlacitko->signal_connect(text_se_zmenil => sub {print "Signal o zmene textu na tlacitku zachycen!";});
```

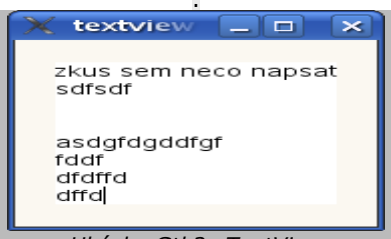
Perl (126) - Gtk2 - textové okno a práce s pozicemi

Základním widgetem pro zobrazování a editaci textu v Gtk2 je textové pole. Umí řadu zajímavých věcí - mimo očekávaných funkcionalit také formátování textu a vkládání widgetů. Na to vše je potřeba dobře rozumět reprezentaci pozic, což jsou objekty určující místa v textovém poli.

Gtk2::TextView je často používaný widget na zobrazování a editaci textu. Tento widget má ještě podstatně více různých funkcí a nastavení, než textová pole v Tk a Wx, která jsme si již představovali.
Na úvod se podívejme na základní použití.

```
$text = Gtk2::TextView->new;
$buffer = $text->get_buffer();
$buffer->set_text("zkus sem neco napsat");
```

Právě jsme vytvořili textové pole a napsali do něj text. Uživatel ho může libovolně editovat.



Ukázka Gtk2::TextView

Na první pohled je kód zdánlivě zbytečně složitý (Proč nepoužít něco jako Gtk2::TextView->set_text? Proč to musíme řešit přes buffer?). Uvidíme, že to zas o tolik složitější není a že lze takto s textem pohodlně manipulovat. Text (společně s informací, jak má být zobrazen) je zde reprezentován speciálním widgetem Gtk2::TextBuffer a až ten můžeme nastavit pomocí set_buffer.
Podívejme se na úvod jen ve stručnosti na některé základní metody Gtk2::TextView.

Metoda	Význam
set_wrap_mode	metoda zalamování; možné hodnoty jsou none, char nebo word
set_editable	false pro read-only, true pro editaci
set_cursor_visible	viditelnost kurzoru (kurzor bychom měli schovat vždy, když používáme read-only režim)
set_justification	zarovnání textu; možné hodnoty jsou left, right, center
set_left_margin, set_right_margin	okraje
set_indent	odsazení

Existuje-li metoda s názvem set_*něco*, pak také často existuje metoda get_*něco*, která detekuje aktuální nastavení.
Objekt typu Gtk2::TextIter reprezentuje pozici v bufferu. Pozice je platná vždy jen do okamžiku, kdy se obsah bufferu (tj. text v Gtk2::TextView) změní. Protože často chceme pozice zachovat i po změně obsahu bufferu, byly vytvořeny objekty typu Gtk2::TextMark, což jsou hýbající se pozice.
Vlastnosti TextBufferu
Gtk2::TextBuffer je vytvořen vždy, když vytvoříme Gtk2::TextView. Samozřejmě si ale můžeme standardní cestou vytvořit TextBuffer vlastní.

```
$textbuffer = Gtk2::TextBuffer->new;
```

Podívejme se na metody, kterými lze s textem v TextBufferu manipulovat.

Metoda	Význam

set_text	nastaví text
insert($textiter, $text)	vloží $text na pozici $textiter (jak víme, pozice je objekt typu Gtk2::TextIter; více o tom dále)
insert_at_cursor	vloží text na aktuální pozici
insert_range($kam, $zacatek, $konec)	výsek určený dle $zacatek, $konec se uloží na pozici $kam
get_text($zacatek, $konec, $ukazat_skryte)	vrátí výsek
get_slice($zacatek, $konec, $ukazat_skryte)	vrátí výsek, obrázky budou vráceny jako znak 0xFFFC; více o tom dále
get_line_count, get_char_count	zjistí počet řádků, resp. znaků

Práce s pozicemi

Již víme, že pozici v TextBufferu uchovává objekt Gtk2::TextIter. Podívejme se na to, jak s pozicemi pracovat. Pozici v bufferu definujeme jednou z následujících metod zavolaných nad TextBufferem.

Příkaz	Kde se vytvoří TextIter?
$textiter=$textbuffer->get_start_iter;	před prvním znakem
$textiter=$textbuffer->get_end_iter;	za posledním znakem
$textiter=$textbuffer->get_iter_at_offset(50);	před padesátým znakem
$textiter=$textbuffer->get_iter_at_line($radek);	před prvním znakem řádku č. $radek
$textiter=$textbuffer->get_iter_at_line_offset(5, 10);	před 10. znak na 5. řádku
$textiter=$textbuffer->get_iter_at_mark($mark);	na pozici existujícícho Gtk2::TextMark
($ti1,$ti2)=$buffer->get_selection_bounds;	na hranice vybrané oblasti

Nad objekty typu Gtk2::TextIter lze pak volat obrovské množství dalších metod. Podívejme se na seznam těch, které se mohou hodit.

Metoda	Význam
Detekce pozice v kontextu slov a řádků	
starts_word, ends_word, inside_word, starts_sentence, ends_sentence, inside_sentence, starts_line, ends_line, is_end, is_start, is_cursor_position	zjišťování informací o pozici
in_range($zacatek, $konec)	jsme v daném rozmezí?
Zjišťování absolutní pozice	
get_offset	vrátí číslo znaku
get_line	vrátí číslo řádku
get_line_offset	vrátí číslo znaku na řádku
Pohyb TextIterů	
forward_char, backward_char, forward_word_end, backward_word_start, forward_sentence_end, backward_sentence_start, forward_line, backward_line, forward_to_line_end, forward_cursor_position, backward_cursor_position, forward_to_end	pohyb na pozici v kontextu slov a řádků
forward_chars($kolik), backward_chars($kolik), forward_word_ends($kolik), backward_word_starts($kolik), forward_sentence_ends($kolik), backward_sentence_starts($kolik), forward_lines($kolik), backward_lines($kolik), forward_cursor_positions($kolik), backward_cursor_positions($kolik)	to samé vícenásobně
set_offset($offset)	nastaví na danou pozici
set_line($radek)	nastaví na daný řádek
set_line_offset($offset_na_radku)	nastaví na pozici na aktuálním řádku
Získávání textu z TextBufferu	
get_char	vrátí znak
get_text($do)	vrátí úsek textu
get_slice($do)	vrátí úsek textu
Detekce dalších objektů na dané pozici	
get_marks	vrátí seznam objektů Gtk2::TextMark
get_tags	vrátí seznam tagů, tj. objektů typu Gtk2::TextTag
get_pixbuf	vrátí seznam objektů Gtk2::PixBuf
get_child_anchor	vrátí seznam objektů Gtk2::TextChildAnchor
has_tag, begins_tag, ends_tag	detekce tagů

Hledání v textu	
forward_search($retezec, 'text-only', $kam_az)	vrátí dvouprvkový seznam s počáteční a koncovou pozicí typu Gtk2::TextIter, hledání dopředu; druhý parametr je typu Gtk2::TextSearchFlags
backward_search($retezec, 'text-only', $kam_az)	vrátí dvouprvkový seznam s počáteční a koncovou pozicí typu Gtk2::TextIter, hledání dozadu; druhý parametr je typu Gtk2::TextSearchFlags
Ostatní	
get_attributes	vrátí objekt typu Gtk2::TextAttributes

Gtk2::TextMark je podobný jako Gtk2::TextIter, avšak zachovává pozice při změnách TextBufferu.
Podívejme se opět, jak lze vytvořit objekt typu Gtk2::TextMark.

Příkaz	Kde se vytvoří TextMark?
$textmark = $textbuffer->get_insert	aktuální pozice kurzoru
$textmark = $textbuffer->get_selection_bound	druhý konec výběru (prvním koncem je pozice kurzoru)
$textmark = $textbuffer->create_mark($nazev, $textiter, $zleva)	Gtk2::TextMark s názvem $nazev se vytvoří na pozici $textiter s tím, že při vkládání na tuto pozici se posouvá/neposouvá doleva

TextMarky lze ručně posouvat pomocí TextIterů jedním z následujících příkazů.

```
$textbuffer->move_mark($mark, $kam);
$textbuffer->move_mark_by_name($nazev, $kam);
```

Metodou get_mark získáme objekt typu Gtk2::TextMark na základě svého názvu.

Tagy - formátování textu

Objekty typu Gtk2::TextTag jsou dalším obsahem TextBufferů. Obsahují informace o formátování. Každý tag funguje na daných pozicích a má nějaký svůj efekt (například zvětšuje písmo na druhém řádku).
Společně s bufferem se vždy vytvoří i objekt typu Gtk2::TextTagTable. Zde je uchováván seznam tagů.
Tag vytvoříme metodou create_tag zavolanou nad TextBufferem.

```
$tag = $textbuffer->create_tag($nazev, %vlastnosti);
```

Poté jsou dvě možnosti, jak tagy aplikovat. Buď přímo nebo opět podle názvu.

```
$textbuffer->apply_tag($tag, $zacatek, $konec);
$textbuffer->apply_tag_by_name($nazev, $zacatek, $konec);
```

Stejně tak můžeme tagy odstraňovat.

```
$textbuffer->remove_tag($tag, $zacatek, $konec);
$textbuffer->remove_tag_by_name($nazev, $zacatek, $konec);
```

Vlastnosti lze nastavovat metodou set_property.
Pomocí set_priority nastavujeme pro tag prioritu mezi 0 a velikostí Gtk2::TextTagTable.
Podívejme se na hash parametrů, které ovlivňují chování tagu. Jaké všechny parametry můžeme používat?

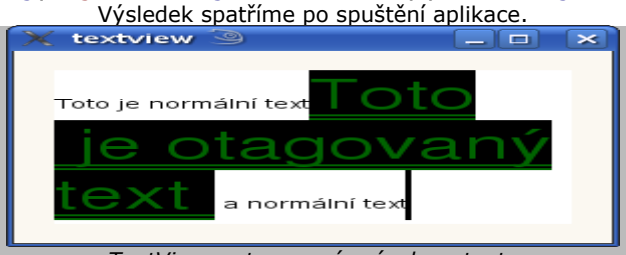
Klíč	Význam a hodnoty
background, background-gdk	barva pozadí pomocí řetězce, resp, Gtk2::Gdk::Color
foreground, foreground-gdk	barva popředí pomocí řetězce, resp, Gtk2::Gdk::Color
background-stipple, foreground-stipple	použije se na pozadí / popředí maska (bitmapa)
font	písmo (například Times 12)
size, size-points	velikost písma v Pango bodech, resp. v bodech
scale	relativní velikost písma oproti okolí
font-desc	styl písma (jako objekt typu Gtk2::Pango::FontDescription)
family	například Times
underline	podtržené písmo
style, variant, weight, stretch	hodnoty jsou v dokumentaci
pixels-above-lines, pixels-below-lines	mezera v pixelech nad / pod řádkem
wrap-mode	jak zalamovat (none, word, char)
justification	left, right, center
direction	směr toku textu (right-to-left, left-to-right)
left-margin, right-margin	okraje
indent	odsazení odstavce
strikethrough	dělení
rise	posunutí nahoru (dolů při záporném parametru)
background-full-height	má-li tag ovlivnit celý řádek ve smyslu barvy pozadí

Zkusíme si vytvořit nějaký tag.

```
$tag = $buffer->create_tag("zvyrazneny_text",
"font" => "Helvetica 30",
"underline" => PANGO_UNDERLINE_DOUBLE,
"foreground" => "darkgreen",
"background-gdk" => Gtk2::Gdk::Color->new(60, 0, 200));
```

```
$buffer->apply_tag($tag, $buffer->get_iter_at_offset(3), $buffer->get_iter_at_offset(7));
```

Výsledek spatříme po spuštění aplikace.



TextView s otagovaným úsekem textu

Vkládání widgetů

Do textového pole lze vkládat také obrázky nebo rovnou celé widgety. Obrázky reprezentované jako Gtk2::PixBuf můžeme vkládat metodou insert_pixbuf na danou pozici. Takový obrázek se bude chovat jako unicodový znak 0xFFFC.

```
$textbuffer->insert_pixbuf($textiter, $pixbuf)
```

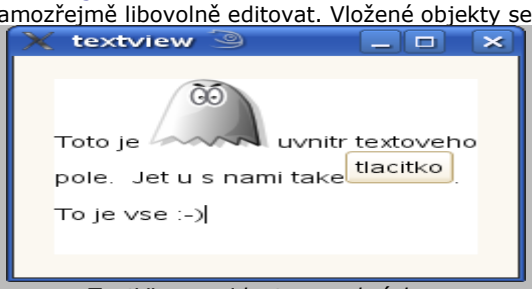
S widgety je to trochu složitější, ale v zásadě podobné. Vytvoříme objekt typu Gtk2::TextChildAnchor na místě, kde chceme, aby byl widget. Poté na toto místo widget vložíme.

```
$anchor = Gtk2::TextChildAnchor->new;
$textbuffer->insert_child_anchor($iter, $anchor);
$textview->add_child_at_anchor($widget, $anchor);
```

Ukážeme si konkrétnější příklad - TextView, do kterého vložíme text, obrázek a tlačítko.

```
$textview = Gtk2::TextView->new;
$buffer = $textview->get_buffer();
$buffer->set_text("nejaky text");

$tlacitko = Gtk2::Button->new("tlacitko");

$anchor = Gtk2::TextChildAnchor->new;
$buffer->insert_child_anchor($buffer->get_start_iter, $anchor);
$textview->add_child_at_anchor($tlacitko, $anchor);

$buffer->insert_pixbuf($buffer->get_start_iter, Gtk2::Gdk::Pixbuf->new_from_file("./Ghost.png"));
```

Text pak můžeme samozřejmě libovolně editovat. Vložené objekty se chovají jako znaky.



TextView s widgetem a obrázkem

Příklad - zvýraznění textu pod kurzorem myši

Ukážeme si, jak lze zachytávat pozici myši a manipulovat s textem (popřípadě tagem) na jejím aktuálním místě. Díky signálu motion_notify_event, který je emitován při změně pozice myši nad daným widgetem můžeme vyvolat akci. Ta bude například zvýrazňovat místo, kde je aktuálně myš.

```
$textview->signal_connect(motion_notify_event => \&akce);
```

Díky metodě window_to_buffer_coords zjistíme pixelové souřadnice. Ty nám metoda get_iter_at_location převede na pozici v textu, tj. objekt typu Gtk2::TextIter. Pak není nic jednoduššího, než s danou pozicí manipulovat.

Vytvoříme tedy tag, kterým budeme zvýrazňovat aktuální znak.

```
my $tag = $buffer->create_tag(undef, "foreground" => "white", "background" => "black");
```

Aplikujeme ho. Na to ovšem musíme mít dva TextItery (potřebujeme začátek i konec tagu). Vytvoříme si tedy kopii našeho TextIteru a posuneme ho o pozici vpřed. TextIter nemá vhodný konstruktor na kopírování objektů a poradíme si tak trochu oklikou.

```
my $textiter2 = $buffer->get_iter_at_offset($textiter->get_offset);
$textiter->forward_char;
$buffer->apply_tag($tag, $textiter2, $textiter);
```

Ještě bychom měli uchovávat starý tag a mazat ho po opuštění pozice. Zavedeme si tedy proměnnou $kurzor_tag, která bude reprezentovat týž objekt jako $tag do doby, než $tag zapomeneme. Tag pak smažeme přes tabulku tagu, kterou získáme z bufferu.

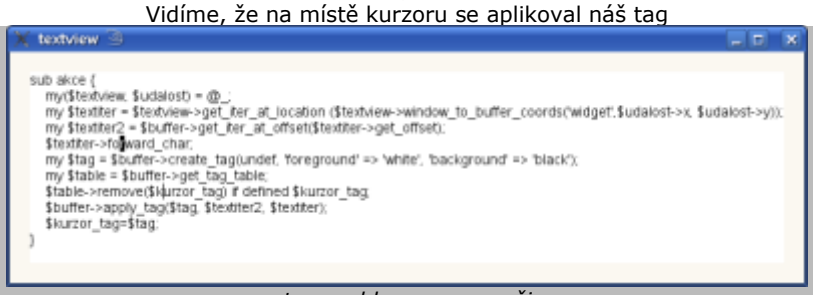
```
my $table = $buffer->get_tag_table;
$table->remove($kurzor_tag) if defined $kurzor_tag;
```

Podívejme se na celý zdrojový kód naší akce.

```
sub akce {
    my($textview, $udalost) = @_;
    my $textiter = $textview->get_iter_at_location(
$textview->window_to_buffer_coords("widget",$udalost->x, $udalost->y));
    my $textiter2 = $buffer->get_iter_at_offset($textiter->get_offset);
    $textiter->forward_char;
my $tag = $buffer->create_tag(undef, "foreground" => "white", "background" => "black");
    my $table = $buffer->get_tag_table;
    $table->remove($kurzor_tag) if defined $kurzor_tag;
    $buffer->apply_tag($tag, $textiter2, $textiter);
    $kurzor_tag=$tag;
```

}

Vidíme, že na místě kurzoru se aplikoval náš tag



tag pod kurzorem myši

Perl (127) - Gtk2 - hierarchické seznamy

Hierarchický seznam používáme v aplikacích pro tok informací oběma směry - od uživateli k aplikaci i naopak. Knihovna Gtk2 umožňuje vytvářet aplikace nejen s editovatelnými položkami, ale i editovatelnou hierarchií. Jednodušší variantu výběru ze seznamu pak Gtk2 nabízí ještě ve formě dalšího widgetu.

Hierarchické seznamy

TreeView je widget s položkami v hierarchické struktuře a umožňuje uživateli následující činnosti.

- vybrat z hierarchické nabídky
- přepisovat položky v hierarchické nabídce
- přeskupovat položky (měnit hierarchii)
- řadit položky dle kritérií
- atd.

Nejprve obvykle vytváříme objekt typu Gtk2::TreeStore. Metodou Gtk2::TreeStore->append v něm tvoříme jednotlivé položky, k nimž nastavujeme popisky voláním Gtk2::TreeStore->set. Z tohoto objektu pak tvoříme Gtk2::TreeView. Je třeba ještě vytvořit příslušný počet sloupců a poté již můžeme výsledek zobrazit.

Zvykem bývá umísťovat TreeView do Gtk2::ScrolledWindow, což je okno s posuvníky. To proto, že uživatel se může rozbalováním nabídek často dostat mimo obrazovku.

Jednoduchý TreeView

Vytvořme tedy hlavní okno a do něj okno s posuvníky.

```
$okno = Gtk2::Window->new("toplevel");
$okno->set_border_width(20);
$okno->set_title("treeview");

$scrolled = Gtk2::ScrolledWindow->new(undef, undef);
$scrolled->set_size_request(150, 200);
$okno->add($scrolled);

#zde vytvoříme strom

$okno->show_all;
Gtk2->main;
```

TreeStore vytvoříme standardně. Vložme do něj pět položek s tím, že poslední dvě budou podřízené druhé. Syntaxe je intuitivní.

```
my $tree_store = Gtk2::TreeStore->new(qw(Glib::String));
my $iter1 = $tree_store->append(undef);
my $iter2 = $tree_store->append(undef);
my $iter3 = $tree_store->append(undef);
my $iter21 = $tree_store->append($iter2);
my $iter22 = $tree_store->append($iter2);
$tree_store->set($iter1,0 => "Polozka 1");
$tree_store->set($iter2,0 => "Polozka 2");
$tree_store->set($iter3,0 => "Polozka 3");
$tree_store->set($iter21,0 => "Polozka 2.1");
$tree_store->set($iter22,0 => "Polozka 2.2");
```

Z této struktury vytvoříme TreeView.

```
my $tree = Gtk2::TreeView->new($tree_store);
```

Zbývá nám přidat sloupce. Zatím ponechme bez komentáře následující řádky.

```
my $tree_column = Gtk2::TreeViewColumn->new();
my $renderer = Gtk2::CellRendererText->new;
$tree_column->pack_start($renderer, FALSE);
$tree_column->add_attribute($renderer, text => 0);
```

Přidejme ještě titulek sloupce.

```
$tree_column->set_title("Seradit");
```

TreeView umí automaticky řadit položky podle určeného sloupce. Určeme ho tedy.

```
$tree_column->set_sort_column_id(0);
```

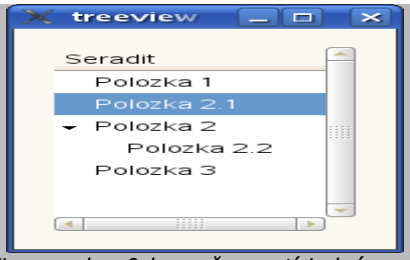
Též můžeme uživateli dovolit, aby metodou drag&drop přesunoval položky.

```
$tree->set_reorderable(TRUE);
```

Na závěr musíme náš strom přidat do okna.

```
$scrolled->add($tree);
```

To je základní použití TreeView.



TreeView po drag&drop přesunutí jedné z položek

Editovatelnost

Zkusme změnit položky nabídky na editovatelné. K tomu slouží tento příkaz.

```
$renderer->set_property("editable", TRUE);
```

Vyzkoušíme-li, zjistíme, že sice položky editovatelné jsou, ale po kliknutí někam pryč se zase vrátí původní text. To je sice v pořádku, ale většinou to nechceme.

Jak to napravit? Používá se na to speciální signál edited, který je emitován právě v okamžiku, kdy k editaci pole došlo.

```
$renderer->signal_connect(edited => \&zmen_polozku, $tree_store);
```

Z toho plyne, že tedy musíme sami napsat funkci, která se nám o změnu textu v položce postará. Zavoláme set_value nad Gtk2::TreeStore. Při volání musíme znát lokalizaci položky ve stromu, sloupec (získáme z $bunka->get_data) a samozřejmě nový text.

```
sub zmen_polozku {
my($bunka, $lok_string, $zmeneno_na, $store) = @_;
$store->set_value($store->get_iter(Gtk2::TreePath->new_from_string($lok_string)), 0, $zmeneno_na);
}
```

Výběr položek

Další vlastností TreeView je, že umí dělat výběry položek. Existuje několik módů. Buď vybírat vůbec nejde (none), jde vybírat jedna položka (single) nebo více položek (multiple). Seznam módů je uchován v Gtk2::SelectionMode.

Výběr je objekt typu Gtk2::TreeSelection a získáme ho metodou get_selection zavolanou nad Gtk2::TreeView. Mód výběru nastavujeme pomocí Gtk2::TreeSelection->set_mode.

Metodou get_selected získáme objekt typu Gtk2::TreeIter, jsme-li v módu single. Jsme-li v módu multiple, pak použijeme metodu get_selected_rows. Ta vrátí seznam Gtk2::TreeIter objektů.

Renderery

Dost divoké věci můžeme dělat s Renderery. Existuje více druhů (nejen pro text). Zde se odkážeme na dokumentaci. Ukažme si jediný jednoduchý příklad, který přebarví pozadí buněk na zeleno.

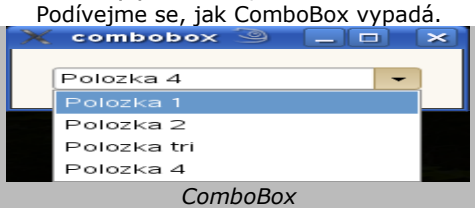
```
$renderer->set_property("cell-background", "green");
```

Rozbalovací nabídka

Rozbalovací nabídka je widget typu Gtk2::ComboBox. Vytvoření probíhá intuitivně.

```
my $combobox = Gtk2::ComboBox->new_text;
$combobox->append_text("Polozka 1");
$combobox->append_text("Polozka 2");
$combobox->append_text("Polozka tri");
$combobox->append_text("Polozka 4");
$combobox->set_active(2); #druhá položka bude zaškrtnuta implicitně
```

Podívejme se, jak ComboBox vypadá.



ComboBox

Jakmile uživatel změní položku, emituje to signál changed. Odchyťme ho.

```
$combobox->signal_connect("changed" => \&combobox);
```

Pomocí $combobox->get_active pak získáme aktuální hodnotu.

K vytvoření comboboxu můžeme využít také Gtk2::ListStore. To je podobně jako Gtk2::TreeStore struktura položek.

```
my $list_store = Gtk2::ListStore->new(qw(Glib::String));
$iter1 = $list_store->append;
$list_store->set($iter1,0,"popisek 0");
$iter2 = $list_store->append;
$list_store->set($iter2,0,"popisek 1");
```

I zde pak můžeme používat triky s renderery.

```
my $combobox = Gtk2::ComboBox->new($list_store);
my $renderer = Gtk2::CellRendererText->new();
$renderer->set_property("cell-background", "green");
$combobox->pack_start($renderer, TRUE);
$combobox->add_attribute($renderer,"text",0);
```

Chceme-li získat editovatelnost, pak použijeme widget Gtk2::ComboBoxEntry. Vše ostatní zůstává stejné. Akorát je opět nutné odchytit signál changed a zajistit přepis položek.

Perl (128) - Gtk2 - dialogy

Dialogem rozumíme speciální okno, které buď obsahuje zprávu pro uživatele nebo nějakou jednoduchou otázku. Naučíme se, jak dialogová okna vytvářet a jak reagovat na uživatelovu odpověď.

Gtk2 má opravdu velmi pestré varianty různých dialogových oken, což si zasluhuje vlastní díl.

Základní dialogovou třídou, kolem které se to všechno odehrává, je Gtk2::MessageDialog.

Standardní dialogová okna

Nejprve si řekněme, co máme za možnosti. V dialozích se zobrazují ikony. Máme 4 možnosti.

- info
- warning
- question
- error

Dále máme 6 možných druhů tlačítek, které se v dialogu objeví.

- ok
- none
- close
- cancel
- ok-cancel
- yes-no

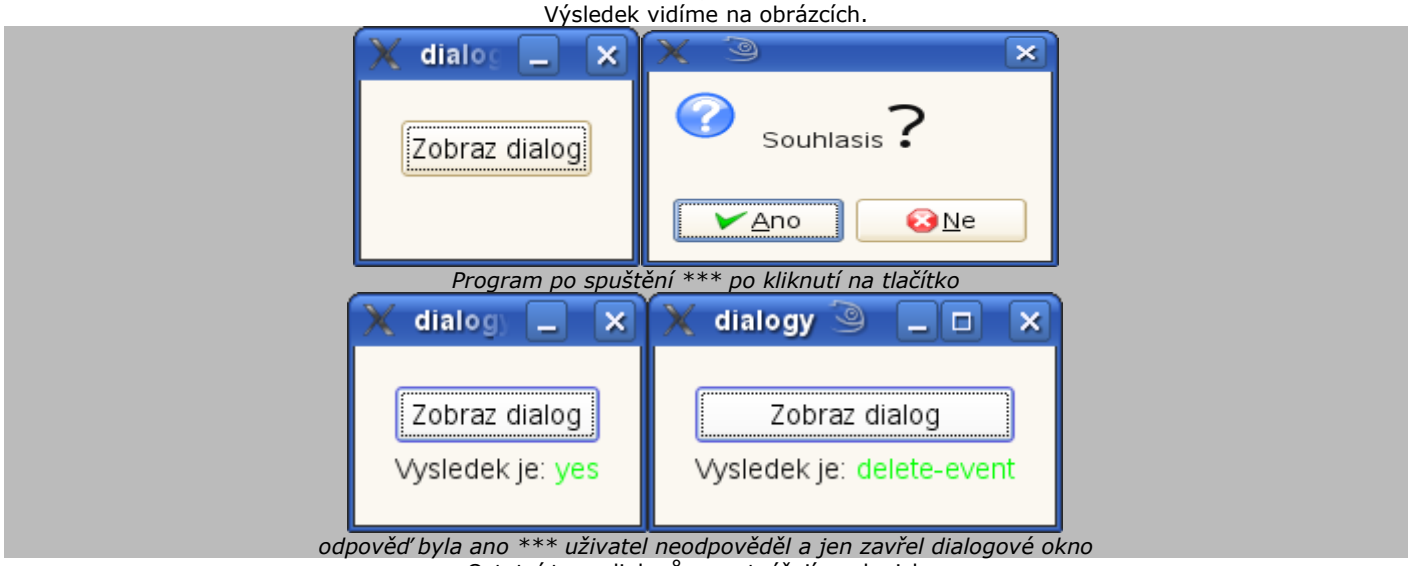
Zde je aplikace, která po kliknutí na tlačítko zobrazí ukázkový dialog. Jakmile klikneme na tlačítko, volá se podprogram zobraz_dialog, ve kterém je pro nás to důležité.

Widget Gtk2::MessageDialog můžeme vytvořit pomocí dvou konstruktorů. Buď bude v dialogovém okně prostý text (pak použijeme standardní new) nebo markup jazykem formátovaný text (v takovém případě zvolíme new_with_markup). Jako parametr vkládáme nadřízené okno. Můžeme vložit též undef, avšak pak se dialog zobrazí uprostřed displeje (nikoliv nad nadřízeným oknem) nebo se může ztratit pod okny (ta budou neaktivní), což je pro uživatele rušivé. Dále jsou parametry samozřejmě typ ikony, tlačítka a obsah sdělení.

$dialog->run zobrazí dialog a vrátí výsledek (například dialogy yes-no vrací buď yes, no nebo delete-event, podle toho, na co uživatel klikne). Jakmile uživatel klikne, dialogové okno zavřeme metodou destroy.

```
$vbox = Gtk2::VBox->new(0, 5);

$tlacitko = Gtk2::Button->new("Zobraz dialog");
$tlacitko->signal_connect("clicked" => \&zobraz_dialog);
$vbox->pack_start($tlacitko, 1, 1, 0);

$label = Gtk2::Label->new;
$vbox->pack_start($label, 1, 1, 0);

$okno->add($vbox);

sub zobraz_dialog {
$dialog = Gtk2::MessageDialog->new_with_markup(
$okno,[qw(modal destroy-with-parent no-separator)],"question","yes-no","Souhlasis<span size='30000'>?</span>");
$label->set_markup("Vysledek je: <span color='green'>".$dialog->run."");
$dialog->destroy;
}
```

Výsledek vidíme na obrázcích.



*Program po spuštění *** po kliknutí na tlačítko*

*odpověď byla ano *** uživatel neodpověděl a jen zavřel dialogové okno*

Ostatní typy dialogů se vytvářejí analogicky.

Vlastní návrh dialogových oken

Pro lepší interakci s uživatelem budeme chtít přidávat do dialogových oken další widgety a přizpůsobovat tlačítka. To lze dělat díky obecnějšímu widgetu Gtk2::Dialog.

Ten vytváříme tak, že specifikujeme, která tlačítka chceme použít. Vybíráme obvykle ze seznamu Gtk2::Stock->list_ids, který jsme si již představovali. Vždy jim přiřadíme nějaké id, podle kterého dále zjistíme, na které z nich uživatel klikl.

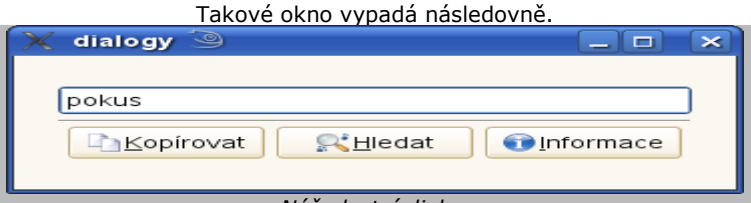
```
$okno = Gtk2::Dialog->new("dialogy", undef, [qw(modal destroy-with-parent)],
"gtk-copy" => 1,
"gtk-find" => 2,
"gtk-info" => 3,
);
```

Můžeme do něj přidat například jednořádkové formulářové pole.

```
$vbox = $okno->vbox;
```

```
$entry = Gtk2::Entry->new();
$vbox->pack_start($entry,1,1,5);
$okno->show_all;
```

Takové okno vypadá následovně.



Náš vlastní dialog

Akce

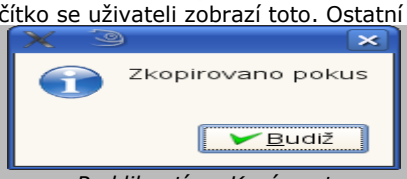
A nyní můžeme samozřejmě pracovat s akcemi. Jakmile si uživatel vybere tlačítko, emituje se signál response. Ten si s sebou nese informaci v podobě id tlačítka, jež bylo stisknuto.

```
$okno->signal_connect(response => \&akce);
```

Do podprogramu akce si již můžeme napsat cokoliv. My třeba otevřeme dialogová okna a oznámíme uživateli, co právě udělal.

```
sub akce {
my $zprava;
$zprava="Zkopirovano" if $_[1] == 1;
$zprava="Nenalezeno" if $_[1] == 2;
$zprava="Toto jsou informace" if $_[1] == 3;
$zprava .= " ".$entry->get_text();
$dialog = Gtk2::MessageDialog->new_with_markup(
$okno,[qw(modal destroy-with-parent)],"info","ok",$zprava);
$dialog->run;
$dialog->destroy;
}
```

Po kliknutí na první tlačítko se uživateli zobrazí toto. Ostatní tlačítka fungují stejně.



Po kliknutí na Kopírovat

Perl (129) - Gtk2 - skládání widgetů

Ruční psaní kódu pro umísťování widgetů obvykle obnáší řadu různých parametrů a spoustu kódu. Tak tomu je i v Gtk2. Protože to ale patří k základům navrhování aplikací, nelze toto téma vynechat.

Při vytváření libovolných GUI aplikací narážíme vždy na problém, jak uspořádat v oknech widgety. Nejjednodušší je obvykle použití nějakého vývojového prostředí, která často umožňují umístit widgety přetáhnutím myší. Tomu se budeme věnovat později. Naučme se však nyní, jak se skládají widgety "ručně". Nevýhodou bývá, že to obvykle obnáší hodně psaní navíc.

Widgety (včetně kontejnerů) budeme skládat do kontejnerů. To jsou speciální neviditelné widgety, které do sebe skládají danou metodou jiné widgety. Tyto metody mohou být obecně libovolné. Časté je skládání vertikální, horizontální a do mřížky. Nicméně, jak uvidíme, kontejnerů je více.

Vertikální a horizontální skládání

Gtk2 obsahuje kontejner Gtk2::HBox pro horizontální skládání a Gtk2::VBox pro vertikální skládání.

Oba zmíněné kontejnery používají dvě základní metody, jimiž jsou pack_start a pack_end. Těmi skládáme widgety do daného kontejneru. pack_start pracuje shora dolů (v případě Gtk2::VBox) nebo zleva doprava (v případě Gtk2::HBox) a pack_end přesně naopak.

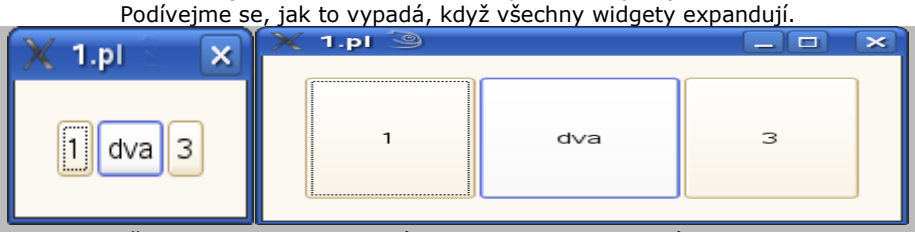
Existuje obrovské množství různých nastavení pro skládání widgetů ohledně velikosti widgetů, orientace atd. Ačkoliv to vyžaduje jistou dávku trpělivosti, zkusme se alespoň na některá užívaná nastavení podívat.

Podívejme se nejprve na to, jak voláme funkce pack_start, resp. pack_end.

```
$kontejner->pack_start($widget, $expanze, $roztahnuti, $padding);
```

Proměnnou $expanze specifikujeme, zda mají widgety zabrat volné místo okolo (do kterého se mohou buď roztáhnout nebo nikoliv; to záleží na parametru $roztahnuti), či zda mají skromně zabrat jen místo, které potřebují (v takovém případě s nimi ještě můžeme ve volném prostoru hýbat).

Podívejme se, jak to vypadá, když všechny widgety expandují.

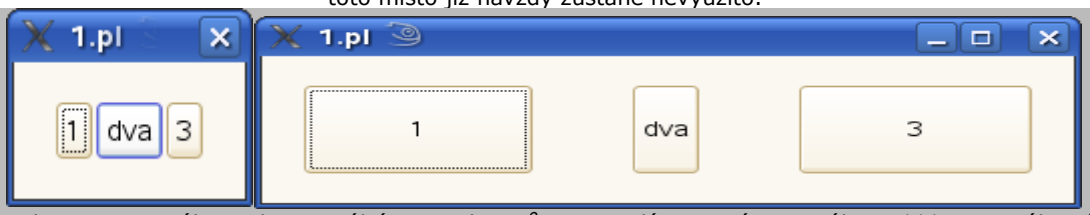


*Všechny widgety expandují *** zkusme trochu roztáhnout okno*

Dále zakažme expanzi prostředního tlačítka.

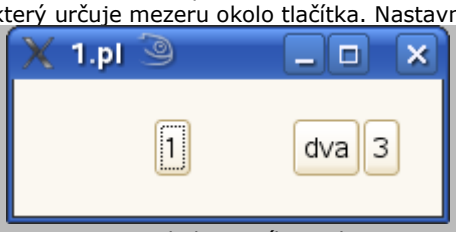
*Prostřední widget neexpanduje, ostatní ano *** po roztáhnutí vidíme rozdíl*

Parametr roztáhnutí funguje jen tehdy, když je expanze zapnutá. Pohrajme si nyní s tímto parametrem. Vypněme roztažení u prostředního tlačítka. Pak si takové tlačítko sice pro sebe zabere spoustu místa, ale samo se do něj nezvětší. To znamená, že toto místo již navždy zůstane nevyužito.



*Prostřední widget se neroztáhne v horizontálním smyslu a zůstane malé, ostatní se roztáhnou *** po roztáhnutí okna to je patrné*

Ještě se podívejme na poslední parametr, který určuje mezeru okolo tlačítka. Nastavme u prvního tlačítka mezeru o velikosti 50.

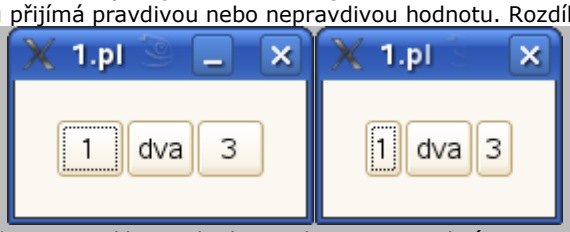


Mezera okolo prvního widgetu

Metodu voláme nad objektem Gtk2::Box, ze kterého dědí kontejnery. Poznamenejme ještě, jaké parametry přijímají konstruktory těchto základních kontejnerů. Zde máme strukturu volání, ze které by měl být význam jednotlivých parametrů patrný.

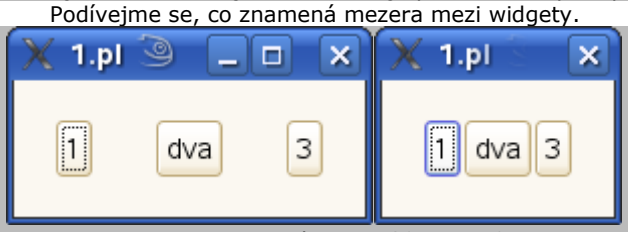
```
$hbox = Gtk2->HBox->new($stejna_velikost_objektu_v_boxu, $mezera_mezi_widgety);
```

$stejna_velikost_objektu_v_boxu přijímá pravdivou nebo nepravdivou hodnotu. Rozdíl je patrný z těchto dvou obrázků.



Hodnota $stejna_velikost_objektu_v_boxu je pravdivá, resp. nepravdivá

Podívejme se, co znamená mezera mezi widgety.



Mezera mezi widgety je 30, resp. 0

Chcete-li si sami zaexperimentovat a lépe tak zjistit, jak jednotlivé parametry fungují, lze použít následující zdrojový kód, na kterém jsme skládání widgetů ilustrovali pomocí obrázků.

```
use Gtk2 "-init";

$okno = Gtk2::Window->new("toplevel");
$okno->set_border_width(20);

#vytvoříme HBox s
$hbox = Gtk2::HBox->new(0, 0);

$tlacitko1 = Gtk2::Button->new("1");
$tlacitko2 = Gtk2::Button->new("dva");
$tlacitko3 = Gtk2::Button->new("3");
$hbox->pack_start($tlacitko1, 1, 1, 0);
$hbox->pack_start($tlacitko2, 1, 1, 0);
$hbox->pack_start($tlacitko3, 1, 1, 0);
$okno->add($hbox);

$tlacitko1->show;
$tlacitko2->show;
$tlacitko3->show;
```

```
$hbox->show;
$okno->show;

Gtk2->main;
```

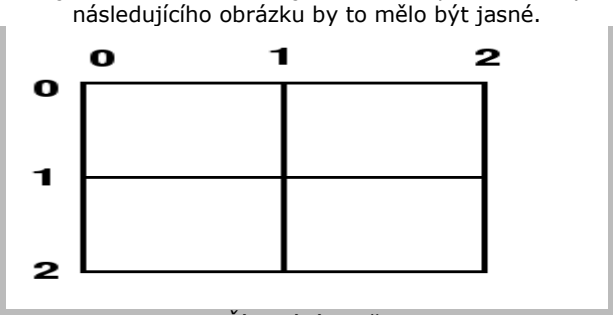
Mřížka

Mřížka (grid), v Gtk zvaná tabulka, je speciální typ kontejneru. Funguje tak, že prostor rozdělíme na několik oblastí a do nich poté přiřazujeme widgety podle toho, kam je chceme umístit.

Mřížku vytvoříme následujícím příkazem. Je třeba určit počet řádků a sloupců. Také můžeme změnit implicitní nastavení a požadovat stejnou velikost všech buněk (pak bude každá buňka stejně velká jako ta, ve které je umístěn největší widget).

```
$mrizka = Gtk2::Table->new($radku, $sloupcu, $stejna_velikost_objektu_v_boxu);
```

Všechny vertikální a horizontální linie mají svá čísla. Vertikální jdou zleva doprava od nuly, horizontální analogicky shora dolů. Z následujícího obrázku by to mělo být jasné.



Číslování buněk

Metodou attach nyní přiřazujeme do jednotlivých buněk (či skupin buněk) widgety.

```
$mrizka->attach($widget, $leva, $prava, $horni, $dolni,
$horizontalni_nastaveni, $vertikalni_nastaveni, $horizontalni_padding, $vertikalni_padding);
```

Lze použít i následující jednodušší volání.

```
$mrizka->attach_defaults($widget, $leva, $prava, $horni, $dolni);
```

Proměnné $leva, $prava, $horni, $dolni určují hrany. Tyto hrany určí životní prostor widgetu$widget. $horizontalni_nastaveni, $vertikalni_nastaveni určují věci jako expanze, roztahovaní apod. Možné hodnoty jsou expand, fill, shrink.

Pomocí následujících metod můžeme nastavovat mezery mezi sloupci a řádky.

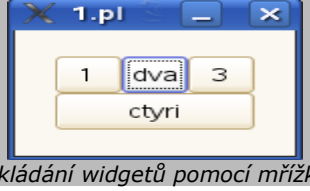
```
$mrizka->set_row_spacing($radek, $mezera);
$mrizka->set_col_spacing($sloupec, $mezera);
$mrizka->set_row_spacings($mezera);
$mrizka->set_col_spacings($mezera);
```

Použití mřížky

Význam jednotlivých příkazů byl dostatečně vysvětlen, takže jen pro představu uveďme jednoduché použití mřížky. Již nebudeme experimentovat s nastaveními typu expanze a roztažení, protože je to analogické jako u HBox a VBox.

```
use Gtk2 "-init";

$okno = Gtk2::Window->new("toplevel");
$okno->set_border_width(20);

$mrizka = Gtk2::Table->new(2, 3, 1);

$tlacitko1 = Gtk2::Button->new("1");
$tlacitko2 = Gtk2::Button->new("dva");
$tlacitko3 = Gtk2::Button->new("3");
$tlacitko4 = Gtk2::Button->new("ctyri");
$mrizka->attach_defaults($tlacitko1, 0, 1, 0, 1);
$mrizka->attach_defaults($tlacitko2, 1, 2, 0, 1);
$mrizka->attach_defaults($tlacitko3, 2, 3, 0, 1);
$mrizka->attach_defaults($tlacitko4, 0, 3, 1, 2);

$okno->add($mrizka);

$okno->show_all;

Gtk2->main;
```

Tento kód vyprodukuje jednoduchou aplikaci, která vypadá stejně jako na tomto obrázku.



Skládání widgetů pomocí mřížky

Perl (130) - Gtk2 - menu a toolbary

Naprostá většina běžných GUI aplikací má jednu věc společnou: pro přehlednou navigaci mají v horní části menu. I proto je tento widget plnohodnotně zakotven v Gtk2. Uvedeme si ještě toolbary, kterými se lze k cíli dostat rychleji a přidávají se hlavně pro často používané akce.

Menu

Pojďme se pro začátek podívat, jak vytvoříme menu v naší aplikaci. Postupujme krok za krokem.
Vytvoříme tedy nejprve kontejner, na jehož horní část menu umístíme. Uvědomme si zde, že menu je widget jako každý jiný a tedy ho musíme jako všechny ostatní widgety umístit sami.

```
my $vbox = Gtk2::VBox->new(0,0);
```

Nyní můžeme vytvářet nabídky. Nabídkou rozumíme seznam položek, který se objeví po stisknutí některého z tlačítek menu. Příslušný widget se jmenuje Gtk2::Menu. Do této nabídky budeme pomocí metody append přidávat položky.

```
my $menu_soubor = Gtk2::Menu->new();
```

Různé typy položek

Do horní části nabídky vložme tear-off položku, což je velmi praktické tlačítko, díky němuž může zůstat menu trvale otevřené (ve zvláštním "okně").

```
$menu_soubor->append(Gtk2::TearoffMenuItem->new);
```

A nyní již vkládejme klasické položky. Lze vkládat například stock tlačítka. Hierarchie je taková, že do nabídky můžeme zařazovat položky. Položku tedy nejprve standardně vytvoříme a poté zařadíme.

```
my $polozka = Gtk2::ImageMenuItem->new_from_stock("gtk-find", undef);
$menu_soubor->append($polozka);
```

Dále můžeme vkládat tlačítka s vlastním obrázkem.

```
$polozka = Gtk2::ImageMenuItem->new("Ikony");
$polozka->set_image(Gtk2::Image->new_from_file("./Ghost.png"));
$menu_soubor->append($polozka);
```

Samozřejmě je k dispozici separátor.

```
$menu_soubor->append(Gtk2::SeparatorMenuItem->new());
```

A vyzkoušejme ještě checkbutton položku, což je přepínač, který může být buď vypnutý nebo zapnutý.

```
$polozka = Gtk2::CheckMenuItem->new("_Zalamovani radku");
$menu_soubor->append($polozka);
```

Když se rozhodneme, že máme v menu vše, co potřebujeme, vytvoříme záhlaví nabídky a nabídku mu přiřadíme.

```
my $soubor= Gtk2::MenuItem->new("_Soubor");
$soubor->set_submenu($menu_soubor);
```

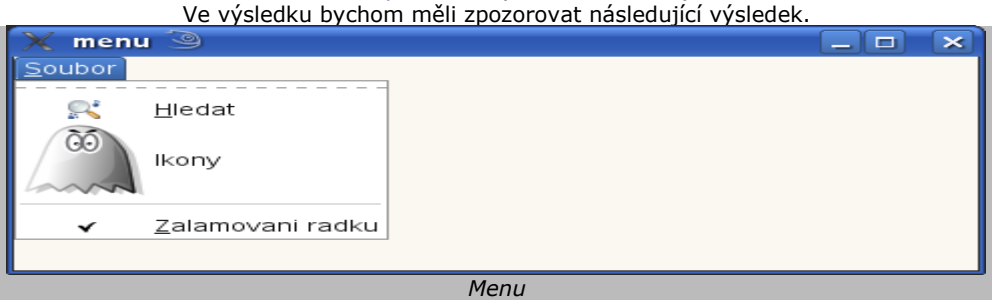
Záhlaví se skládají do menubaru.

```
my $menubar = Gtk2::MenuBar->new;
$menubar->append($soubor);
```

A na závěr menu vložíme na vrchol VBoxu.

```
$vbox->pack_start($menubar,0,0,0);
```

Ve výsledku bychom měli zpozorovat následující výsledek.



Menu

Aktivace položek

Zatím ale naše menu nic neumí. Chceme-li aktivovat tlačítka, pak je nutné spojit každé z nich s nějakou akcí. Běžné tlačítko emituje signál activate. Checkbutton a radiobutton emitují signál toggled.
Není tedy potřeba učit se nic nového, protože odchytávat signály již dovedeme.

```
$tlacitko_ulozit->signal_connect("activate" => \&ulozit);
```

Další možnosti menu

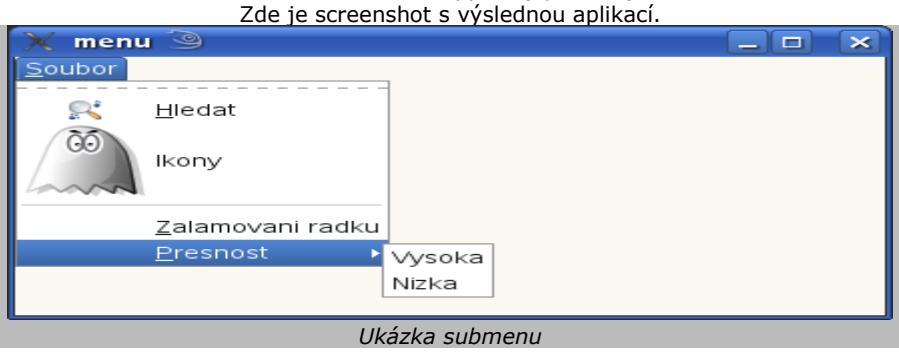
Chceme-li, aby se záhlaví menu zobrazovalo implicitně vpravo, zavoláme metodu set_right_justified.

```
$soubor->set_right_justified(1);
```

Chceme-li vytvořit submenu, použijeme set_submenu. Podívejme se v náznaku jak se to dělá. Do menu si přidáme další položku.

```
my $submenu = Gtk2::Menu->new();
$polozka = Gtk2::MenuItem->new("Vysoka");
$submenu->append($polozka);
$polozka = Gtk2::MenuItem->new("Nizka");
$submenu->append($polozka);

$polozka = Gtk2::MenuItem->new("_Presnost");
$polozka->set_submenu($submenu);
$menu_soubor->append($polozka);
```

Zde je screenshot s výslednou aplikací.



Ukázka submenu

Pro složitější menu, která jsou závislá na akcích lze použít nástroj Gtk2::UIManager, který jejich tvorbu značně usnadňuje generováním z XML.

Toolbary

Toolbar je widget podobného významu jako Menu a používá se u položek, které očekáváme, že budou mimořádně často využívány.

gtk2-perl využívá pro vytváření toolbarů widgetu Gtk2::Toolbar. Jednotlivé položky na toolbaru jsou objekty typu Gtk2::ToolItem.

Položky toolbaru mohou obsahovat buď jen text nebo jen ikony nebo obojí zároveň. V toolbaru mohou být i složitější widgety (stačí se podívat na internetový prohlížeč - zde je navíc například adresní řádek nebo kolonka pro vyhledávání).

Toolbary mohou být přesouvatelné. To umí Gtk2 samozřejmě také.

Pojďme se podívat na to, jak přidáme toolbar do naší aplikace.

Začněme widgetem Gtk2::HandleBox, pomocí kterého můžeme toolbar metodou drag&drop přemísťovat. Vložíme ho opět do kontejneru VBox. V kontextu aplikace bude vypadat vložení toolbaru takto.

```
use Gtk2 "-init";
$okno = Gtk2::Window->new("toplevel");
my $vbox = Gtk2::VBox->new(0,0);
my $handlebox = Gtk2::HandleBox->new;

#vytvoření toolbaru

$handlebox->add($toolbar);
$vbox->pack_start($hb,0,0,0);
$okno->add($vbox);
$okno->show_all;
```

Do HandleBoxu pak vložíme toolbar.

```
my $toolbar = Gtk2::Toolbar->new;
```

Do toolbaru můžeme vkládat několik druhů objektů. Podívejme se, které.

Objekt	Význam
Gtk2::ToolButton	obyčejné tlačítko
Gtk2::RadioToolButton	radiobutton
Gtk2::ToggleToolButton	checkbutton
Gtk2::MenuToolButton	vysouvací menu
Gtk2::ToolItem	vkládání dalších widgetů
Gtk2::SeparatorToolItem	klasický separátor

Vložme si do toolbaru dva radiobuttony. Nejprve vytvoříme tlačítka a poté je vložíme do toolbaru metodou insert.

```
my $radio1 = Gtk2::RadioToolButton->new(undef);
$radio1->set_label("Zalamovat");
$radio1->set_icon_widget(Gtk2::Image->new_from_file("./2.gif"));
$toolbar->insert($radio1,-1);

my $radio2 = Gtk2::RadioToolButton->new_from_widget($radio1);
$radio2->set_label("Nezalamovat");
$radio2->set_icon_widget(Gtk2::Image->new_from_file("./1.gif"));
$toolbar->insert($radio2,-1);
```

Úplně stejně bude vypadat vložení separátoru.

```
$toolbar->insert(Gtk2::SeparatorToolItem->new ,-1 );
```

Podívejme se ještě, jak vytvoříme vysunovací menu.

```
my $menubutton = Gtk2::MenuToolButton->new_from_stock("gtk-undo");
my $menu = Gtk2::Menu->new();
$menu->append(Gtk2::MenuItem->new("O _1 krok zpet"));
$menu->append(Gtk2::MenuItem->new("O 1_0 kroku zpet"));
$menu->show_all();
$menubutton->set_menu($menu);
$toolbar->insert($menubutton,-1);
```

Pomocí ToolItem lze vkládat do toolbaru další widgety. Vložme zde často užívaný widget Entry, který může reprezentovat adresní řádek.

```
my $toolitem = Gtk2::ToolItem->new;
$toolitem->add(Gtk2::Entry->new);
$toolbar->insert($toolitem,-1);
```

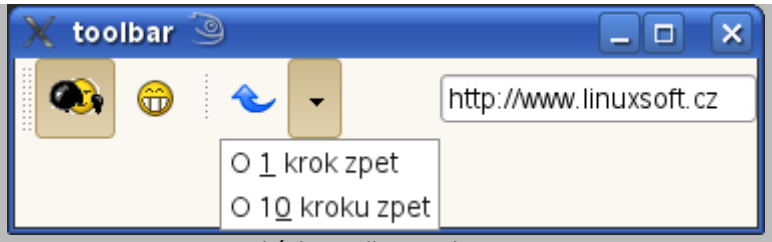
Kdyby někdo nevěděl, co sem má napsat, poradíme mu. Díky Gtk2::Tooltips lze vytvářet nad objekty v toolbaru žluté bubliny. Napišme do ní tedy, že se sem vkládá WWW adresa.

```
$toolitem->set_tooltip(Gtk2::Tooltips->new, "WWW adresa", "");
```

Kdybychom chtěli vložit do toolbaru pružnou mezeru, která se bude dle potřeby zvětšovat a zmenšovat, udělali bychom to pomocí expandujícího separátoru, který by se nezobrazil.

```
my $mezera = Gtk2::SeparatorToolItem->new;
$mezera->set_draw(0);
$mezera->set_expand(1);
$toolbar->insert($mezera,-1 );
```

Tím jsme si ukázali, jak přidávat do toolbaru objekty.

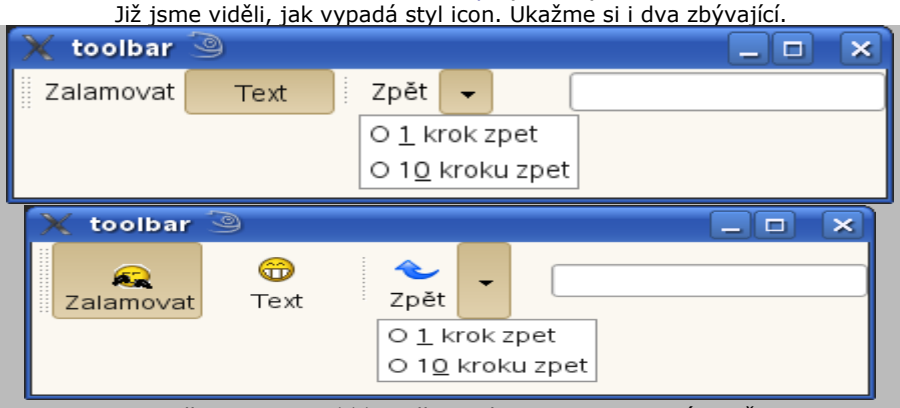


Ukázka toolbaru s ikonami

Na závěr se zmiňme o metodě set_style, která nastavuje, zda budou tlačítka textová, s ikonou nebo obojí. Možné volby jsou text, icon a both.

```
$toolbar->set_style("both");
```

Již jsme viděli, jak vypadá styl icon. Ukažme si i dva zbývající.



*toolbar s textem *** toolbar s ikonami i textem zároveň*

Perl (131) - Gtk2 - transparentní okna, tray ikona, výběr souborů

Transparentní okna jsou základním kamenem pro vytváření aplikací s OSD efektem. Ty umí uživatele informovat pomocí zprávy umístěné kamkoliv do prostoru obrazovky mimo okno aplikace. Tray ikona je tlačítko na panelu v grafických prostředích, které zpřístupňuje aplikaci nebo její vybrané části.

Transparentní okna - OSD aplikace

Gtk2 umí vytvářet speciální druh oken, kdy žádné okno vlastně vidět není a zobrazuje se pouze jeho obsah. Takovým aplikacím se dává přívlastek OSD (on screen display).

Zpravidla jde o zobrazování textu, ale patří sem také řadu efektů se zobrazováním obrázků. Gtk2 tyto pokročilejší záležitosti umí také.

OSD se používá často u mnoha běžných aplikací. Například u hudebních přehrávačů pro informaci o názvu skladby a interpretovi, ukazování času nebo třeba jako indikace došlých emailů či jabber zpráv.

Podíváme se, jak lze vytvářet transparentní okna v Gtk2.

```
use Gtk2 -init;
$okno = Gtk2::Window->new("popup");
```

Vytvořme kreslící plochu Gtk2::DrawingArea.

```
$drawing_area = Gtk2::DrawingArea->new;
$drawing_area->set_size_request(1000, 1000);
$okno->add($drawing_area);
$okno->show_all();
```

Do ní nastavíme to, co chceme, aby se později zobrazilo. Nastavíme zvětšený text AHOJ.

```
$text = $drawing_area->create_pango_layout("");
$text->set_markup("<span background='#000000' size='250000'>AHOJ</span>");
($w, $h)=$text->get_pixel_size;
```

Zkonvertujeme obrázek do bufferu pixelů.

```
$pixmap = Gtk2::Gdk::Pixmap->new($drawing_area->window, $w, $h, -1);
$pixmap->draw_layout($drawing_area->style->white_gc, 0, 0, $text);
$pixbuf = Gtk2::Gdk::Pixbuf->new ("rgb", TRUE, 8, $w, $h);
$pixbuf->get_from_drawable($pixmap, Gtk2::Gdk::Colormap->get_system, 0, 0, 0, 0, $w, $h);
```

Dále pixelům přidáme složku průhlednosti.

```
$pixbuf = $pixbuf->add_alpha (TRUE, 0, 0 , 0);
```

Nyní získáme masku (to je bitmapa, která určuje, které pixely jsou aktivní a které nikoliv).

```
($pm, $maska) = $pixbuf->render_pixmap_and_mask(255);
```

Na závěr tedy zvoditelníme pouze pixely z masky.

```
$okno->shape_combine_mask($maska, 0, 0);

Gtk2->main;
```

A výsledek? Takto může po spuštění aplikace vypadat levý horní roh pracovní plochy.

Část plochy po spuštění aplikace s OSD

Tray ikona

Napíšeme aplikaci, která využije místa na panelu na grafickém prostředí a vytvoří tray ikonu. Ta se běžně používá například pro přepínání jazyka, indikaci, kolik máme nepřečtených emailů, zda vyšly nové RSS zprávy, zda přišla zpráva přes jabber nebo jen pro aktivaci aplikací, které právě běží na pozadí.

Abychom mohli tray ikonu použít, musíme zvlášť modul Gtk2::TrayIcon nainstalovat.

```
# cpan Gtk2::TrayIcon
```

Poté již nic nebrání vytvoření jednoduché aplikace. Nejprve vytvoříme ikonu a pomocí EventBoxu k ní přiřadíme obrázek. To je vše, co musíme udělat.

```
use Gtk2 "-init";
use Gtk2::TrayIcon;

my $img= Gtk2::Image->new_from_file("./ikona.gif");
my $eventbox = Gtk2::EventBox->new;
$eventbox->add($img);

my $ikona= Gtk2::TrayIcon->new("tray");
$ikona->add($eventbox);

$ikona->show_all;

Gtk2->main;
```

Ještě bychom si měli ukázat, jak se vytváří akce. Po kliknutí na tray ikonu se emituje signál button-press-event. Pokud na ni jen najedeme kurzorem myši, emituje se enter-notify-event.

Tyto dvě události odchytíme. Jakmile se myš dostane do prostoru tray ikony, vypíšeme na standardní výstup informaci. Po kliknutí na tray ikonu program ukončíme.

```
$eventbox->signal_connect("enter-notify-event" => sub{print "Jsi v oblasti tray ikony\n";});
$eventbox->signal_connect("button-press-event" => sub{Gtk2->main_quit;});
```

Spusťme aplikace a podívejme se na panel. Měla by se tam objevit tray ikona.

Naše tray ikona

Vybírání souborů

K dispozici máme několik widgetů, které vyberou z adresářové struktury soubor. Představíme si Gtk2::FileChooserButton a Gtk2::FileChooserDialog.

Použití prvního je jednoduché.

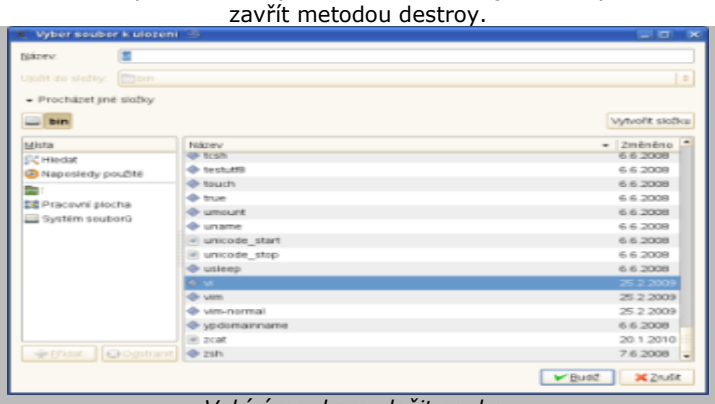
```
my $soubor = Gtk2::FileChooserButton->new("Vyber soubor" , "open");
my $adresar = Gtk2::FileChooserButton->new("Vyber adresar" , "select-folder");
```

Gtk2::FileChooserDialog se používá buď pro výběr místa uložení souboru nebo pro výběr souboru, který má být otevřen. Vybereme si tedy nejprve jednu variantu. Dále se musíme rozhodnout pro tlačítka, která zobrazit.

Vytvořme si dialog pro uložení souboru s obvyklými tlačítky "OK" a "Zrušit".

```
my $vyber =  Gtk2::FileChooserDialog->new(
"Vyber soubor k ulozeni", undef, "save", "gtk-ok" => "ok", "gtk-cancel" => "cancel");
my $vysledek = $vyber->run;
```

Uživatel si vybere a výsledek obdržíme v proměnné $vysledek. Poznamenejme, že bychom měli po obdržení výsledku dialog zavřít metodou destroy.



Vybíráme, kam uložit soubor

Zajímavé je též používání filtrů. Pomocí filtru můžeme v nabídkách zobrazovat jen soubory podle určité masky nebo MIME typu.

```
my $filter = Gtk2::FileFilter->new();
$filter->set_name("HTML soubory");
$filter->add_mime_type("text/html");
```

Chceme-li použít raději masku, pak napíšeme následující.

```
$filter->add_pattern("*.html");
$filter->add_pattern("*.htm");
```

Filtr pak přidáme metodou add_filter.

```
$vyber->add_filter($filter)
```

Velmi efektně může v aplikacích vypadat zakomponování přetahování dat myší. Všichni to známe jako uživatelé. Jak ale drag&drop funguje v pozadí aplikace? Zdánlivě trochu komplikovanější postup si rozebereme na elementární kroky. Druhým tématem je nástroj na pohodlnou konfiguraci aplikace.

Drag & Drop

Přetahování objektů myší patří mezi další zajímavé schopnosti knihovny Gtk2. Přetahování probíhá v několika fázích. Nejprve existuje nějaký zdrojový objekt. Tím může být v podstatě jakýkoliv widget. Jakmile ho uživatel chytí na kurzor, stává se z něj jiný objekt, který se stará o animaci. Po uvolnění tlačítka myši se objekt přestane pohybovat a stává se z něj cílový objekt. Uvolnění tlačítka může nastat i nad widgetem z jiné aplikace.

Nejprve je třeba aktivovat widgety, na kterých bude drag&drop proces probíhat.

- pomocí metody Gtk2::Widget->drag_source_set nastavíme, že je widget připravený k procesu
- pomocí metody Gtk2::Widget->drag_dest_set nastavíme všechny možné cíle pro přetahovaný objekt

Samotný proces pak můžeme rozdělit do několika kroků.

1. Uživatel uchopí widget
2. Emituje se signál drag-data-get, zde je třeba zapamatovat si uchopená data
3. Uživatel se posunuje
4. Uživatel upustí widget nad nějakým jiným widgetem
5. Emituje se signál drag-data-received, zapamatovaná data zde můžeme použít

Napíšeme si aplikaci, na které si jednotlivé kroky vysvětlíme. Bude obsahovat nějaký seznam objektů, do kterého budeme moci upouštět objekty nové. Po upuštění obsahu vždy do seznamu přidáme novou položku.

Drop - destinace a upouštění

Nastavme nejprve cílový seznam, kam budeme moci objekty upouštět. Seznam vytvoříme pomocí Gtk2::SimpleList. Položky budou vždy obsahovat ikonu a popisek.

```
my $list = Gtk2::SimpleList->new("ikona" => "pixbuf","status"  => "markup");
```

Dále je třeba nastavit, která data bude náš cíl přijímat. Nyní se spokojme pouze s přijímáním obyčejného textu text/plain.

```
$list->drag_dest_set("all", ["copy"], {"target" => "text/plain",  "flags" => []},);
```

Protože seznam bude potenciálně obsahovat hodně položek, vložíme ho do okna s posuvníky. Zajistíme tak, aby se okno zbytečně samo nezvětšovalo.

```
my $scrolled = Gtk2::ScrolledWindow->new(undef,undef);
$scrolled->set_policy("never", "automatic");
$scrolled->add($list);
$scrolled->set_size_request(300,200);
```

Poté ještě nastavíme akci, která se provede, jakmile náš seznam obdrží data.

```
$list->signal_connect("drag-data-received" => \&upusteno, $list);
```

Funkce upusteno pouze vygeneruje ikonu a přidá do seznamu novou položku, což je tato ikona s textem. Předali jsme si $list jako parametr, tudíž nám nic nebrání s ním manipulovat.

```
sub upusteno {
my ($widget, $kontext, $x, $y, $data, $info, $cas, $list) = @_;
my $icon_factory = Gtk2::IconFactory->new();
my $iconset = $icon_factory->lookup_default("gtk-ok");
my $pixbuf = $iconset->render_icon(Gtk2::Style->new(), "none", "normal", "menu", undef);
push @{$list->{data}}, [ $pixbuf, $data->data ];
}
```

Drag - uchopení objektu

Nyní je polovina programu za námi. Zbývá dodefinovat zdrojové objekty. Vytvořme si například label, který budeme moci přesunout.

```
my $label = Gtk2::Label->new("Zkus me vzit a presunout");
```

Nyní z něj pomocí EventBoxu vytvoříme zdroj. Přidáme tedy label do EventBoxu a metodou drag_source_set nastavíme chování při pokusu o přetažení.

Povolíme pouze přetahování prvním a třetím tlačítkem myši. Jako první parametr tedy uvedeme ['button1_mask', 'button3_mask']. Budeme kopírovat obyčejný text.

```
my $event_box = Gtk2::EventBox->new();
$event_box->add($label);
$event_box->drag_source_set (
["button1_mask", "button3_mask"],
["move"],
{"target" => "text/plain","flags" => []}
);
```

Pomocí flags se dá nastavit, zda máme přijímat pouze obsah, který je přetahován z též aplikace (pak nastavíme GTK_TARGET_SAME_APP) nebo téhož widgetu (pak použijeme konstantu GTK_TARGET_SAME_WIDGET).

Vytvořme opět akci, která se provede, jakmile uživatel začne náš objekt přetahovat. Zde akorát musíme nastavit data, která si přetahovaný objekt ponese s sebou.

```
$event_box->signal_connect ("drag-data-get" => \&sebrano,
"<span foreground='green' size='15000'>Hotovo!</span>");

sub sebrano {
my ($widget, $kontext, $data, $info, $cas, $retezec) = @_;
$data->set_text($retezec, -1);
}
```

Tím je hotový jednoduchý program, který je schopen procesu drag&drop. A nyní si zkuste přetáhnout nějaký text z jiné aplikace (například adresní řádek v internetovém prohlížeči). Co se stane?

Úplně stejně lze postupovat při přetahování obrázků. Vždy je třeba vytvořit zdroj pomocí drag_source_set a v cíli metodou drag_dest_set povolit obrázky jakožto data, která náš seznam přijme.

Druid - konfigurační obrazovky

Druid (častěji nazývaný wizard) je soubor několika po sobě jdoucími okny, které obvykle pokládají uživateli otázky. Uživatel nastavuje hodnoty a kliká na tlačítka "Další", "Zpět" apod. To se děje často při konfigurování aplikace po prvním spuštění.

Budeme používat modul Gnome2::Druid. Měli bychom tedy nainstalovat modul Gnome2.

```
# cpan Gnome2
```

Nebudeme vytvářet čistou Gtk2 aplikaci, ale Gnome2 aplikaci. Je třeba na to pamatovat kvůli horší přenositelnosti.

V aplikaci tedy načteme tento modul.

```
#!/usr/bin/env perl
use strict;
use warnings;
use Gnome2;
use Gtk2 "-init";
```

Dále vložíme informace o naší aplikaci a verzi.

```
Gnome2::Program->init("NaseAplikace", "0.1", "");
```

A nyní již můžeme vytvořit druida. Pro nás nejzajímavějším parametrem je zde titulek okna.

```
my($druid, $okno) = Gnome2::Druid->new_with_window("Pruvodce konfiguraci", undef, 1);
```

Dále budeme postupně přidávat stránky. Obvykle máme úvodní stránku, závěrečnou stránku a nějaké stránky mezi tím. Úvodní a závěrečnou stránku vytváříme pomocí Gnome2::DruidPageEdge. Na všechny prostřední stránky obvykle používáme widget Gnome2::DruidPageStandard.

Začněme tedy popořádku úvodní stránkou. Zde nastavíme hlavní nadpis úvodní stránky a nějaký uvítací text. Taktéž zde nastavujeme tři obrázky, které se zobrazí - logo a dva vodotisky (liší se pouze tím, na které části okna se zobrazují). My nastavíme dva z nich.

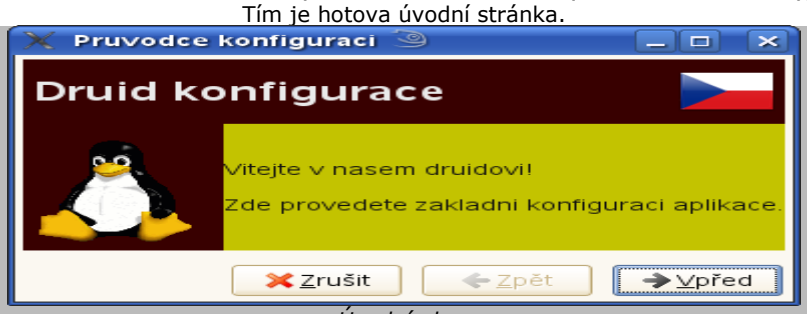
Na závěr nesmíme zapomenout okno přiřadit našemu druidovi.

```
my $druid_uvod = Gnome2::DruidPageEdge->new_with_vals("start", 0,
"Druid konfigurace",
"Vitejte v nasem druidovi!\n\nZde provedete zakladni konfiguraci aplikace.",
Gtk2::Gdk::Pixbuf->new_from_file("./czech.png"),
Gtk2::Gdk::Pixbuf->new_from_file("./linux.png"),
);
$druid->append_page($druid_uvod);
```

Dále nastavíme barvy. K tomu máme k dispozici několik metod podle toho, která část okna bude ovlivněna. Nastavíme barvu pozadí, pozadí textu a titulku.

```
$druid_uvod->set_bg_color(Gtk2::Gdk::Color->new(80000,0,0));
$druid_uvod->set_textbox_color(Gtk2::Gdk::Color->new(50000,50000,0));
$druid_uvod->set_title_color(Gtk2::Gdk::Color->new(60000,60000,60000));
```

Tím je hotova úvodní stránka.

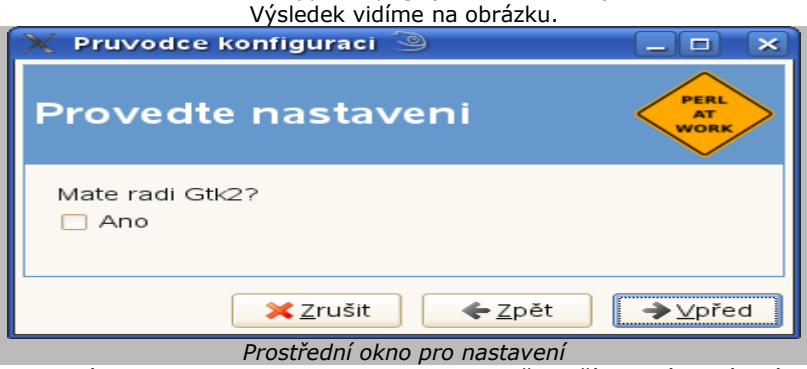


Úvodní okno

Podívejme se na další stránku. Vytvoříme ji z widgetu Gnome2::DruidPageStandard. Opět zde nastavíme nějaký nadpis. Metodou append_item navíc můžeme vkládat další widgety. Vložme sem tedy zaškrtávací políčko. Na závěr opět musíme přidat stránku do druida.

```
my $druid_strana1 = Gnome2::DruidPageStandard->new_with_vals("Provedte nastaveni",
Gtk2::Gdk::Pixbuf->new_from_file("./druid2.png"), undef);
$druid_strana1->append_item("Mate radi Gtk2?", Gtk2::CheckButton->new("Ano"), "");
$druid->append_page($druid_strana1);
```

Výsledek vidíme na obrázku.



Prostřední okno pro nastavení

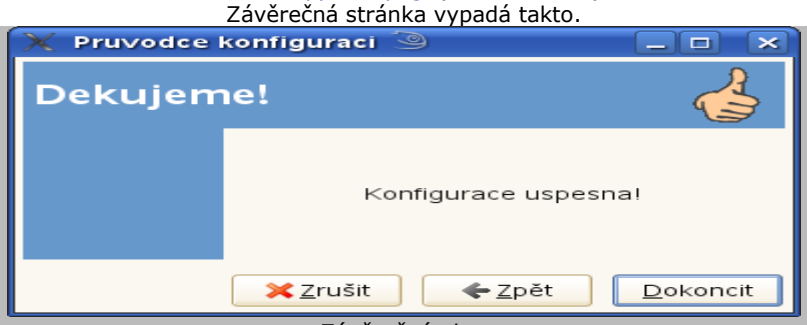
Na závěr napíšeme poslední okno. Gnome2::DruidPageEdge jsme již používali u úvodní stránky. Zde je to stejné. Pozornost si však zaslouží nastavení dlačítka "Dokončit", které na konci obvykle nahrazuje tlačítko "Další".

Dále si zde také vyzkoušíme odchytit signál. Jakmile je kliknuto na "Dokončit", vytiskneme na standardní výstup informaci o tom, že konfigurace byla dokončena.

```
my $druid_konec = Gnome2::DruidPageEdge->new_with_vals("finish", 0,
"Dekujeme!",
"Konfigurace uspesna!",
```

```
Gtk2::Gdk::Pixbuf->new_from_file("./druid4.png"),
undef,
undef);
$druid->finish->set_label("_Dokoncit");
$druid->finish->signal_connect(clicked => sub{
print
"Konfigurace dokoncena! Muzeme pokracovat.";
Gtk2->main_quit;
});
$druid->append_page($druid_konec);
```

Závěrečná stránka vypadá takto.



Závěrečné okno

Abychom zde měli kód kompletní, doplňme standardní zakončení programu. Doplňme, že je zvykem, aby se druid objevoval vždy uprostřed obrazovky.

```
$okno->set_position("center-always");
$okno->show_all;
Gtk2->main;
```

Na závěr poznamenejme, že každá stránka emituje po potvrzení uživatelem signál next.

Perl (133) - Gtk2 - úpravy vzhledu aplikací pomocí rc

Odbočme trochu stranou a naučme se stanovovat defaultní hodnoty pro vzhled Gtk2 pomocí resource-file souborů. Díky tomu budeme moci v našich aplikacích využívat vlastní motivy Gtk2.

V adresáři /usr/share/themes nalezneme motivy pro vzhled Gtk2 aplikací. Zde se dají také upravovat implicitní hodnoty pro různá nastavení.

Zkusme stručně nahlédnout, jak témata vlastně fungují.

Témata jsou zapsané v tzv. rc souborech, které mají speciální syntaxi. Jsou uložené zpravidla v adresáři /home/uživatel/.themes/ nebo na veřejném místě v /usr/share/themes/, případně /usr/local/share/themes/.

Nastavení témat pro naší aplikaci

Pro každý widget nastavujeme téma zvlášť. Můžeme ale využít dědičnosti. Na gnome.org najdeme widgetovou hierarchii. Protože všechny widgety dědí od Gtk::Widget, lze snadno nastavit všechny widgety najednou.

Je několik možností, jak téma použít uvnitř naší aplikace. Nejjednodušší je napsat si vlastní rc soubor a uvnitř aplikace ho pomocí Gtk2::Rc nastavit. To se dělá následovně.

```
Gtk2::Rc->parse("/usr/share/themes/MojeTema/gtk-2.0/gtkrc");
```

Analogický je následující příkaz.

```
Gtk2::Rc->parse_string("include "/usr/share/themes/MojeTema/gtk-2.0/gtkrc"");
```

Parametr příkazu je v tomto případě již psaný v rc syntaxi, přičemž include je příkaz pro načtení rc souboru.

Díky parse_string můžeme vkládat rc syntaxi přímo do naší aplikace. Můžeme tedy psát příkazy následujícího typu.

```
Gtk2::Rc->parse_string(<<EOF);
include '/usr/share/themes/MojeTema/gtk-2.0/gtkrc'
style 'normal' {
font_name ='serif 30'
}
EOF
```

Vytváření témat

Podívejme se na syntaxi rc souborů. Chceme-li vytvořit téma s názvem MojeTema, pak se bude náš rc jmenovat /usr/share/themes/MojeTema/gtk-2.0/gtkrc.

Na začátku rc souboru může být série include příkazů, které se používají pro načítání nastavení z jiných souborů. Chceme-li téma logicky rozdělit do několika souborů (například soubor /usr/share/themes/MojeTema/gtk-2.0/tlacitka.rc může řešit vzhled tlačítek, soubor /usr/share/themes/MojeTema/gtk-2.0/dialogy.rc může řešit vzhled dialogů apod.), vložíme na začátek gtkrc následující řádky.

```
include "tlacitka.rc"
include "dialogy.rc"
include "ostatni.rc"
```

Dále definujeme takzvané styly. Obvykle vytvoříme jeden výchozí styl s názvem default a pak můžeme vytvořit několik dalších. Styly mohou ovlivňovat buď implicitní nastavení (například velikost rámečku atd.) nebo zde můžeme definovat tzv. enginy.

Styl se vytváří následovně.

```
style "default" {
#nastavení
}
```

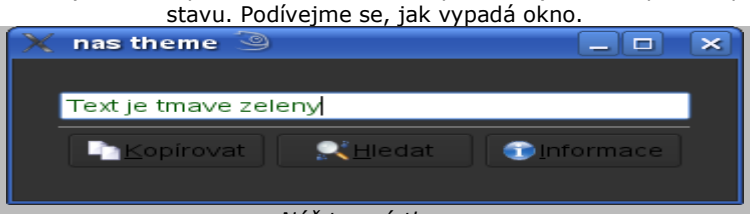
Vytvořme si na úvod jednoduchý styl. Náš gtkrc soubor bude vypadat takto.

```
style "default" {
xthickness = 1
ythickness = 1
```

GtkButton ::child-displacement-x = 6
GtkButton ::child-displacement-y = 2

bg[NORMAL] = "#333"
fg[NORMAL] = "#f0f0f0"
text[NORMAL] = "#0ff"
}

class "GtkWidget" style "default"
Nastavili jsme pro ukázku velikost mezery mezi textem a hranicí widgetu (xthickness, ythickness; používá se na různých místech), hloubku stisku tlačítka (o kolik se posune text na tlačítku při stisku) a barvu pozadí, popředí a textu v normálním stavu. Podívejme se, jak vypadá okno.



Náš tmavý theme

Podívejme se na některá nastavení.
Nejprve si všimněme nastavování implicitních hodnot. My jsme nastavili hodnoty child-displacement-x a child-displacement-y pro GtkButton. Chceme-li nastavit nějakou implicitní hodnotu pro daný widget, je vhodné se podívat do dokumentace, kde je seznam všech možných vlastností, možné hodnoty a vysvětlení, co která hodnota znamená. Tento seznam lze nalézt na live.gnome.org.
Jakmile nalezneme vhodnou vlastnost, stačí ji zapsat do stylu v rc souboru ve tvaru *JménoWidgetu*::*vlastnost* = *hodnota*.
Poznamenejme jen, že jméno widgetu v rc souborech neobsahuje :: (rc soubory nemají s Perl syntaxí nic společného).

Barvy

Barvy jsou vlastnost, která nás asi bude zajímat nejvíce. Jsou čtyři kategorie barev.

- bg - barva pozadí
- fg - barva popředí
- text - barva textu
- base - barva pozadí pro některé speciální widgety (například TextView, TreeView atd.)

Dále je pět stavů, ve kterých se může widget nacházet.

- NORMAL
- PRELIGHT - přejezd myší
- ACTIVE - pro právě stisknutá tlačítka
- SELECTED - pro vybranou oblast (například označený text)
- INSENSITIVE - neaktivní widgety (například formuláře, jejichž hodnoty nelze měnit)

Barvy nastavujeme vždy pro kategorii a stav. To zapisujeme kategorie[stav]. Tedy konkrétně například bg[NORMAL].
Barvy můžeme zadávat mnoha způsoby.

- přes kód barvy ve tvaru #RGB, #RRGGBB, #RRRGGGBBB, #RRRRGGGGBBBB
- {R, G, B}, kde hodnoty jsou mezi 0 a 1
- symbolickými názvy
- vlastními symbolickými názvy (ty se ve stylu definují příkazem barva["název"] = "#RRGGBB")
- pomocí efektů na změnu barev:
  - lighter(barva)
  - darker(barva)
  - mix(typ, barva1, barva2)
  - shade(typ, barva)
- pomocí proměnných

Pokud definujeme barvu pomocí proměnných, pak bychom měli ještě před definici stylů přiřadit do gtk_color_scheme a nastavit tím tak barvy pro celé téma. Hodnota pro gtk_color_scheme je seznamem hodnot, které jsou oddělené znakem nového řádku. Hodnoty jsou tvaru *kategorie*:*barva*, tedy například bg_color:#000. Používáme-li Gnome, je vhodné dodržovat následujících 8 kategorií: fg_color, bg_color, base_color, text_color, selected_bg_color, selected_fg_color, tooltip_bg_color, tooltip_fg_color.
Zde je příklad barevného schématu.
gtk_color_scheme = "fg_color:#010\nbg_color:#0f0\nbase_color:#fff\ntext_color:#f0f\n
selected_bg_color:#8ad\nselected_fg_color:#eee\ntooltip_bg_color:#ffb\ntooltip_fg_color:#000"
Barvy se potom zapisují jako @*kategorie*; například @bg_color.

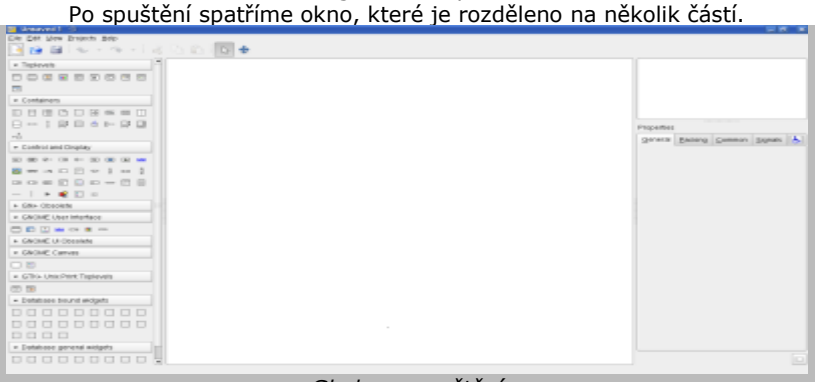
Použití stylu

Již jsme si napsali jednoduchý styl. Gtk2 ale ještě neví, co s ním má dělat. Musíme nějak specifikovat, jaké styly se mají použít na jakých místech.
Příkazem class je možné asociovat styly jednotlivým prvkům a využít hierarchie widgetů. Podívejme se na následující řádky.
class "GtkButton" style "tlacitka"
class "GtkMenu" style "default"
class "GtkWidget" style "default"
Všechny widgety, které dědí od Gtk2::Button (například Gtk2::ToggleButton, Gtk2::VolumeButton, Gtk2::ColorButton) budou mít styl tlacitka. Všechny widgety dědící od Gtk2::Menu budou mít styl default. Pro všechny widgety, které nemají nastaven styl, jsme nastavili styl default.
Poznamnejme, že lze využít i žolíkových znaků.
Jiný příkaz pro aplikaci stylu je widget_class. Ten využívá vnoření widgetů do sebe. Máme-li v okně tlačítko a v něm nějaký text, jde o vnořené widgety. Vnoření popisku tlačítka v okně tak můžeme zapsat jako GtkWidget.GtkButton.GtkLabel.
Chceme-li tedy nastavit pro widgety uvnitř tlačítek typu Gtk2::Button styl uvnitr_tlacitek, zapíšeme to do souboru gtkrc tímto řádkem.

widget_class "*.GtkButton.*" style "uvnitr_tlacitek"
Kdybychom to chtěli aplikovat i na jiné typy tlačítek (tj. na všechny widgety, které od Gtk2::Button dědí), upravili bychom příkaz do nové podoby.
widget_class "*.<GtkButton>.*" style "uvnitr_tlacitek"
Ještě existuje příkaz widget, kterým nastavujeme vzhled pro konkrétní widget. Všechny widgety mohou mít svá jména, která nastavujeme příkazem set_name.
$widget->set_name("nebezpecne-tlacitko-spusteni-stepne-reakce");
Vzhled pak nastavíme již zmíněným příkazem widget.
widget "nebezpecne-tlacitko*" style "nebezpeci"
Enginy
Enginy umožňují použít uvnitř našeho stylu nějaký existující motiv. Pro inspiraci se můžeme podívat do /usr/share/themes/ (či podobného adresáře), kde máme motivy uložené.
style "nas-styl" {
engine "thinice" {
#naše úpravy
}
}
Perl (134) - Gtk2 - Glade Interface Designer

Na závěr série o Gtk2 se podíváme na vývojové prostředí Glade. Lze v něm snadno a bez řešení technických detailů navrhovat aplikace, což nám ušetří výrazné množství času.

Vytváření aplikace bylo až dosud obrovské množství psaní. Každý widget zabral několik řádků. Kdybychom takto měli tvořit něco komplexnějšího s tisícovkami widgetů, museli bychom vynaložit spoustu mechanické práce. Proto vznikají u všech rozšířenějších grafických knihoven takzvané interface designery nebo-li obecněji RAD.
Interface designer je prostředí, ve kterém si snadno naklikáme, jak má naše aplikace vypadat. Obsahuje obvykle seznam widgetů, které můžeme klikáním přidávat z nabídky do našeho projektu a umísťovat (malým nedostatkem Glade možná je, že to nelze dělat metodou drag&drop). Díky tomu se nadále nemusíme starat o tvorbu widgetů.
Interface designer převede widgety, které jsme si naklikali, do nějakého kódu - obvykle XML. Z něj pak lze vzezření aplikace rekonstruovat. Otázkou je, jak. Každý interface designer má svoji metodu, jak z XML dostat naši aplikaci. Obvykle k tomu budeme potřebovat nějaký program (takovým programům se někdy říka sketchery). Některé sketchery konvertují XML přímo do zdrojového kódu daného jazyka, některé vytvoří pouze vazebný kód a používají XML ke generování aplikace při každém spuštění. Pro každý jazyk je samozřejme potřeba vlastní sketcher. Výhodou je, že XML kód bývá univerzální a lze ho použít v různých jazycích. Později se budeme zabývat tím, jak to funguje u Glade. Nejprve si ukážeme, jak se s Glade pracuje.
Glade má již poměrně dlouhou tradici. První verze spatřila světlo světa již v roce 1998. Dnes je k dispozici poslední verze 3.10.0. Ve většině linuxových distribucí již existuje balíček glade nebo glade-3. Pro jiný operační systém je třeba stáhnout Glade ze stránek projektu a dle instrukcí nainstalovat.
Vytváření aplikací v Glade
Spusťmě tedy Glade. Obvykle se to dělá příkazem glade-2, glade-3, glade3 nebo glade. Měli bychom ho také najít v nabídce našeho grafického prostředí.
Po spuštění spatříme okno, které je rozděleno na několik částí.



Glade po spuštění

V prostředním panelu budeme vytvářet náš projekt. Budeme přitom používat widgety z levého panelu. Pravá část okna je rozdělena na dvě části. Nahoře je seznam widgetů podle toho, jak jsou do sebe vnořené. Dole budeme nastavovat widgetům vlastnosti.
Zbytek je již jen klikání. Ovládání je velmi intuitivní. Vše se zde dá snadno dohledat, protože Glade neobsahuje nic zbytečného navíc.
Na úvod je nejlepší proklikat si všechny widgety a podívat se, co všechno se dá dělat. Každý widget se pak dá smazat stiskem klávesy delete, označíme-li ho v hierarchii widgetů.
Vložme na úvod do projektu pouze hlavní okno a uložme ho například jako nic.glade. Pokud ho otevřeme, uvidíme následující XML kód.
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<!--Generated with glade3 3.4.3 on Sun Dec 19 13:38:54 2010 -->
<glade-interface>
<widget class="GtkWindow" id="window1">
<child>
<placeholder/>
</child>
</widget>
</glade-interface>

Aplikace glade souborů

Podívejme se, jak lze z glade souborů vytvářet aplikace. Nejprve použijeme příkaz gtk-builder-convert, který fakticky pouze z jednoho XML souboru vytvoří jiný, který je čitelný modulem Gtk2::Builder. Proč tento mezikrok? První XML soubor je surovým výstupem Glade. Skript toto XML upraví do podoby, ve které pro nás bude dále čitelný. Kdybychom například chtěli aplikovat původní glade v jiném jazyku než v Perlu, mohl by skript vypadat jinak.

Zkusíme tedy nejprve zkonvertovat náš soubor nic.glade, který kóduje pouze prázdné okno.

gtk-builder-convert nic.glade nic.xml

Nahlédneme-li do zdrojového kódu, uvidíme, že vypadá téměř stejně jako glade soubor. V novějších verzích bude možné generovat přímo toto XML bez mezikroku.

```
<?xml version="1.0"?>
<!--Generated with glade3 3.4.3 on Sun Sep 19 13:38:54 2010 -->
<interface>
<object class="GtkWindow" id="window1">
<child>
<placeholder/>
</child>
</object>
</interface>
```

Tento soubor ale již můžeme použít přímo ze zdrojového kódu naší aplikace. Avšak pamatujme na to, že pro změny v uživatelském rozhraní pomocí Glade budeme dále původní glade soubor potřebovat.

Je několik možností, jak z vzniklého XML vytvořit aplikaci. My využijeme modulu Gtk2::Builder. Podívejme se jak.

```
use Gtk2 "-init";

$builder = Gtk2::Builder->new();
$builder->add_from_file("nic.xml");

$okno = $builder->get_object("window1");

$okno->show();
Gtk2->main();
```

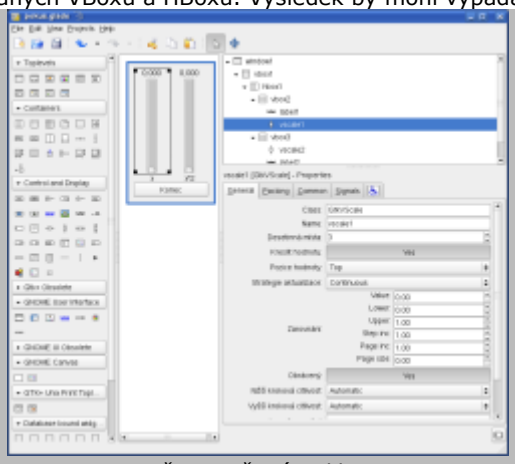
Tento kód již funguje. Zatím jsme pouze chtěli ukázat kostru postupu vytváření aplikací, takže jsme zvolili natolik jednoduchý příklad, že aplikace nereaguje žádné signály.

Všimněme si volání metody get_object. Ta umí pomocí jména widgetu (tak, jak jsme ho pojmenovali při vytváření aplikace v Glade) získat objekt, který tento widget reprezentuje. Zde jsme to použili pouze pro hlavní okno, ale jinak to je potřeba pro každý widget, se kterým budeme pracovat.

Příklad se signály

Vytváření aplikací v Glade je celkově velmi intuitivní. Zkusíme si jako ukázku vytvořit program, který již něco málo bude dělat. Bude obsahovat dvě škály (například widget Gtk2::VScale), které budou vzájemně propojené. Pohneme-li s jednou, změní se automaticky druhá tak, aby ukazovala druhou mocninu té první. Obě budou pro jednoduchost fungovat na intervalu [0,1], protože na tomto definičním oboru má funkce x^2 tentýž obor hodnot.

Naklikejme si tedy dle libosti vzhled aplikace. Měla by obsahovat dvě škály, pro každou popisek a můžeme přidat tlačítko pro ukončení aplikace. Ještě před přidáním widgetů je dobré si rozmyslet rozvržení. Buď můžeme použít mřížku nebo pár vhodně poskládaných VBoxů a HBoxů. Výsledek by mohl vypadat takto.



Naše navržená aplikace

Vzhled aplikace nakonec uložíme například do souboru pokus.glade a vytvoříme cílové XML.

gtk-builder-convert pokus.glade pokus.xml

Budeme určitě chtít manipulovat s hlavním oknem, oběmi škálami a tlačítkem. Tyto widgety si uložíme pomocí get_object. Poté již stačí jen odchytit pár signálů.

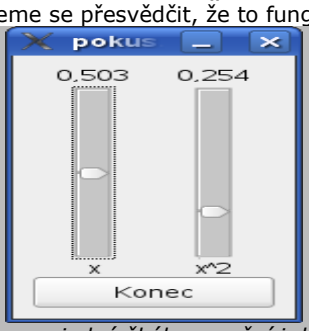
- Když se změní první škála, změníme druhou
- Když se změní druhá škála, změníme první
- Po kliknutí na tlačítko ukončíme aplikaci
- Aplikaci ukončíme též po zavření hlavního okna

Přepsáním do kódu získáme hotovou aplikaci.

```
use Gtk2 "-init";

$builder = Gtk2::Builder->new();
$builder->add_from_file("pokus.xml");
```

```
$okno = $builder->get_object("window1");

$skala1 = $builder->get_object("vscale1");
$skala2 = $builder->get_object("vscale2");
$konec = $builder->get_object("button1");

$konec->signal_connect("clicked" => sub{Gtk2->main_quit});

$skala1->signal_connect("change-value" => sub {$skala2->set_value($_[2]**2);});
$skala2->signal_connect("change-value" => sub {$skala1->set_value(sqrt($_[2]));});

$okno->signal_connect(delete_event => sub{Gtk2->main_quit});
$okno->show();
Gtk2->main();
```

Můžeme se přesvědčit, že to funguje.



Posunem jedné škály se mění i druhá

Perl (133) - Gtk2 - úpravy vzhledu aplikací pomocí rc

Odbočme trochu stranou a naučme se stanovovat defaultní hodnoty pro vzhled Gtk2 pomocí resource-file souborů. Díky tomu budeme moci v našich aplikacích využívat vlastní motivy Gtk2.

V adresáři /usr/share/themes nalezneme motivy pro vzhled Gtk2 aplikací. Zde se dají také upravovat implicitní hodnoty pro různá nastavení.

Zkusme stručně nahlédnout, jak témata vlastně fungují.

Témata jsou zapsané v tzv. rc souborech, které mají speciální syntaxi. Jsou uložené zpravidla v adresáři /home/uživatel/.themes/ nebo na veřejném místě v /usr/share/themes/, případně /usr/local/share/themes/.

Nastavení témat pro naší aplikaci

Pro každý widget nastavujeme téma zvlášť. Můžeme ale využít dědičnosti. Na gnome.org najdeme widgetovou hierarchii. Protože všechny widgety dědí od Gtk::Widget, lze snadno nastavit všechny widgety najednou.

Je několik možností, jak téma použít uvnitř naší aplikace. Nejjednodušší je napsat si vlastní rc soubor a uvnitř aplikace ho pomocí Gtk2::Rc nastavit. To se dělá následovně.

```
Gtk2::Rc->parse("/usr/share/themes/MojeTema/gtk-2.0/gtkrc");
```

Analogický je následující příkaz.

```
Gtk2::Rc->parse_string("include "/usr/share/themes/MojeTema/gtk-2.0/gtkrc"");
```

Parametr příkazu je v tomto případě již psaný v rc syntaxi, přičemž include je příkaz pro načtení rc souboru.

Díky parse_string můžeme vkládat rc syntaxi přímo do naší aplikace. Můžeme tedy psát příkazy následujícího typu.

```
Gtk2::Rc->parse_string(<<EOF);
include '/usr/share/themes/MojeTema/gtk-2.0/gtkrc'
style 'normal' {
font_name ='serif 30'
}
EOF
```

Vytváření témat

Podívejme se na syntaxi rc souborů. Chceme-li vytvořit téma s názvem MojeTema, pak se bude náš rc jmenovat /usr/share/themes/MojeTema/gtk-2.0/gtkrc.

Na začátku rc souboru může být série include příkazů, které se používají pro načítání nastavení z jiných souborů. Chceme-li téma logicky rozdělit do několika souborů (například soubor /usr/share/themes/MojeTema/gtk-2.0/tlacitka.rc může řešit vzhled tlačítek, soubor /usr/share/themes/MojeTema/gtk-2.0/dialogy.rc může řešit vzhled dialogů apod.), vložíme na začátek gtkrc následující řádky.

```
include "tlacitka.rc"
include "dialogy.rc"
include "ostatni.rc"
```

Dále definujeme takzvané styly. Obvykle vytvoříme jeden výchozí styl s názvem default a pak můžeme vytvořit několik dalších. Styly mohou ovlivňovat buď implicitní nastavení (například velikost rámečku atd.) nebo zde můžeme definovat tzv. enginy.

Styl se vytváří následovně.

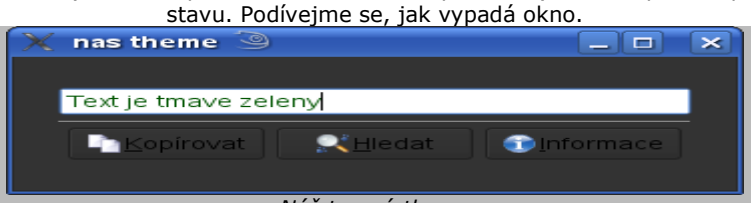
```
style "default" {
  #nastavení
}
```

Vytvořme si na úvod jednoduchý styl. Náš gtkrc soubor bude vypadat takto.

```
style "default" {
 xthickness = 1
 ythickness = 1

GtkButton ::child-displacement-x = 6
GtkButton ::child-displacement-y = 2
```

```
bg[NORMAL] = "#333"
fg[NORMAL] = "#f0f0f0"
text[NORMAL] = "#0ff"
}

class "GtkWidget" style "default"
```

Nastavili jsme pro ukázku velikost mezery mezi textem a hranicí widgetu (xthickness, ythickness; používá se na různých místech), hloubku stisku tlačítka (o kolik se posune text na tlačítku při stisku) a barvu pozadí, popředí a textu v normálním stavu. Podívejme se, jak vypadá okno.



Náš tmavý theme

Podívejme se na některá nastavení.

Nejprve si všimněme nastavování implicitních hodnot. My jsme nastavili hodnoty child-displacement-x a child-displacement-y pro GtkButton. Chceme-li nastavit nějakou implicitní hodnotu pro daný widget, je vhodné se podívat do dokumentace, kde je seznam všech možných vlastností, možné hodnoty a vysvětlení, co která hodnota znamená. Tento seznam lze nalézt na live.gnome.org. Jakmile nalezneme vhodnou vlastnost, stačí ji zapsat do stylu v rc souboru ve tvaru *JménoWidgetu*::*vlastnost* = *hodnota*. Poznamenejme jen, že jméno widgetu v rc souborech neobsahuje :: (rc soubory nemají s Perl syntaxí nic společného).

Barvy

Barvy jsou vlastnost, která nás asi bude zajímat nejvíce. Jsou čtyři kategorie barev.

- bg - barva pozadí
- fg - barva popředí
- text - barva textu
- base - barva pozadí pro některé speciální widgety (například TextView, TreeView atd.)

Dále je pět stavů, ve kterých se může widget nacházet.

- NORMAL
- PRELIGHT - přejezd myší
- ACTIVE - pro právě stisknutá tlačítka
- SELECTED - pro vybranou oblast (například označený text)
- INSENSITIVE - neaktivní widgety (například formuláře, jejichž hodnoty nelze měnit)

Barvy nastavujeme vždy pro kategorii a stav. To zapisujeme kategorie[stav]. Tedy konkrétně například bg[NORMAL].

Barvy můžeme zadávat mnoha způsoby.

- přes kód barvy ve tvaru #RGB, #RRGGBB, #RRRGGGBBB, #RRRRGGGGBBBB
- {R, G, B}, kde hodnoty jsou mezi 0 a 1
- symbolickými názvy
- vlastními symbolickými názvy (ty se ve stylu definují příkazem barva["název"] = "#RRGGBB")
- pomocí efektů na změnu barev:
  - lighter(barva)
  - darker(barva)
  - mix(typ, barva1, barva2)
  - shade(typ, barva)
- pomocí proměnných

Pokud definujeme barvu pomocí proměnných, pak bychom měli ještě před definici stylů přiřadit do gtk_color_scheme a nastavit tím tak barvy pro celé téma. Hodnota pro gtk_color_scheme je seznamem hodnot, které jsou oddělené znakem nového řádku. Hodnoty jsou tvaru *kategorie*:*barva*, tedy například bg_color:#000. Používáme-li Gnome, je vhodné dodržovat následujících 8 kategorií: fg_color, bg_color, base_color, text_color, selected_bg_color, selected_fg_color, tooltip_bg_color, tooltip_fg_color. Zde je příklad barevného schématu.

```
gtk_color_scheme = "fg_color:#010\nbg_color:#0f0\nbase_color:#fff\ntext_color:#f0f\n
selected_bg_color:#8ad\nselected_fg_color:#eee\ntooltip_bg_color:#ffb\ntooltip_fg_color:#000"
```

Barvy se potom zapisují jako @*kategorie*; například @bg_color.

Použití stylu

Již jsme si napsali jednoduchý styl. Gtk2 ale ještě neví, co s ním má dělat. Musíme nějak specifikovat, jaké styly se mají použít na jakých místech.

Příkazem class je možné asociovat styly jednotlivým prvkům a využít hierarchie widgetů. Podívejme se na následující řádky.

```
class "GtkButton" style "tlacitka"
class "GtkMenu" style "default"
class "GtkWidget" style "default"
```

Všechny widgety, které dědí od Gtk2::Button (například Gtk2::ToggleButton, Gtk2::VolumeButton, Gtk2::ColorButton) budou mít styl tlacitka. Všechny widgety dědící od Gtk2::Menu budou mít styl default. Pro všechny widgety, které nemají nastaven styl, jsme nastavili styl default.

Poznamnejme, že lze využít i žolíkových znaků.

Jiný příkaz pro aplikaci stylu je widget_class. Ten využívá vnoření widgetů do sebe. Máme-li v okně tlačítko a v něm nějaký text, jde o vnořené widgety. Vnoření popisku tlačítka v okně tak můžeme zapsat jako GtkWidget.GtkButton.GtkLabel.

Chceme-li tedy nastavit pro widgety uvnitř tlačítek typu Gtk2::Button styl uvnitr_tlacitek, zapíšeme to do souboru gtkrc tímto řádkem.

```
widget_class "*.GtkButton.*" style "uvnitr_tlacitek"
```

Kdybychom to chtěli aplikovat i na jiné typy tlačítek (tj. na všechny widgety, které od Gtk2::Button dědí), upravili bychom příkaz do nové podoby.

```
widget_class "*.<GtkButton>.*" style "uvnitr_tlacitek"
```

Ještě existuje příkaz widget, kterým nastavujeme vzhled pro konkrétní widget. Všechny widgety mohou mít svá jména, která nastavujeme příkazem set_name.

```
$widget->set_name("nebezpecne-tlacitko-spusteni-stepne-reakce");
```

Vzhled pak nastavíme již zmíněným příkazem widget.

```
widget "nebezpecne-tlacitko*" style "nebezpeci"
```

Enginy

Enginy umožňují použít uvnitř našeho stylu nějaký existující motiv. Pro inspiraci se můžeme podívat do /usr/share/themes/ (či podobného adresáře), kde máme motivy uložené.

```
style "nas-styl" {
engine "thinice" {
#naše úpravy
}
}
```

Perl (134) - Gtk2 - Glade Interface Designer

Na závěr série o Gtk2 se podíváme na vývojové prostředí Glade. Lze v něm snadno a bez řešení technických detailů navrhovat aplikace, což nám ušetří výrazné množství času.

Vytváření aplikace bylo až dosud obrovské množství psaní. Každý widget zabral několik řádků. Kdybychom takto měli tvořit něco komplexnějšího s tisícovkami widgetů, museli bychom vynaložit spoustu mechanické práce. Proto vznikají u všech rozšířenějších grafických knihoven takzvané interface designery nebo-li obecněji RAD.

Interface designer je prostředí, ve kterém si snadno naklikáme, jak má naše aplikace vypadat. Obsahuje obvykle seznam widgetů, které můžeme klikáním přidávat z nabídky do našeho projektu a umísťovat (malým nedostatkem Glade možná je, že to nelze dělat metodou drag&drop). Díky tomu se nadále nemusíme starat o tvorbu widgetů.

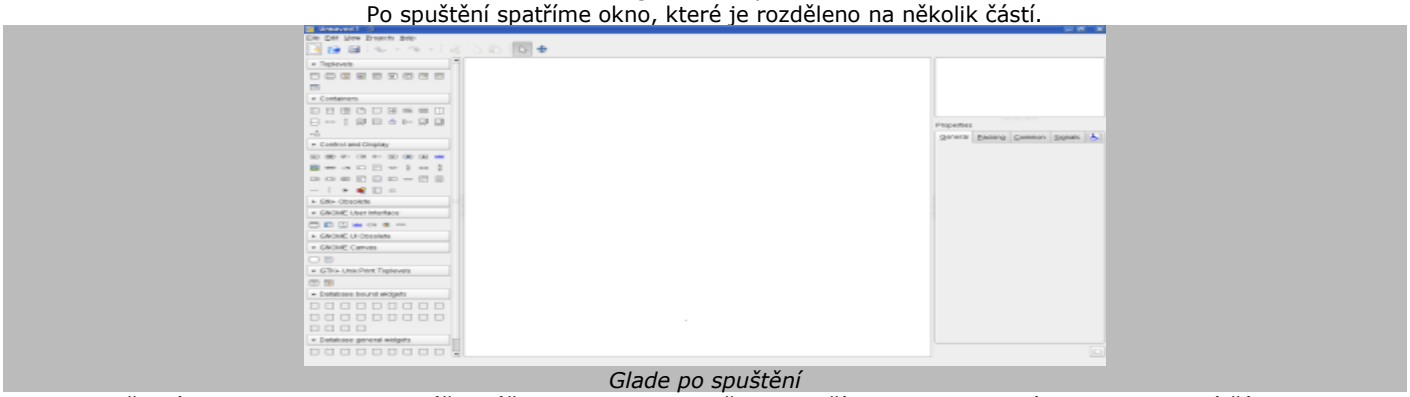
Interface designer převede widgety, které jsme si naklikali, do nějakého kódu - obvykle XML. Z něj pak lze vzezření aplikace rekonstruovat. Otázkou je, jak. Každý interface designer má svoji metodu, jak z XML dostat naši aplikaci. Obvykle k tomu budeme potřebovat nějaký program (takovým programům se někdy říka sketchery). Některé sketchery konvertují XML přímo do zdrojového kódu daného jazyka, některé vytvoří pouze vazebný kód a používají XML ke generování aplikace při každém spuštění. Pro každý jazyk je samozřejme potřeba vlastní sketcher. Výhodou je, že XML kód bývá univerzální a lze ho použít v různých jazycích. Později se budeme zabývat tím, jak to funguje u Glade. Nejprve si ukážeme, jak se s Glade pracuje.

Glade má již poměrně dlouhou tradici. První verze spatřila světlo světa již v roce 1998. Dnes je k dispozici poslední verze 3.10.0. Ve většině linuxových distribucí již existuje balíček glade nebo glade-3. Pro jiný operační systém je třeba stáhnout Glade ze stránek projektu a dle instrukcí nainstalovat.

Vytváření aplikací v Glade

Spusťmě tedy Glade. Obvykle se to dělá příkazem glade-2, glade-3, glade3 nebo glade. Měli bychom ho také najít v nabídce našeho grafického prostředí.

Po spuštění spatříme okno, které je rozděleno na několik částí.



Glade po spuštění

V prostředním panelu budeme vytvářet náš projekt. Budeme přitom používat widgety z levého panelu. Pravá část okna je rozdělena na dvě části. Nahoře je seznam widgetů podle toho, jak jsou do sebe vnořené. Dole budeme nastavovat widgetům vlastnosti.

Zbytek je již jen klikání. Ovládání je velmi intuitivní. Vše se zde dá snadno dohledat, protože Glade neobsahuje nic zbytečného navíc.

Na úvod je nejlepší proklikat si všechny widgety a podívat se, co všechno se dá dělat. Každý widget se pak dá smazat stiskem klávesy delete, označíme-li ho v hierarchii widgetů.

Vložme na úvod do projektu pouze hlavní okno a uložme ho například jako nic.glade. Pokud ho otevřeme, uvidíme následující XML kód.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<!--Generated with glade3 3.4.3 on Sun Dec 19 13:38:54 2010 -->
<glade-interface>
<widget class="GtkWindow" id="window1">
<child>
<placeholder/>
</child>
</widget>
</glade-interface>
```

Aplikace glade souborů

Podívejme se, jak lze z glade souborů vytvářet aplikace. Nejprve použijeme příkaz gtk-builder-convert, který fakticky pouze z jednoho XML souboru vytvoří jiný, který je čitelný modulem Gtk2::Builder. Proč tento mezikrok? První XML soubor je surovým

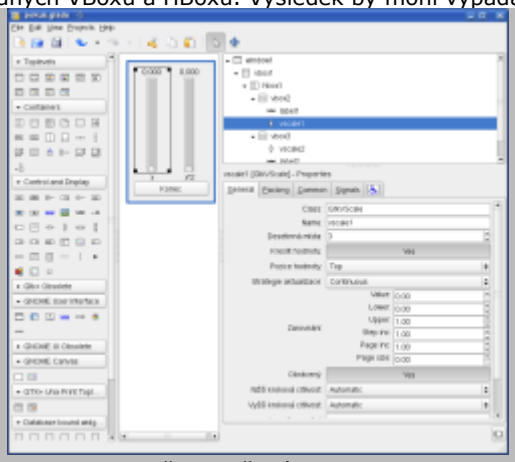
výstupem Glade. Skript toto XML upraví do podoby, ve které pro nás bude dále čitelný. Kdybychom například chtěli aplikovat původní glade v jiném jazyku než v Perlu, mohl by skript vypadat jinak.
Zkusíme tedy nejprve zkonvertovat náš soubor nic.glade, který kóduje pouze prázdné okno.
gtk-builder-convert nic.glade nic.xml
Nahlédneme-li do zdrojového kódu, uvidíme, že vypadá téměř stejně jako glade soubor. V novějších verzích bude možné generovat přímo toto XML bez mezikroku.

```
<?xml version="1.0"?>
<!--Generated with glade3 3.4.3 on Sun Sep 19 13:38:54 2010 -->
<interface>
<object class="GtkWindow" id="window1">
<child>
<placeholder/>
</child>
</object>
</interface>
```

Tento soubor ale již můžeme použít přímo ze zdrojového kódu naší aplikace. Avšak pamatujme na to, že pro změny v uživatelském rozhraní pomocí Glade budeme dále původní glade soubor potřebovat.
Je několik možností, jak z vzniklého XML vytvořit aplikaci. My využijeme modulu Gtk2::Builder. Podívejme se jak.

```
use Gtk2 "-init";

$builder = Gtk2::Builder->new();
$builder->add_from_file("nic.xml");

$okno = $builder->get_object("window1");

$okno->show();
Gtk2->main();
```

Tento kód již funguje. Zatím jsme pouze chtěli ukázat kostru postupu vytváření aplikací, takže jsme zvolili natolik jednoduchý příklad, že aplikace nereaguje žádné signály.
Všimněme si volání metody get_object. Ta umí pomocí jména widgetu (tak, jak jsme ho pojmenovali při vytváření aplikace v Glade) získat objekt, který tento widget reprezentuje. Zde jsme to použili pouze pro hlavní okno, ale jinak to je potřeba pro každý widget, se kterým budeme pracovat.
Příklad se signály
Vytváření aplikací v Glade je celkově velmi intuitivní. Zkusíme si jako ukázku vytvořit program, který již něco málo bude dělat. Bude obsahovat dvě škály (například widget Gtk2::VScale), které budou vzájemně propojené. Pohneme-li s jednou, změní se automaticky druhá tak, aby ukazovala druhou mocninu té první. Obě budou pro jednoduchost fungovat na intervalu [0,1], protože na tomto definičním oboru má funkce x^2 tentýž obor hodnot.
Naklikejme si tedy dle libosti vzhled aplikace. Měla by obsahovat dvě škály, pro každou popisek a můžeme přidat tlačítko pro ukončení aplikace. Ještě před přidáním widgetů je dobré si rozmyslet rozvržení. Buď můžeme použít mřížku nebo pár vhodně poskládaných VBoxů a HBoxů. Výsledek by mohl vypadat takto.



Naše navržená aplikace

Vzhled aplikace nakonec uložíme například do souboru pokus.glade a vytvoříme cílové XML.
gtk-builder-convert pokus.glade pokus.xml
Budeme určitě chtít manipulovat s hlavním oknem, oběmi škálami a tlačítkem. Tyto widgety si uložíme pomocí get_object. Poté již stačí jen odchytit pár signálů.

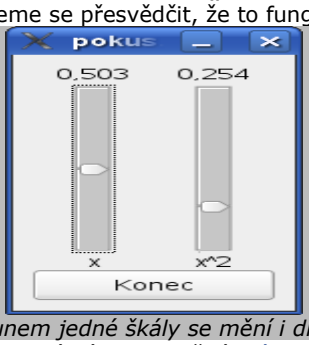
- Když se změní první škála, změníme druhou
- Když se změní druhá škála, změníme první
- Po kliknutí na tlačítko ukončíme aplikaci
- Aplikaci ukončíme též po zavření hlavního okna

Přepsáním do kódu získáme hotovou aplikaci.

```
use Gtk2 "-init";

$builder = Gtk2::Builder->new();
$builder->add_from_file("pokus.xml");

$okno = $builder->get_object("window1");

$skala1 = $builder->get_object("vscale1");
```

```
$skala2 = $builder->get_object("vscale2");
$konec = $builder->get_object("button1");

$konec->signal_connect("clicked" => sub{Gtk2->main_quit});

$skala1->signal_connect("change-value" => sub {$skala2->set_value($_[2]**2);});
$skala2->signal_connect("change-value" => sub {$skala1->set_value(sqrt($_[2]));});

$okno->signal_connect(delete_event => sub{Gtk2->main_quit});
$okno->show();
Gtk2->main();
```

Můžeme se přesvědčit, že to funguje.



Posunem jedné škály se mění i druhá

Zdrojové kódy je možné stáhnout.

Perl (135) - XML - čtení a zápis

XML je univerzálním jazykem pro přenos nejrůznějších dokumentů. Jak si s jeho parsováním poradí Perl?

Jazyk XML (Extensible Markup Language) netřeba dlouze představovat. Jde o značkovací jazyk, který se masivně používá pro přenos dokumentů. Příklad XML dokumentu je k vidění na Wikipedii.

Nyní budeme zkoumat nástroje, které dokáží v našich programech s XML pohodlně zacházet. Díky oblibě XML jich je k dispozici celá řada.

Základním nástrojem pro nejjednodušší operace s XML daty je modul XML::Simple z archivu modulů CPAN. XML::Simple umí jedinou věc - konvertuje XML data do datové struktury Perlu a naopak.

Hned na úvod poznamenejme, že XML::Simple má jeden zásadní nedostatek. Nelze mixovat obsah a elementy uvnitř jednoho elementu. S tím si XML::Simple neporadí a je třeba použít něco silnějšího.

Další potenciální slabinou XML::Simple je, že je náročný na paměť a u extrémně velkých souborů to může působit problémy. S tím souvisí i to, že nelze zpracovávat data dávkově. K tomu slouží specializované moduly, které pracují s nástrojem zvaným SAX.

Modul XML::Simple obsahuje dvě hlavní metody. XMLin pro načtení XML souboru a naopak XMLout pro export do XML.

Čtení XML dat

Budeme na chvíli pracovat s následujícím souborem.

```
<?xml version="1.0" encoding="UTF-8"?>
<katalog>
<kniha id="1">
<nazev>Stopařův průvodce Galaxií 1.</nazev>
<autor>Adams Douglas</autor>
<foto src="stoparuv.jpg" alt="Titulní strana - Stopařův průvodce"/>
<popis>
Nezapomeňte: <citace>ručník je dost možná tou nejužitečnější věcí ve vesmíru</citace>.
</popis>
<cena>319</cena>
</kniha>
<kniha id="2">
<nazev>Alchymista</nazev>
<autor>Coelho Paulo</autor>
<foto src="alchymista.jpg" alt="Titulní strana - Alchymista"/>
<popis>
Cesta za pokladem.
</popis>
<cena>399.90</cena>
</kniha>
</katalog>
```

Podívejme se na úvod, jak tedy načteme XML soubor katalog.xml.

```
use XML::Simple;
use Data::Dumper;
$katalog = XMLin("katalog.xml");
print Dumper $katalog;
```

Zároveň jsme díky modulu Data::Dumper vytiskli strukturu právě vytvořené proměnné $katalog.

```
$VAR1 = {
'kniha' => {
'1' => {
'foto' => {
'alt' => "Titulní strana - Stopařův průvodce",
'src' => 'stoparuv.jpg'
},
```

```
'autor' => 'Adams Douglas',
'popis' => {
'citace' => "ručník je dost možná tou nejužitečnější věcí ve vesmíru",
'content' => [
"
Nezapomeňte: ",
'.
'
]
},
'nazev' => "Stopařův průvodce Galaxií 1.",
'cena' => '319'
},
'2' => {
'foto' => {
'alt' => "Titulní strana - Alchymista",
'src' => 'alchymista.jpg'
},
'autor' => 'Coelho Paulo',
'popis' => '
Cesta za pokladem.
',
'nazev' => 'Alchymista',
'cena' => '399.90'
}
}
};
```

Jako parametr XMLin jsme uvedli jméno souboru. Kdybychom použili metodu bez argumentů, hledal by se soubor stejného názvu jako skript (přesněji __FILE__) s tím, že by se přípona pozměnila na .xml. Další možností je vložit přímo řetězec ve formátu XML.

Modul funguje i s objektovým přístupem.

```
XML::Simple->new()->XMLin("neco.xml")
```

Uveďme zde ještě pár poznámek, jak se data ukládají. Především, ukládání do datové struktury je ztrátové, ale není to nic kritického. Na několika příkladech si nyní vysvětlíme, jak konverze probíhá. Podle nich uvidíme, kde dochází k nejednoznačnostem, které se potom pokusíme odstranit. Ne u všech to půjde.

Uvnitř elementu může být buď text nebo další struktura elementů. Je-li tam text, pak se též objeví jako příslušná hodnota hashe. V opačném případě bude touto hodnotou nějaká další anonymní datová struktura.

Pokud máme XML opakované elementy, pak jsou položky (knihy) reprezentovány jako prvky anonymního pole.

```
<kniha>...</kniha>
<kniha>...</kniha>
...
```

Pokud je rozlišíme parametrem s názvem id (tak jsme to udělali v případě knih i v úvodním příkladu), pak se ukládají jako hash tvaru {1=>{...}, 2=>{...}, ...}.

```
<kniha id=1>...</kniha>
<kniha id=2>...</kniha>
...
```

Dodejme ještě, že úplně stejně by se do datové struktury uložilo následující.

```
<kniha><id>1</id>...</kniha>
<kniha><id>2</id>...</kniha>
...
```

Nabízí se otázka: Jde celé toto "chytré" chování nějak obejít? Odpověď je: Ano, stačí data načítat s parametrem ForceArray => 1. Výsledná datová struktura je složitější a strukturou více popisuje vstupní data.

```
XMLin("ukazka.xml", ForceArray => 1);
```

Opět zde narážíme na nedostatky XML::Simple. Vždy jde o to, co vlastně chceme. Pokud jen chceme získat několik jednotlivých dat a nejednoznačnosti nejsou podstatné, bude se lépe pracovat s původní verzí (avšak zde poznamenejme, že robusnější řešení nabízí xPath). Pokud chceme XML upravit a uložit, bude lepší využít ForceArray, neboť výsledné XML by se pak mohlo hodně změnit (to uvidíme podrobněji dále).

Dalších parametrů je celá řada. Pro další informace se odkažme na dokumentaci.

Ještě si všimněme, jak se element <citace></citace> objevil ve výsledné datové struktuře. Již jsme na to narazili v úvodu. Pokud se uvnitř elementu objevuje text (což je zde fyzicky obsah elementu <content></content>) a další elementy, pak XML::Simple vůbec neumí rozlišit jejich pořadí. Měli bychom tedy použít nějaký jiný nástroj.

Práce s jednotlivými daty a xPath

Samozřejmě můžeme pomocí pravidel pro datové struktury tisknout i konkrétní skaláry. Například následující volání má za následek vytisknutí autora 1. knihy v seznamu.

```
print $katalog->{kniha}->{1}->{autor};
```

Pravdou však je, že to není příliš pohodlné. Je ale třeba si uvědomit, že pro tisk jakýchkoliv dat je uvedení cesty k nim nezbytné. Přirozenou otázkou je, zda neexistuje nějakým způsobem standardizovaná cesta k jednotlivým datům ve formě řetězce. Ano, samozřejmě již existuje a daný jazyk se nazývá xPath.

Nebudeme se jím podrobně zabývat, pouze si uvedeme příklad. Na CPANu je řada modulů, které xPath podporují. Uveďme například XML::XPath::Simple. Autora první knihy zde získáme takto.

```
use XML::XPath::Simple;
$xpath = new XML::XPath::Simple(xml => "ukazka.xml");
print $xpath->valueof("/katalog/kniha[1]/autor");
```

Pro zájemce o xPath je k dispozici například tutoriál na w3schools.com .

Zápis XML dat

Nyní bychom samozřejmě mohli s touto strukturou pracovat. To zatím ponechme stranou a podívejme se, jak z datové struktury vytvoříme XML.

O výstup do XML dat se stará Metoda XMLout. Vytiskněme na úvod naši datovou strukturu $katalog na standardní výstup ve formátu XML.

```
print XMLout($katalog);
```

Výstup nás možná překvapí, protože náš původní XML soubor vypadal trochu jinak. Důvod jsme již uvedli. Struktura dat nicméně zůstává stejná.

```
<opt>
<kniha name="1" autor="Adams Douglas" cena="319" nazev="Stopařův průvodce Galaxií 1.">
<foto alt="Titulní strana - Stopařův průvodce" src="stoparuv.jpg" />
<popis citace="ručník je dost možná tou nejužitečnější věcí ve vesmíru">
<content>
Nezapomeňte: </content>
<content>.
</content>
</popis>
</kniha>
<kniha name="2" autor="Coelho Paulo" cena="399.90" nazev="Alchymista" popis="
Cesta za pokladem.
">
<foto alt="Titulní strana - Alchymista" src="alchymista.jpg" />
</kniha>
</opt>
```

Předně, velmi záleží na tom, zda jsme načetli XML soubor s parametrem ForceArray. Předcházející soubor nám vznikl bez ForceArray. Jeho nastavením získáme lehce změněný výstup, který je podobnější úvodní verzi.

```
<opt>
<kniha name="1">
<autor>Adams Douglas</autor>
<cena>319</cena>
<foto alt="Titulní strana - Stopařův průvodce" src="stoparuv.jpg" />
<nazev>Stopařův průvodce Galaxií 1.</nazev>
<popis>
<citace>ručník je dost možná tou nejužitečnější věcí ve vesmíru</citace>
<content>
Nezapomeňte: </content>
<content>.
</content>
</popis>
</kniha>
<kniha name="2">
<autor>Coelho Paulo</autor>
<cena>399.90</cena>
<foto alt="Titulní strana - Alchymista" src="alchymista.jpg" />
<nazev>Alchymista</nazev>
<popis>
Cesta za pokladem.
</popis>
</kniha>
</opt>
```

Nastavením dodatečných parametrů pro metodu XMLout můžeme docílit ještě dalších změn.

Parametrem KeepRoot => 0 se zbavíme obalujícího elementu <opt></opt>.

Také se nám cestou ztratila úvodní deklarace z XML souboru. Můžeme ji doplnit parametrem XMLDecl => "<?xml version='1.0'?>".

Abychom nemuseli vrácený řetězec složitě zapisovat do souboru, lze užít také parametr OutputFile => 'novy_katalog.xml', který vytiskne výsledek do uvedeného souboru na místo standardního výstupu. Výsledné volání nyní vypadá takto.

```
XML::Simple::XMLout($katalog,
                    KeepRoot   => 0,
                    XMLDecl    => "<?xml version='1.0'?>",
                    OutputFile => "new.xml",
                    );
```

Perl (136) - XML - DOM a SAX přístupy

Naťukněme stručně několik témat souvisejících s XML. Bude to jen úvod, ze kterého by mělo vycházet další studium.

Použití XML::LibXML - reprezentace pomocí DOM

Představme si velmi stručně ještě jeden z nejlepších modulů pro práci s XML, kterým je XML::LibXML od českého autora Petra Pajase. Umí parsovat XML soubory jako DOM. DOM je aplikační rozhraní pro čtení nebo zápis obsahu a struktury XML dokumentů.

Opět budeme pracovat se souborem ukazka.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<katalog>
<kniha id="1">
<nazev>Stopařův průvodce Galaxií 1.</nazev>
<autor>Adams Douglas</autor>
<foto src="stoparuv.jpg" alt="Titulní strana - Stopařův průvodce"/>
```

<popis>
Nezapomeňte: <citace>ručník je dost možná tou nejužitečnější věcí ve vesmíru</citace>.
</popis>
<cena>319</cena>
</kniha>
<kniha id="2">
<nazev>Alchymista</nazev>
<autor>Coelho Paulo</autor>
<foto src="alchymista.jpg" alt="Titulní strana - Alchymista"/>
<popis>
Cesta za pokladem.
</popis>
<cena>399.90</cena>
</kniha>
</katalog>
Následující kód parsuje soubor ukazka.xml a načte data do objektu.

```
use XML::LibXML;

$parser = XML::LibXML->new;
$d = $parser->parse_file("ukazka.xml");
```

Pro opětovný výpis dat ve formátu XML nyní budeme volat metodu toString.

```
print $d->toString(1);
```

Velmi stručně se podívejme, jak přidávat nové elementy. Budeme k tomu potřebovat odkaz na kořenový element.

```
$root = $doc->documentElement();
```

Přidejme nejprve další knihu.

```
$kniha = $root->addNewChild("", "kniha");
```

Poté dovnitř můžeme přidat elementy pro autora a název.

```
$autor = $kniha->addNewChild("", "autor");
$nazev = $kniha->addNewChild("", "nazev");
```

Do těchto elementů již můžeme vepsat data.

```
$autor->addChild($doc->createTextNode("Antoine de Saint-Exupéry"));
$nazev->addChild($doc->createTextNode("Malý princ"));
```

Pro další podobné metody lze nahlédnout do dokumentace.
Modul XML::LibXML je velmi komplexní a umí obrovské množství věcí. Pro zájemce je k dispozici dokumentace.
SAX - postupné zpracování XML dokumentu
Ještě se stručně podívejme, jak lze pracovat s nástrojem SAX. Ten nabízí trochu odlišný přístup na XML dokument. SAX chápe XML jako tok dat a událostí. Každý element vyvolá příslušnou událost a na základě nich postupně můžeme zpracovávat data. XML::LibXML má SAX rozhraní, avšak je pouze simulované, protože data jsou tak jako tak načteny do paměti. Je třeba se poohlédnout zase po něčem novém. Podívejme se blíže na XML::SAX::ExpatXS.

```
use XML::SAX::ParserFactory;
use XML::SAX::Writer;

$XML::SAX::ParserPackage = "XML::SAX::ExpatXS";
$parser = XML::SAX::ParserFactory->parser(Handler => XML::SAX::Writer->new);
$parser->parse_file("ukazka.xml") ;
```

Na programátorovi je určit ke každé události, jak se na ni má reagovat. My jsme v příkladu použili standardní XML::SAX::Writer, který vypíše na výstup XML.
Rozšiřováním XML::SAX::Base pomocí navrhování handlerů můžeme definovat vlastní reagování na události. Funguje to tak, že přetížíme některé z metod, které jsou uvedené v dokumentaci. Zkusme přetížit metodu start_element, která je volána vždy poté, kdy narazíme na první z páru elementů. Můžeme pomocí SUPER zdědit chování nadřízené třídy nebo nějak jinak manipulovat s daty.

```
package MujHandler;
use Data::Dumper;
use base qw(XML::SAX::Base);

sub start_element {
my ($self, $data) = @_;
$self->SUPER::start_element($data);
}

#další metody

1;
```

Stojí za to se podívat, co vlastně v proměnné $data máme.

```
print Dumper $data;
```

Podívejme se orientačně, jak vypadá výstup pro elementy autor a foto.

```
$VAR1 = {
'LocalName' => 'autor',
'Prefix' => '',
'Attributes' => {},
'Name' => ${\$VAR1->{'LocalName'}},
'NamespaceURI' => ${\$VAR1->{'Prefix'}}
};
$VAR1 = {
'LocalName' => 'foto',
'Prefix' => '',
```

```
'Attributes' => {
'{}alt' => {
'LocalName' => 'alt',
'Prefix' => ${\$VAR1->{'Prefix'}},
'Value' => "Tituln\x{ed} strana - Stopa\x{159}\x{16f}v pr\x{16f}vodce",
'Name' => ${\$VAR1->{'Attributes'}{'{}alt'}{'LocalName'}},
'NamespaceURI' => ${\$VAR1->{'Prefix'}}
},
'{}src' => {
'LocalName' => 'src',
'Prefix' => ${\$VAR1->{'Prefix'}},
'Value' => 'stoparuv.jpg',
'Name' => ${\$VAR1->{'Attributes'}{'{}src'}{'LocalName'}},
'NamespaceURI' => ${\$VAR1->{'Prefix'}}
}
},
'Name' => ${\$VAR1->{'LocalName'}},
'NamespaceURI' => ${\$VAR1->{'Prefix'}}
};
```

Abychom si také ukázali nějaký příklad vlastního handleru, zkusíme vypsat stromový seznam všech otevíracích a uzavíracích elementů.

Jak tedy bude vypadat náš handler? Vytvoříme dvě metody - jednu pro otevírací elementy a druhou pro uzavírací. Z proměnné $data získáme snadno název elementu. Také budeme uchovávat hloubku zanoření.

```
package MujHandler;
use Data::Dumper;
use base qw(XML::SAX::Base);

our $zanoreni = 0;

sub start_element {
my ($self, $data) = @_;
print " " x $zanoreni . $$data{"LocalName"}." - zacatek elementu\n";
$zanoreni++;
}

sub end_element {
my ($self, $data) = @_;
$zanoreni--;
print " " x $zanoreni . $$data{"LocalName"}." - konec elementu\n";
}

1;
```

Nyní tento handler použijeme. V úvodním programu nahradíme XML::SAX::Writer za náš MujHandler.

```
use XML::SAX::ParserFactory;
use MujHandler;

my $handler = MujHandler->new();
$XML::SAX::ParserPackage = "XML::SAX::ExpatXS";
my $parser = XML::SAX::ParserFactory->parser(Handler => $handler);

$parser->parse_uri("ukazka.xml") ;
```

A výsledek?

```
katalog - zacatek elementu
kniha - zacatek elementu
id - zacatek elementu
id - konec elementu
nazev - zacatek elementu
nazev - konec elementu
autor - zacatek elementu
autor - konec elementu
foto - zacatek elementu
foto - konec elementu
popis - zacatek elementu
citace - zacatek elementu
citace - konec elementu
popis - konec elementu
cena - zacatek elementu
cena - konec elementu
kniha - konec elementu
kniha - zacatek elementu
id - zacatek elementu
id - konec elementu
nazev - zacatek elementu
nazev - konec elementu
autor - zacatek elementu
autor - konec elementu
```

foto - zacatek elementu
foto - konec elementu
popis - zacatek elementu
popis - konec elementu
cena - zacatek elementu
cena - konec elementu
kniha - konec elementu
katalog - konec elementu
Nyní si lze snadno představit, jak přizpůsobivý nástroj máme.

Vlákna zajišťují souběžné provádění několika různých podprogramů v rámci jednoho procesu.

Proces lze rozdělit na části, kterým se říká vlákna. Můžeme si je představit jako několik programů, které běží zároveň. Každé vlákno je jeden program. Přitom si mohou předávat (resp. sdílet) data.
Vlákna v Perlu jsou dostupné uvedením

```
use threads;
```

Jak to funguje
Již máme představu o tom, co to vlákna jsou. Pojďmě se podívat na základní kroky, které musíme v programu využívajícím vlákna udělat.
Nejprve je třeba vytvořit jedno nebo několik vláken. Na to máme konstruktor create nebo new. Nejdůležitějším parametrem konstruktoru je samozřejmě zadání úkolu, který má vlákno udělat. To se dělá ve formě anonymního či pojmenovaného podprogramu nebo i předáním jeho jména. Další parametry se předají tomuto podprogramu.
Nyní můžeme dělat cokoliv. Naše vlákno si teď samo paralelně běží a nemusíme se o něj starat. Klidně můžeme vytvořit další vlákna. Jakmile budeme chtít získat výsledky vlákna, zavoláme nad ním metodu join. join počká na to, až vlákno skončí (pokud se tak již nestalo) a předá návratovou hodnotu.
Příklad
Pojďme se podívat na konkrétní kód.

```
#!/usr/bin/env perl
use strict;
use warnings;

use threads;

use IO::Handle;
STDOUT->autoflush(1);

print "Začátek\n";

my $vlakno1 = threads->new(\&ukol, 1);
sleep 1;
my $vlakno2 = threads->new(\&ukol, 2);

my $return1 = $vlakno1->join;
print "Vlákno 1 vrátilo $return1\n";

my $return2 = $vlakno2->join;
print "Vlákno 2 vrátilo $return2\n";

print "Konec\n";

sub ukol {
my $n = shift;
print "$n: Vlákno spuštěno\n";
sleep 2;
print "$n: Jsem uprostřed\n";
sleep 2;
print "$n: Končím\n";
return $n;
}
```

Co se tam děje? Pojďme se na to podívat.
Za prvé používáme IO::Handle kvůli autoflush pro vyprázdnění výstupního bufferu, což nás ale v tuto chvíli nezajímá, protože to s vlákny nemá nic společného. Kdybychom to neudělali, měli bychom problém se střídáním výstupu na STDOUT a funkce sleep.
Soustřeďme se nyní na to podstatné. Projděme si schematicky běh programu sekundu po sekundě.

Kolikátá sekunda běží	Co se děje
0	Spustíme program
	Vytiskne se "Začátek\n"
	Vytvoří se vlákno 1
	Tím pádem se vytiskne (už ve vlákně 1) text "1: Vlákno spuštěno\n"
1	Vytvoří se vlákno 2
	Vytiskne se (ve vlákně 2) text "1: Vlákno spuštěno\n" ...
	... zatímco v hlavním programu zavoláme join a čekáme, až nám vlákno 1 vrátí výsledek

2	Vlákno 1 tiskne "1: Jsem uprostřed\n"
	Hlavní program stále čeká
3	Vlákno 2 tiskne "2: Jsem uprostřed\n"
	Hlavní program stále čeká
4	Vlákno 1 tiskne "1: Končím\n"
	Vlákno 1 končí a vrací 1
	Na to reaguje hlavní program - join mu konečně vrátila výsledek.
	Hlavní program neprodleně tiskne "Vlákno 1 vrátilo 1\n"
	Hlavní program se posunuje na druhé join a čeká na ukončení druhého vlákna
5	Vlákno 2 tiskne "2: Končím\n"
	Hlavní program opět získává návratovou hodnotu a tiskne "Vlákno 2 vrátilo 2\n"
	Hlavní program tiskne "Konec\n" a program končí

Jen stručně si přibližme další metody. Voláním threads->yield uvnitř vlákna dáme najevo, že toto vlákno je nejdůležitější a že má dostat největší CPU čas. Co to přesně znamená však není nikde specifikováno.
Metoda is_running nám řekne, zda ještě vlákno běží.
Vlákna jsou přetížené vzhledem k operaci ==, takže je tímto operátorem můžeme rozlišovat.
Metoda detach odtrhne vlákno. To znamená hlavně to, že nebude možné získat jeho návratovou hodnotu pomocí join.
Detachovat se může i samo vlákno - tak lze učinit voláním threads->detach nebo oklikou pomocí získání vlastního objektu. To lze udělat nejlépe pomocí threads->self nebo, známe-li id objektu, threads->object($tid). Dodejme, že metodami is_joinable, is_detached lze detekovat aktuální stav.
Každému vláknu se automaticky přiřadí id, které lze získat metodou tid. Vlákna se číslují přirozenými čísly počínaje 1. Zajímavé je také to, že když zavedeme threadss parametrem stringify,

```
use threads qw(stringify);
```

bude objekt vlákna přetížen při použití uvnitř řetězce a bude se tak chovat jako proměnná uchovávající id vlákna.
Sdílení dat
Budeme pracovat se zavedeným threads::shared.

```
use threads;
use threads::shared;
```

Nyní můžeme proměnné označovat atributem :shared. Nejprve se ale podívejme na následující kód.

```
use threads;
my $data = 1;
threads->create(sub {$data = 2})->join();
print $data;
```

Co se vytiskne? 1 nebo 2? Vytiskne se 1, protože data mezi vlákny nejsou sdílena.
Co když ale sdílet chceme? Pak označme proměnnou atributem shared.

```
use threads;
use threads::shared;
my $data :shared = 1;
threads->create(sub {$data = 2})->join();
print $data;
```

Nyní program tiskne 2.
Jaký je rozdíl mezi vlákny a forkem?
Nelze v tomto dílu nevzpomenout na fork. Mohlo by se zdát, že jde o totéž jako vlákna. Pravdou je, že ve většině aplikací lze tyto přístupy zaměnit. Je zde ale několik rozdílů.
Obecně platí, že při forku získáme méně závislé běhy programu než při použití vláken. V případě forku jsou rodič a potomek dva nezávislé procesy, které mají různé id procesu a nesdílejí systémové prostředky. Rodič i potomek mají vlastní adresní prostor. Komunikace mezi nimi je podstatně náročnější než u vláken, která paměť sdílejí.
Vlákna mají stejné id procesu a řadu dalších společných věcí - některá data, ovladače, signály atd. Nejzásadnější věc, kterou nesdílejí, je CPU, která jim je přidělována nezávisle. Vlákna celkově potřebují méně režie. Nastartování vlákna je podstatně rychlejší než nastartování forku (pokud ovšem nejde o 'lazy' implementaci forku, při které jsou data zkopírována až před prvním čtením nebo zápisem, což nemusí někdy ani nastat).
Ovšem se sdílenými prostředky se pojí také řada nebezpečí. Například změna hodnoty proměnné se promítne v obou vláknech. Pokud jedno vlákno náhle selže, má to vliv i na všechna ostatní. Psaní vícevláknového programu je díky těmto vlastnostem oproti forku podstatně méně čitelné.
Z důvodu průhlednosti se vyplatí dát přednost forku. Na vlákna se přechází buď nejsme-li spokojeni s rychlostními testy nebo když potřebujeme sdílet data.
Perl (138) - Memoizace - cachování podprogramů

Zejména u projektů zpracovávajících velká množství dat je již ve fázi implementace nezbytné přemýšlet o optimalizaci jednotlivých programových úseků. Jedním z efektivních nástrojů s jednoduchou myšlekou je memoizace.

Memoizace je optimalizační technika, která zefektivňuje chod vhodných úseků programu. Používá se u podprogramů, které jsou často spouštěny se stejnými vstupními daty.
Princip memoizace je v tom, že po zavolání podprogramu nejprve zkontrolujeme, zda jsme ho již se stejnými vstupními daty nespouštěli. Když zjistíme, že ne, spustíme ho, vrátíme výsledek a navíc si ho někde ponecháme uložený. Když ale zjistíme, že totožný výpočet již běžel, volání podprogramu přeskočíme a vrátíme uložený výsledek.
To však neznamená, že memoizace je něco, co bychom mohli bezhlavě používat všude. Memoizace je obchod - měníme rychlost programu za paměťové nároky. Cenu určuje charakter podprogramu - čím více je možností vstupu, tím je rychlost dražší.

Co memoizovat

Typičtí kandidáti na memoizaci často bývají rekurzivní funkce, které se rozběhnou vždy stejným způsobem. Ve většině případů je rekurzivní varianta výpočtu sice intuitivnější, avšak méně efektivní. Zefektivnění lze vždy provést přepsáním algoritmu s rekurzí na algoritmus bez ní (což lze mimochodem udělat algoritmicky pro libovolnou rekurzi).

Může nastat i extrémní případ, kdy rekurze způsobuje exponenciální složitost u programu, který by mohl být třeba i lineární. Ukážeme si jeden takový příklad a pak zkusíme cachovat.

Ještě před tím uveďme, že kandidáti pro memoizaci pochází často také z nerekurzivních funkcí. Bez kontextu sice nelze skoro nikdy říci, zda jde o vhodné kandidáty, ale uveďme jen pro představu pár příkladů, kde to většinou výhodné bude. Každý jistě vymyslí mnoho dalších.

- matematické výpočty - různé posloupnosti (zejména rekurzivně zadané), náročné operace pro malý definiční obor, konkrétně např. faktoriál
- mirrorování webu, nebudeme opakovaně stahovat stejnou URL
- různé konverze na malém definičním oboru

Příklad typického neefektivního podprogramu - Fibonacciho posloupnost

Fibonacciho posloupnost je definovaná jako 1 pro nultý a první člen (někdy 0 a 1) a pro ostatní jako součet dvou předchozích. Začíná tedy čísly 1 1 2 3 5 8 13 21 34 55.

Podívejme se na následující triviální podprogram, který pro dané *n* spočítá *n*tý člen posloupnosti.

```
sub fibonacci {
my $n = shift;
return 1 if ($n <= 1);
return fibonacci($n - 1) + fibonacci($n - 2);
}
```

Zkusme si teď spočítat prvních 36 členů posloupnosti a stopnout, jak dlouho to bude trvat.

```
use Time::HiRes qw(time);

for(0 .. 35){
my $t= time();
my $f = fibonacci($_);
print sprintf("f(%2d) = %8d, doba: %f\n", $_, $f, time() - $t);
}
```

Zajímá-li vás výstup, zde je:

```
f( 0) =        1, doba: 0.000024
f( 1) =        1, doba: 0.000007
f( 2) =        2, doba: 0.000013
f( 3) =        3, doba: 0.000012
f( 4) =        5, doba: 0.000018
f( 5) =        8, doba: 0.000022
f( 6) =       13, doba: 0.000032
f( 7) =       21, doba: 0.000047
f( 8) =       34, doba: 0.000073
f( 9) =       55, doba: 0.000116
f(10) =       89, doba: 0.000190
f(11) =      144, doba: 0.000471
f(12) =      233, doba: 0.000483
f(13) =      377, doba: 0.000909
f(14) =      610, doba: 0.001499
f(15) =      987, doba: 0.002130
f(16) =     1597, doba: 0.003329
f(17) =     2584, doba: 0.005423
f(18) =     4181, doba: 0.008336
f(19) =     6765, doba: 0.016447
f(20) =    10946, doba: 0.027183
f(21) =    17711, doba: 0.039360
f(22) =    28657, doba: 0.058051
f(23) =    46368, doba: 0.098867
f(24) =    75025, doba: 0.150952
f(25) =   121393, doba: 0.241542
f(26) =   196418, doba: 0.398880
f(27) =   317811, doba: 0.637901
f(28) =   514229, doba: 0.996751
f(29) =   832040, doba: 1.626886
f(30) =  1346269, doba: 2.650302
f(31) =  2178309, doba: 4.216838
f(32) =  3524578, doba: 6.935077
f(33) =  5702887, doba: 11.239907
f(34) =  9227465, doba: 18.135487
f(35) = 14930352, doba: 29.593933
```

Výpočet pro 35 již trvá půl minuty. Jak je to možné? Vždyť v celém programu opakovaně provádíme pouze sčítání dvou malých čísel!

Všimněme si posloupnosti dob výpočtů vpravo. Přibližně platí, že každá hodnota je součtem dvou předcházejících. To naznačuje, že v *n*tém kroku vždy provádíme tolik operací jako v dvou předcházejících dohromady.

Pro f(0) a f(1) pouze vrátíme výsledek. Řekněme, že potřebujeme jednu operaci. Pro f(2) voláme f(0) a f(1) a ty sčítáme, dohromady 3 operace. Pro f(3) voláme f(1) a f(2) a opět sčítáme. To je 5 operací. Posloupnost počtu výpočtů je 1 1 3 5 9 15 25 41 67 109 177 287 465 753 1219 1973 3193 5167 8361 13529 21897 atd. Půjdeme-li na konec, dostaneme pro f(35) 29 860 703 volání naší funkce. Na to, že jde o velmi snadný výpočet je to trochu moc. Jak tomu zamezit?

Jak funguje memoizace

Koho zajímá jen výsledný efekt, může bez obav přeskočit na sekci o modulu Memoize.

Pojďme zkusit upravit náš podprogram fibonacci tak, aby pracoval rychleji. K tomu účelu si vytvoříme nějakou cache ve formě hashe nebo pole, tj. globální proměnnou nebo proměnnou uzavřenou do bloku společně s podprogramem.

Nám bude stačit obyčejné pole. Na začátku podprogramu se podíváme, zda jsme již náhodou stejné volání neprováděli. Pokud ano, přeskočíme celý velký strom výpočtů a okamžitě vrátíme výsledek. Pokud ne, výsledek spočítáme jako dříve a uložíme ho do cache.

```
{
my @vypocitane;

sub fibonacci {
my $n = shift;

return $vypocitane[$n] if defined $vypocitane[$n];

my $vysledek;

if ($n <= 1) {
$vysledek = 1;
}
else {
$vysledek = fibonacci($n - 1) + fibonacci($n - 2);
}

return $vypocitane[$n] = $vysledek;
}
}
```

Jaká je zde posloupnost počtu výpočtů? 1 1 3 3 3 3 3 3 3 atd. Je velký rozdíl dělat 30 000 000 nebo 3 výpočty. Proto dostaneme výsledky hned.

```
f( 0) =       1, doba: 0.000026
f( 1) =       1, doba: 0.000007
f( 2) =       2, doba: 0.000016
f( 3) =       3, doba: 0.000009
f( 4) =       5, doba: 0.000010
f( 5) =       8, doba: 0.000009
f( 6) =      13, doba: 0.000009
f( 7) =      21, doba: 0.000009
f( 8) =      34, doba: 0.000009
f( 9) =      55, doba: 0.000009
f(10) =      89, doba: 0.000009
f(11) =     144, doba: 0.000009
f(12) =     233, doba: 0.000010
f(13) =     377, doba: 0.000009
f(14) =     610, doba: 0.000009
f(15) =     987, doba: 0.000009
f(16) =    1597, doba: 0.000009
f(17) =    2584, doba: 0.000009
f(18) =    4181, doba: 0.000009
f(19) =    6765, doba: 0.000009
f(20) =   10946, doba: 0.000009
f(21) =   17711, doba: 0.000009
f(22) =   28657, doba: 0.000014
f(23) =   46368, doba: 0.000009
f(24) =   75025, doba: 0.000008
f(25) =  121393, doba: 0.000009
f(26) =  196418, doba: 0.000009
f(27) =  317811, doba: 0.000009
f(28) =  514229, doba: 0.000010
f(29) =  832040, doba: 0.000009
f(30) = 1346269, doba: 0.000009
f(31) = 2178309, doba: 0.000009
f(32) = 3524578, doba: 0.000009
f(33) = 5702887, doba: 0.000008
f(34) = 9227465, doba: 0.000008
f(35) = 14930352, doba: 0.000008
```

Modul Memoize

Abychom nemuseli dělat tento proces pokaždé, napsal Damian Conway modul Memoize, který ho provede za nás.

```
use Memoize;
memoize("fibonacci");

sub fibonacci {
my $n = shift;
return 1 if ($n <= 1);
return fibonacci($n - 1) + fibonacci($n - 2);
}
```

U větších projektů s mnoha memoizovanými funkcemi pak poslouží ještě lépe např. Attribute::Memoize:

```
use Attribute::Memoize;
sub fibonacci :MEMOIZE {
my $n = shift;
return 1 if ($n <= 1);
return fibonacci($n - 1) + fibonacci($n - 2);
}
```

Co vlastně modul Memoize udělá? Nejprve se vytvoří nový anonymní memoizovaný podprogram. Ten se pojmenuje stejně jako ten náš původní, který tak je zapomenut.

Další možnosti modulu Memoize

Funkce memoize má několik parametrů. Například lze memoizovanou funkci pojmenovat jinak než původní podprogram. To se může hodit pro účely testování rychlosti.

```
memoize("fibonacci", INSTALL => "memoized_fibonacci");
```

Pomocí volby NORMALIZER => normalizacni_funkce vytváříme třídy ekvivalence mezi voláními. Co když například v následujících volání nezávisí na pořadí?

```
f(a => 1, b => 2);
f(b => 2, a => 1);
```

Pak jsou ekvivalentní a pro větší efektivitu memoizace to můžeme specifikovat pomocí normalizační funkce. Často je ale třeba testovat, zda se to vyplatí.

Normalizační funkce má jako vstup parametry původního podprogramu a výstupem je unikátní řetězec pro každou třídu ekvivalence. V našem případě by například parametry a => 1, b => 2 stejně jako b => 2, a => 1 transformovala například na řetězec "a, 1, b, 2". Standardní normalizační funkce zřetězí parametry speciálním ASCII znakem 28 (file separator, hexa 1c). Podívejme se na příklad normalizační funkce, která zahladí rozdíly mezi pořadím dvojic.

```
sub normalizacni_funkce {
my %hash = @_;
my $id = '';
for (sort keys %hash){
$id .= $_ . "\x{1c}" . $hash{$_} . "\x{1c}";
}
return $id;
}
```

Normalizační funkci lze použít i k opačnému účelu. Pokud náš program závisí na čase, nelze memoizovat. Pokud však závisí jen na určité složce (například sekundě od 0 do 59), stačí ji přidat do výstupního řetězce.

Když už budeme memoizovat, často budeme chtít uchovávat cache i pro další spuštění programu. Paměť se obvykle sama maže. Jak zajistit perzistenci? V dokumentaci je naznačeno použití mechanizmu tie a souborové databáze.

```
use DB_File;
tie %cache => "DB_File", "soubor_s_daty", O_RDWR|O_CREAT, 0666;
memoize("funkce", SCALAR_CACHE => [HASH => \%cache]);
```

Cache můžeme kdykoliv vyprázdnit voláním

```
flush_cache("funkce")
```

Co nememoizovat

- (kritické) Funkce závislé na něčem jiném než parametrech (typicky čas, náhoda, globální proměnné, čtení databáze, čtení souborů atd.)
- (kritické) Funkce, které vrací odkaz na strukturu, která by se mohla měnit
- (kritické) Funkce, které tisknou něco na výstup nebo působí jinou činnost (zápis do databáze nebo souboru, změna globálních proměnných atd.)
- Funkce, jejichž definiční obor je příliš velký a je velká šance, že se parametry trefí opět do nových hodnot
- Funkce, jejichž memoizovaná verze je pomalejší než originální (vždy bychom to měli otestovat)

Perl (139) - Profilling - efektivní odhalování pomalých míst v programu

Napsali jste program, který je pomalý? Nevíte si rady, jak ho zoptimalizovat? Pak zkuste profiler - nástroj, který odhalí, jaké části programu trvají nejdéle.

Kdysi jsme se zabývali modulem Benchmark. To je vynikající nástroj pro případy, kdy potřebujeme výsledek typu porovnání rychlosti dvou podprogramů. Co když ale potřebujeme komplexní analýzu naší aplikace, která neběží tak rychle, jak bychom očekávali? Pak je třeba použít daleko silnější nástroje.

Mezi takové patří tzv. profilery, které dělají dynamickou analýzu běhu programu.

Existují profilery měřící výkon i profilery měřící použití paměti. Mohou fungovat tak, že měří větší úseky (podprogramy nebo nějaké vybrané části) nebo menší úseky (jednotlivé výrazy). Jejich výstupem je rozbor zdrojového kódu v různé formě, statistická analýza nebo něco jiného podle zaměření konkrétního nástroje.

Profiling a Perl

Perl má to štěstí, že existuje nástroj Devel::NYTProf, který zastane většinu běžně požadovaných úkolů.

```
$ cpan Devel::NYTProf
```

Příklad

Většinou se profiler používá na nějaké rozsáhlejší aplikace, které používají řadu modulů, kde tyto moduly používají jiné moduly atd. My si pro jednoduchost ukážeme běh profileru pouze na malém kousku kódu.

Minule jsme se zabývali efektivitou algoritmů na výpočet Fibonacciho posloupnosti. Zkusme se tématu věnovat dále. Podívejme se na následující trojici podprogramů.

```
use Memoize;

sub fibonacci_neopt {
my $n = shift;
return 1 if ($n <= 1);
return fibonacci_neopt($n - 1) + fibonacci_neopt($n - 2);
}

{
```

```
my @vypocitane;

sub fibonacci_opt {
my $n = shift;

return $vypocitane[$n] if defined $vypocitane[$n];

my $vysledek;

if ($n <= 1){
$vysledek = 1;
}
else {
$vysledek = fibonacci_opt($n - 1) + fibonacci_opt($n - 2);
}

return $vypocitane[$n] = $vysledek;
}
}

memoize("fibonacci_memoize");
sub fibonacci_memoize {
my $n = shift;
return 1 if ($n <= 1);
return fibonacci_memoize($n - 1) + fibonacci_memoize($n - 2);
}

fibonacci_neopt(25);
fibonacci_opt(25);
fibonacci_memoize(25);
```

Každý z nich jsme použili k získání 26. čísla z Fibonacciho posloupnosti. Co bude trvat nejděle a jak dlouho? Zkusme se na to podívat pomocí profileru a jen pozorujme, co všechno se dozvíme.

Použití profileru

Předpokládejme, že předchozí kód máme uložen v souboru fib.pl. Použití je velmi jednoduché. Celá teorie tohoto dílu se nyní vejde do dvou příkazů.

Pomocí toho prvního spustíme program a změříme časy pro všechno, co lze.

```
$ perl -d:NYTProf fib.pl
```

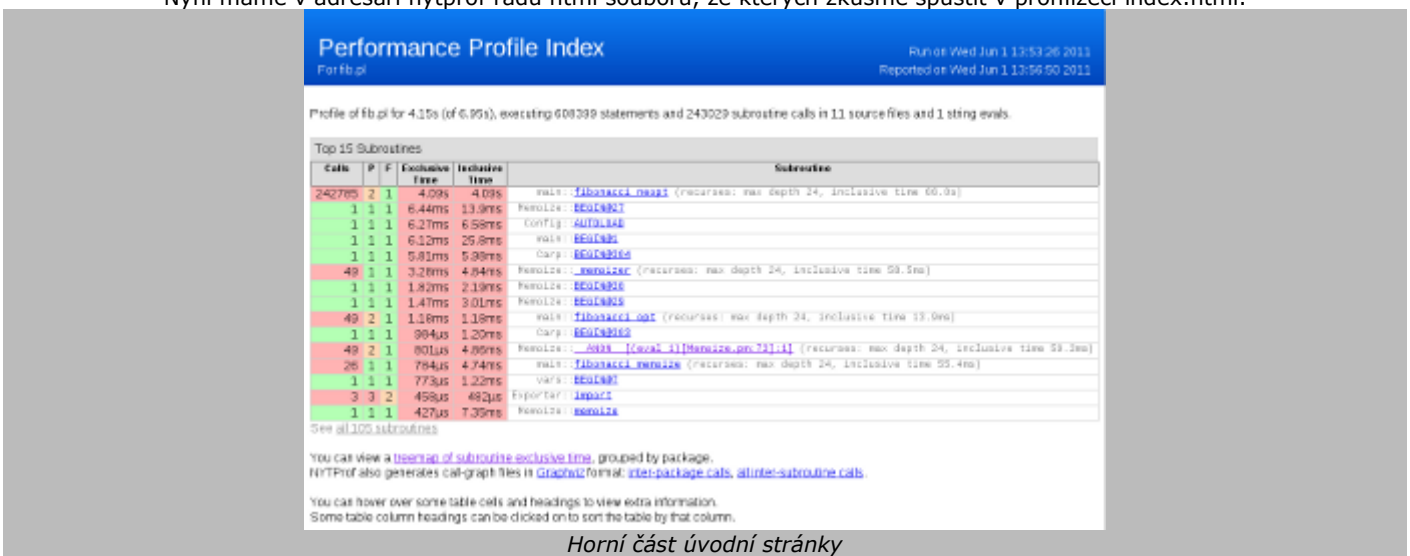
Vznikne nám soubor nytprof.out, kde jsou surová data. Rádi bychom je viděli v nějaké čitelné podobě, takže je zkonvertujeme do html:

```
$ nytprofhtml
```

To je vše, data jsou připravena.

Výsledky analýzy

Nyní máme v adresáři nytprof řadu html souborů, ze kterých zkusme spustit v prohlížeči index.html.



Horní část úvodní stránky

Co můžeme vidět? První řádek nám dá velmi zajímavou informaci:

Profile of fib.pl for 4.15s (of 6.95s), executing 608399 statements and 243029 subroutine calls in 11 source files and 1 string evals.

Profiler počítá s veškerým kódem, který proběhl - tedy i s importovanými moduly. Dva časy ("exclusive time" a "inclusive time") jsou uváděny všude. První znamená čas, který jsme strávili právě uvnitř aktuálního podprogramu; pokud z něj voláme jiné podprogramy, tento čas se nepočítá. Druhý čas pak započítává i čas strávený ve volaných podprogramech. "Inclusive time" je tedy vždy větší nebo roven "exclusive time" a součet všech "inclusive time" může výrazně převyšovat dobu běhu programu, protože se některé úseky započítavají vícekrát.

Dalším údajem zde jsou sloupce se záhlavím P resp. F. Sloupec P udává počet míst, odkud je daný podprogram volán. F je počet souborů, odkud je podprogram volán (tedy P je vždy větší nebo rovno než F).

Analýza podprogramů

Nyní se podívejme na první tabulku. V ní jsou seřazené všechny použité funkce a podprogramy podle toho, kolik času s nimi procesor strávil.

Všimněme si prvního řádku - naše neoptimalizovaná verze pro výpočet 26. čísla Fibonacciho posloupnosti byla volána 240 000 × jen kvůli výpočtu jednoho jednoduchého čísla. Tato volání zabrala astronomických 4,09 sekundy.

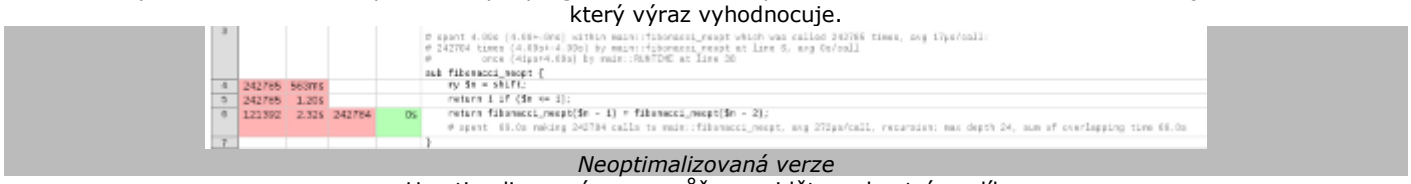
Naše optimalizovaná verze volala sama sebe jen 49× a zabrala 1,18 milisekund, tedy odhadem 4000× méně. Navíc další volání by byla se zapamatovanými daty ještě podstatně rychlejší.

Memoizovaná verze byla zdánlivě ještě dvakrát rychlejší, avšak ve výsledku není započítána režie okolo (jen samotné volání memoize zabralo skoro půl milisekundy).

Všimněme si obarvení na škále mezi zelenou a červenou. Čím je dané číslo červenější, tím větší je náročnost toho, k čemu je přiřazeno. Ovšem implicitní nastavení není příliš citlivé a tak se většinou stává, že červené je skoro všechno. Je to však dobré minimálně k jedné věci: tomu, co není nejčervenější, se nemá smysl při optimalizaci věnovat.

Barevně jsou ve výsledku profileru označeny také místa, kde používáme proměnné $&, $`, $', které výrazně zpomalují regulární výrazy.

Zkusme nyní kliknout na některý z našich podprogramů. U toho neoptimalizovaného můžeme krásně vidět, jak mnohokrát se který výraz vyhodnocuje.

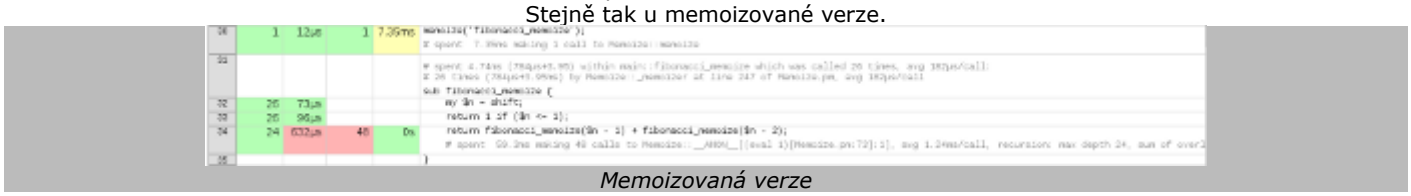


Neoptimalizovaná verze

U optimalizované verze můžeme vidět markantní rozdíl.

Optimalizovaná verze

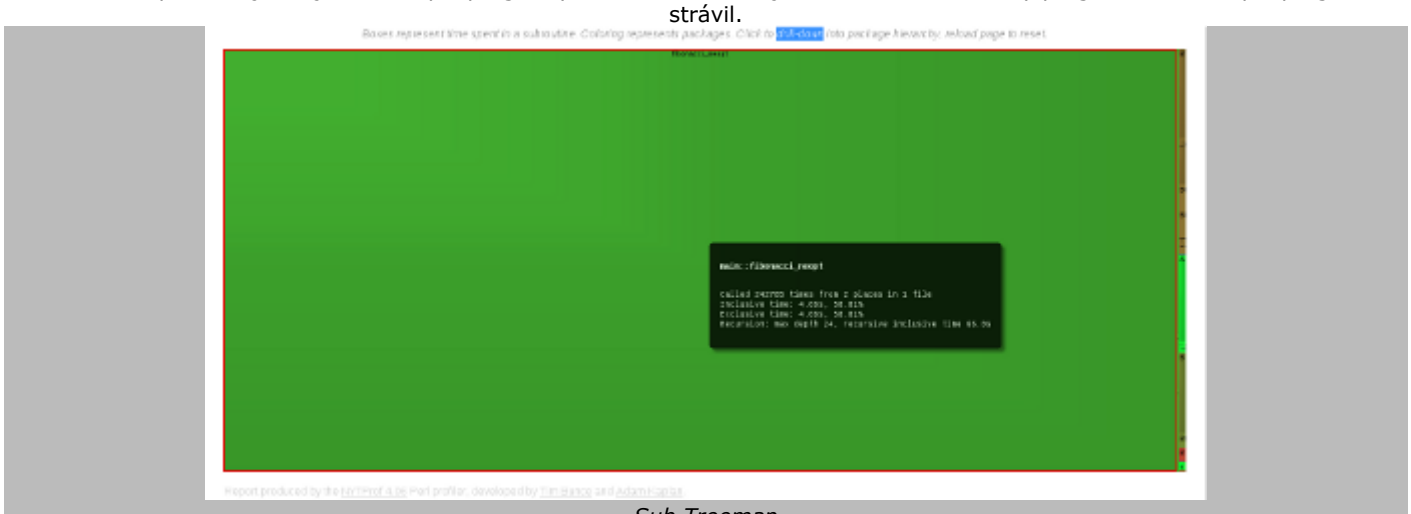
Stejně tak u memoizované verze.



Memoizovaná verze

Doplňme, že nahoře lze přepínat mezi block view, line view a sub view - podle toho, tak podrobně se mají statistiky ukazovat.

Další zajímavou věci je Subroutine Exclusive Time Treemap. Výsledkem je odbélníkový graf, který se sestává z menších obdélníků reprezentujících jednotlivé podprogramy. Obsah obdélníku jsou úměrné času, který program v daném podprogramu strávil.



Sub Treemap

Treemap z takto malého programu, kde má navíc jeden podprogram tak obrovskou majoritu, vůbec nevypadá jako treemap. Každý si však může zkusit profilovat jakýkoliv jiný program a pak může výsledek vypadat např. jako ukázka na blogu Tima Bunceho.

Analýza souborů

Podívejme se nyní na spodní tabulku úvodní stránky.

Dolní část úvodní stránky

Zde vidíme, z kterých všech souborů jsme používali kód. To je samo o sobě zajímavé. Ještě zajímavější však je pohled na to, kolik výrazů bylo v daném souboru vyhodnoceno a kolik času to celkem zabralo včetně nějaké základní statistické analýzy. Kliknutím na line, block nebo sub opět dostaneme rozepsání zdrojového kódu s podrobnějšími výsledky.

Jak optimalizovat

Optimalizace kódu bývá obvykle velmi nepříjemná. Je třeba postupovat systematicky od malých úprav a algoritmy pokud možno neměnit, protože to se může snadno zvrtnout v nekonečnou bezvýslednou práci.

Uveďme si pár triků, které ale snadno udělat lze a mnohdy mohou k požadovanému výsledku stačit. Když stačit nebudou, je to špatné a asi nezbude než se zamyslet nad algoritmem.

- Volání return bychom měli vždy provést ihned, jakmile známe výsledek. Stejně tak lze odfiltrovat některé triviální případy. Dostaneme takovou konstelaci parametrů, že lze použít snažší postup, přeskočit část výpočtu, hned vrátit výsledek apod.? Pak toho využijme.
- Opakované volání stejných metod nad stejnými daty je zbytečné - v případě, že je to v kódu "blízko", je nejlepší to řešit uložením do dočasné proměnné; nastává-li to dokonce globálně, pak zvažme použití memoizace.
  - Cokoliv, co může být mimo cyklus, umístíme mimo něj.
  - Jakmile lze z cyklu vyskočit (již máme vše potřebné), uděláme to.
  - Cokoliv, co může být v podmínce (u které není složité ověřování testu), umístíme do ní.
- Občas existují rychlejší externí moduly, které se dají použít (ať už místo nějakých jiných externích modulů nebo místo vlastního kódu).
- Místo cyklů, jejichž hlavní náplní je volání nějaké metody, lze napsat metodu, která zpracovává seznam parametrů. Tato metoda bývá často rychlejší.

Zjišťování kvality testů

Pomocí profileru Devel::Cover lze získávat informaci o tom, kolik procent kódu bylo použito. To se hodí zejména při testování, nakolik test skripty pokrývají modul. Jako výsledek získáme informaci o tom, jaký kód jsme otestovali a jaký ještě ne.

Další informace

Stojí za to zhlédnout prezentaci o Devel::NYTProf z OSCON 2010. Je zde interaktivně vysvětleno, jak profiling využívat.

Perl (140) - Profilling - píšeme si vlastní profiler / debugger

Již známe profiler Devel::NYTProf. Nutno říct, že existuje řada jednodušších. Ale přesto leckoho napadne otázka: Na jakém principu vlastně taková věc funguje?

Překvapivě není vůbec těžké napsat si vlastní profiler nebo debugger. Rozhodně je dobré si to zkusit, protože potom člověk sváže pochopí, jak vlastně debugger (potažmo všechny profilery) fungují.

K tomu je třeba napsat modul a v něm zadefinovat podprogram DB::DB, který je automaticky volán během zpracovávání každého řádku. Výsledky pak můžeme zobrazit v bloku END. Použití se neliší od profileru Devel::NYTProf.

Debugger/profiler pomocí systémové proměnné

Nejjednodušší (avšak velmi limitovaný) způsob, jak napsat vlastní debugger, je pomocí proměnné prostředí PERL5DB. V ní můžeme nastavit pomocí podprogramu DB::DB chování. Například následující nastavení vždy vyhodnotí řádek a počká si před dalším pokračováním na ENTER.

PERL5DB="sub DB::DB {scalar <STDIN>}"

Takto tedy můžeme spustit libovolný program.

PERL5DB="sub DB::DB {scalar <STDIN>}" perl -d program.pl

Příklad - počítání řádků

Jakkoliv je proměnné PERL5DB zajímavá, je vhodnější si pro nějaké smysluplné profilery vytvořit přímo balík, ve kterém se o všechny požadované operace postaráme.

Ukážeme si zde jednoduchý příklad. Napíšeme si takovou hodně odlehčenou verzi modulu Devel::Cover, o kterém jsme se letmo zmínili minule. Budeme pro libovolný program počítat, kolikkrát jsme vyhodnocovali který řádek.

Jediné, o co se tedy budeme starat, je udržování nějakého pole, které nám bude uchovávat pro každý řádek počet jeho volání.

Vytvořme tedy soubor Devel/MujProfiler.pm, který se nachází v cestě z @INC. V něm bude celý náš profiler.

```
package Devel::MujProfiler;
```

Naším hlavním úkolem je přetížit podprogram DB::DB. Změňme tedy balík a zadefinujme v něm podprogram DB::DB.

```
package DB;

sub DB {
    ...tady budeme počítat volání...
}
```

Budeme používat nějakou globální proměnnou na počítání řádků. Informace do ní získáme pomocí funkce caller, která vrací jméno balíku, programu a aktuální řádek. Uvědomme si, že DB::DB je volán na každém řádku, takže nyní již pouze stačí přičíst 1 k aktuálnímu řádku.

```
my($balik, $soubor, $radek) = caller;

if($soubor eq $0){
```

```
$radky[$radek]++;
}
```
Data máme posbíraná, nyní je potřebujeme nějak zobrazit nebo někam vyexportovat. Zde je nejvhodnějším způsobem umístění do bloku END, který se provádí těsně před koncem programu.
Proměnná @radky je globální a tak není problém k ní přistupovat. Otevřeme tedy sama sebe (tj. soubor z proměnné $0) a budeme po řádcích číst. Každý řádek vytiskneme a z proměnné @radky ještě dodáme číslo, kolikkrát byl právě tento řádek volán (nebo nulu, pokud volán nebyl).
```
END {
my $radek = 0;

open KOD, $0;
while(my $kod = <KOD>){
printf("%2d | %s", $radky[++$radek] || 0, $kod);
}
}
```
Udělejme ještě pár drobných úprav a potom již celý profiler můžeme uložit do Devel/MujProfiler.pm.
```
package Devel::MujProfiler;
use strict;
use warnings;

package DB;

our @radky;

print "Spoustim program. Vystup:\n";

sub DB {
my($balik, $soubor, $radek) = caller;

if($soubor eq $0){
$radky[$radek]++;
}

return;
}

END {
my $radek = 0;
print "\nProgram skoncil.\n\nVysledek profileru:\n\n";
open KOD, $0;
while(my $kod = <KOD>){
printf("%2d | %s", $radky[++$radek] || 0, $kod);
}
print "\n";
}

1;
```
Použití vlastního profileru
Vyberme si nyní nějaký program, který budeme chtít profilovat. Zde je výstup programu, který obsahuje některé řídící struktury. Povšimněme si například, že uvnitř cyklu jsme napočítali 10 vyhodnocení.
```
$ perl -d:MujProfiler program.pl
Spoustim program. Vystup:
Zkusme si cyklus:123456789109 != 10
Program skoncil.

Vysledek profileru:

 0 | #!/usr/bin/env perl
 0 | use strict;
 0 | use warnings;
 0 |
 1 | print "Zkusme si cyklus:";
 0 |
 1 | for (1 .. 10){
10 |     print;
 0 | }
 0 |
 1 | if(9 == 10){
 0 |     print '9 == 10';
 0 | }else{
 1 |     print '9 != 10';
 0 | }
$
```
Příklad - počítání podprogramů
Dalším zajímavým podprogramem pro nás bude DB::sub. Ten je volán pokaždé, když je volán nějaký podprogram. Jako parametry přitom dostane stejné parametry, jako onen volaný podprogram.

V proměnné $DB::sub máme vždy dostupné jméno aktuálního podprogramu. Díky tomu můžeme pracovat s celými podprogramy úplně stejně jako se řádky. Již bez dalšího komentáře si tedy uveďme, jak by mohl vypadat profiler, který by počítal volání jednotlivých podprogramů.

```
package Devel::SpocitejVolaniPodprogramu;
use strict;
use warnings;

package DB;

our %subs;

print "Spoustim program. Vystup:\n";

sub DB {}

sub sub {
$subs{$DB::sub}++;
}

END {
print "\nVolane podprogramy:\n";
for (keys %subs){
sub printf("%2d | %s\n", $subs{$_} || 0, $_);
}
print "\n";
}

1;
```

Použijme tento profiler na libovolný program. Na výstupu dostaneme seznam a počet použití jednotlivých podprogramů.

```
$ perl -d:SpocitejVolaniPodprogramu sub.pl
Spoustim program. Vystup:

Volane podprogramy:
19 | main::new
 5 | main::union
 1 | main::init
$
```

Perl (141) - Formátování kódu, deparsování, perltidy

Co když někdo záměrně napsal nečitelný kód a my chceme zjistit, jak jeho program funguje? Máme šanci? Nebo naopak, pomůžeme si tím, když chceme svůj kód uchránit? Jak Perl chápe náš kód?

Nahlédněme dnes trochu do vnitřností Perlu. Podíváme se, jak zjistit, jakým způsobem Perl chápe kód.

Na třídě všech programů v Perlu lze nalézt (netriviální) ekvivalenci, která tuto třídu faktorizuje na podtřídy tak, aby každá třída ekvivalence obsahovala programy, které Perl chápe stejně. Výraz "chápe stejně" znamená, že je kompiler přeloží do stejných instrukcí. Tento výrok je sice v podstatě tautologický, ale minimálně je zajímavé si jeho pravdivost uvědomit.

Díky modulu B::Deparse lze každý zdrojový kód převést na něco jako standardizovanou formu. Tento modul funguje tak, že nejprve převede zdrojový kód do nějakých instrukcí, kterým rozumí jen Perl. Zobrazení ze zdrojového kódu do interních instrukcí není prosté. Následně B::Deparse provede inverzní proces, čímž získáme opět zdrojový kód. Protože ono první zobrazení nebylo prosté, může se tento kód od původního lišit.

Příklad

Co dělá následující kód?

```
$_ = "abcdefgh";
s$g$g$g;
print
```

Vypadá to jako překlep, ale po spuštění interpret nezahlásí žádnou chybu.

```
$ perl -w kod.pl
abcdefgh
$
```

Toto je poměrně jednoduchá ukázka, navíc nám pomohlo zvýraznění syntaxe. Lze tedy tušit, že prostřední řádek znamená substituci s trochu netradičním oddělovačem. Podívejme se ale, co nám řekne B::Deparse.

```
$ perl -MO=Deparse test.pl
$_ = 'abcdefgh';
s/g/g/g;
print $_;
test.pl syntax OK
$
```

Na našem kódu se změnilo několik věcí. Především, v prostředním řádku se změnil oddělovač z dolaru na lomítko, odkud je hned jasné, že je to skutečně substituce. Dále byly ještě nahrazeny uvozovky apostrofem a výchozí proměnná byla uvedena všude tam, kde došlo k její použití.

Řetězec test.pl syntax OK je vypsán na STDERR, pokud je kód validní. V opačném případě jsou vypsány chyby, nicméně B::Deparse se pokusí aspoň zčásti kód vytisknout.

Nečitelný kód

B::Deparse nepředstavujeme ani tak kvůli praktickému využití jako spíš zajímavost, pomocí které lze pochopit řadu věcí tak, jak je chápe překladač.

Avšak někdy se může modul B::Deparse hodit například i pro pochopení toho, jak fungují nečitelné programy.

Nejlepším zdrojem pro nečitelné programy v Perlu je tzv. Obfuscated Perl Contest, což bývala soutěž o nejnečitelnější kód. Podívejme se na následující program.

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU / lreP rehtona tsuJ";sub p{
@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=$f=!fork;map{$P=$P[$f^ord
($p{$_})&6];$p{$_}=/ ^$P/ix?$P:close$_}keys%p}p;p;p;p;p;map{$p{$_}=~/^[P.]/&&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$_=$d[$q];sleep rand(2)if/\S/;print
```

Spustíme-li ho, vypíše se pomalu text

Just another Perl / Unix hacker

Jak ale tento program skutečně funguje, to je záhada možná i na několik hodin. V případě, že nepoužijeme B::Deparse. Zkusme aplikovat B::Deparse na předchozí program. Dostaneme již podstatně přátelštější kód.

```
@P = split(??, '.URRUUxR', 0);
@d = split(??, "\nrekcah xinU / lreP rehtona tsuJ", 0);
sub p {
@p{"r$p", "u$p"} = ('P', 'P');
pipe "r$p", "u$p";
++$p;
($q *= 2) += $f = !fork;
map {$P = $P[$f ^ ord $p{$_} & 6];
$p{$_} = / ^$P/xi ? $P : close $_;} keys %p;
}
p ;
p ;
p ;
p ;
p ;
map {close $_ if $p{$_} =~ /^[P.]/;} %p;
wait until $?;
map {<$_> if /^r/;} %p;
$_ = $d[$q];
sleep rand 2 if /\S/;
print $_;
```

Sice se nedá říct, že bychom už na první pohled mohli přesně říci, co se děje, ale již nás asi napadne, že různé části textu jsou tisknuty různými procesy a jejich synchronizace probíhá pomocí rour. Použití B::Deparse nebo nástroje perltidy by měl být první krok při čtení takto nepřehledného kódu.

Zajímavý příklad

Poradí si opravdu B::Deparse s jakýmkoliv kódem? Pro zpestření si ještě jednou pro inspiraci sáhněme do soutěže Obfuscated Perl Contest. Zde je další zajímavý program.

```
#:: ::-| ::-| .-. :||-:: 0-| .-| ::||-| .:|-. :||
open(Q,$0);while(<Q>){if(/^#(.*)$/){for(split('-',$1)){$q=0;for(split){s/\|
/:.:/xg;s/:/../g;$Q=$_?length:$_;$q+=$q?$Q:$Q*20;}print chr($q);}}}print"\n";
#.: ::||-| .||-| :|||-| ::||-| ||-:: :|||-| .:|
```

Zkusme ho spustit.

```
$ perl kod.pl
The Perl Journal
$
```

A jak funguje? Zkusme na něj použít nám známý modul B::Deparse, který nám kód promění do podoby, v jaké ji chápe Perl.

```
$ perl -MO=Deparse poc.pl
open Q, $0;
while (defined($_ = )) {
if (/^#(.*)$/) {
foreach $_ (split(/-/, $1, 0)) {
$q = 0;
foreach $_ (split(' ', $_, 0)) {
s/\|
/:.:/gx;
s/:/../g;
$Q = $_ ? length $_ : $_;
$q += $q ? $Q : $Q * 20;
}
print chr $q;
}
}
}
print "\n";
poc.pl syntax OK
$
```

Výstup si uložme do souboru a pošleme na něj interpret Perlu. Výsledek nás asi překvapí, protože se nic nevypíše. Proč zde B::Deparse nefunguje?

Ve skutečnosti B::Deparse funguje, problém je někde jinde. Nahlédněme zpět do zdrojového kódu, přesněji hned na první řádek. Tento skript je totiž zároveň jak programem, tak jeho daty. Program parsuje svůj vlastní kód a z komentářů extrahuje data. Ale B::Deparse všechny komentáře odstraní a tím pádem se kód stává nefunkční. Musíme tedy nejprve přidat ony dva komentáře.

perltidy - nástroj na formátování kódu

Pokud nám jde čistě o zpřehlednení kódu, je na to asi nejlepším nástrojem příkaz perltidy. Tento nástroj umí formátovat zdrojový kód Perlu několika způsoby.

Zavoláním následujícího příkazu se kód přeformátuje do podoby, která by měla co nejvíce odpovídat manuálové stránce perlstyle(1). Na výstupu vznikne soubor program.pl.tdy.

$ perltidy program.pl
Kdo má raději formátování GNU, může přidat příslušný přepínač.
$ perltidy -gnu program.pl
Zajímavé je také HTML zvýrazňování syntaxe. Přidáním -html dostaneme v souboru program.pl.html HTML stránku.
$ perltidy -html program.pl
Další možnosti tohoto nástroje nalezneme v dokumentaci.

Perl (142) - Způsoby konfigurování

CPAN obsahuje několik modulů pro práci s konfiguračními soubory. Na tuto otázku se ovšem podíváme ještě o něco šířeji, neboť konfigurovat lze i jinými způsoby.

Každý větší a i řada menších programů vyžaduje konfigurovatelnost. Pro běh v různých podmínkách bude vyžadovat různě nastavené implicitní hodnoty. Program by měl být v tomto směru dostatečně flexibilní, abychom, pokud možno, nikdy nemuseli pro změnu výchozího nastavení zasahovat do zdrojového kódu algoritmu.

Konfigurace pomocí proměnných ve zdrojovém kódu

První způsob konfigurace, který nás většinou napadne, je umístění speciálního deklaračního bloku do zdrojového kódu, případně import souboru s konfigurací pomocí require v bloku BEGIN. V něm si uživatel modulu výchozí hodnoty nastaví.

Program pak může začínat následovně.

```
#!/usr/bin/perl
use strict;
use warnings;

### NASTAVENÍ - ZAČÁTEK###

my $login = "vase_jmeno";
my $heslo = "vase_heslo";
my $host  = "localhost";

### NASTAVENÍ - KONEC ###
```

Zdánlivou výhodou je jednoduchost. Tento postup lze v odůvodněných případech použít v pomocných programech pro vlastní potřebu, ale už vůbec není vhodný pro jakoukoliv masovou distribuci.

Nevýhodou je zejména zasahování uživatele do zdrojového kódu. Rozhraní a zdrojový kód by totiž měly být striktně oddělené, což zde vůbec není pravda. Na uživatele je také kladena nutnost alespoň základní znalosti syntaxe Perlu, což je nárok, který by tu být vůbec nemusel. V neposlední řadě při hromadném použití je třeba upravovat každou instanci zvlášť, což na přehlednosti nepřidá.

Tato metoda se dobře hodí prakticky jen pro vnitřní potřebu autora kódu, nikoliv pro uživatelské nastavení. Například se lze často setkat s něčím jako je

```
my $DEBUG = 0;
```

YAML soubor

Výrazně lepší je import datové struktury ve formátu YAML ze speciálního souboru. YAML slouží k uchovávání proměnných napříč různými skriptovacími jazyky. Perl jej podporuje také. Předtím, než začnete psát konfigurační soubory v YAML formátu, čtěte dále. Dostaneme se k modulu Config::Auto, který umí číst mimo jiné YAML konfiguraci.

Konfigurace pomocí voleb příkazového řádku

Podstatně čistější metoda pro konfiguraci je také umístění dat do argumentů příkazového řádku. Pro každé volání tak volíme parametry zvlášť.

$ ./program.pl localhost pepa heslo

Lepší je však nechtít heslo jako parametr, ale říci si o něj interaktivně (například pomocí modulu Term::ReadPassword).

Interaktivita se nám však často nemusí hodit a pak nezbude než použít nějaký konfigurační soubor.

Volba -s příkazu perl může sama rozpoznávat přepínače typu pravda/nepravda. Můžeme též použít krátké přepínače

$ ./program.pl -h localhost -u pepa

nebo dlouhé přepínače.

$ ./program.pl --host localhost --login pepa

Tímto tématem jsme se již zabývali a proto v případě zájmu následujte uvedené odkazy.

Interaktivní konfigurace

Již jsme naznačili též možnost postupně se na začátku běhu programu zeptat uživatele na všechny nutné údaje. To často bývá zdlouhavá činnost a nelze to dělat hromadně. Je to nevhodné i v případech, kdy uživatel často zadává stejné hodnoty (leda že by byla automaticky nabízena očekávaná hodnota přečtená například z konfiguračního souboru).

Nezřídka je program dokonce volán jiným programem a tam by bylo zadávání dat na standardní výstup komplikované. V některých případech se však interaktivní zadávání hodí.

Konfigurace pomocí proměnných prostředí

Také jsme se věnovali přístupu k proměnným prostředí.

Jen poznamenejme, jak lze proměnné prostředí nastavovat zvenku. Pro jednorázové použití proměnné prostředí můžeme její hodnotu zamontovat do příkazu.

$ HOST=localhost LOGIN=pepa ./program.pl

Nyní budeme mít uvnitř programu v $ENV{HOST} a $ENV{LOGIN} nastavené hodnoty.

Chceme-li toto nastavení zachovat pro zbytek sezení shellu, použijeme příkaz export v Bashi, případně v jiném shellu set, setenv apod.

$ export HOST=localhost
$ export LOGIN=pepa
$ ./program.pl

Konfigurační soubory

Na CPANu máme k dispozici několik modulů pro čtení konfiguračních souborů. Jde o nejsofistikovanější volbu nejen pro rozsáhlé aplikace s velkým množstvím nastavitelných hodnot.

Nejjednodušší konfigurační soubory

Pro jednoduché konfigurační soubory nám plně postačí modul ConfigReader::Simple. Tento modul čte konfigurační soubor, ve kterém je na každém řádku pár *klíč-hodnota*. *Klíč* a *hodnota* mohou být odděleny prvním rovnítkem, mezerami nebo rovnítkem a mezerami okolo. Takto mohou vypadat řádky v konfiguračním souboru.

#komentář
klic1 hodnota
klic2 = hodnota
klic3 hodnota
klic4 hod no ta
klic5 = hod no ta
klic6 "hod no ta"
klic7 'hod no ta'

Nevýhodou je omezení na skalární hodnoty. Použití je ale zato intuitivní. Nejprve si předchozí konfigurační soubor (nebo jakýkoliv jiný) uložme do souboru mujprogramrc. Poté ho načteme a rozparsujeme pomocí ConfigReader::Simple.

```
use ConfigReader::Simple;

$config = ConfigReader::Simple->new("mujprogramrc");
```

Získali jsme objekt $config, z kterého získáme metodou get hodnotu libovolného klíče. Takto vytiskneme jednu z hodnot.

```
print $config->get("klic4");
```

Konfigurační soubory lze i editovat. Přidejme si klíč login s hodnotou pepa a uložme nový konfigurační soubor do novyrc.

```
$config->set("login", "pepa");
$config->save("novyrc");
```

Dodejme, že ConfigReader::Simple umí zpracovávat i více konfiguračních souborů zároveň.

```
$config = ConfigReader::Simple->new_multiple(Files => ["rc1", "rc2"]);
```

Automatické zjišťování formátu konfigurace

Modul Config::Auto umí sám od sebe rozpoznat jméno konfiguračního souboru i jeho formát souboru. Lepší však často bývá zadat alespoň formát ručně, neboť rozpoznávání funguje pouze heuristicky. Rozpoznávání jména konfiguračního souboru funguje na základě jména programu a je blíže popsáno v manuálové stránce.

Seznam aktuálně podporovaných formátů získáme voláním metody Config::Auto->formats.

irssi bind ini space perl colon yaml xml equal list

Pojďme však nyní vyzkoušet parsovat nějaké soubory. Co třeba /etc/resolv.conf? Poradí si Config::Auto?

```
use Config::Auto;
use Data::Dumper;

$config = Config::Auto::parse("/etc/resolv.conf");

print Dumper $config;
```

Ano, poradí. Výsledkem je následující datová struktura.

```
$VAR1 = {
'search' => [
'vashost.cz',
'linux'
],
'nameserver' => '192.168.15.94'
};
```

Při parsování pomocí Config::Auto stačí vynaložit minimální úsilí, modul se sám o všechno postará.

Ještě se podívejme na to, jak zadat přesně formát konfiguračního souboru.

```
$config = Config::Auto::parse("config.xml", format => "xml");
```

Perl (143) - Struktura datových typů, správa paměti

Jednou z nevýhod dynamicky typovaných jazyků je neefektivní spotřeba paměti. Měli bychom aspoň trochu tušit, jak si vede konkrétně Perl.

Již jsme několikrát narazili na měření rychlosti programů nebo jejich úseků. Zatím však vůbec nepřišla řeč na druhý významný systémový zdroj - paměť. Částečně i proto, že na její měření nám nestačí pouhé stopky.

Programátora v Perlu nezajímá, co se kdy do paměti má ukládat. Má na to své "lidi". Alokaci a s tím související věci vůbec neřeší. Přesto je dobré, minimálně kvůli informaci, jak Perl funguje (nebo i kvůli jiným věcem, například debugging XS programů), alespoň tušit, jak Perl s pamětí přidělenou operačním systémem pracuje.

Jako vždy jde mezi pamětí a rychlostí o kompromis. Pro dobrý příklad se vraťme k memoizaci. Pomocí ní můžeme zvýšit rychlost, ale vždy za cenu větších paměťových nároků.

Jak na tom Perl tedy je? Vzhledem k velké rychlosti programů v Perlu se dá o paměťových nárocích leccos usuzovat a asi to nikoho moc nepřekvapí. Perl je významným spotřebitelem paměti, a tak si pro dnešek sundáme růžové brýle (ale jen na chvíli).

Není tajemstvím, že Perl preferuje výkon nad úsporou paměti. Perl si raději i jednoduchou věc uloží, než aby ji musel znovu vypočítat (později se podíváme na konkrétní příklady). Když přidáme pokročilé datové typy a dynamické typování (v důsledku čehož má například integer minimálně 16 bajtů, float 24, řetězec 28 atd.), pramení z toho vysoká paměťová náročnost.

Když libovolný proces potřebuje paměť, požádá o ni operační systém. Ten mu přidělí adresy, které může využívat. Perl se operačnímu systému nebojí říct si vždycky o velký kus (který celý momentálně ani nepotřebuje) a k zabírání paměti celkově přistupuje s filozofií "na horší časy". Když přestane část paměti využívat, tak ji operačnímu systému nevrací - co kdyby ji ještě na něco potřeboval.

Vnitřní reprezentace proměnných

Prozkoumejme však práci s pamětí a s tím související vnitřní strukturu datových typů podrobněji pomocí několika z CPANu dostupných modulů. Budou nás zajímat hlavně tyto dva základní.

```
use Devel::Peek qw(Dump);
use Devel::Size qw(size total_size);
```

Proměnná obvykle mívá hodnotu. Ta může být jedním z následujících typů.

- SV - skalární
- AV - pole
- HV - hash
- RV - odkaz
- GV - typeglob

- CV - podprogram

Tyto zkratky jsou vlastně datovými typy v jazyce C a každý typ má nějaké API, se kterým lze pracovat. Pro skalární hodnoty jsou všechny tyto funkce deklarovány v hlavičkovém souboru sv.h, který můžeme použít v našem C programu. Bývá umístěn v adresáři jména typu /usr/lib/perl5/5.12.3/i586-linux-thread-multi/CORE. Jsou zde nízkoúrovňové funkce pro vytvoření nového skaláru, přiřazení do skaláru (zvlášť pro celé číslo, desetinné číslo, řetězec atd.) a podobně. Například pro funkci, která z řetězce vytvoří novou skalární hodnotu, máme deklarovanou následující hlavičku.

```
SV* newSVpv(char* str, int len);
```

Tomu se zde nebudeme více věnovat, ale koho by více zajímalo použití těchto hlavičkových souborů nebo vnitřní struktura jazyka obecně, nechť se podívá na manuálovou stránku perlguts(1).

Skalární proměnná v paměti

Skalární proměnná může obsahovat buď celé číslo (IV), celé číslo bez znaménka (UV), desetinné číslo (NV), řetězec (PV) nebo speciální magickou hodnotu (MG). Později uvidíme, že může obsahovat i kombinace některých hodnot.

Vytvořme na začátku nějakou proměnnou. Nic do ní nebudeme přiřazovat, pouze ji deklarujeme. Pomocí funkcí Dump a size můžeme zkoumat některé nízkoúrovňové informace.

```
my $skalar;
print Dump($skalar);
print "Velikost: ", size($skalar);
```

Výstupem Dump je následující text.

```
SV = NULL(0x0) at 0x832d31c
  REFCNT = 1
  FLAGS = (PADMY)
```

Jak již víme, SV znamená Scalar Value. Za SV následuje NULL(0x0), což znamená, že proměnná je prázdná a na konci řádku máme její adresu. V REFCNT je počet odkazů na tuto proměnnou a ve FLAGS jsou různé flagy (například, že je proměnná deklarovaná pomocí my, také zde ale můžeme mít informaci, že jde o objekt atd.).

Výstupem funkce size pak je počet bajtů, které proměnná zabírá. V našem případě je to 16! To je dost vysoké číslo na to, že tato proměnná nedrží žádnou informaci. Každá proměnná tedy zabírá minimálně 16 bajtů v paměti.

Dále můžeme zkusit do proměnné přiřadit řetězec.

```
my $skalar = "qwerty";
print Dump($skalar);
print "Velikost: ", size($skalar);
```

Opět se můžeme podívat na výstup Dump.

```
SV = PV(0x830e724) at 0x832d31c
  REFCNT = 1
  FLAGS = (PADMY,POK,pPOK)
  PV = 0x8328024 "qwerty"\0
  CUR = 6
  LEN = 8
```

Přibylo zde několik nových řádků. CUR je počet znaků a LEN počet alokovaných míst pro znaky. Ještě tedy můžeme trochu přidat a pak bude potřeba přialokovat další paměť. LEN je vždy alespoň o 1 větší, protože je zde započítán i jeden speciální znak. S tím souvisí i velikost. Ta vzrostla na 32 bajtů. Zajímavé je zkusit si přidávat nebo ubírat postupně znaky. Uvidíme, že velikost vždycky jednou za čas podle LEN a CURskočí, ale často se nemění. Také je zajímavé vyzkoušet přidávat jiné než ASCII znaky.

Dále zde je PV (pointer value), což, znamená, jak opět víme, že jde o řetězec. Zajímavé je, že pokud řetězec někde použijeme jako číslo, změní se PV na PVIV a bude mít jak řádek PV tak i IV - tedy pro oba datové typy bude celá informace uložena zvlášť. Toto lze udělat i pro další dvojice, například můžeme dostat PVUVnebo PVNV. Podívejme se tak potom vypadá výpis Dump pro následující situaci.

```
my $c = 1;   # použijeme jako celé číslo
print "$c";  # použijeme jako řetězec
print Dump($c);
```

Zde je výsledek.

```
SV = PVIV(0x8323378) at 0x832d31c
  REFCNT = 1
  FLAGS = (PADMY,IOK,POK,pIOK,pPOK)
  IV = 1
  PV = 0x8328024 "1"\0
  CUR = 1
  LEN = 4
```

Takové chování se samozřejmě se projeví i na paměťové náročnosti - uložené je obojí, takže celkem je zabráno 32 bajtů. Tady je pěkně vidět, jak Perl s pamětí hospodaří.

Ještě jsme na začátku zmínili magický typ. Taková proměnná se vyznačuje tím, že má nějakou speciální vlastnost, například spojení s nějakou akcí. To, že k takové proměnné přistoupíme může například způsobit spuštění nějakého jiného procesu. Typicky jde o navázané proměnné. Několik speciálních proměnných má magický typ, například $!. Funkce Dump zde odhalí následující.

```
SV = PVMG(0x83536a0) at 0x832d334
  REFCNT = 1
  FLAGS = (GMG,SMG)
  IV = 0
  NV = 0
  PV = 0
  MAGIC = 0x833b8e4
  MG_VIRTUAL = &PL_vtbl_sv
  MG_TYPE = PERL_MAGIC_sv(\0)
  MG_OBJ = 0x832d324
  MG_LEN = 1
  MG_PTR = 0x8334f44 "!"
```

Další způsob, jak získat magickou hodnotu, je režim nakažení. Každá nakažená proměnná je typu MG. S využitím této vlastnosti pak funguje například modul Safe, který umožňuje s nakaženými daty například i provedení eval, pokud není výsledkem vyhodnocení nějaká zakázaná instrukce.

Odkaz v paměti

Dump zobrazuje u odkazů hierarchii. Takto mohou vypadat informace pro odkaz na číslo 123.

```
SV = IV(0x832d4f8) at 0x832d4fc
  REFCNT = 1
  FLAGS = (PADMY,ROK)
  RV = 0x832d31c
SV = IV(0x832d318) at 0x832d31c
  REFCNT = 2
  FLAGS = (PADMY,IOK,pIOK)
  IV = 12
```

Funkce size ukazuje v případě odkazu velikost toho, na co se odkazuje, což taky většinou chceme. V případě odkazu na odkaz na hodnotu však již dostaneme skutečně velikost odkazu na hodnotu, nikoliv hodnotu.

Volání Dump pro odkazy se bude hodit zejména pro zkoumání polí a hashů.

Pole v paměti

Podívejme se i na ostatní datové typy. Pole (AV) je ve skutečnosti dynamickým seznamem odkazů na skaláry.

V perlguts(1) nebo hlavičkovém souboru av.h opět nalezneme API seznamů.

Tiskneme-li výstup Dump u odkazu na pole (například pole [1 .. 8]), získáme jak celkovou analýzu (počet prvků, poslední prvek, typ hodnot, kolik paměti je alokováno a kdy bude potřeba přialokovat atd.) tak i postupně informace o několika úvodních prvcích. Výstup tedy vypadá takto.

```
SV = IV(0x832d318) at 0x832d31c
  REFCNT = 1
  FLAGS = (PADMY,ROK)
  RV = 0x8310874
SV = PVAV(0x83118dc) at 0x8310874
  REFCNT = 1
  FLAGS = ()
  ARRAY = 0x832c6e4
    FILL = 7
    MAX = 7
    ARYLEN = 0x0
    FLAGS = (REAL)
    Elt No. 0
SV = IV(0x83109c0) at 0x83109c4
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 1
    Elt No. 1
SV = IV(0x83115d0) at 0x83115d4
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 2
    Elt No. 2
SV = IV(0x83115c0) at 0x83115c4
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 3
    Elt No. 3
SV = IV(0x8311600) at 0x8311604
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 4
```

Funkce size nám u polí přestává fungovat tak, jak bychom si představovali. Měří pouze velikost struktury, nikoliv už dat v ní uložených. Pro každé stejně velké pole vrátí size stejnou hodnotu. Abychom získali velikost i s daty, je třeba použít total_size, která následuje reference.

Například struktura následujícího desetiprvkového pole zabere 28 bajtů. S obsahem je to však dohromady už 1028.

```
my $pole_r = ("." x 100) x 10;
```

Hash v paměti

Podobné informace získáme i pro hashe. Hash je dvojice jednoznačného řetězce a ukazatele na skalár. Řetězec lze přitom namapovat na celé číslo a pak lze hash reprezentovat jako obyčejné dynamické pole. Mapování přitom funguje poměrně zajímavě. Na bázi nějakého hashovacího algoritmu (poznamenejme, že odtud pochází název pro datový typ hash) totiž ke každému klíči přiřadíme celé číslo mezi 1 a 2^n, kde *n* je nějak zvolené.

Přitom může docházet ke kolizím! To znamená, že se dva klíče mohou namapovat na stejné číslo. Skutečnost je taková, že se vlastně uchovává pro každé číslo spojový seznam všech hodnot, jejichž klíče se na toto číslo zobrazily hashovací funkcí. Ideální stav by byl, kdyby byly prvky hashe v poli co nejrovnoměrněji rozloženy (to jest, aby spojové seznamy měly pokud možno právě jeden prvek). To zajišťujeme za prvé vhodným hashovacím algoritmem. S ním mimochodem můžeme pomocí API manipulovat (třeba i proto, že máme nějak specificky zvolenou množinu klíčů hashe a výchozí algoritmus tím pádem nevyhovuje, protože tyto klíče mapuje často na stejné hodnoty). Stačí předat číslo prvku (tj. námi alternativně zahashovanou hodnotu) funkci hv_store deklarované v hlavičkovém souboru hv.h, která ukládá prvek hashe do tabulky. Výchozí hashovací algoritmus je definován jako makro PERL_HASH(hash, key, klen), kde klen je délka klíče.

```
hash = 0;
while (klen--)
    hash = (hash * 33) + *key++;
hash = hash + (hash >> 5);      /* od verze 5.6 */
```

Zopakujme, že pokud tedy proženeme náš klíč tímto algoritmem, získáme v proměnné hash nějaké číslo, pod které se uloží jako další prvek spojového seznamu naše hodnota.

A za druhé co nejrovnoměrnější rozdělení zajišťujeme vhodnou volbou *n*. Jak hash roste na velikosti, roste i *n*. Pokud počet prvků hashe dosáhne 2^n (tj. na jedno číslo jeden prvek hashe), rozšíří se tabulka automaticky zvýšením *n* o 1.

Dodejme, že tiskneme-li hash ve skalárním kontextu, získáme dvě čísla oddělená lomítkem:

- Počet prvků hashe, tj. počet klíčů
- 2^n, tj. velikost tabulky, do které se mapují prvky hashe na základě hashovacího algoritmu

Například následující kód vytiskne 2/8.

```
%x = (prvni => 1, druhy => 2);
print scalar %x;
```

Kdybychom přidali dalších 6 prvků, dostali bychom na výstupu 8/8. Kdybychom však dostali ještě jeden další prvek, již by se vytisklo 9/16, protože by se zvýšilo *n*. Jakmile se zvýší, není třeba nic přepočítávat. Prvky již přítomné si svoje číslo ponechají nadále.

Čím vyšší je poměr těchto dvou čísel, tím je algoritmus paměťově úspornější. Naopak, čím je poměr nižší, tím rychleji jsme schopni z hashe číst, protože je menší pravděpodobnost, že budeme nuceni prohledávat delší spojový seznam.

Uveďme nevýznamnou, avšak zajímavou poznámku na okraj. Pokud očekáváme hodně prvků, lze velikost tabulky nastavit přímo v programu následujícím příkazem.

```
keys %x = 1000;
```

Díky němu bude překladač vědět, že má dynamické pole naalokovat větší než původně chtěl, protože to časem bude potřeba. Po jeho zavolání se *n* nastaví tak, aby 2^n byla nejbližší vyšší mocnina dvou. Přidáním tohoto řádku by tak měl předchozí kód za následek vytisknutí 2/1024.

Podívejme se ale, jak bude vypadat výstup funkce Dump u hashů. Nyní k němu asi už není třeba podrobného komentáře.

```
SV = IV(0x83115a0) at 0x83115a4
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x832d35c
SV = PVHV(0x83186dc) at 0x832d35c
  REFCNT = 3
  FLAGS = (OOK,SHAREKEYS)
  ARRAY = 0x83351f4  (0:1022, 1:2)
  hash quality = 125.0%
  KEYS = 2
  FILL = 2
  MAX = 1023
  RITER = -1
  EITER = 0x0
  Elt "prvni" HASH = 0x84356c8d
  SV = IV(0x8310870) at 0x8310874
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 1
  Elt "druhy" HASH = 0xa05ccdc7
  SV = IV(0x83109f0) at 0x83109f4
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 2
```

Ostatní datové typy

Samozřejmě to nejsou všechny datové typy. Typegloby jsou reprezentovány hashovou tabulkou a funkce Dump ukáže například následující.

```
SV = PVGV(0x8346888) at 0x832d334
  REFCNT = 4
  FLAGS = (MULTI,IN_PAD)
  NAME = "x"
  NAMELEN = 1
  GvSTASH = 0x8310764   "main"
  GP = 0x8334f8c
  SV = 0x832d344
    REFCNT = 1
    IO = 0x0
    FORM = 0x0
    AV = 0x832d3b4
    HV = 0x0
    CV = 0x0
    CVGEN = 0x0
    LINE = 1
    FILE = "-e"
    FLAGS = 0xa
  EGV = 0x832d334     "x"
```

Pro úplnost se podívejme na výstup Dump pro anonymní podprogram.

```
SV = IV(0x8310870) at 0x8310874
  REFCNT = 1
  FLAGS = (TEMP,ROK)
  RV = 0x832d32c
SV = PVCV(0x83290dc) at 0x832d32c
  REFCNT = 2
  FLAGS = (PADMY,ANON,WEAKOUTSIDE)
```

COMP_STASH = 0x8310764 "main"
START = 0x8334d48 ===> 0
ROOT = 0x8327cb0
GVGV::GV = 0x832d36c "main" :: "__ANON__"
FILE = "-e"
DEPTH = 0
FLAGS = 0x90
OUTSIDE_SEQ = 94
PADLIST = 0x832d33c
PADNAME = 0x832d34c(0x8334ebc) PAD = 0x832d3ac(0x8334f64)
OUTSIDE = 0x8310a04 (MAIN)

Pro více informací lze opět doporučit stránku perlguts(1).

Memchmark

Existuje analogie modulu Benchmark pro měření paměťových nároků. Memchmark exportuje funkci cmpthese, která spočítá, kolik paměti zabírá který podprogram. Funguje tak, že vytvoří nový proces, zjišťuje v intervalech jeho paměťové nároky a hledá maximální hodnotu. Díky tomu se však nemusí modul vždy chovat 100% správně. Zde je malý příklad použití.

```
use Memchmark qw(cmpthese);
cmpthese(
maly  => sub { ("." x 1000) x  10_000 },
velky => sub { ("." x 1000) x 100_000 }
);
```

Na výstupu dostaneme přibližnou paměťovou náročnost jednotlivých podprogramů.

```
test: maly, memory used: 9871360 bytes
test: velky, memory used: 99872768 bytes
```

Perl (144) - POE - událostmi řízené programování

Jak lze v Perlu psát událostmi řízené programy a co to vlastně je?

POE je slovy svého autora framework pro multitasking a networking pro libovolné událostní smyčky. Znamená Perl Object Environment (nebo libovolný z dalších akronymů ze stránky poe.perl.org).

Před bližším popisem si pro navození alespoň nějaké představy vzpomeňme na používání grafických toolkitů. Zpravidla jsme vždy volali funkci s názvem mainloop nebo podobným. Ta sloužila k rozběhnutí smyčky událostí. Program vždy čekal na události od uživatele, na které vzápětí nějak reagoval.

Pojďme se na to ale podívat postupně a podrobněji.

Událost

Co je to událost? Podívejme se na pár příkladů "ze života".

- bazén je napuštěn
- voda dosáhla bodu varu
- jídlo je hotové
- autobus přijel na zastávku

Dalo by se s nadsázkou říct, že reálný svět je naprogramován právě pomocí reakcí na události. Vše má svůj důvod a vše je reakcí na nějakou událost.

Našim účelům budou blíže méně abstraktní příklady z IT života. Co může být událostí?

- start programu
- klik levým tlačítkem do určené oblasti
- stisk ENTER
- konec reakce na jinou událost

Událostmi řízené programování

Na základě emitování událostí a jejich obsluhování k tomu napsanými ovladači lze vytvořit spletitou síť, která bude něco smysluplného dělat. Pro analogii bychom opět mohli zajít do reálného světa.

Dostaneme-li na začátku programu událost _start, můžeme ji na základě okolností, vstupů atd. masivně rozvinout, přičemž nám jako programátorům stačí pouze napsat ovladače. O ostatní se postará toolkit (v našem případě POE). Naopak, vykonáno bude jen to, co je důsledkem události, která skutečně nastala. Bez událostí se žádná činnost sama neděje.

Toto paradigma se nazývá událostmi řízené programování (event-driven programming).

Událostmi řízené programování, POE a Perl

Dvěma základními součástmi POE, které budeme používat v každém programu, jsou POE::Kernel a POE::Session. Stručně řečeno, instance POE::Session jsou úkoly nebo procesy, které jsou spravovány pomocí POE::Kernel. Každý úkol má například vlastní data nebo zdroje. POE::Kernel vše řídí a rozvrhuje, kdy se má co dělat.

Vytvoření POE programu tedy zahrnuje vytvoření jednoho nebo několika úkolů (sessions), které budou obsluhovat nějaké události. Například můžeme obsloužit událost _start, která se emituje jako vůbec první událost po spuštění smyčky událostí. Existují již některé předdefinované události (první událost a poslední událost), ostatní si musíme dle potřeb nadefinovat sami.

Jak můžeme obsluhovat události? K události může (pak je obsloužena) nebo nemusí (pak je ignorována) být přiřazen ovladač. Ovladač je obyčejný podprogram, který se po vyvolání události provede.

Jakmile již není na obzoru žádná událost, úkol vyvolá událost _stop a ukončí se.

Hello World

Na úvod si ukážeme program, který bude mít jeden jednoduchý úkol - po startu vypsat text.

Vytvoříme tedy objekt typu POE::Session, kterému předáme požadované parametry. Již víme, že po spuštění smyčky událostí se emituje signál _start. Pro něj tedy napíšeme jednoduchý ovladač. Také si napíšeme ovladač pro událost _stop, která nastane těsně před zánikem POE::Session.

Po nadefinování ukolů nesmíme nikdy zapomenout spustit smyčku událostí. To provedeme voláním POE::Kernel->run.

Takto tedy bude vypadat náš první POE program.

```
#!/usr/bin/env perl
use strict;
use warnings;
use POE;
```

```
POE::Session->create(
inline_states => {
_start => sub {
print "Hello World\n";
},
_stop => sub {
print "Bye World\n";
}
},
);

POE::Kernel->run;
```

Vlastní data a vyvolávání vlastních událostí

Podívejme se na další příklad. Napíšeme si simulaci odpočítávání a startu. Schéma programu bude stejné, akorát změníme nastavení POE::Session.

Opět musíme obsloužit událost _start. Ta bude sloužit jako inicializace. Poté musíme vyvolat jinou událost, která se bude starat o odpočítávání. Nazvěme ji například countdown.

Ale popořádku. Jak tedy vyvoláme událost? Musíme zavolat metodu yield s parametrem, kterým je název události. Metodu budeme volat nad naší instancí POE::Kernel. Tu získáme jako jeden z parametrů ovladače. Nemusíme si pamatovat který, protože již máme do programu vyexportované konstanty, které toto pořadí mapují. Jinými slovy v $_[KERNEL] máme tento objekt uložen.

Druhou podstatnou věcí jsou soukromá data procesu. Ta můžeme ukládat pomocí parametru $_[HEAP](HEAP je opět importovaná konstanta z POE), který je odkazem na datovou strukturu. My si takto uložíme počet sekund, které zbývají do startu.

Jak bude vypadat ovladač pro událost countdown? Zde musíme udělat několik věcí: Odečíst jednu sekundu, počkat a dále se zachovat podle aktuální situace. Jestliže již žádné sekundy nezbývají, došlo na start. V opačném případě opět budeme emitovat signál vyvolávající událost countdown.

Celý program může vypadat následovně.

```
#!/usr/bin/env perl
use strict;
use warnings;
use POE;

our $sekund = 10;

POE::Session->create(
inline_states => {
_start => sub {
my $kernel = $_[KERNEL];
my $data   = $_[HEAP];

$data->{sekund} = $sekund;

print "POE::Session běží\n";
print "  Start za $sekund sekund\n";

$kernel->yield("countdown");
},
_stop => sub {
print "POE::Session končí\n";
},
countdown => sub {
my $kernel = $_[KERNEL];
my $data   = $_[HEAP];

sleep 1;

$data->{sekund}--;

if($data->{sekund} > 0){
print "  Start za " . $data->{sekund} . " sekund\n";
$kernel->yield("countdown");
}
else{
print "  Odstartováno!\n";
}
}
}
);

print "POE::Kernel se spouští\n";
POE::Kernel->run;
print "POE::Kernel ukončen\n";
```

Zkusíme-li ho spustit, vypíše se postupně následující výstup.

```
$ perl start.pl
```

```
POE::Session běží
Start za 10 sekund
POE::Kernel se spouští
Start za 9 sekund
Start za 8 sekund
Start za 7 sekund
Start za 6 sekund
Start za 5 sekund
Start za 4 sekund
Start za 3 sekund
Start za 2 sekund
Start za 1 sekund
Odstartováno!
POE::Session končí
POE::Kernel ukončen
$
```

Dodejme, že ovladači pro _stop již není poskytován parametr $_[KERNEL]. Nelze tedy již vyvolávat další signály a úkol se nekompromisně ukončí.

Multitasking

Jak si POE poradí, když chce více úkolů reagovat na tentýž signál? Co když POE::Session->create voláme vícekrát? Nejprve bude užitečné vědět, že pomocí proměnné $_[SESSION] lze uvnitř ovladače získat proměnnou reprezentující aktuální objekt typu POE::Session. Pomocí metody ID získáme jeho jednoznačný identifikátor, který se nám bude hodit pro rozlišování mezi více úkoly.

```
my $session = $_[SESSION];
my $id = $session->ID;
```

Každý si nyní může sám zkusit připsat si vlastní POE::Session->create volání. V případě lenosti stačí zkopírovat stávající volání dvakrát (či vícekrát) po sobě (ideálně s doplněným ID do všech komentářů uvnitř ovladačů, aby bylo hezky vidět, který komentář je generovaný kterou session).

Parametry ovladačů

Metodě yield lze volitelně předat další parametry. K těm se pak uvnitř ovladače dostaneme opět přes speciální indexy, například $_[ARG0] (konstanty ARG0 až ARG9 zastupují čísla 10 až 19).

```
#!/usr/bin/env perl
use strict;
use warnings;
use POE;

POE::Session->create(
inline_states => {
_start => sub {
my $kernel  = $_[KERNEL];
my $session = $_[SESSION];
print "Session " . $session->ID . " spuštěna\n";
$kernel->yield("udalost", $session->ID);
},
udalost => sub {
my $session = $_[SESSION];
my $arg     = $_[ARG0];
print "Session " . $session->ID . " zpracovala signál. Obdrželi jsme parametr: $arg\n";
},
}
);

POE::Kernel->run;
```

Další příklad

Zajímavou aplikací POE je modul Acme::POE::Knee, který simuluje závody poníků.

Zvyšováním abstrakce lze z procesů a nízkoúrovňových událostí postupně budovat stále větší a větší celky. Na CPANu máme řadu modulů, které fungují například jako socket servery.

Na základě POE lze vytvářet i aplikace pro meziprocesovou komunikaci. Typicky slouží k vytváření klientů a serverů. Existuje velké množství modulů, které začínají na POE::. Ty obsahují různé nástroje, které nám usnadní vytváření většiny POE aplikací. Nejprve se tedy stručně podívejme, co za nástroje nám vlastně POE:: moduly nabízejí.

- Komponenty (POE::Component::) - komponenta je proces, který spravuje jiné procesy, může jít například o předpřipravený server
- Wheels (POE::Wheel::) - základní algoritmy, které pomáhají s praktickými úkoly napříč různými servery; například vytvoření spojení se serverem
- Drivery (POE::Driver::) - implementace IO vrstvy
- Filtry (POE::Filter::) - konvertují nějaká data z jednoho formátu do jiného

Zajímat nás budou hlavně moduly typu Wheel a Component. Za chvíli se s nimi setkáme a uvidíme více.

Příklad - server jako kalkulačka

Budeme si demonstrovat základní postup při používání různých POE:: modulů.

Zkusíme si napsat jednoduchý server a klient. Server bude zpracovávat dotazy na výpočet a posílat je zpět klientům. Na zpracování můžeme použít například nějaký kalkulátor typu bc nebo dc.

Server

Především použijeme již připravený modul POE::Component::Server::TCP. To nám výrazně usnadní práci a výsledný zdrojový kód několikanásobně zkrátí. Abychom si demonstrovali také nějaký POE::Wheel modul, použijeme POE::Wheel::Run, který nám bude externě spouštět úlohy v kalkulátoru.

Používáme-li nějaké doplňkové moduly, můžeme je uvést bez POE:: jako parametr use POE. Jinak řečeno, pokud využijeme POE::Component::Server::TCP, můžeme POE zavést tímto příkazem.

```
use POE qw(Component::Server::TCP);
```

Nadále však bude fungovat i několikanásobné volání use, kterému zde budeme dávat přednost.

Pak již bude základní kostra programu skoro stejná jako u našich příkladů z minulého dílu. Jediným drobným rozdílem bude to, že místo POE::Session budeme vytvářet instanci POE::Component::Server::TCP, která se o vytvoření POE::Session postará sama.

```
#!/usr/bin/perl
use warnings;
use strict;
use POE;
use POE::Component::Server::TCP;
use POE::Wheel::Run;

POE::Component::Server::TCP->new(
# tady bude všechno důležité
);

POE::Kernel->run;
```

Nyní tedy stačí napsat vnitřnosti serveru, to jest parametry ve tvaru klíč - hodnota. Ostatně, jak uvidíme i dále, vše v POE jsou vlastně parametry tvaru klíč - hodnota. Jedním z klíčů bude InlineStates, se kterým jsme se setkali již minule. Ten bude uchovávat některé ovladače.

```
InlineStates => {
# ovladače
}
```

Dalším klíčem bude Port, kde uvedeme, na kterém portu budeme očekávat klienty.

```
Port  => 22224,
```

Pak ještě ošetříme dvě speciální události, které mají definovaný vlastní klíč (o to se stará POE::Component::Server::TCP) - ClientConnected, ClientInput. To jsou události, které nastanou, jakmile se klient připojí, odpojí resp. pošle dotaz. Samozřejmě, že takových událostí existuje celá řada. Jejich vyčerpávající přehled je uveden v dokumentaci.

Jakmile se uživatel připojí, slušelo by se poslat mu uvítací zprávu. Vyvoláme tedy událost vitejte, kterou budeme muset později dopsat do InlineStates.

```
ClientConnected => sub {
$_[KERNEL]->yield("vitejte");
}
```

Nejkomplikovanější ovladač bude ClientInput. Všimněme si, že jako parametr označený ARG0 dostaneme vstup od klienta, který můžeme dále zpracovávat.

Nás nyní bude zajímat hlavně to, jak použijeme avizované POE::Wheel::Run, které se má postarat o zavolání kakulátoru. Předně si ujasněme, jaký shellový příkaz budeme vlastně volat. Zvolme si například kalkulátor bc, který příkazy zpracovává takto.

```
$ echo '2+3' | bc -l
```

Zavoláme tedy POE::Wheel::Run a předáme mu šablonu pro tento příkaz pod klíčem Program. Ještě musíme předat další dva parametry s hodnotami - StdoutEvent(parametrem bude jméno ovladače, který bude volán, jakmile náš příkaz vyprodukuje výstup na STDOUT) a CloseEvent (to je zas znamení, že je výstup dat z volaného příkazu u konce).

Takto tedy může vypadat ošetření v ovladači ClientInput.

```
ClientInput => sub {
my $data    = $_[HEAP];
my $vypocet = $_[ARG0];

if ($data->{cmd} or $vypocet =~ m/\\'/) {
return;
}

$data->{cmd} = POE::Wheel::Run->new(
Program      => "echo '$vypocet' | bc -l",
StdoutEvent  => "odpoved",
CloseEvent   => "konec",
);
},
```

Nyní nám zbývá dopsat ovladače vitejte, odpoved a konec. Poznamenejme nejprve, že data klientovi odešleme metodou put. První dva budou prostým odesláním dat klientovi. Ten třetí uklidí nepořádek, který v $_[HEAP]->{cmd} nechal POE::Wheel::Run.

```
InlineStates => {
odpoved => sub {
my $odpoved = $_[ARG0];
$_[HEAP]->{client}->put($_[ARG0]);
},
konec => sub {
delete $_[HEAP]->{cmd};
},
vitejte => sub {
$_[HEAP]->{client}->put("Vítejte na POE serveru");
},
```

```
},
```
Dodejme, že u tohoto serveru uživatel musí zadávat korektní dotazy, jinak se odpovědi nedočká.

Připojení na server

K serveru se samozřejmě můžeme připojit běžnými nástroji jako například telnet. Spustťme tedy server a na nějaké jiné konzoli spustíme telnetového klienta.

```
$ telnet localhost 22224
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Vitejte na POE serveru
5*7
35
```

Klient

Ačkoliv se můžeme připojit telnetem nebo jinými běžnými klienty, napíšeme si také svého vlastního. Ten bude opět používat POE.

Klient bude fungovat na základě podobné logiky jako server. Pro usnadnění použijeme komponentu POE::Component::Client::TCP. Kostra aplikace opět vypadá následovně. Rovnou přidáme zřejmé parametry pro port a hostitele.

```
#!/usr/bin/perl
use warnings;
use strict;

use POE;
use POE::Component::Client::TCP;

POE::Component::Client::TCP->new(
RemoteAddress => "localhost",
RemotePort => 22224,
# tady bude náš klient
);

POE::Kernel->run();
```

POE::Component::Client::TCP opět sám generuje řadu událostí, na které je snadné reagovat. Například událost Connected se objeví, jakmile jsme připojeni k serveru.

```
Connected => sub {
print "Jsme připojeni\n";
}
```

Nás však bude zajímat především událost ServerInput, ke které dojde, jakmile nám server pošle data. Data opět máme dostupná pod proměnnou $_[ARG0]. Data tedy přijmeme, zobrazíme (poprvé to bude uvítací zpráva, následně pak výsledky na naše dotazy) a poté můžeme provádět dotazy. Na to si vytvoříme vlastní ovladač proved.

```
ServerInput => sub {
my $odpoved = $_[ARG0];
print "Výsledek: $odpoved\n";
$_[KERNEL]->yield("proved");
},
```

Uvnitř proved se uživatele nejprve zeptáme na příkaz a poté ho pošleme serveru. Ještě můžeme například ošetřit, zda uživatel náhodou nechce program ukončit.

```
InlineStates => {
proved => sub {
print "příkaz> ";
my $cmd = <STDIN>;
if($cmd =~ m/\A(?:quit|q|exit|konec)/){
$_[KERNEL]->yield("shutdown");
return;
}
$_[HEAP]{server}->put($cmd);
}
}
```

Závěr

Nyní tedy máme jak server tak klienta. Ke stažení jsou zde: server.pl, klient.pl. Můžeme si zkusit spustit server a jednoho nebo několik klientů. S klientem lze pracovat například takto.

```
$ perl klient.pl
Jsme připojeni
Výsledek: Vitejte na POE serveru
příkaz> scale=50;4*a(1)
Výsledek: 3.14159265358979323846264338327950288419716939937508
příkaz> 2+3
Výsledek: 5
příkaz>
```


Na Perl 6 jsme čekali již před začátkem tohoto seriálu. Stále čekáme. Ale bude stát za to.

Perl 6 je rodící se skriptovací jazyk, který vzniká na základě Perlu 5. Bude dalším stupněm nejen ve vývoji Perlu, ale půjde o velký krok v evoluci programovacích jazyků obecně. Jeho hlavními filozofiemi jsou hesla "make easy things easier" a "keep Perl 6 Perl".

Vznik

Původním cílem bylo odstranit některé nešvary z Perlu 5. Larry Wall ve své přednášce na Perl Conference prohlásil: "Easy things should stay easy, hard things should get easier, and impossible things should get hard".

Základní myšlenky Perlu 6 byly na této konferenci tedy položeny již v červenci roku 2000. Následovala veřejná diskuze ve formě několika RFC (request for comments).

Poté Larry Wall napsal několik článků (zvaných Apokalypsy), kde představil, které změny by měl Perl 6 obsahovat, které ne a které možná. Číslování Apokalyps je shodné s číslováním kapitol v knize Programming Perl. V každé Apokalypse je tedy popsáno, co by se mělo z dané kapitoly změnit.

Z Apokalyps poté vznikly vychytáním sporů mezi jednotlivými Apokalypsami Synopses, ze kterých vychází specifikace jazyka. To je zásadní rozdíl oproti Perlu 5. V Perlu 5 specifikace neexistovala a museli jsme se spokojit s popisem chování (fakticky byla tedy specifikace dána zdrojovými kódy interpretu Perlu 5 a dokumentace byla pouze jejich popisem). Nyní je v Synopses jasně deklarováno, jak by se Perl měl chovat a podle nich bude interpret fungovat. Pokud se tedy bude lišit specifikace a chování interpretu Perlu 6, bude třeba interpret opravit.

Také existuje série tzv. Exegesis, což jsou výklady Apokalyps (na příkladech je zde vysvětleno nové chování).

Interprety Perlu 6

Zásadní změnou oproti Perlu 5 bude to, že nově bude moci existovat více implementací interpretu jazyka.

Existuje několik projektů, které vyvíjejí interpret Perlu 6, avšak žádný ještě nedospěl do svého konce. Již mnohokrát bylo anoncováno datum, kdy Perl 6 spatří světlo světa, ale zatím nabírá stále nová zpoždění. Od jisté doby se manažeři projektu vyjadřují slovy "Perl 6 has no schedule".

Již nyní můžeme pomocí těchto projektů zkoušet syntaxi Perlu 6. Stačí, když si stáhneme nějaký z nabídky aktuálně vyvíjených interpretů. Jednou z možností je instalace Rakudo Perlu, po které nám vznikne spustitelný soubor perl6. Nakopírujme ho do některého z adresářů v proměnné prostředí $PATH. Nyní můžeme spouštět naše programy v interpretu perl6 (je třeba si uvědomit, že řada věcí je zde ještě neimplementovaných).

```
$ perl6 -v
This is Rakudo Perl 6, version 2010.11 built on parrot 2.10.1Null PMC access in get_bool()
current instr.: 'perl6;Perl6;Compiler;version' pc 326465 (src/gen/perl6-grammar.pir:16967)
called from Sub 'perl6;PCT;HLLCompiler;command_line' pc 1855
(compilers/pct/src/PCT/HLLCompiler.pir:917)
called from Sub 'perl6;Perl6;Compiler;main' pc 326404 (src/gen/perl6-grammar.pir:16945)
$ perl6 hello_world.pl6
```

Také lze spustit perl6 bez argumentu. Objeví se perl6 shell, který se hodí pro naše další pokusy.

```
$ perl6
> note 123
123
>
```

Vztah Perl 6 s předchozími verzemi

Perl 6 nebude s předchozími verzemi zpětně kompatibilní. To znamená, že zdrojový kód programu napsaného v Perlu 5 nemusí fungovat (nebo může fungovat jinak). Důležité však je, že archiv CPAN bude i v Perlu 6 nadále k dispozici, neboť bude možné používat Perl 6 ve speciálním módu, kde kompatibilita bude.

Změny oproti Perlu 5

Již bylo řečeno, že asi nejzlomovějším rozdílem je existence specifikace. Na základě ní tedy lze kdykoliv vytvořit interpret jazyka. Avšak kdyby šlo jen o ni, tak by se z praktického hlediska v Perlu 6 nic nezměnilo. Věnujme se tedy dále změnám v syntaxi jazyka. Změn je obrovské množství. Máme řadu nových funkcí, operátorů (ačkoliv "Operators are just functions with funny names and syntax"), konstrukcí a změn v syntaxi. Zde si představíme jen ty nejviditelnější změny. Následující text by měl sloužit zejména pro orientaci, co je v Perlu 6 možné, nikoliv jako komplexní příručka. Ta by totiž vydala na desítky dílů.

Koho zajímá více, než jen letmý pohled, měl by se podívat do specifikace jazyka, kde najde skutečně všechno.

Použití kódu Perlu 5

I v Perlu 6 budeme moci používat nespecifikvaný kód Perlu 5.

```
use v5;
{
# Perl 5 kód
}
use v6;
{
# Perl 6 kód
}
```

Výpis textu

Existuje příkaz say jako alternativa print, která navíc vytiskne znak nového řádku.

```
say "Hello world";
```

Sigil se nemění pro získání prvku pole

Sigil (znak, který označuje začátek proměnné) zůstává v Perlu 6 stejný i po operaci "prvek pole" nebo "prvek hashe". Konkrétněji, následující příklad bude běžným zápisem (v Perlu 5 bychom měli změnit u @pole[2]zavináč na dolar).

```
my @pole = (1 .. 10);
my $treti_element = @pole[2];
```

Twigils

Také se objevují tzv. twigils. To jsou proměnné se dvěma sigil znaky. Vyskytují se následující twigils:

Twigil	Význam
.	přístup k datům objektu, veřejná data
!	přístup k datům objektu, soukromá data
?	pro data známé v době kompilace; například $?LINE místo starého __LINE__

*	například $*IN, $*OUT, $*ENV, $*PID, $*ARGV
^	proměnná jako poziční parametr, používá se i například jako $^a, $^b, $^c atd. (řazené lexikograficky) místo starých $a a $b
:	pojmenovaný parametr
<	$<*zachyceni*> zachytávání zapamatovaných hodnot u regulárních výrazů; využijeme v příštím dílu
=	POD proměnná
~	subjazyky (například $~MAIN, $~Q, $~P5Regex)

Deklarátory

Podívejme se, jakými způsoby lze deklarovat proměnné.

Deklarátor	Význam
my	lexikálně vymezené proměnné
state	lexikálně vymezené perzistentní proměnné
our	balíkové proměnné
has	atributy objektů
anon	anonymní jména
augment	přidává definice (například regulární výraz do gramatiky, uvidíme v příštím dílu)
supersede	mění existující definice

Končí local, ale lze používat temp.

Méně závorek

U často používaných konstrukcí mohou být vynechávány kulaté závorky. Cykly a podmínky tak lze psát v tomto stylu (dříve to bylo možné pouze v postfixovém zápisu).

```
if 5 < 10 {
say "OK";
}
```

Definice pole

Podobně je to s definicí pole. Operátor čárka se zde již chová jinak (v Perlu 5 by následující kód měl úplně jiný význam - toto lze uvést jako důkaz nekompatibility Perlu 6 s předchozími verzemi).

```
@pole = 1, 2, 3;
```

Pro vytváření pole řetězců nyní máme místo qw novou syntaxi.

```
@retezce = <a b c>
```

Nově lze snadno na pole konvertovat proměnnou s mezerami.

```
@retezce = <<a $prvky_oddelene_mezerou c>>
```

Exsituje také speciální operátor Q pro uvozování, který přijímá různé modifikátory.

Líné vyhodnocování

Lze vytvořit pole s "nekonečně mnoha" prvky uvedením hvězdičky nebo Inf jako horní meze. Následující kód bude fungovat.

```
my @pole = 0 .. Inf;
say @pole[10000];
```

Obsah pole se vyhodnocuje líně. Dokud ho nepotřebujeme, nebude se počítat. Příslušná část se vyhodnocuje, jakmile je to potřeba (čtení hodnoty).

S tím souvisí i zavedení konstrukce gather...take. Vytvoříme pole @nadruhou všech čtvercových čísel od 0 do nekonečna.

```
@nadruhou = gather for 0 .. Inf {
take $_ ** 2;
};
```

Celý proces se zde vyhodnocuje líně, což znamená, že příslušné hodnoty prvků pole @nadruhoubudou dopočteny teprve až dojde k jejich přečtení. Díky tomu není plýtváno cennými jednotkami CPU, pokud výpočet nebude potřeba.

Hybridní typový systém

V Perlu 5 jsme se vůbec nemuseli starat o datové typy používaných proměnných. Pracovali jsme s různými druhy dat a Perl si je v tichosti sám podle nějakých pravidel konvertoval.

To v Perlu 6 zůstává, ale nově bude existovat i možnost explicitního uvedení datového typu programátorem při deklaraci proměnné.

```
#!/usr/bin/env perl
my Str $retezec1 = "1";
my Str $retezec2 = "cokoliv";
my Int $cislo = 1;
say $retezec1;          #tiskne 1
say $retezec1 + $cislo; #tiskne 2 - také v pořádku, k vyhodnocení se použije konverze
                        # řetězce na číslo
say $retezec2 + $cislo; #tiskne 1 - také v pořádku, obsah $retezec2 byl vyhodnocen jako 0
$cislo = "cokoliv";     #chyba - nelze přiřadit řetězec do celočíselné proměnné
```

K dispozici máme následující datové typy.

Kód	Název
Any	cokoliv (dynamické chování jako v Perlu 5; takto se chovají proměnné bez deklarace)
Int	celé číslo
Str	řetězec
Num	číslo
Rat	racionální číslo (zlomek)

Bool	true/false
NejakaTrida	objekt daného typu

Výraz lze testovat na datový typ pomocí v Perlu 6 oblíbeného operátoru "odpovídat si", který se zapisuje ~~. Například výraz 1 ~~ Int je pravdivý.

Lze definovat vlastní datové typy. Například datový typ pro sudá čísla bychom definovali takto.

```
subset Even of Int where { $_ % 2 == 0 }
```

Rozepisování datových struktur

Pro pohled do nitra datové struktury jsme se naučili používat modul Data::Dumper. Nyní ale pro základní orientaci postačí následující.

```
my $a = [(1, 2), (3, 4), [5, 6]];
say $a.perl; # vytiskne řetězec [(1, 2), (3, 4), [5, 6]]
```

Přetěžování kontextu

V Perlu 5 máme funkci scalar, která vynutí skalární kontext. V Perlu 6 lze již vynutit libovolný kontext. Následující tabulka uvádí, jak vynutit jednotlivé (již specializovanější než jen prázdný, skalární a pole) datové typy.

Syntaxe	Význam
~něco	řetězec
?něco	boolean
+něco	číslo bez znaménka
-něco	číslo se znaménkem
$(něco)	skalár
@(něco)	pole
@@(něco)	řez polem (narozdíl od kontextu pole řezy vzájemně neinteragují)
%(něco)	hash

Díky této změně budeme moci zjednodušit spoustu zápisů. Jak bychom například v Perlu 6 zjistili neprázdnost pole?

Podprogram MAIN

Podprogram s názvem MAIN si automaticky vezme parametry z příkazového řádku. Ukažme si, jak to lze použít.

```
sub MAIN ($arg1, $arg2) {
say "Parametry: $arg1 $arg2";
}
```

A teď se podívejme, jak bude program reagovat na volání.

```
$ perl6 main.pl 1 2
Parametry: 1 2
$
```

Co když uvedeme jiný počet parametrů? Pak se automaticky zavolá podprogram USAGE, který implicitně vytiskne následující zprávu.

```
$ perl6 main.pl 1 2 3 4 5
Usage:
add.pl arg1 arg2
$
```

Pokud místo klíčového slova sub uvedeme multi, lze vytvořit různé MAIN pro různé prototypy. Také lze zadávat parametry --arg1 --arg2.

Pojmenování parametrů v hlavičce podprogramů, konec @_

V hlavičce podprogramů je třeba v Perlu 6 pojmenovat parametry, protože již nebudeme používat proměnnou @_. Podprogramy tak budou vypadat následovně.

```
sub nadruhou($cislo){
return $cislo ** 2;
}
```

I zde je možné explicitně uvést datový typ.

```
sub nadruhou(Rat $cislo){
return $cislo ** 2;
}
```

Důležité však je, že tyto parametry budou pouze pro čtení a při pokusu o změnu jejich hodnoty bude vyvolána výjimka. Toto chování lze změnit atributem is rw.

Existují tři módy předávání parametrů:

- Poziční - Klasické parametry u nichž záleží na pořadí.
- Pojmenované - Jsou předávány pomocí svého jména, nezáleží na seřazení a deklarují se znakem :
- Srkající (slurpy) - Tyto proměnné absorbují všechny neabsorbované parametry a označují se znakem *. Pole takto označené absorbuje všechny poziční jinam nezařazené parametry a hash pohltí ty pojmenované.

Parametry mohou být povinné i nepovinné. Standardně jsou povinné a pro změnu chování je třeba uvést za pojmenovaný parametr znak ! a za poziční znak ?. Srkající parametry jsou vždy nepovinné.

Podívejme se na příklad, který ukazuje chování dvou různých volání.

```
sub funkce($pozicni, :$pojmenovany, *@srkajici){
...
}
funkce(1, 2, 3);                    # $pozicni=1, $pojmenovany=2, @srkajici=(3)
funkce(:pozicni<1>, :pojmenovany<2>, 3); # $pozicni=1, $pojmenovany=2, @srkajici=undef
```

Enumerace

Pomocí následujícího kódu lze vytvářet něco jako speciální datový typ pro konečné množství hodnot.

```
enum Rok (2011 2012 2013);
```

Hodnotu pak reprezentujeme jako Rok::2011.

Změny v cyklech

for a while cyklus můžeme psát v nové syntaxi. V jedné iteraci můžeme ze seznamu převzít i více než jeden prvek.

```
for @seznam -> $i, $j {
say "Máme prvky $i, $j.";
}
```

Céčkovský for cyklus je nyní přejmenován na loop. Cyklus loop {} se dá také použít jako nekonečná smyčka.

Operace se soubory

Koncept handlerů z Perlu 5 samozřejmě funguje dále. Perl 6 ale nabídne několik zajímavých usnadnění. Podívejme se, jak budeme moci otevřít soubor a číst z něj.

```
my $fh = open "nejaky_soubor";
print $fh.get;  # tiskne řádek
print $fh.getc; # tiskne znak
$fh.close
```

Jak vytvoříme nový soubor (analogie unixového touch)?

```
open("novy_soubor", :w).close
```

A co zápis?

```
open "nejaky_soubor", :w;
$fh.say("toto zapíšeme do souboru");
$fh.print("toto taky");
```

Za zmínku stojí též funkce slurp s názvem souboru jako parametrem, která vrátí obsah souboru.

Změnilo se také testování souborů. Nyní budeme testovat pomocí nového operátoru ~~ pro "odpovídání si", se kterým můžeme porovnávat skoro všechno se vším. Ještě se s ním setkáme později. Například existenci souboru ověříme následovně.

```
if ("jmeno_souboru".IO ~~ :e){
say "existuje";
}
```

Změny v OOP

V Perlu 5 vznikaly objekty pomocí funkce bless. To je možné stále, ale existuje nově také paralelní model, který je inspirován i již existujícím Moose.

Okamžitě viditelnou je změna operátoru šipky na tečku při přístupu k metodám a atributům.

Takto bychom mohli začít psát třídu pro manipulaci s komplexními čísly. Musíme deklarovat atributy pro reálnou a imaginární část pomocí deklarátoru has. Atributy se označují uvedením tečky hned po sigil znaku.

```
class KomplexniCislo is rw {
has $.re;
has $.im;
}
```

Takto potom vytvoříme číslo 1 + 2i.

```
my $komplex = KomplexniCislo.new(re => 1, im => 2);
```

Dále s ním samozřejmě můžeme manipulovat.

```
$komplex.re = 3;
$komplex.im = 4;
say "Nová hodnota: ", $komplex.re, " + ", $komplex.im, "i"
```

Nové OOP bude podporovat role, které budou fungovat podobně jako v Moose. Role jsme již probrali dříve a proto zde jen uveďme, jak se budou v Perlu 6 používat.

```
class Organizmus {...}
class Clovek is Organizmus {...}
role Vareni {...}
class Kuchar is Clovek does Vareni {...}
```

Nyní bychom mohli tuto roli přiřadit konkrétnímu objektu.

```
my $clovek = new Clovek;
$clovek does Vareni;
```

Přípojky (junctions)

Přípojky jsou hodnoty, které se mohou vyskytovat ve více stavech. Jde o velmi zajímavou myšlenku, která se poprvé v Perlu objevila jako modul Quantum::Superpositions.

Zadefinujme proměnnou, která se bude vyskytovat v 5 stavech. Nejprve se podívejme na přípojku typu "any", která reprezentuje některou z hodnot.

```
my $mnozina1 = 0 | 1 | 2 | 3 | 6;
```

Nyní krátce zkoumejme vlastnosti této proměnné. Můžeme zkusit testovat, jaké číslo tato proměnná reprezentuje. Následující test dopadne úspěšně.

```
if 2 == $mnozina1 {
say "2 je OK";
}
```

Zajímavé je, že úspěšně dopadne i následující test. U tohoto typu přípojky je totiž pro rovnítko lepší intepretací operace "být prvkem množiny".

```
if 2 == $mnozina1 == 3 {
say "4 je OK";
}
```

Vybereme-li číslo mimo množinu, test neuspěje.

```
if 5 == $mnozina1 {
say "5 je OK";
}
```

Leckoho napadne, jak se chová $mnozina1 při běžných aritmetických operacích.

```
say $mnozina1 * 2;
```

Asi ale nikoho nepřekvapí, že výsledným výrazem je další přípojka, protože je to jediný konzistentní způsob. Výraz se tedy vyhodnotí na any(0, 2, 4, 6, 12).

Ještě existuje druhý typ přípojky a to typ "all". Ta vyjadřuje všechny hodnoty.

```
my $mnozina2 = 1 & 2 & 5;
```

Rozdíl si můžeme ukázat například při porovnávání. Když porovnáváme dvě "any" přípojky mezi sebou, výraz je pravdivý, pokud existuje v první přípojce prvek, který je obsažen i ve druhé. Porovnáváme-li "all" přípojky, musí mít pro pravdivou hodnotu všechny stavy identické.

Při porovnávání "any" a "all" přípojky musí existovat každý stav "all" přípojky i v "any" přípojce. Následující výraz je potom nepravdivý.

```
$mnozina1 == $mnozina2
```

Dodejme, že nezávisí na pořadí prvků v přípojkách.

Měli bychom připomenou i to, že bitové operátory (jako které by přípojky identifikoval Perl 5) mají nyní novou syntaxi. Více v dokumentaci.

Poznamenejme, že s přípojkami souvisí ještě to, že výraz obsahující all(@pole) se může začít samovolně zpracovávat ve více vláknech. Uvedení all se totiž chápe tak, že nezáleží na pořadí, ve kterém se budou výrazy zpracovávat.

Využití může vypadat i takto.

```
if $p ~~ all(@regularni_vyrazy) {...}
```

Makra

Na okraj zmiňme také makra. V Rakudo Perlu zatím nejsou implementované, ale v budoucnu bude fungovat kód následujícího typu.

```
macro hello($what) {
q:code { say "Hello { {{{$what}}} }" };
}
```

Další změny stručně

- Pomocí operace transliterace lze nahradit tr///. Příkaz say "ABCA".trans("BCA" => "ZYX"); vypíše XZYX. Lze používat dvoutečkový operátor rozsahu.
- Pro přehlednější výpis pole lze použít say ~@pole (vizte kontexty).
- Funkce reverse dělala v Perlu 5 dvě věci - z pole udělala seznam pozpátku a v řetězci invertovala pořadí prvků. Nyní bude invertovat pouze seznamy.
- Na invertování řetězce máme novou funkci flip.
- Novou operací je inverze hashe. V hashi můžeme zaměnit klíče s hodnotami zápisem %invert = %hash.invert.
- Pro práci s časem existuje funkce time vracející POSIXový čas a funkce now, která vrací objekt typu Instant, tj. tvaru Instant:2011-06-26T17:54:44.343521Z.
- Počet prvků pole zjistíme pomocí @pole.elems, poslední index pomocí @pole.end.
- Formáty končí - tyto věci nepatří do jádra jazyka a mohou být řešeny externími knihovnami.
- Přetížit standardní chování funkce lze nejlépe pomocí multi sub GLOBAL::funkce.
- Končí mimo jiné tyto operátory a funkce:
  - % je zobecněné.
  - Funkce pro DBM, jsou nahrazeny externími moduly, například DB_File.
  - each - nově máme metodu .pairs.
  - length - nově je třeba uvést jednotky, máme metody .chars a .bytes.
  - pos - lze použít $/.
  - ref - na typ můžeme testovat pomocí ~~ TYP.
  - tie - lze nahradit novými proměnnými, makry atd.
- Končí autokvotace hashů. Nelze tedy psát %hash{klic}. Nově se použije buď %hash{'klic'} nebo %hash<klic>.
- Řada vestavěných funkcí se stává metodami, například sort, print atd.
- Místo blokového eval máme nově try, v němž můžeme použít blok CATCH.

Perl (147) - Perl 6 - regulární výrazy, nové operátory

V minulém dílu jsme opomněli dvě zásadní témata. Regulární výrazy prošly od Perlu 5 obrovskou změnou. Podobně bychom mohli mluvit o nových operátorech.

Regulární výrazy (rules)

Na začátek povězme, že staré regulární výrazy lze nadále používat. Syntaxe pro to je následující.

```
if $p ~~ m:Perl5/^\w\d[a-d]$/ {
say "OK";
}
```

Avšak regulární výrazy prošly revolučními změnami. V Perlu 5 mají dle Larryho Walla několik zásadních nedostatků. Je určitě zajímavé zamyslet se nad následujícími body.

- too compact and cute
- too much reliance on too few metacharacters
- little support for named captures
- little support for grammars
- poor integration with real language

Perl 5 podporoval tzv. regexes, což jsou regulární výrazy rozšířené oproti jejich formální definici (my v seriálu nejsme úplně korektní a tyto pojmy zaměňujeme). Perl 6 bude obsahovat novou soustavu regulárních výrazů pod názvem rules (pravidla). Pro detailní informace je nejvhodnější nahlénout do specifikace.

V Perlu 6 je syntaxe a možnosti regulárních výrazů hodně odlišná od předchozích verzí. Mění se i filozofie, s jakou je k regulárním výrazům přistupováno. V Perlu 5 byly regulární výrazy od zbytku programu dost oddělené - byl to speciální řetězec, jehož význam se řídil vlastními pravidly a na první pohled bylo jasně vidět, že regulární výraz je něco jiného než zbytek programu. To se nyní změnilo a regulární výrazy jsou do jazyka daleko více zabudované.

Co se nezměnilo

Nejzákladnější vlastnosti se neliší.

- Znak | pro alternativy
- Čísla, písmena, podtržítko a escapované znaky pomocí \ matchují doslovně
- (...) funguje jako zachycení řetězce

- Pro kvantifikátory zde máme znaky ? * + ?? *? +?, již však nemáme kvantifikátor {*od*, *do*}

Drobné změny

Téměř vše ostatní se ale změnilo. Podívejme se, co se stalo s několika často používanými konstrukcemi.

- Místo nezachytávacích závorek (?:...) nyní budeme používat [...], které dosud sloužily jako závorky pro množinu znaků.
- Pro vkládání kódu použijeme <?{...}>
- Nově nezáleží na bílých znacích uvnitř regulárního výrazu, tj. je automaticky zapnut modifikátor x
- Mění se syntaxe pro pozitivní/negativní pohled dopředu/dozadu. Pro pozitivní resp. negativní pohled dopředu použijeme <before ...> resp. <!before ...>. Podobně pro pohled dozadu s after.
- Kvantifikátory pro rozsahy mají tvar **{2 .. 7}, **{2 .. 7}?.
- Místo operátorů =~ a !~ máme nově ~~ a !~~.
- ^^ je nová kotva pro začátek řádku, $$ pro konec řádku
- Pojmenované zachytávání se děje následující speciální konstrukcí: /Cislo: $<jmeno_promenne>:=(\d+)/. Potom budeme mít zachycená data v proměnné $jmeno_promenne a nebudeme muset používat $0. Také je vhodné zmínit, že výsledkem zachytávání bude zanořená datová struktura, nikoliv pouze seznam hodnot.
- Tečka nově zastupuje i znak nového řádku

Formální gramatiky, modifikátory

V Perlu 6 lze daleko intuitivněji definovat pravidla pro formální gramatiky. Například, mějme jazyk, který můžeme chápat jako množinu slov $L = \{a^n b a b^n \mid n > 0\}$ (mimochodem, nejde o regulární jazyk). To lze v Perlu přepsat do pravidla následovně pomocí deklarátoru rule (které se používá k deklaraci pojmenovaného nebo anonymního regulárního výrazu).

```
rule S {(a+) ba (b+) <{$0.elems == $1.elems}>}
```

Zde je jiný způsob. Velmi přehledně lze přepisovat gramatiky. Pravidla pro tvorbu slov jsou v našem jazyku následující.

```
S -> a S b
S -> b a
```

Totéž tedy bude dělat toto pravidlo.

```
rule S {a S b | b a}
```

Dále existují také deklarátory regex, rx a token, které se liší v tom, jaké modifikátory jsou automaticky aktivní. Máme na výber z mnoha modifikátorů. Uvádějí se uvozené dvojtečkou za jméno pravidla (tj. podobně jako atributy u podprogramů) nebo před regulární výraz (je-li anonymní, například m:i:g/hledane_slovo/).

Zde je několik příkladů modifikátorů.

Modifikátor	Popis
:P5, :Perl5	zapne starou verzi regulárních výrazů
:ratchet	bez backtrackingu
:i, :ignorecase	nehledí na velikost písmen
:m, :ignoremark	nehledí na různé druhy interpunkce
:g, :global	provede porovnání vícekrát
:ov, :overlap	provede porovnávání vícekrát, ale navíc i s překrývajícími se výskyty
:s, :sigspace	bílé znaky nebudou ignorovány
:7x, :x(7)	matchovat 7×
:7th, :nth(7)	7. match
:nth(2,4,6...*)	matchuje sudé výskyty
:ex, :exhaustive	matchuje všemi způsoby (přesvědčit se lze v proměnné @())

Perl 6 navíc umožňuje definici vlastních modifikátorů. Pro obsáhlejší informace o modifikátorech lze doporučit nahlédnutí do dokumentace.

Klíčové slovo grammar deklaruje jmenný prostor pro pravidla. Názorněji si představme grammar jako analogii package. package obsahuje metody, podobně grammar obsahuje nějaké regulární výrazy.

Uveďme si hodně umělý příklad na validaci emailové adresy, na kterém ale uvidíme, jak se výše uvedené používá. Používá se Backus-Naurova normální forma.

```
grammar Email::Simple {
regex email {<uzivatel> @ <server> \. <koncovka>}
token uzivatel {[\w\.\-]+}
token server {[\w\.\-]+}
token koncovka {cz|com|info}
}
```

Nyní se můžeme vně jmenného prostoru například na regex email odvolávat jako na Email::Simple::email. Jak tedy ověříme správnost emailu?

```
if $email ~~ /Email::Simple::email/ {
say "OK";
}
```

Nové operátory

Podívejme se, jaké nové operátory nám Perl 6 přináší. Již jsme se s některými setkali v minulém dílu a nyní se věnujme dalším. Na úvod je dobré se alespoň letmo podívat na kompletní přehled operátorů, kde uvidíme desítky úplně nových.

Výraz pro vícenásobné porovnávání

Nově lze zřetězit operátory pro porovnávání. Lze tak vytvořit následující podmínku.

```
if (1 <= 5 < 10){
say "OK";
}
```

Nové operátory pro kartézský součin a operace po složkách

Operátor X je tzv. metaoperátor, což je velmi zajímavá věc. Základní použití je následující.

```
say ((1, 2) X (4, 5)).perl; # tiskne ((1, 3), (1, 4), (2, 3), (2, 4))
```

Použití však je daleko širší, neboť lze specifikovat, jakým operátorem mají interagovat každé dva prvky. Implicitně je to čárka. Lze ale použít i jiné operátory. Operátor uvedeme za X.

```
say ((1, 2) X~ (3, 4)).perl   # tiskne ("13", "14", "23", "24")
say ((1, 2) X* (3, 4)).perl   # tiskne (3, 4, 6, 8)
say ((1, 2) X== (3, 4)).perl  # tiskne (Bool::False, Bool::False, Bool::False, Bool::False)
```

Stojí za zamyšlení, jak bychom libovolný z uvedených příkladů napsali v Perlu 5.

Pro operace po složkách lze stejným způsobem použít operátor Z. Vždy spolu interagují *n*tý prvek v pravém seznamu a *n*tý prvek v levém seznamu. Podívejme se, jak efektně můžeme v Perlu 6 sčítat vektory nebo tvořit hashe.

```
say ((1, 2) Z+ (3, 4)).perl                # tiskne (4, 6)
%hash = @zavodnici Z=> @jejich_osobni_rekordy   # vytvoří hash
```

Definice vlastních operátorů

V Perlu 6 máme následující typy operátorů.

Operátor s operandy	Typ
7 + 7	infix
+7	prefix
$p++	postfix
<7 8 9>	circumfix
@p[7]	postcircumfix

Jak bychom například vytvořili a použili operátor pro zaokrouhlování (za předpokladu, že máme funkci round)?

```
multi sub prefix:<°> (Rat $arg) {
return round($arg);
}
say °3.14;
```

Měli bychom ale také specifikovat prioritu právě definovaného operátoru a asociaci. To uděláme v případě priority pomocí slov equiv, tighter, looser vzhledem k již existujícím operátorům a pomocí assoc s parametrem left, right nebo none. Například takto.

```
multi sub infix:<°> is equiv(&infix:<*>){ ...  }
multi sub infix:<°> is tighter(&infix:<*>){ ... }
multi sub infix:<°> is looser(&infix:<*>){ ... }
multi sub infix:<°> is assoc("left"){ ... }
```

Pomocí multi můžeme také přetěžovat již existující operátory.

Generování posloupností čísel

Perl 6 zavádí nový třítečkový operátor pro rozsahy s líným vyhodnocováním. Díky němu můžeme elegantně generovat různé aritmetické a geometrické posloupnosti.

```
1, 2, 4 ... *     # generuje posloupnost mocnin dvou
1, 3, 9 ... *     # generuje posloupnost mocnin tří
0, 2 ... *        # generuje posloupnost sudých čísel
1, 3 ... *        # generuje posloupnost lichých čísel
5, 4 ... *        # generuje posloupnost čísel od 5 do minus nekonečna
1.1, 1.2 ... 5.3  # generuje posloupnost čísel od 1,1 do 5,3 s krokem 0.1
```

Co když chceme omezenou posloupnost, ale nevíme z hlavy poslední člen? Co například mocniny dvou do 10000?

```
1, 2, 4 ... 10000
```

To bohužel fungovat nebude, protože vygenerovaný člen je vždy porovnáván s posledním na rovnost. Příklad tedy vygeneruje nekonečnou posloupnost. Avšak existuje trik, který ji ukončit dokáže, i když poslední člen nevíme.

```
1, 2, 4 ... * >= 10000
```

Místo >= lze použít i jiný operátor. Tato konstrukce je zkratkou za uzávěr -> $a {$a >= 10000}. Totéž lze nezkráceně napsat takto.

```
1, 2, 4 ... -> $a {$a >= 10000}
```

Když chceme generovat posloupnost, která není ani aritmetická ani geometrická, musíme ji definovat jako lambda funkci. Vzpomeňme si, jak jsme psali podprogram na výpočet Fibonacciho posloupnosti. Takto ji přiřadíme do pole v Perlu 6.

```
my @fibonacci := 0, 1, -> $a, $b {$a + $b} ... *;
```

S trochou magie lze totéž přepsat takto.

```
my @fibonacci := 0, 1, * + * ...^ *
```

Výsledek po operaci ... lze samozřejmě přiřadit do pole.

```
@mocniny_dvou = 1, 2, 4 ... *;
```

Prvních 6 elementů vytiskneme takto.

```
say @mocniny_dvou[^6];
```

Další změny v operátorech

Podívejme se v bodech na několik zajímavých změn.

- Je zaveden nový feed operátor <==, který dělá současné používání příkazů typu map, grep atd. podstatně čitelnější.
- Operátor := se používá k ztotožnění dvou proměnných.
- Operátor =:= rozhodne, zda dvě proměnné ukazují na totéž.
- Operátor ~ se používá pro zřetězení.
- Podmínkový operátor má nyní novou syntaxi: podmínka ?? v_případě_true !! v_případě_false.
- Operátor pro identitu: ===, eqv.
- Nové relační operátory pro negaci: !===, !=:=, !==, !eq, !eqv.
- Zaujmout by mohly též operátory pro minimum a maximum, které lze použít následovně: 5 min 7 max 2. Samozřejmě existují i odpovídající max= a min=.
- Máme dva replikační operátory: x je pro skaláry a xx pro seznamy.

Na odlehčení věnujme jeden díl kultuře Perlu. Představíme si několik kultovních projektů, které jsou pevně spjaté s jeho historií.

Jedním z měřítek úspěšnosti jazyka je bezesporu to, jak velkým terčem se stane pro jeho kritiky (nebo zastánce). Díky bezpočtu neotřelých přístupů patří Perl mezi nejdiskutovanější jazyky vůbec. Narážky na jeho syntaxi můžeme najít na každém aspoň trochu souvisejícím diskuzním fóru.

Na internetu existuje řada stránek s názvem Perl Humour, které odkazují na zajímavé programy, citace, odkazy do diskuzí, recesistické moduly na CPANu nebo narážky na mnoho let zpožděný Perl 6.

Abychom se zorientovali v této oblasti, podívejme se letmo na několik legendárních počinů, o kterých je dobré něco málo vědět.

Just Another Perl Hacker (JAPH)

JAPH je asi nejznámějším projektem z kategorie Perl Humour. Jeho cílem bylo demonstrovat heslo TIMTOWTDI (there is more than one way to do it). Jde o kolekci programů od různých přispěvovatelů, které vypíší text "Just Another Perl Hacker" co nejzajímavějším způsobem.

Podívejme se na některé zajímavé příspěvky. Zde je jeden od Randala Schwartze:

```
@ARGV=split(/\S\n?/,<<EOF);$a+=length(shift),vec($_,$a,1)=1while@ARGV;print;
:-)  :-)   :-)  :-)  :-)  :-)  :-) :-)  :-) :-)   :-) :-)    :-)  :-) :-)
 :-)      :-)   :-)    :-) :-)   :-) :-)  :-)  :-) :-)  :-) :-) :-)  :-)
 :-)    :-)  :-) :-) :-)    :-)  :-) :-)  :-)  :-)   :-) :-)   :-)   :-) :-)
  :-)      :-)      :-)  :-)  :-)  :-)   :-) :-)   :-)   :-) :-) :-)    :-)
 :-)  :-) :-)      :-)     :-)  :-) :-)  :-)    :-) :-)  :-) :-)   :-) :-)
:-) :-)  :-)  :-) :-)  :-)  :-)   :-) :-)   :-)   :-) :-) :-)    :-) :-)  :-)
EOF
```

Ještě jeden od stejného autora:

```
print join('',grep(s/.*"(.)".*/$1/,split(/\n/,<<)));
Gimme a "J"!
Gimme a "u"!
Gimme an "s"!
Gimme a "t"!
Gimme a " "! :-)
Gimme an "a"!
Gimme an "n"!
Gimme an "o"!
Gimme a "t"!
Gimme an "h"!
Gimme an "e"!
Gimme an "r"!
Gimme a " "! :-)
Gimme a "P"!
Gimme an "e"!
Gimme an "r"!
Gimme an "l"!
Gimme a " "! :-)
Gimme an "h"!
Gimme an "a"!
Gimme a "c"!
Gimme a "k"!
Gimme an "e"!
Gimme an "r"!
Gimme a ","! :-)
What's that spell??
```

Zde je jiný způsob od Abigaila.

```
print 74.117.115.116.32;
print 97.110.111.116.104.101.114.32;
print 80.101.114.108.32;
print 72.97.99.107.101.114.10;
```

Na závěr se podívejme na program ve tvaru dvojšroubovice DNA od MeoChowa (koho zaujme, může se podívat na Acme::DoubleHelix, což je modul, který jakýkoliv program do tohoto tvaru přepíše):

```
$ _=
qq
.CG
T--A
A---T
A----T
C----G
T----A
A---T
G--C
CG
CG
C--G
G---C
G----C
C----G
A----T
C---G
```

G--C
AT
CG
A--T
A---T
G----C
A----T
G----C
C---G
A--T
GC
CG
G--C
A---T
G----C
G----C
G----C
A---T
T--A
AT
CG
T--A
C---G
A----T
A----T
G----C
A---T
A--T
CG
TA
T--A
G---C
G----C
C----G
A----T
C---G
G--C
CG
TA
T--A
C---G
T----A
C----G
A----T
C---G
G--C
GC
TA
G--C
C---G
G----C
G----C
G----C
A---T
T--A
CG
AT
G--C
A---T
A----T
C----G
C----G
A---T
A--T
CG
CG
G--C
G---C
G----C
A----T
T----A
C---G
G--C
CG
TA
A--T

```
A---T
G----C
A----T
A----T
C---G
A--T
GC
TA
G--C
C---G
T----A
G----C
G----C
C---G
T--A
AT
CG
G--C
G---C
T----A
C----G
G----C
C---G
C--G
CG
AT
T--A
C---G
G----C
G----C
A----T
T---A
G--C
CG
TA
G--C
A---T
G----C
A----T
G----C
.;@_{A
=> C
=>
G=>
T=>}
=0..3
;s;. *
(\w).*
(\w).*
\n;$_
{ $-
++
/9
%2?$
2:$ 1
};gex;
s;(.)(
.)(.)(.
);chr
64*$
1+
16
*$2+
4 *$
3 +$
4 ;gex
; eval
```

Ostatní programy lze najít například na CPAN.org. Tato kolekce je mnohem více než jen zábava. Stojí za to jim věnovat trochu času, protože uvnitř je v koncentrované podobě skryto obrovské množství triků, které stojí za to se naučit nebo o nich aspoň vědět.

Poezie

Existují básně psané v Perlu. Vyznačují se tím, že jsou validním kódem Perlu a obsahují příběh. Pro představu jeden příklad od suaveanta s názvem "Fish dinner":

```
use Carp;
unpack fish, spices;
croak fish if $alive;
```

```
study recipe, pop in, time;
BEGIN {meal};
tied %bib;
scalar fish;
fork, split;
sqrt lemon;
glob tartarsauce;
chop, open(MOUTH), chomp;
unlink flesh;
truncate bone, chomp;
untie %bib;
push plate;
END {meal};
sleep now;
```

Řada básní je na vláknu věnovaném poezii. Také existovala ne příliš úspěšná soutěž o nejlepší báseň.

Perl Golf Apocalypse (PGA)

PGA byla další programátorskou soutěží. Jde o týmovou hru s několika úkoly, které je třeba naprogramovat v Perlu s co nejmenším počtem znaků.

Podívejme se pro zajímavost na jeden příklad. Zde je jeho zadání.

* Difficulty: Easy

* Time: 3 minutes

* Task: Write a Perl one-liner that prints the following to STDERR:

Can't find label TPC at -e line 5.

A zde je možné řešení na 62 znaků:

```
perl -e"print STDERR qq{Can't find label TPC at -e line 5.\n}"
```

Zvládli byste to ale na méně než 25 znaků?

Vlákno s dalšími úkoly s komentáři je na perlmonks.org.

Obfuscated Perl Contest

Konala se také soutěž o nejnečitejnější kód. Roli při hodnocení hrála hlavně nečitelnost a estetika kódu a použití co nejširší palety nástrojů. Soutěžilo se například v těchto disciplínách:

- Print a human-readable "The Perl Journal"
- Create a diversion
- Do something powerful
- Be creative
- Make believe you're another language
- Best four line signature

Zde je například příspěvek, který vypíše řetězec "The Perl Journal" (což byla jedna z kategorií, ve které se soutěžilo):

```
#!/usr/bin/perl -l

print ocr(<<TPJ);
  #  # # ## ##  ## ##  #   #  #  # # ##  #  #  #  #
 ### # # #  # # #  # # #   # # # # # # ## # # # #
#  ### ## ##  ## ##  #   # # # # # ##  # ## ### #
  #  # # #  #  # # #  # # # # # # # #  # # # #
#  # # ## #  ## # # ## ###  #  ### # # #  # # # ##
TPJ

sub ocr{@{$-[$@++]}=split$,for(split'\n',shift);for$@(0..4){for(0..51){++$_{$_
}if($-[$@][$_]=~$")}}@&=(-1);for(sort{$a<=>$b}keys%_){push@&,$_ if($_{$_}>4)
}push@&,52;for$@(0..13){@{$|[$@][$_]}=@{$-[$_]}[$&[$@]+1..$&[$@+1]-1]for(0..
4)}for(@|){**=$_;$w=@{$*[$^=$$=0]}-1;for$@(0..4){for(1..$w){$^++if$*[$@][$_
]ne$*[$@][$_-1]}}for(0..$w){for$@(1..4){$$++ if$*[$@][$_]ne$*[$@-1][$_]}}
for(0..20){push@},chr$_+65if(7*(8,4,2,9,2,3,7,8,1,$@,5,4,9,10,10,6,3,8,4,
8,8)[$_]+(5,8,3,3,4,2,1,2,8,2,7,1,5,4,6,$@,3,6,8,4,1)[$_]==7*$^+$$)}}@}}
```

Jednou z kategorií pozoruhodných programů jsou takové, jejichž výstupem je vlastní zdrojový kód. Zde je tedy skript, který vytiskne sám sebe.

```
#!/usr/local/bin/perl
$a='#!/usr/local/bin/perl%c$a=%c%s%c;printf($a,10,39,$a,39,10);%c';printf($a,10,39,$a,39,10);
```

Takové programy se označují slůvkem quine a kdyby někoho zajímaly více, je dobrým zdrojem například The Quine Page.

A na závěr, zkuste hádat, v jakém jazyce je napsán tento program. Jako parametr mu dejte jméno existujícího souboru.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#define open(a,b) open(b,a)
#define $ARGV argv
#define $i i
#define x : /* aren't four #define way too much?
unshift @ARGV, $_ = $ARGV[0]; "*/
main(int argc, char *argv[]) { // "; {
int m=1, i[14]; char * pp; int p=-1;
int q, F=3; char * qq = "Hello\, world!\n";
i[12]=537463307; i[13]=3085; //,; $_ = "" if(length!=2);
```

```
if(m+-p?(argc>1&&!strcmp(argv[1],"-p"))?p+i? 1 : 1 x 0 x 0) {
printf(qq/*\bThe Perl Journal\n/#*/
); exit(0); }
qq="=;#"; argv[0][0]='\0'; memset(i,0,48);
$i[10]=($i[11]=(q/*\b/&&scalar@ARGV))-1;#*/=0) + argc)-1;
do{
if($i[11]<2) { $i[10]=1; q/*/&&*F=*STDIN;#*/=F=0;
} else { open(O_RDONLY, $ARGV[$i[11]-$i[10]]);//; *F=*O_RDONLY;
}
while(read(F, $i, 1)>0) {
++$i[4]^(q=/*.=,$_=$i);#*/0); pp=i;
$i[3]+=m=( *pp^0x0A)?/*\n=;#*/0:1; for(qq=&i[12];*qq;*pp^*qq++||(q=1));
if(m=/*[    \n\f\r\xB]=#*/q
) { if($i[1]){$i[$i[1]]++; $i[1]=0; }} else { $i[1]=2;}
}
if($i[1]){$i[$i[1]]++;};
printf("%7d %7d %7d %s\n",$i[3],$i[2],$i[4],$ARGV[$i[11]-$i[10]]);
close(F);
if($i[11]>2){for($i[1]=2;$i[$i[1]+4]+=$i[$i[1]];$i[1]++){$i[$i[1]]=0;};$i[1]=0;}
} while(--$i[10]);
if($i[11]>2) { printf("%7d %7d %7d total\n",$i[7],$i[6],$i[8]); }
}
```

Pro více informací a další zajímavé programy si zkuste zalistovat archivem Perl Journalu, kde jsou mimo jiné staré výsledky The Obfuscated Perl Contest.

Acme moduly

Moduly s názvem začínajícím Acme:: konvertují zdrojový kód do nějakých netradičních forem. Například Acme::EyeDrops udělá z vašeho zdrojového kódu obličej vybraného Perl vývojáře, velblouda nebo jiného z přednastavených objektů. Nebo Acme::Morse zakóduje celý váš kód do Morseovy abecedy a Acme::Bleach ho skryje do netisknutelných znaků. Přitom celý program bude fungovat stejně jako předtím.

Zajímají-li vás Acme moduly více, podívejte se pro lepší představu o paletě možností na jedno vlákno na perlmonks.org, které představuje vybraných 60 modulů.

Jak moc znáte Perl?

Jak moc "Perl pure" jste? Udělejte si Perl purity test.

Dále se můžete zkusit zařadit do jedné z úrovní podle Seven Levels of Perl Mastery.

Jste "geek" a nevíte, jak to dát najevo ostatním? Pak stačí, když si sestavíte vlastní geek code. Geek code lehce připomíná zdrojový kód Perlu a uvedete ho například do patičky příspěvků v diskuzních fórech. Pro sestavení můžete použít nějaký z enkoderů.

Perl 7

Na závěr se určitě stojí za to podívat, co si perloví monkové myslí o Perlu 7. Zajímavá anketa je na perlmonks.org.