C/C++

Proč právě C/C++

Existuje mnoho programovacích jazyků, nejen C a C++. Možná jste už slyšeli o Javě, C# (čti c šarp), Pythonu, Perlu, PHP a mnoha dalších. C a C++ patří k těm nejstarším, které se dnes široce používají. Jsou na naučení a používání těžší než většina ostatních, ale ostatní jazyky z nich přímo či nepřímo vycházejí, takže naučit se se znalostí C/C++ jiné jazyky už pro vás bude hračka. Mnohé osatní jazyky před programátorem skrývají důležité rysy programování. Snižují tak šanci na chybu progamátora, urychlují vývoj, ale za tu cenu, že programy jsou pomalejší a objemnější. Proto tyto programovací jazyky umožňují programátorům vytvářet svá vlastní rozšíření. A hádejte, v jakém jazyce?

V jazyce C je také napsáno mnoho důležitých aplikací, které umožňují vytvářet své rozšíření v jazyce C. Proto si myslím, že je důležité, abyste se naučili jazyk C (a nejen C++). C++ je něco jako „novější C", takže když se naučíte C, v C++ se budete učit (téměř) jen to, co je nové. Určitě je lepší se nejdříve naučit C a později C++, než obráceně. Na druhou stranu, pokud si myslíte, že jazyk C využívat nikdy nebudete (třeba chcete psát jen nové programy), nemusíte se bát začít učit rovnou C++.

Nemusíte se ani bát naučit se rovnou nějkterý z „jednodušších", modernějších jazyků, jako například Javu. To už závisí případ od případu, co je pro vás konkrétně nejlepší. Určitě v životě potkáte spoustu lidí, kteří vám budou tvrdit, že ten jejich programovací jazyk je ten nejlepší :). Každopádně, tento kurz vás provede programováním tak jednoduše a polopaticky, jakto jen jde :).

Co musíte vědět

Jistě jste již netrpěliví začít programovat. Nejdříve vám však položím několik otázek, na které byste měli odpovědět kladně, jinak nejste na programování dostatečně připraveni.

1. Umíte číst, psát a počítat?
2. Víte, co je to počítač?
3. Víte, co je to soubor, adresář, adresářová struktura?
4. Umíte psát na počítači, vytvářet a ukládat soubory?
5. Víte, jaký je rozdíl mezi hardwarem a softwarem?
6. Říkají vám něco přípony .txt, .exe, .com, .bat, nebo co to znamená spustitelný soubor?
7. Víte, co je to harddisk, paměť RAM, procesor, grafická karta?
8. Víte, co je to OS (Operační systém) a nepletete si jej s pojmem Windows?
9. Víte, co je to Linux?
10. Viděli jste film Matrix?

Na prvních 7 otázek musíte odpovědět ano, osmou otázku během výuky dovysvětlím, devátá je tu hlavně kvůli tomu, že se v textu budu věnovat (později výhradně) programování v Linuxu a pokud jste na 10. otázku odpověděli záporně, tak to rychle napravte :-).

Co se nenaučíte

Někdo je po seznámení s počítačem natolik okouzlen, že by chtěl hned umět všechny ty programy a aplikace vytvářet. Možná byste si také chtěli naprogramovat svůj OS jako Windows, nějaké ty office jako Word, Excel, nějakou tu hru, jako Quake atp. Bohužel, v tom vás zklamu. Tak rozsáhlé programy nemůže zvládnout jeden člověk, jedná se totiž o milióny řádek kódu a velice sofistikované matematické algoritmy. Zde se naučíte vytvářet jednoduché textové programy (tj. bez grafického rozhraní), které budou například sčítat dvě čísla, hledat řetězec v textovém souboru atp. V části věnované Linuxu se dostane i na nějaké ty grafické nadstavby. Pokud se jen s tímto nespokojíte, nevadí. Všechno totiž začíná tím, co se zde dozvíte a nic z toho, co se zde dozvíte, nebude pro vás zbytečné, i když chcete umět dělat profesionální aplikace. Počítejte jen s tím, že v takovém případě budete muset studiu věnovat několik let!

Na druhou stranu, i jako programátor jednotlivec můžete poměrně snadno vytvářet jednoduché hry, GUI aplikace (tj. aplikace s grafickým uživatelským rozhraním, např. ve Windows) nebo se spolupodílet na vývoji velkých projektů.

Mám i jednu dobrou zprávu na závěr, a to zejména pro uživatele Linuxu. Spousta programů je napsána v jazyce C a spousta jich je dodávána i se zdrojovými kódy (pod licencí GNU). Až se naučíte programovat v C, budete si je moci upravovat k obrazu svému, učit se z nich, nebo se podílet na jejich vývoji.

Co se naučíte

Úplně na začátku si objasníme některé pojmy, aby mezi námi nedocházelo k nedorozumění. Poté se dozvíte, jak program vzniká a co je k tomu třeba. Naučíte se základy programovacího jazyka C a později „jeho rozšířenou verzi" jazyk C++. Nakonec se budu stručně věnovat programování pod Linuxem. Ostatně, to vás bude provázet celým výukovým textem. Samozřejmě se také naučíte programovat pod DOSem a dozvíte se něco o programování ve Windows. Myslím, že o tom „co bude" toho již bylo řečeno dost, a tak není důvod proč se nevrhnout na věc. Příjemnou zábavu.

Číselné soustavy

Desítková a dvojková soustava

Ačkoliv se o dvojkové soustavě učí již na základní škole, ne každý si ji pamatuje. Proto jsem se rozhodl na začátek této kapitoly vsunout malé připomenutí toho, oč se vlastně jedná. Pozorně si přečtěte následující dva odstavce a porovnejte je.

Desítková soustava je to, co asi všichni běžně používáme.

Desítková se jí říká proto, protože máte k dispozici právě deset znaků, kterými zapisujeme všechna čísla. Jsou to znaky: 0123456789.

Všimtěte si, že desítková soustasva již nemá znak pro číslo deset, to se musí složit ze dvou znaků: 10.

Všimtěte si, jak se takové číslo v desítkové soustavě vytváří. Dokud máte znak na zapsání čísla, tak jej použijete. Například když chcete napsat číslo devět, můžeme použít znak 9. Ale pokud chcete napsat číslo deset, už žádný další znak nemáte (opakuji, máte znaky jen pro čísla nula až devět). V takovém případě napíšete vlevo jedničku, jako že jste jednou vyčerpali všechny dostupné znaky a nulu k tomu. (Vlevo proto, protože jsme zdědili čísla od arabů).

Příklad: 11(v desítkové s.) = $1*(10^1)+1*(10^0)$ = 10 + 1 = 11 (v desítkové s.)

Dvojková soustava je to, co běžně používají programátoři.

Dvojková se jí říká proto, protože máte k dispozici právě dva znaky, kterými zapisujete všechna čísla. Jsou to znaky: 0 a 1.

Všimtěte si, že dvojková soustava již nemá znak pro číslo dva, to se musí složit ze dvou znaků: 10.

Všimtěte si, jak se takové číslo ve dvojkové soustavě vytváří. Dokud máte znak na zapsání čísla, tak jej použijete. Například když chcete napsat číslo jedna, můžeme použít znak 1. Ale pokud chcete napsat číslo dva, už žádný další znak nemáte (opakuji, máte znaky jen pro čísla nula až jedna). V takovém případě napíšete vlevo jedničku, jako že jste jednou vyčerpali všechny dostupné znaky a nulu k tomu. (Vlevo proto, protože jsme zdědili čísla od arabů).

Příklad: 1011(v dvojkové s.) = $1*(2^3) + 0*(2^2) + 1*(2^1) + 1*(2^0)$ = 8 + 0 + 2 + 1 = 11 (v desítkové s.)

Za domácí úkol popište osmičkovou, šestnáctkovou a třeba dvacetisedmičkovou soustavu :)

Nápověda: 13(v osmičkové s.) = $1*(8^1) + 3*(8^0)$ = 8 + 3 = 11 (v desítkové s.)

Na světě je 10 druhů lidí - tí, co rozumí dvojkové soustavě a tí, co ji nerozumí.

Příklad sčítání čísel ve dvojkové soustavě. Zčítá se z prava do leva, podobně jako když sčítáte v desítkové soustavě:

```
00110101   (= 53 desítkově)
10001100   (=140)
--------
11000001   (=193)
```

Snad vám tento stručný úvod do dvojkové soustavi stačil. Níže ještě proberu převody mezi různými soustavami a ukážu příklady.

Bit a bajt

Abyste dobře porozuměli programování a počítačům vůbec, musíte vědět, co je to binární kód. V počítači se pomocí binárního kódu zapisuje všechno. Jsou tak zapsány nejenom programy, ale třeba i textové dokumenty, obrázky, videa atp. Je to dáno tím, že hardware je schopen rozeznávat **jen dva stavy** – například kladný a záporný elektrický náboj, nebo „je v CD důlek" a „není v CD důlek". Tyto stavy reprezentují čísla 1 nebo 0 (mohli bychom je reprezentovat například pejskem a kočičkou, ale s tím by se asi špatně počítalo).

Stav (1 nebo 0) je nejmenší jednotkou informace a nazývá se **bit** (čti bit). Je jasné, že čím více bitů, tím více informací můžeme zaznamenat (například na DVD jsou menší „důlky" než na CD, proto se na stejnou plochu vejde více informací – více muziky :). Pomocí jednoho bitu můžete zapsat jen čísla nula a jedna. Pomocí 2 bitů už nula až tři, pomocí 8 bitů 0 až 255 atd. Podívejte se na následující tabulku:

Zápis čísel v desítkové a dvojkové soustavě

Desítková soustava	Binární (1 bit)	Binární (2 bity)	...	Binární (8 bitů)
0 (nula)	0	00	...	0000 0000
1(jedna)	1	01	...	0000 0001
2(dva)	x	10	...	0000 0010
3(tři)	x	11	...	0000 0011
...	...	...	...	...
255 (dvěstě padesát pět)	x	x	...	1111 1111

Dlouhá řada jedniček a nul se špatně čte, proto je zvykem zapisovat čísla ve dvojkové soustavě po čtyřech (a také kvůli snadnému převodu do šestnáctkové soustavy, ale o tom až zachvilku). Takže místo 11111111 budu psát 1111 1111 – mezera je tam jen kvůli čitelnosti.

Byte (čti bajt) je **posloupnost 8 bitů**. Například největší možné číslo zapsané pomocí jednoho bajtu je 1111 1111 (v desítkové soustavě 255). Číslo 0111 1111 je desítkově 127 atp.

Bajt má 8 bitů z historických důvodů. Prostě dříve nebylo dost peněz na víc bitů :) Hardware byl moc drahý. Určitě jste někdy slyšeli pojem 8-bitové počítače …

Je tu ale ještě jeden problém, jak do toho DVD vypálím **číslo se znaménkem**, když můžu jen vypálit či nevypálit „důlek", tedy když mám k dispozici jen ty jedničky a nuly?

Můžete si říct, že budete zapisovat čísla se znaménkem tak, že první bit určuje znaménko. Na vlastní hodnotu čísla pak zbyde 7 bitů, takže lze zapsat čísla od -127 do +127.

Bajt 1000 0001 je **se znaménkem** -1 (kdežto bez znaménka by to bylo 129).

Tohle řešení vypadá hezky, ale je tu drobný problém s nulou. Dá se zapsat jako 0000 0000, ale také jako 1000 0000 (záporná nula)! S tím by se počítačům špatně počítalo.

Při operacích jako je násobení, sčítání a odčítání by si musel processor na tento „extra" bit dávat pořád pozor. Proto přišli matematici s lepším řešením, tzv. **doplňkovým kódem**.

V **doplňkovém kódu** je nula pouze 0000 0000, -1 je 1111 1111, -2 je 1111 1110, -3 je 1111 1101, a nejnižší možné číslo -128 je 1000 0000. Největší číslo pro byte se znaménkem je 127 (0111 1111).

Jestli jde o byte se znaménkem (rozsah 127 až -128) nebo bez (rozsah 0 až 255) nelze nijak poznat, musíme se na tom dohodnout vždy dopředu. Proto existují v jazyku C datové typy se znaménkem (signed) a bezeznaménka (unsigned). K tomu se dostaneme později.

Doplňkový kód, nebo též **dvojkový doplněk** má další zajímavé (a žádané) matematické vlastnosti, kvůli kterým se používá. Podrobný popis vlastností dvojkového doplňku je však nad rámec této kapitoly a myslím, že se bez něj ve svých začátcích obejdete. Pokud vás zajímají výhody toho to zápisu, najděte si podrobnější informace někde jinde.

V této části jsem ukázal, jak se ukládají celá čísla (se znaménkem nebo bez). Co jsem napsal o jednom bajtu platí obdobně i pro dva bajty (dva bajty, tj 16 bitů, se nazývají *slovo*, ale to je asi fuk). Dnešní 64-bitové počítače jsou navrženy na práci s čísly skládajícími se z (chvilka napětí) 64-bitů. Pokud chce počítač počítat s čísly tak velkými že se do 64 bitů nevejdou, musí to nějak „očůrat", což výpočty zdržuje.

Ještě větší sranda je s čísly s desetinnou čárkou. Ty se ukládají tak složitě, že vám to milosrdně zatajím. Nicméně jen pro zajímavost uvedu, že počítače neumí s čísly s desetinnou čárkou počítat moc přesně. (Dokonce nezvládají uložit přesně ani číslo 0.1 a už vůbec ne 1/3. Důsledkem může být, že výpočet (1/3)*3 vyjde 0.9999999999).

Převody soustav

Převod z binární soustavy do desítkové je jednoduchý. Začněte zprava a součet (2^pozice zprava)*bit je hodnota čísla v desítkové soustavě (dvě umocníte na číslo odpovídající pozici bitu zprava (začíná se od 0) a vynásobíte bitem, tj. jedničkou nebo nulou).

Podívejte se na 4 bitové číslo (převádím z prava do leva):

$$1010 = (2^0)*0 + (2^1)*1 + (2^2)*0 + (2^3)*1 = 1*0 + 2*1 + 4*0 + 8*1 = 0+2+0+8 = 10$$

Nejlépe uděláte, když si zapamatujete tuhle tabulku:

bin:	0000 0001	0000 0010	0000 0100	0000 1000	0001 0000	0010 0000	0100 0000	1000 0000
dec:	1	2	4	8	16	32	64	128

Pak už stačí jenom „najít" odpovídající jedničky z čísla, které převádíte a sečíst hodnoty z druhého řádku. Tj. číslu 1010 odpovídají druhý a čtvrtý sloupeček, tj 2 + 8 = 10.

Převod z desítkové do binární není o moc těžší. Číslo vydělíte dvěma a pokud vám vyjde výsledek beze zbytku, zapíšete 0, jinak 1. Celočíselný výsledek stále dělíte dvěma a 0 nebo 1 zapisujete zprava doleva.

Podívejte se na převod čísla 10d (d označuje desítkovou soustavu):

Deset desítkově se zapíše v dvojkové soustavě jako 1010

Krok ve výpočtu	**celočíselné dělení (desítkově)**	**mezivýsledek**
1.	10/2 = **5** beze zbytku, zapisuji 0	0
2.	**5**/2 = **2** a zbytek 1, zapisuji 1	10
3.	**2**/2 = **1** beze zbytku, zapisuji 0	010
4.	**1**/2 = 0 a zbytek 1, zapisuji 1	1010

Šestnáctková soustava

Při programování se také setkáte s šestnáctkovou (hexadecimální) soustavou. Ta, jak už název napovídá, využívá k zápisu 16 znaků. Postupně 0, 1, 2 … 9, A, B, C, D, E a F. (Jestli píšete znaky A až F malými nebo velkými písmeny je jedno). Tj. A odpovídá 10d, f je 15d atd.

1 byte může nabývat hodnot 00 až ff (0d až 255d, pokud to iterpretujete beze znaménka).

V jazyce C lze používat k zápisu čísel desítkovou nebo hexadecimální soustavu. Dvojkovou nikoliv. Pokud budete pracovat s bity, je dobré umět převést dvojkovou soustavu na hexadecimální. Převod se dá zjednodušit tím, že budete převádět vždy jen 4 bity do hexadecimální soustavy:

Při převod bajtu (8 bitů) do šestnáctkové soustavy lze převést horní a dolní 4 bity zvlášť.

desítkově	**binárně**	**hexadecimálně**
11	1011	B
12	1100	C
188	1011 1100	BC
203	1100 1011	CB

Ještě se na závěr zmíním, že se v jazyce C používá i **osmičková soustava**. Osmičková soustava se z dvojkové snadno převádí po 3 bitech (obdobně jako šestnáctková po 4).

Čísla v osmičkové soustavě se v jazyce C zapisují pomocí nuly na začátku. Šestnáctková soustava se zapisuje pomocí nuly a písmena x na začátku.

Ukázka zápisu čísel v různých číselných soustavách jazyce C/C++

desítkově	**dvojkově**	**osmičkově**	**šestnáctkově**
255	11 111 111	0377	0xff

0377 musí mít na začátku tu nulu, 377 by se bralo v jazyce C jako desítkové číslo.

Jaký zápis použijete je úplně jedno, nejčastěji asi budete používat desítkový. Jestli ale někomu vyhovuje více osmičkový zápis, nechť jej klidně používá :-). V určitých situacích, například při adresování paměti, je výhodnější používat šestnáctkovou soustavu. Vždy jde hlavně o to, aby to bylo čitelnější pro člověka, který čte zdrojový kód.

Kilobyte vs Kibibyte

Tyto pojmy se často pletou. 1 kilobyte = 1000 bytů. Ale při práci s počítačem se pracuje s kibibyty. 1 kibibyte = 1024 bytů. To je přesně 2^{10}.

Počítejte semnou: 8, 16, 32, 64, 128, 256, 512, 1024.

Jak vidíte, na počátku všeho stojí osmička. A ne náhodou je tato číselná řada násobkem dvou – dvojková soustava ruleeez!

1 Mebibyte = 1024 kibitů = 1048576 bytů.

Zkratky pro kibibyte a mebibyte jsou **KiB** a **MiB**. Často se ale velmi nesprávně uvádí KB a MB. Třeba výrobci disků rádi uvádějí velikosti v MB (teda spíš už v GB). Oni i správně uvedou jednotku GB. Když se pak na disk podíváte v PC, počítač vám ohlásí velikost v GiB a vy se divíte, kam se vám poděli stovky megabytů (nebo mebibytů? :-). Widnows vám dokonce drze oznámí, že je velikost uváděna v GB, ikdyž je v GiB.[1)]

Jestli nechcete být za hlupáky, uvádějte vždy správnou jednotku!

MB a MiB znamenají něco jiného a je na vás, abyste vybrali tu správnou.

Co je to OS

Než vám řeknu, co je to OS (operační systém), podíváme se na to, jak vlastně celý počítač funguje a co se v něm děje od jeho spuštění. Po zapnutí se začne napájet hardware počítače elektřinou a spustí se BIOS. **BIOS**je program, který je umístěn přímo na základní desce vašeho PC. Hned po spuštění počítače začne zjišťovat, co je na desce za další hardware (např. grafická karta, procesor, pevné disky, paměť). Pokud BIOS najde všechen potřebný hardware pro běh počítače, pokusí se spustit operační systém. Většinou lze nastavit v BIOSu, kde se má OS hledat (na disketě, na CD, USB, nebo na harddisku …). Pokud se BIOSu podaří najít spouštěcí oblast na některém médiu (nebo síti), pak spustí OS a předá mu kontrolu. **Operační systém** je také program, který má za úkol komunikovat z hardwarem počítače, se vstupními a výstupními zařízeními (klávesnice, myš, modem, monitor, tiskárna …). Operační systém se také stará o běh dalších procesů (programů).

Běžné programy by s hardwarem komunikovat vůbec neměli, a měli by to nechávat na OS. Například, pokud chce program něco vytisknout, předá data operačnímu systému, který je pošle na tiskárnu. K tomu, aby si OS s periferiemi rozuměl (např. s tou tiskárnou, zvukovou kartou atp.) se vytvářejí tzv. **ovladače**. Ovladače většinou vyrábí výrobce hardwaru pro konkrétní OS. Ovladač je program, který umí na jednu stranu komunikovat s OS a na druhou stranu s hardwarem.

Z předchozího řečeného vyplývají dvě zprávy. Jedna dobrá a jedna špatná. Ta dobrá je, že se nemusíte při psaní programů starat o to, jaký máte hardware v počítači, o to se postará OS. (Při psaní ovladačů by jste se o to samo sebou už starat museli). A teď ta špatná zpráva. Různé OS vytvářejí různí lidé. Proto také programy, o které se OS stará vypadají různě. Vytváříte-li program, pak takový, aby s ním OS dokázal komunikovat. To znamená, že program pro jeden OS nebude fungovat pod jiným OS. (Existují i výjimky, ale program do Windows nebude pracovat v Linuxu a naopak). Ovšem nic není tak zlé, jak se na první pohled vypadá. K tomuto problému (tzv. problém přenositelnosti) se později na chvilku vrátím.

Program a instrukce

Program není opět nic jiného než jenom hromada jedniček a nul. Některé ty jedničky a nuly reprezentují data (např. text, obrázek, cokoliv vás napadne) a některé instrukce. **Instrukce** jsou příkazy, které říkají, co se má udělat. Instrukce jsou určeny pro procesor, který je umí vykonat (například sečtení dvou čísel) a také pro OS, který jim musí rozumět (například volání funkce pro tisk). Existují různé typy procesorů, například Intel ix86, sparc, mips atp. Samozřejmě co procesor, to jiná sada instrukcí :-(. Pokud tedy vytváříte program, nejste závislí jen na OS, pro který jej vytváříte, ale i na procesoru. Například program pro Windows a 64-bitový procesor Intel nebude fungovat na Windows s 32-bitovým Intel procesorem.

Program je jakýsi návod pro OS, co má s čím dělat, jaké instrukce předat procesoru a tak podobně. Program v binární podobě je pro obyčejného smrtelníka nečitelný shluk jedniček a nul. Ačkoli po prostudování tohoto kurzu jazyka C již normální smrtelníci nebudete, stále pro vás nebude program v binárním kódu čitelný :-).

Dříve se programy psali na děrovacích štítcích (co štítek, to instrukce), ale dnes je již takový způsob programování téměř nemyslitelný. Jak se dá vytvořit program rychle a jednoduše, to se dozvíte v příští kapitole.

Jak program ve skutečnosti pracuje? Co se stane když poklepete ve Windows na ikonku programu? OS se podívá do souboru s programem a zjistí, pro jakou verzi jakého OS byl napsán. Pokud je s tímto programem kompatibilní, začne číst a vykonávat jeho instrukce. Instrukce mohou požadovat například sečtení dvou čísel, vytvoření souboru, nebo vytištění dat (Jinak řečeno, zavolání funkce OS pro tisk. Všimněte si, že program v tomto případě musí vědět jak OS o zavolání funkce požádat, jaká funkce to má být, jak této funkci předat data k vytištění, nebo třeba také to, jak zjistit, zda tisk proběhl v pořádku. To všechno nás opět přivádí k problému přenositelnosti programu mezi různými OS zmíněnému výše.)

Teď už znáte problémy, tak se můžete těšit na jejich řešení v dalších kapitolách :-)

program

Vzniku velkého programu předchází spousta procesů. Nejdříve je třeba provést analýzu, návrh funkčnosti programu, uživatelského prostředí. Je třeba si rozmyslet, jak bude program vypadat, co bude jeho funkcí, pod jakými OS bude pracovat, budou-li v budoucnu potřeba nějaké změny v programu (jaké), kolik lidí bude na programu pracovat a jak dlouho, jak se bude program následně udržovat aktuální a funkční atd. Z toho pak vyplyne, jakými postupy program vznikne. Dobrou pomůckou vám při tom může být například znalost jazyka UML. Prostě, jak se říká, dvakrát měř, jednou řež. Já se zde ničím podobným zabývat nebudu, není to obsahem tohoto výukového textu. Zde se dozvíte pouze to, jak program „naprogramovat".

- Programovací jazyk, zdrojový kód a překladač
- Přenositelnost
- Trocha historie C a C++
- Ukázka zdrojových kódů

Programovací jazyk, zdrojový kód a překladač

Jak jsem psal v minulé kapitole, program je soubor s daty a instrukcemi. Dříve se programovalo pomocí děrovacích štítků, ovšem s příchodem nových periférií, jako klávesnice a monitor, vznikaly i nové způsoby programování. Teď se pokusím vysvětlit některé pojmy, které s tím souvisejí.

Prvním důležitým pojmem je **zdrojový kód** programu. Zdrojový kód programu není nic jiného, než hromada textových souborů, v nichž je zapsáno pomocí **programovacího jazyka** (nikoliv tedy pomocí českého nebo anglického jazyka) co má program dělat.

Překladač (angl. compiler) je program, který si přečte zdrojový kód a vytvoří z něj program.

Odjakživa bylo snahou vytvořit takový programovací jazyk, který by byl co nejjednodušší na pochopení a naučení se pro člověka a zároveň, aby bylo proveditelné vytvořit překladač, který by takovému jazyku porozuměl (proto, alespoň zatím, nelze psát zdrojové kódy v lidské řeči, musí se na to vymyslet speciální jazyk – programovací jazyk).

Bylo vytvořeno mnoho různých programovacích jazyků a k nim samozřejmě mnoho překladačů (co jazyk, to překladač(e)). Mezi prvními programovacími jazyky byl assembler. Pomocí tohoto jazyka se přímo zapisovali instrukce procesoru. Například sečtení dvou čísel v assembleru může vypadat takto:

```
MOV AX,a
MOV BX,b
ADD AX,BX
MOV a,AX
```

AX a BX je označení registrů procesoru, „a" a „b" jsou proměnné, které obsahují nějaká čísla. MOV je instrukce která data přesune (nejdříve data z „a" do AX, pak z „b" do BX…) a ADD je instrukce, která přičte druhé číslo k prvnímu (tj. k AX přičte BX). Nakonec se výsledek zapíše do proměnné *a*. Překladač assembleru tento zdrojový kód převede do binárního kódu (tedy programu), který je již procesoru srozumitelný.

Později vznikly „vyšší" programovací jazyky, mezi něž patří i jazyk C, které již nepracovali přímo s instrukcemi procesoru. Předchozí kód se v jazyce C zapíše následovně:

```
a = a + b;
```

Nyní je již na překladači, aby tento zápis převedl do procesoru srozumitelných instrukcí. Vyšší programovací jazyk znamená vyšší míru abstrakce. Už nepracujete přímo s registry procesoru, ale necháte na překladači zdrojového kódu, aby zdrojový kód přeložil do binárního kódu srozumitelného konkrétnímu procesoru na konkrétním OS. Vy jen říkáte „ sečti mi *a* a *b* a výsledek ulož do *a*". (Kde *a* a *b* zastupují nějaké čísla uložené v bytech.)

Překladače (a vývojová prostředí) proberu hned v další kapitole.

Přenositelnost

Přenositelnost programu je pojem, který nám říká, kde všude náš program poběží. Například program pro Windows 3.11 poběží i pod Windows 95, ale nepoběží pod Linuxem. Program pro 32-bitová Windows poběží, při troše štěstí, i pod 64-bitovými Windows, ale program pro 64-bitová Windows pod 32-bitovými Windows nepoběží.

Přenositelnost programovacího jazyka je vlastnost, která jednoduše určuje, na kterých platformách lze zdrojový kód tohoto jazyka přeložit. Jinak řečeno, pro které platformy (operační systémy a procesory) existuje překladač.

Čím více existuje překladačů pro různé procesory a OS, tím je program „přenositelnější". V tomto ohledu patří jazyk C k jednomu z nejlepších. Ještě lepší je Java, která vám poběží na mobilním telefonu, nebo třeba ve vaší pračce :-).

Java, mimochodem, vychází z jazyků C/C++, jako ostatně spousta dalších moderních programovacích jazyků. (Takže pokud se naučíte programovat v C/C++, bude vám syntaxe Javy povědomá.)

Překladače pro různé platformy vytvářejí různí lidé, či organizace, proto vznikají **standardy** jazyka, které říkají, čemu všemu by měl překladač rozumět. Pokud se standardů budete držet, bude váš zdrojový kód přenositelný. Pokud budete využívat nějaká nestandardní vylepšení vašeho překladače, pak se nejspíš přenositelnost vašeho programu (resp. zdrojového kódu) sníží.

Bohužel, v počátcích vývoje jazyka C žádný standard neexistoval. Standardy se také postupem času mění a podpora toho či onoho standardu je v tom či onom překladači na různé úrovni. Jazyk C se už dlouho nevyvíjí, proto by jeho podpora měla být v překladačích na vysoké úrovni. Existuje standard **ANSI C** vytvořený organizací pro „americký standard" www.ansi.org a z něj odvozený **ISO/IEC 9899:1999** vytvořený organizací pro mezinárodní standard www.iso.org. Odvážnější z vás si mohou stáhnout „Logický výklad standardu C99" v angličtině.

Přenositelnost zdrojového kódu je dána přenositelností programovacího jazyka a dodržováním **standardů**. Některé překladače totiž umí „něco navíc", co obvykle usnadňuje psaní zdrojového kódu, ale snižuje přenositelnost.

Další faktor, který má vliv na přenositelnost zdrojového kódu je používání knihoven.

Knihovny jsou soubory obsahující funkce, které může váš program využívat (říká se, že je program volá). Za roky vývoje jazyka C bylo vytvořeno mnoho knihoven funkcí a jejich používáním si ušetříte velkou spoustu času. Existují standardní knihovny, které jsou dostupné na všech podporovaných platformách. Například knihovny pro práci se soubory. Některé takové knihovny budu probírat. Jiné knihovny jsou však dostupné jen na některých platformách. Tak třeba pokud budete chtít naprogramovat nějakou aplikaci v systému Windows a využívat přirozené grafické prostředí Windows, tj. všech těch pěkných tlačítek, rozbalovacích menu, přepínačů, okének atp., budete k tomu využívat knihovny Windows, které, pochopitelně, jinde než na Windows nebudou fungovat. Váš program se tak stane na Windows závislý. Existují knihovny pro grafické uživatelské rozhraní (angl. zkratka GUI), které jsou funkční na Windows i Linuxu, ale zase nejsou přenositelné nikde jinde a uživatel takového programu si na první pohled všimne, že to vypadá a chová se jinak, než běžné grafické aplikace. Je to dáno tím, že standard jazyka C (ani C++) nezahrnuje GUI (na rozdíl třeba od Javy, která GUI má. Díky tomu můžete v Javě psát snadno přenositelný program s GUI, ovšem za tu cenu, že bude vypadat ve všech OS stejně – ošklivě.).

Některé knihovny funkcí jsou zdarma, některé jsou zdarma pro nekomerční použití, některé jsou za peníze. Některé vyvíjejí dobrovolníci, některé komerční firmy, nebo i neziskovky. Některé jsou udržované a **pořád se vyvíjejí**, některé jsou už zastaralé. To je holt život.

Trocha historie C a C++

Programovací jazyk C je známý svou přenositelností a rychlostí. Byl navržen jako poměrně malý jazyk, kombinující efektivitu a výkonnost. Byl vytvořen v Bellových laboratořích AT&T Denisem Ritchiem v létech 1969 až 1973. Dále se na vzniku jazyka podíleli Brian Kernighan a Ken Thompson. Byl napsán pro snadnou přenositelnou implementaci Unixu (Unix je OS, na jehož základě se později vyvinul Linux). Postupem času vznikla ANSI (American National Standard Institute) norma jazyka C. Shodný standard definuje ISO (International Stnadards Organisation) norma. Proto se někdy uvádí spojení ANSI/ISO norma. Pokud budete ANSI normu dodržovat, máte velkou šanci, že budete moci svůj program přenášet na různé platformy.

Jazyk C++ je pokračovatelem jazyka C (jehož vývoj se již téměř zastavil). Nejdříve byl označován jako C with Classes, od roku 1983 C++. Vyvinul jej Bjarne Stroustrup v AT&T.

Jazyk C++ přináší novinky, především OOP (objektově orientované programování). Pojem OOP vysvětlím až dojde na C++.

První oficiální norma jazyka C++ byla přijata v roce 1998.

Původně fungoval jazyk C++ tak, že jen navíc umožňoval objektový zápis, který se překladačem nejdříve přepsal do klasického jazyka C a pak se teprve přeložil. Mohli jste tedy psát v C++ cokoliv jako v C a fungovalo to. Ty doby jsou už však dávno pryč a dnes je C++ prostě jiný programovací jazyk než C. Pořád ale platí (více méně), že co se naučíte v C, to uplatníte i v C++.

Tolik stručně z historie. Pokud by vás zajímala více, na internetu jistě najdete podrobnější informace o historii.

Ukázka zdrojových kódu

Jak jsem napsal dříve, program vytváří ze zdrojového kódu překladač. Podívejte se tedy nejdříve na to, jak vypadá takový zdrojový kód v jazyce C. Připomínám, že si pro tento kurz můžete stáhnout všechny zdrojové kódy. Název souboru, ve kterém je zdrojový kód uložen jsem vždy napsal do komentáře na začátek zdrojáku (Takže první zdrojový kód najdete v souboru hello.c)

```
1.  /*----------------------------------------------*/
2.  /* c03/hello.c                                  */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.      printf("Hello World\n");
9.      return 0;
10. }
11. /*----------------------------------------------*/
```

Pokud vidíte na začátcích řádků čísla, tak ty nejsou součástí zdrojového kódu. Jsou tu na mé webové stránce jen pro to, abych se na čísla řádků mohl odkazovat v textu (např. na řádku 2 vidíte název souboru c03/hello.c).

Tak málo stačí napsat k tomu, abyste mohli vytvořit kompletní funkční program. Jak snadné, že? Význam toho co je tam napsáno osvětlím později. Program vytvořený z tohoto zdrojového kódu neudělá nic jiného, než že vytiskne na obrazovku (v

„textovém režimu“, tj. například v okně DOSu nebo příkazového řádku) „Hello World“ a přesune kurzor na další řádek (odřádkuje).
V C++ se předchozí program zapíše takto:

```
/*-----------------------------------------------*/
  /* c03/hello.cpp                               */

#include <iostream>

int main(void)
  {
std::cout << "Hello World\n";
  return 0;
  }
/*-----------------------------------------------*/
```

*Kód jazyka C můžete přeložit i pomocí překladače jazyka C++, ale zdroják C++ již jen pomocí překladače C++. Překladač jazyka C nezná (by neměl znát) operátor **<<**, ani std::cout.*
Tímto vás nechci nabádat, abyste používali na jazyk C překladač jazyka C++, protože (v tom lepším případě) by vás mohli obtěžovat zbytečné varování překladače, v tom horším případě dříve nebo později narazíte na nějakou nekompatibilitu, protože C a C++ jsou už prostě dva různé programovací jazyky, ačkoliv v základu velmi, velmi podobné.
Zdrojové kódy v jazyce C se zapisují do souboru s příponou **.c** a v jazyce C++ s příponou **.cpp**. Některé (starší) překladače vyžadují místo cpp příponu cc (nejen shodnout se na příponě zdrojových kódů bylo pro tvůrce překladačů nadlidský úkol). Některým překladačům je přípona ukradená. Důležité je, že zdrojový kód je **obyčejný textový soubor**, který otevřete třeba v notepadu (poznámkový blok). Zdrojový kód rozhodně nemůžete uložit ve formátu Microsoft Word atp.
O tom jak z těchto zdrojových souborů vytvořit program bude další kapitola.

Vývojová prostředí a překladače

V minulé kapitole jsem ukázal zdrojové kódy pro jazyk C a C++. Nyní vám představím programy, které vám pomohou z těchto zdrojáků vytvořit skutečné programy.

- Windows API
- Překladač pro Linux
- Code::Blocks
- Bloodshed Dev-C++ a wxDev-C++
- CodeLite
- DJGPP
- Visual Studio
- Visual C++ Express Edition
- Borland C++ Builder
- Embarcadero Free C++ Compiler / Stareter edition
- Borland C++ Builder Compiler
- Eclipse
- NetBeans
- Anjuta
- Závěr

Samotný překladač umožní vytvořit ze zdrojáků program, ale s rostoucími projekty budete možná chtít více než jen překladač. **Vývojová prostředí** (zkratka **IDE** = Integrated development environment) je sada nástrojů, která vám budou pomáhat s laděním programů, hledáním chyb, udržováním přehledu ve zdrojovém kódu, nápovědou k funkcím atd.
Vývojová prostředí nejsou jen chytré textové editory s barevným zvýrazňováním zdrojového kódu, ale kompletní sada nástrojů, která vám bude pomáhat se psaním zdrojového kódu.
Překladač se anglicky nazývá compiler, proto někdy uslyšíte místo „překladač“ počeštěné „kompiler“ a místo „překládat“ „kompilovat“.
Pro začátek ale nebude chyba, když si budete myslet, že vývojové prostředí je jen chytřejší textový editor, který umí barevně zvýrazňovat syntaxi zdrojového kódu a spustit překladač, který vytvoří ze zdrojového kódu program. Důležité je si uvědomit, že je jedno jestli zdroják napíšete v nějakém nabušeném vývojovém prostředí, nebo notepadu – program ze zdrojáku vytváří **překladač** (který můžete spustit sami, nebo pomocí vývojového prostředí).

Windows API

Než vám představím několik překladačů do Windows, udělám úkrok stranou a zmíním se o tzv. **Windows API**(API = application programming interface, tj. rozhraní pro programování aplikací). Windows API není nic jiného, než sada knihoven (instrukcí napsaných pro jazyk uložených v nějakém souboru), které můžete využívat při programování k vytváření „windowsovských“ aplikací. Do vašeho kódu stačí vložit tento řádek:

```
#include <windows.h>
```

Toto vloží hlavičkový soubor windows.h. V hlavičkovém souboru jsou popsány funkce knihovny windows (překladač ten popis potřebuje). Pak můžete používat funkce z windows knihovny, jako například WinMain()(namísto funkce main(), viz příklad hello.c), nebo CreateWindow() pro vytvoření okna atd. Můžete tak pomocí tohoto rozhraní snadno vytvářet a ovládat standardní objekty Windows (jako ta zmíněná okna, nabídky, tlačítka, roletové menu atd. atd.).
Hlavičkový soubor windows.h a přislušnou knihovnu funkcí nevytváří nikdo jiný, než sám Microsoft. Nejedná se tedy o standardní knihovnu a jinde než na Windows se s ní nesetkáte.
Později se v tomto kurzu naučíte psát vlastní hlavičkové soubory a k čemu vlastně jsou.
API je něco, co si může vymyslet a vytvořit každý, není součást standardu jazyka C ani C++. Já se Windows API věnovat nebudu. Tím, že jsem se zmínil že existuje, kdo ho vytváří a k čemu je, považuji téma Windows API za uzavřené :-). Pouze se dále budu zmiňovat o tom, které překladače umí Windows API a které ne, protože si myslím, že se k Windows API jednou nejspíš dostanete. Chcete-li se naučit používat Windows API, stejně se nejdříve musíte naučit jazyk C/C++.

Překladač pro Linux

Protože jsem se rozhodl věnovat programování v Linuxu v samostatném kurzu, seznámím vás teď jen letmo s překladačem **gcc**, resp. **g++**, které se používají v Linuxu k překladu nejčastěji. Zde uvedené informace by vám měli vystačit až do konce tohoto kurzu o C/C++.

Teď tedy ukáži, jak použít překladač gcc k vytvoření programu. (Už jste napjatí?) Uživatelé Windows mohou tuto část s klidem přeskočit.

Překladač gcc je většinou již v Linuxu nainstalován. Pokud ne, určitě bude součástí standardních balíčků vaší distribuce.

Vezměte si zdrojové kódy z předchozí kapitoly a přeložte je příkazem:

```
$ gcc -o hello hello.c
```

respektive pro C++:

```
$ g++ -o hello hello.cpp
```

„hello" je název výsledného spustitelného programu. Spustíte jej jednoduše příkazem:

```
$ ./hello
```

Soubory hello.c (hello.cpp) jsou textové soubory obsahující zdrojový kód. Místo **gcc** můžete mít překladač cc a místo **g++** c++. Vyzkoušejte, který máte. (Záleží na vaší distribuci Linuxu/Unixu).

*Používejte textové editory, které dokáží barevně zvýraznit syntaxi programovacího jazyka. V Linuxu je jich hafo (vim, emacs, nedit atd.). I ve Windows se najdou dobré editory (například PSPad). Jinou možností je používat komplexní **vývojová prostředí**, o kterých bude řeč dále. Vždycky však ukládejte zdrojové kódy jako text a ne jako např. „dokument Microsoft Word".*

Překladač **gcc** se v Linuxu používá k překladu většiny programů napsaných v C/C++, dokonce i k překladu samotného jádra Linuxu. Některá z vývojových prostředí níže popsaných existují i pro Linux a k překladu využívají právě gcc. (I když u většiny vývojových prostředí je možné si nastavit, jaký překladač chcete používat).

Kromě překladače gcc existuje v Linuxu i novější překladač **clang**. Ten si vytknul za cíl být rychlejší než gcc a vytvářet i rychlejší (optimalizovanější) kód. Nejspíš ho také najdete mezi balíčky vaší linuxové distribuce. clang má navíc daleko srozumitelnější chybové hlášky než gcc, proto vám jej doporučuji používat místo gcc.

Code::Blocks

Code::Blocks je asi to nejlepší vývojové přostředí, co můžete mít zdarma. Existuje verze pro Linux i Windows. Spolupracuje s mnoha překladači (například s C++ Builder, který umí Windows API, nebo s gcc/g++, a to i ve Windows), dokáže importovat projekty z Bloodshed Dev-C++, je rozšiřitelný mnoha pluginami, je přehledný a snadno v něm vytvoříte multiplatformní programy (pro Linux i Windows). Code::Blocks je určitě správná volba. Code::Blocks obsahuje i „něco jako" prostředí RAD (viz Visual Studio níže), ale to bohužel není ani zdaleka na tak vysoké úrovni jako u komerčních produktů. K dispozici je i český manuál.

Více o Code::Blocks

Bloodshed Dev-C++ a wxDev-C++

Bloodshed je (kdysi jedno z nejpopulárnějších) free vývojové prostředí, využívající k překladu minGW (což je přenesená verze linuxového gcc). Je malý, rychlý a jednoduchý. Je to velice pěkný program a mohu ho začátečníkům jen doporučit, protože jeho jednoduchost je pro ně velikou výhodou. A umí česky. Jeho vývoj, jak se zdá, v roce 2005 umřel. Vývoj Dev-C++ ale převzala jiná parta programátorů, nazvala jej wxDev-C++. Další, aktuálně udržovaný klon tohoto vývojového prostředí, se jmenuje Orwell Dev-C++.

Více o Dev-C++

CodeLite

O CodeLite by se dalo napsat skoro to samé, jako o Code::Blocks. Tyto dvě prostředí jsou přímí konkurenti a je těžké říct, které je lepší. Osobně mám raději Code::Blocks, ale vám se třeba bude víc líbit CodeLite. Budete si je muset vyzkoušet :-). Nesporná výhoda CodeLite je, že je jeho uživatelské rozhraní přeloženo do češtiny.

Více o CodeLite

DJGPP

DJGPP je linuxový překladač gcc přenesený do prostředí Windows a je zdarma. Kromě překladače také obsahuje sadu nástrojů a knihoven, které vám pomohou vytvářet 32-bitové programy v prostředí MS-DOS/MS-Windows. DJGPP pro vás bude tou pravou volbou, pokud budete chtít psát programy pro Linux i MS-DOS, nebo vytvářet přenositelné hry a grafické programy, nebo si prostě zkusit „linuxové programování" v prostředí Windows. Na DJGPP je také výborné, že pracuje jak ve starém MS-DOSu, tak ve Windows 3.0 i v těch nejnovějších Windows které máte. Nezabere také mnoho místa.

Lze jej používat jako gcc z příkazové řádky, nebo můžete využít vývojové prostředí RHIDE, které je velice podobné legendárnímu IDE (Integrated development environment) od Borlandu.

Více o DJGPP

Visual Studio

Visual Studio je nejlepší nástroj pro vývoj aplikací pro Windows. Jedná se o tzv **RAD** (Rapid Applications Development) prostředí. RAD je nejen pěkný marketingový název. V tomto prostředí máte vizuální pomůcky pro sestavování aplikace. Prostě a jednoduše, vidíte před sebou formulář a na ten můžete myší natahat běžné prvky z Windows jako například tlačítka, záložky, menu atp. Změnu vlastností prvků můžete nastavit v přehledném editoru vlastností.

Visual Studio je úzce spjato s technologií .NET.

Na jeho koupi si připravte pár tisíc korun. Můžete si však stáhnout a vyzkoušet jeho Trial verzi (cca 3.3 GiB dat). Zaručuji vám 90 dní plných zábavy a neutuchajícího nadšení. Pro náš kurz programování v C/C++ je to však poměrně nevhodné prostředí (kladivo na komára).

Více o Visual Studiu (a C++ Express Edition)

Visual C++ Express Edition

Visual C++ Express Edition je takový menší bratříček Visual Studia a je **zdarma**. Vlastně není až tak malý, instalace zabírá zhruba 300MB místa :). Rozhraní je totožné, jako u Visual Studia, jen toho umí méně (mnohem méně) a neobsahuje 2GiB nápovědy jako Visual Studio. Přesto je to nejlepší prostředí pro vývoj aplikací ve Windows API, co je zdarma.

Visual je úzce svázán s platformou .NET a je navržen pro programování v C++, ale nehodí se moc pro náš výukový kurz jazyka C. Pro začátečníky je totiž možná až příliš složitý. Přesto, všechny zdrojové kódy z tohoto tutoriálu upravil tak, aby fungovali i ve Visual Studiu.

Více o Visual Studiu (a C++ Express Edition)

RAD Studio

C++ Builder bylo, obdobně jako Visual Studio, **RAD** vývojové prostředí od legendární společnosti Borland. Po neslavném konci firmy Borland převzala vývoj frima Embarcadero. Můžete si stáhnout trial verzi RAD Studia (po zaregistrování). Kdysi to byla jednička ve vývojovém prostředí, nyní je to dvojka (byla mu vyčítána hlavně nestabilita a chybovost, ale to snad už dnes není

pravda). Prostředí je určeno pro jazyk C++. Koupit se dá cca od 95 000 Kč (v roce 2016). Jinak se o něm dá napsat to samé, co o Visual Studiu.

Stáhnout RAD Studio (free trial)

Embarcadero Free C++ Compiler / Stareter edition

Kromě RAD Studia si můžete také stáhnout moderní Free C/C++ Compiler. A nejen to.
Také si můžete stáhnout vývojové prostředí **Starter edition** a dokud nebudete vydělávat víc jak 1000$ za rok, tak ho používat zdarma.

Stáhnout C++Builder Starter Edition (Klikněte na „Buy Now" a vyberte si verzi zdarma)
Stáhnout Free C++ Compiler (Není potřeba, pokud si stáhnete Start Edition)

Borland C++ Builder Compiler

Borland C++ Builder Compiler je pouze překladač pro jazyky C/C++, který můžete ovládat z příkazové řádky. Je to stejný překladač, který jste mohli najít v Borland C++ Builderu, je zdarma (musíte se jen zaregistrovat) a je bez jakéhokoliv vývojového prostředí. Lze jej používat z příkazové řádky (jako např. DJGPP nebo gcc), nebo jej může používat s Code::Blocks (jupíí). Existuje pouze verze pro Windows.
Kompiler je z roku 2000, takže už neodpovídá moderním standardům C++. Na C je snad ještě dobrý, ale nějak mě nenapadá, proč by jste jej měli používat, když je tu tolik alternativ …

Stáhnout C++ Builder compiler (C++ Compiler 5.5).

Existuje i free prostředí s názvem Turbo C++, ale poslední verze je z roku 2006, má netušeně zastaralé požadavky na systém a po jeho instalaci se mi ani nespustil (házelo to nějaké chyby). Prostě s ním nemá smysl ztrácet čas.

Eclipse

Eclipse je velice pěkné vývojové prostředí, ale především pro Javu. Do Eclipse lze pomocí pluginů zakomponovat podporu pro spoustu dalších programovacích jazyků, včetně C/C++. Bohužel, nainstalovat takové pluginy a naučit se je ovládat dá (začátečníkům) dost práce. Prostředí je to vhodné především pro lidi, kteří programují v Javě, nebo pro ty, co programují v několika jazycích a chtějí používat jednotné vývojové prostředí. Eclipse existuje ve verzi pro Windows i Linux.

Více o Eclipse

NetBeans

Vývojové prostředí napsané v Javě a primárně pro Javu, ale pomocí pluginu lze používat i pro C/C++. Existuje verze pro Linux i Windows a je zdarma. Toto prostředí je přímým konkurentem pro Eclipse. Je mladší a osobně jej mám rači :-). Určitě najdete na netu spoustu hádek o tom, které z těchto prostředí je lepší :-). NetBeans (ani Eclipse) nejsou dodávány s překladačem jazyka C, takže si budete muset nějaký doinstalovat a nastavit …

Více o NetBeans

Anjuta

Anjuta je Linuxové vývojové prostředí napsané pro GTK/GNOME, kdysi velmi oblíbené. Bývá součástí standardních balíčků linuxových distribucí. Možná se vám bude líbit. Jako překladač využívá gcc /g++ (co taky jiného v Linuxu :-).

Více o Anjuta

Závěr

Asi vám teď jde ze všech těch vývojových prostředí a překladačů hlava kolem. Pro začátečníka není lehké vybrat si, tak vám zkusím trochu poradit.
Pokud budete programovat v Linuxu, vývojové prostředí vlastně vůbec nebudete v tomto kurzu potřebovat. Programy zde probírané se většinou skládají z jednoho, max. dvou souborů a k jejich správě opravdu nepotřebujete složité vývojové prostředí. Vystačíte si s obyčejným textovým editorem (třeba Vim nebo Emacs) a překladačem gcc / g++ / clang.
Pokud už si chcete vyzkoušet vývojové prostředí a nebojíte se angličtiny, berte Code::Blocks. Pokud se bojíte angličtiny, zkuste CodeLite nebo Anjutu.
Pokud chcete programovat ve Windows i Linuxu, jediná správná volba je Code::Blocks.
Pokud chcete programovat jen ve Windows, můžete začít s tím nejjednodušším – (Orwell) Dev-C++. Nebo, jestli se nebojíte angličtiny, vyberte si Code::Blocks. A nebo, pokud máte pro strach uděláno, jděte rovnou do Visual C++ Express Edition. (Na to byste měli dříve nebo později přejít tak jako tak, pokud se chcete orientovat jen na Windows). Výhoda těchto vývojových prostředí je, mimo jiné, že se instalují rovnou s překladačem.

Struktura programu

Vše o vzniku programu již bylo objasněno a tak se můžeme věnovat čistě jen syntaxi jazyka C. V této kapitole na prvním jednoduchém příkladě osvětlím velkou spoustu nových věcí. Některé z nich se budou probírat později podrobněji. Pokud pochopíte věci vysvětlované v této kapitole, již se můžete začít považovat za programátory amatéry :-).

- První program
- Komentáře
- Knihovny
- Funkce
- Funkce printf()
- Prohlížení výstupu

První program

Správně bych měl říkat první zdrojový kód, ale myslím, že říkat "program" není zase tak velký prohřešek. Tak mě za to nekamenujte. Však vy si z toho zdrojového kódu program vytvoříte.
Podívejte se tedy na něj. Následující kód můžete uložit do souboru s příponou **.c** (např. kod.c). Přípona .c je pro některé překladače (zvláště v Linuxu) povinná, někdy však ne a soubor se může jmenovat libovolně. Nevidím však jediný rozumný důvod, proč ji nepoužívat. **Ve jménech souboru nepoužívejte českou diakritiku ani mezery!**

```
/*-------------------------------------------------*/
  /* c04/kod.c                                     */

  /* Toto je libovolny komentar
   * v souboru kod.c          */

#include <stdio.h>            //standardni knihovna

int main(void)
  {
```

```
    printf("Hello World\n");
    return 0;
}
/*----------------------------------------------*/
```

Pokud vidíte čísla na začátku řádků, tak ty nejsou součástí zdrojového kódu. Budu je však uvádět před každým zdrojovým kódem, abych se na ně mohl v textu odkazovat.

Všimněte si mezer, odsazování a konců řádků ve zdrojovém kódu a vůbec celkové úpravy. Bílé znaky, jako je mezera, tabulátor nebo znak nového řádku většinou ve zdrojovém kódu jazyka C nemají žádný význam. Odsazováním program pouze získá na přehlednosti a čitelnosti. Funkci main() byste mohli napsat do jednoho řádku, překladači to bude jedno.

```
#include <stdio.h>
int main(void){printf("Hello World\n");return 0;}
```

Posuďte sami, co je přehlednější. Konce řádků jsou důležité pouze pro direktivy začínající znakem # (čti „šarp“) a u komentářů začínajících dvěma lomítky (viz komentář //standardni knihovna), protože jejich konec je dán právě koncem řádku. Mezera musí zůstat i mezi int a main (a return a 0), aby se to neslilo do jednoho slova.

Na netu se dají najít 30 stránkové dokumenty o správném formátování zdrojového kódu v C. Pokud se je naučíte, riskujete, že až přijdete pracovat do nějaké firmy, dostanete úplně jiný 30 stránkový manuál ☺.

Komentáře

Tím nejjednodušším, co je v programu vidět, jsou **komentáře**. Vše co je mezi znaky /* a */ je komentář. V jazyce C++ se navíc považuje za komentář vše za dvěma lomítky // až do konce řádku. Většina moderních překladačů C také rozezná dvě lomítka jako komentář (mnohdy je překladač C a C++ jeden a ten samý program).

Až budete dělat větší programy, uvidíte jak je dobré mít správně okomentovaný zdrojový kód. **Dobré komentáře vám pomohou se v kódu orientovat**. Je dobré poznamenávat takové věci, jako třeba *... toto číslo musí být nezáporné z toho a toho důvodu ...*. Při změně programu v budoucnosti se tím vyvarujete chyb.

Knihovny

Existují takzvané standardní knihovny, což jsou knihovny dodávané s překladačem, a uživatelské knihovny, které si vytváří programátor sám. **Hlavičkové soubory** jsou soubory, v nichž jsou uloženy definice funkcí z knihovny.

Jejich přípona je většinou **.h** (jako header), ale není povinná (stejně jako .c). Pro C++ se někdy knihovny označují příponou .hpp.

Direktiva #include říká překladači, jaký hlavičkový soubor má načíst před překladem zdrojového kódu. Pokud je jméno v takovýchto lomených závorkách < >, pak se soubor hledá ve standardních adresářích. Které to jsou, to záleží na překladači.

Standardní knihovna **stdio** (která obsahuje funkci printf()) je tak notoricky používána, že některé překladače ani nevyžadují uvedení direktivy #include <stdio.h>. Přesto, v zájmu kompatibility, tuto direktivu používejte (pokud budete používat nějaké funkce z této knihovny).

*Zkratka stdio = **st**an**d**ard **i**nput **o**utput (knihovna pro standardní vstup a výstup)*

Jména uživatelských hlavičkových souborů jsou uzavřeny ve dvojitých uvozovkách a hledají se v aktuálním adresáři. Například #include "kkk/knihovna.h" je soubor se jménem knihovna.h v podadresáři kkk/aktuálního adresáře. Jak si vytvořit vlastní hlavičkový soubor vysvětlím později. Nejdříve se budete seznamovat se standardními knihovnami a funkcemi.

Funkce

Funkce v C/C++

V příkladě jsou v programu dvě funkce. Funkce main() a funkce printf(). Funkce main() se ve zdrojáku **definuje**. To znamená, že se popisuje co tato funkce bude dělat (co je jejím obsahem). Funkce printf() se pouze **volá**, tj. chceme po ní, aby udělala to, co je dáno její definicí. Je to funkce definovaná v knihovně stdio.h a její definice způsobí vypsání textu na obrazovku.

Funkce má své jméno (main), návratovou hodnotu (int) a argumenty určené v závorkách za jménem funkce ((void)). Návratová hodnota určuje, jaká data funkce vrátí při svém skončení (např. int je celé číslo). K čemu to je, to se dozvíte později. Argumenty jsou pro změnu data, která funkce dostává ke zpracování. Hodnota void znamená „nic“, tedy v příkladu výše to znamená, že funkce main() žádné argumenty neočekává.

Návratová hodnota je vždy jen jedna a její typ se píše před jméno funkce. Argumenty se píší za jméno funkce do závorek a je-li jich více, oddělují se čárkou.

Funkce printf() dostává jako argument řetězec znaků (řetězce jsou vždy ve dvojitých uvozovkách), který pak vytiskne na obrazovku. Posloupnost znaků \n reprezentují nový řádek (ENTER, chcete-li).

Tělo funkce je všechno to, co je mezi složenými závorkami {}. Funkce se ukončuje klíčovým slovem returnza kterým je návratová hodnota funkce (funkce main() vrací celé číslo (int), a to 0).

Funkce jsou poměrně složitou záležitostí a budu se jim ještě věnovat později podrobněji. Důležité je vědět, že **funkce main() je speciální funkce, která je volána[1)] v programu jako první**. Ve funkci main() pak můžete volat další funkce, dle libosti. Z toho také vyplývá, že funkci main() musí mít každý program, jinak by překladač nevěděl čím začít při provádění programu. Funkce main() má vždy návratovou hodnotu typu int (celé číslo).

Funkce printf()

Z toho přísunu nových informací můžete být dost zmateni. Hlavně proto, že jsem u spousty věcí říkal, že je proberu podrobněji později :-). Pokud jste se prokousali až sem, pak již máte vyhráno. Už víte, jak udělat program v jazyce C. Nejdříve začnete funkcí main(), protože ta se volá v programu jako první. Přidáte hlavičkové soubory knihoven pomocí #include, abyste mohli používat v nich nadefinované funkce a ty pak voláte ve funkci main().

Funkce main() skončí příkazem return. Pokud byste jej neuvedli, pravděpodobně byste program také přeložili, ale překladač by vás na tuto chybu upozornil varováním (v jazyce C, v jazyce C++ to chyba není a standard předpokládá return 0. Ach, ty standardy). Stejně tak by bylo odpustitelnou chybou, kdyby jste za jménem funkce main() do závorek nenapsali void ale nechali je prázdné. Překladač by si „void" domyslel sám. Funkce main() může být buďto bez argumentů, nebo se dvěma speciálními argumenty, o kterých vám povím později :-).
Na závěr ukážu ještě jeden program a další využití funkce printf(). Funkce printf() má jako první argument textový řetězec. Ten může obsahovat speciální sekvence. Už jsme se s jednou setkali. Sekvence \npřesune kursor na nový řádek. Funkci printf(). později vysvětlím systematičtěji i se všemi speciálními znaky. Teď vám chci ukázat ještě dva, které se nám budou při výkladu hodit. První je %s. Za tento znak se dosadí textový řetězec, který je dalším argumentem funkce printf(). Druhý je %i, za který se dosadí celé číslo. Prostudujte si následující program a zkuste napsat vlastní.

```
/*----------------------------------------------*/
/* c04/kod2.c                                   */

#include <stdio.h>

int main(void)
{
printf("1 + 1 = %i\n", 1 + 1);
printf("%i + %i = %i\n", 1, -2, 1 + (-2));
printf("%s\n", "Konec programu.");
/* return ukonci program. */
return 0;
printf("Tohle se uz nezobrazi %s!\n");
}

/*----------------------------------------------*/
```

V příkladu jsou některé arugmenty funkce printf() **výrazy**. Například 1 + 1 je výraz použitý jako druhý argument při prvním volání funkce printf(). Prvním argumentem je řetězec, tedy vše v uvozovkách. Zatímco řetězec se nijak nezpracovává, výraz se vyhodnotí a až jeho výsledek (1 + 1 = 2) se stane argumentem funkce.
Všimněte si posledního volání funkce printf(). Je až za příkazem return a proto k jejímu volání v programu nedojde. Navíc obsahuje chybu. V řetězci je %s ale funkce nemá žádný další argument. Pokud by tato funkce byla volána, „sáhla" by si kamsi do paměti pro neexistující řetězec. Buď by se vytiskly nějaké nesmysly, nebo by program skončil s chybou (neoprávněný přístup do paměti; program provedl nepovolenou operaci atp.). Takže si na to dávejte pozor. Na tuto chybu vás překladač nemusí upozornit, protože překladač zkontroluje jen to, že funkce printf() má mít jako první argument textový řetězec (a to má). Jelikož je tato chyba opravdu častá (a taky dost nebezpečná), dokáží moderní překladače tyto chyby odhalit, ale mnohdy se o to musí „požádat" nějakým nastavením nebo přepínačem (třeba s překladačem gcc můžete použít přepínač -Wall).
A to je pro dnešek vše. Přečtěte si tuto kapitolu tolikrát, kolikrát bude potřeba, abyste zdrojový kód kod2.c celý pochopili. Zvykněte si na pojmy „volání funkce", „definice funkce", „argumenty funkce" a „návratová hodnota funkce". Určete všechny části zdrojového kódu, které se k těmto pojmům vztahují.

Prohlížení výstupu

Tak dobře, ještě to není vše. Často mi totiž lidi píšou, že po přeložení a spuštění programu jenom problikne okno a nic nevidí. Proto jsem se rozhodl napsat ještě tento dodatek.
Některé vývojové prostředí fungují tak, že otevřou okno pro výstup programu, spustí program a po jeho ukončení okno zase hned zavřou. Nemáte tak šanci stihnout výstup přečíst.
Existuje několik možností, jak tento problém vyřešit. Například nastavením tzv. breakpointů, nebo nastavením vývojového prostředí, aby okno po zkončení programu neuzavíral. To jsou ale věci, které se vývojové prostředí od vývojového prostředí liší. Jednodušší pro mě bude vám popstat třetí způsob:
Vložte na začátek souboru #include <stdlib.h> a před řádek s return vložte volání funkce system("pause");.
Knihovna <stdlib.h> deklaruje funkci system(). Tato funkce zavolá program pause[2], který nedělá nic jiného, než že čeká až uživatel zmáčkne nějakou klávesu. Tím pádem program neskončí, dokud něco nestisknete a můžete si tak v klidu prohlédnout jeho výstup.
Další způsoby jak pozastavit výstup můžete najít na stránkách, kde popisuji použití jednotlivých vývojových prostředí.

Datové typy, bloky, deklarace a definice

Základní datové typy

V minulé kapitole jste se setkali s datovými typy int a void. void je takový zvláštní datový typ, který vlastně říká, že žádná data nebudou. int je jméno pro celé číslo (nelze jej použít pro číslo s desetinnou částí), které má nějakou velikost (viz tabulka datových typů).

Základní datové typy jazyka C

Název	Bitů	Význam	Příklad
char	8	celé číslo, znak	0, 255, 'a', 'A', 'e'
short[1]	16	krátké celé číslo	65535, -32767

Název	Bitů	Význam	Příklad
int	16/32	celé číslo	--||--
long[1)]	32	dlouhé celé číslo	--||--
long long[1)]	64	ještě delší celé číslo	9223372036854775807, -9223372036854775807
enum	8/16/32	výčtový typ	
float	32	racionální číslo	5.0, -5.0
double	64	racionální číslo s dvojitou přesností	5.0l, 5l, -5.0L
long double	80	velmi dlouhé racionální číslo	5.5L, 5l, -5.0l
pointer	16/32/64	ukazatel	

Kompletní tabulku můžete najít např. na Wikipedii

V prvním sloupci vidíte základní datové typy jazyka C. Ve druhém vidíte to, kolik zabírají (orientačně) bitů v paměti. Čím více bitů, tím větší číslo je možné zapsat. Na druhou stranu, zabírají více paměti a práce s nimi je pomalejší. Ve třetím sloupci jsou významy čísel a ve čtvrtém příklady.

Co je na jazyku C zajímavé (a za co ho někteří nemají rádi) je to, že velikost všech datových typů není standardem pevně dána. Můžete se spolehnout je na to, že sizeof(char) = 1 (1 byte) [2)] a že:

sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

Proč? Závisí to především na procesoru. Na 16-bitovém procesoru bude mít int velikosti 16 bitů (16bitovému procesoru se s takto velkým číslem "nejlépe" pracuje), na 64-bitovém procesoru zase 64-bitů. Ovšem kvůli zpětné kompatiblitě se klidně může stát, že 64-bitový překladač bude používat pro int jen 32 bitů. Pro ukazatel (pointer) by ale měl používat 64-bitový systém 64-bitů, protože 32 bitů není dostatečně velký typ na adresování celého rozsahu paměti.

Mimochodem, možná to ještě nevíte, ale už víte, proč 32-bitové počítače mohou mít max. 4GiB paměti. Hádejte, jak velké je maximální číslo, které se vejde do 32 bitů?

Při výběru vhodného datového typu byste měli počítat s tím, že rozsah datového typu (tj. nejmenší a největší číslo, které do něj můžete nacpat) se bude lišit „počítač od počítače".

Pokud budete mít při návrhu pocit, že by mohlo dojít k „přetečení" rozsahu (takhle se říká tomu, když se snažíte do nějaké proměnné uložit větší čislo, než se tam vejde), používejte pro ověření velikosti čísla konstanty z knihovny <<limits.h>.

Konkrétní rozsahy datových typů můžete najít ve standardní knihovně <limits.h>.

Zjistit velikost datového typu můžete pomocí operátoru sizeof(), který vrací počet bytů (nikoliv bitů) datového typu (nebo proměnné). Například sizeof(char) by se vždy mělo rovnat 1.

Další možnosti ukáži o několik kapitol dále, a to v příkladu s binárními operátory (to teď ale pusťte klidně z hlavy).

Typy char, short, int, long a long long mohou být se znaménkem nebo bez znaménka. Určí se to pomocí klíčových slov **signed** (se znaménkem) nebo **unsigned** (beze znaménka). „signed" je defaultní hodnota, proto ji není třeba psát.

Vzpomeňte si na kapitolu o bitech a bajtech, kde jsem psal o reprezentaci záporných čísel.

K čemu jsou datové typy?. Když počítač počítá s nějakými čísly, musí je mít uložené někde v paměti. Toto místo v paměti získáte tak, že vytvoříte **proměnnou**. A aby počítač věděl, kolik místa takové proměnné má v paměti dát, musíte proměnné určit datový typ. To dává smysl, ne? :-).

Proměnná je tedy nějaké vyhrazené místo v paměti o nějaké velikosti (dané datovým typem), kam si můžete ukládat nějaké čísla (resp. nejen čísla, ale i testové řetězce a další složitější struktury, o kterých bude řeč později).

Následující příklad snad osvětlí použití datových typů a proměnných:

```
/*----------------------------------------------*/
  /* c05/typy.c                                 */

#include <stdio.h>
int a;

int main(void)
{
/* deklarace promennych */
unsigned int b;
float c;

a = 5;
b = 11;
a = a / 2;
c = 5.0;
printf("a=%i\nb=%u\nc=%f\n", a, b, c / 2);

return 0;
}

/*----------------------------------------------*/
```

Výstup z tohoto programu bude následující:

```
a=2
b=11
c=2.500000
```

Všimněte si, že do proměnné *a* lze uložit pouze celé číslo. Takže a/2, tedy 5/2 je 2 a ne 2.5. Celé číslo (int) prostě neukládá desetinnou část. Výsledek se ani nezaokrouhlil, prostě se „useknul".
Vyzkoušejte si do proměnné *b* uložit záporné číslo (třeba -11). Schválně, uhodnete, co se stane?

Boolovské datové typy

V některých programovacích jazycích se používá datový typ **bool**, který může nabývat pouze hodnot **true**(pravda) nebo **false** (nepravda). Tyto hodnoty se používají ve vyhodnocování podmínek při řízení běhu programu a podmíněných operátorů. Oboje budu probírat později. Na tomto místě jen řeknu, že v jazyce C takovýto typ neexistuje.
V jazyce C se jako **nepravda** vrací 0 a jako **pravda** 1. A dále platí, že cokoliv nenulového se vyhodnocuje jako pravda (tj. vše kromě 0 a NULL). Čísla 2, 0.1, -1, všechny jsou *pravda*.
Nic není ztraceno. V jazyce C si můžete vytvářet vlastní datové typy pomocí operátoru typedef, nebo si taky můžete hodnoty *true* a *false* nadefinovat pomocí preprocesoru. Ale na to zapomeňte :-). Standard C99 (stejně jako C++) totiž přináší knihovnu <stdbool.h>, která to už za vás udělala.
Takže odvolávám co jsem odvolal :-). Datový typ **bool** můžete používat, pokud zahrnete do svého zdrojáku <stdbool.h>:

```
#include <stdbool.h>
...
bool x = true;
x = false;
...
```

Příklad vám zatím neukážu, protože to nemá smysl, dokud se nedostanete k vyhodnocování podmínek.
K čemu že je to vlastně dobré si definovat bool? Kvůli čitelnosti. Když si chcete udržovat informaci o tom, zda je něco pravda nebo ne, je čitelnější to ukládat do bool jako true nebo false, než třeba do char jako 0 nebo 1 (nebo 0.1 …). Zdrojové kódy by se měli v první řadě psát tak, aby byli dobře čitelné pro programátory.

Bloky, deklarace a definice

Blokem se rozumí vše, co je mezi složenými závorkami { a }. V příkladě nahoře je jediný blok, a to tělo funkce main(). Všechny proměnné, které deklarujete v bloku, jsou tzv. lokální proměnné a proměnné mimo blok jsou globální. Lokální proměnné platí jen v těle bloku a v blocích vnořených (jinde je překladač neuvidí, tj. jinde, než v daném bloku, s nimi nemůžete pracovat).
Proměnné jsou místa kdesi v paměti počítače, které definujete v programu. Velikost místa v paměti závisí na datovém typu proměnné. Do proměnných lze vkládat hodnoty, měnit je a číst. V příkladě typy.c jsem vytvořil globální proměnnou *a* (je mimo blok, tj. mimo tělo funkce main()) a dvě lokální proměnné *b* a *c*.
Deklarací proměnných se určuje, jaké proměnné se v programu objeví. Nejdříve se určí datový typ a za ním názvy proměnných oddělené čárkou. Deklarace se ukončí středníkem. Při deklaraci je možné do proměnné rovnou vložit hodnotu. Názvy proměnných nesmí obsahovat mezeru, národní znaky (č, ř, š atp.) a nesmí začínat číslem. Jejich maximální délka je závislá na překladači. Jazyk C rozlišuje malá a velká písmena, takže můžete vytvořit proměnné se jmény Ahoj, ahoj, aHoj a program je bude chápat jako různé identifikátory. Dobrým zvykem je **pro proměnné používat vždy jen malá písmena**.

```
unsigned int i,j,k;
signed int cislo1, cislo2 = cislo3 = 25;
```

Dokud neurčíte jakou má proměnná hodnotu, může v ní být jakékoliv číslo. Např. proměnné *cislo2* a *cislo3* mají hodnotu 25, všechny ostatní obsahují náhodnou hodnotu!
Překladač jazyka Java vám nedovolí použít proměnnou před přiřazením hodnoty, překladač jazyka C/C++ ano, ale pořád je to chyba. Nenechte se zmást tím, že v proměnné, které nepřiřadíte hodnotu, je většinou nula. Tak to může být v 999 případech z 1000, ale pak to tak jednou nebude a to může vést k nedpředvídatelným chybám …
Proměnné mohou být v jazyce C deklarovány pouze na začátku bloku funkce – ihned za úvodní závorkou ({) (tzv. lokální proměnné), nebo na začátku programu (globální proměnné). Lokální proměnné „existují" jen v bloku, ve kterém jsou definovány (případně v blocích v něm vnořených) a nelze je tudíž používat mimo něj. V jazyce C++ je možné deklarovat proměnné téměř kdekoliv, ale stále platí, že proměnná má platnost pouze v bloku, ve kterém je deklarována.
Definice říká, jak bude nějaký objekt (funkce, datový typ) vypadat. V programu například definuji funkci main(). Základní datové typy jsou definovány normou ANSI C. Můžete definovat i vlastní typy dat, další funkce atd., ale k tomu se dostanu později. Jen si prosím nepleťte pojmy definice a deklarace. Definovat objekt můžete jen jednou (velikost typu int musí být pro všechny zdrojové soubory stejná), deklarovat vícekrát (můžete deklarovat mnoho proměnných typu int).

Textové řetězce

Už víte, jak si vytvořit proměnnou pro celé čísla různých velikostí, čílsa proměnné velikosti nebo 8-bitový znak[3]. Jak ale zapsat nějaký textový řetězec?
Text jsou vlastně jenom znaky poskládané zasebou. Jinak řečeno, je to pole znaků. Textové pole je souvislá řada znaků, která je v paměti, někde začíná a někde končí. Pro textové řetězce neexistuje žádný datový typ. Místo toho si pouze uložíte ukazatel na začátek pole a pak pracujete s tímto ukazatelem. Takže než ukážu jak pracovat s textem, nejdřív musím vysvětlit, co jsou ukazatele.

Ukazatele

Ukazatele (též zvané pointry) jsou jednou z nejtěžších věcí na pochopení v jazyku C. Ukazatele jsou proměnné, které **uchovávají adresu** ukazující do paměti počítače. Při jejich deklaraci je třeba uvést, na jaký datový typ bude ukazatel ukazovat. Paměť počítače je adresována po bytech. Číslo 0 ukazuje na první bajt v paměti, číslo 1 na druhý bajt atd. Pokud máte šestnáctibitový překladač/processor, bude velikost ukazatele 16 bitů, u 32 bitového překladače 32 bitů.
Deklarace ukazatele na nějaký datový typ vypadá takto:

```
typ * jmeno;
```

Například:

```
float *uf; // Ukazatel na float. Má velikost typu "ukazatel".
int *ui;
void *uv;
```

Proměnné *uf*, *ui* a *uv* jsou ukazatele.

Proměnná *uf* je ukazatel na typ float, *ui* je ukazatel na typ int a *uv je tzv. prázdný ukazatel u kterého není určeno, na jaký datový typ se bude ukazovat. To překladači neumožňuje typovou kontrolu, proto je lepší jej nepoužívat, pokud to není nutné. Do všech třech proměnných lze vložit libovolné (celé) číslo, které pak bude překladačem chápáno jako adresa do paměti. Všichni tři ukazatelé mají stejnou velikost (16, 32 nebo 64 bitů - podle překladače).*
K získání adresy nějaké proměnné slouží operátor & (viz příklad níže). K získání dat z adresy, která je uložena v ukazateli, slouží operátor * (viz příklad).

```
/*----------------------------------------------*/
  /* c05/ukazatel.c                             */

#include <stdio.h>

int main(void)
  {
int i;      /* promenna int */
  float f, f2;
int *ui;    /* ukazatel (na typ int) */
  float *uf;

  f = 5.5;
uf = 50;    /* v ukazateli uf je hodnota 50. Co je v pameti na 50 bajtu
    nikdo nemuze vedet, proto se takto do ukazatele adresa
    nikdy (!) neprirazuje */
uf = &f;    /* v ukazateli uf je hodnota adresy promenne f (tak je to
    spravne :-)) */
  ui = &i;

 f2 = *uf;   /* do f2 se priradi hodnota z promenne f, tedy 5.5 */
*ui = 10;   /* tam, kam ukazuje adresa v ui (tedy do i) se ulozi 10.
    Hodnota promenne ui se nezmeni (je to stale adresa
    promenne i) */

printf("f = %f\n", f); /* hodnota v f se od prirazeni 5.5 nezmenila */
printf("f2 = %f\n", f2); /* do f2 se priradila hodnota z promenne f, na
    kterou jsem se odkazoval pomoci ukazatele uf */
 printf("i = %i\n", i); /* do i jsem pres ukazatel ui priradil 10 */

printf("uf = %u, *uf=%f\n", uf, *uf); /* vytiskne hodnotu uf a hodnotu
    na teto adrese (tj. adresu f2 a
    hodnotu f2) */
  return 0;
  }
/*----------------------------------------------*/
```

Výstup z programu bude následující:

```
f = 5.500000
f2 = 5.500000
i = 10
uf = 3221224276, *uf=5.500000
```

Proměnná *uf* obsahuje adresu paměti, na které byla proměnná *f* uložena. Ta se bude při každém spuštění programu lišit, neboť paměť přiřazuje programu operační systém. Do které části paměti program nahraje nelze předem určit (alespoň my, obyčejní smrtelníci, to nedokážeme).
V programu jsem se dopustil dvou chyb. Jednak to bylo přímé přiřazení hodnoty typu integer do proměnné *uf*(uf = 50;) – měl bych přiřazovat hodnotu typu ukazatel. A pak v poslední funkci printf(), kde se tiskne hodnota ukazatele uf jako celé číslo bez znaménka (unsigned int, spec. znak %u). Je možné, že vám tyto konstrukce překladač ani nedovolí přeložit!
Tisknutím ukazatele (pointeru) jako unsigned int jsem chtěl zdůraznit, že je to číslo (které uchovává nějakou adresu paměti). Ukazatele (pointery) by se správně měli tisknout přes %p (pak se vytisknou v 16tkové soustavě). Nahraďte řádek 31 tímto:

```
printf("uf = %p, *uf=%f\n", uf, *uf); /* vytiskneme si hodnotu uf ...
```

Za zmínku ještě stojí přiřazení f2 = *uf;. Jak to funguje? Nejdříve se vezme hodnota z uf – to je adresa &dnash; a z této adresy se přečte číslo dlouhé **tolik bitů, kolik bitů má typ, pro který je ukazatel určen**. Proměnnou *uf* jsem deklaroval jako ukazatel na typ float, který má 32 bitů. Proto se vezme 32 bitů začínajících na adrese uložené v *uf* a tyto bity se pak chápou jako racionální číslo a jako takové se uloží do *f2*.
Kdybych deklaroval *uf* jako ukazatel na typ char (*char *uf;*), pak by se sice začalo číst ze stejné adresy, ale z ní by se přečetlo jenom 8 bitů!
Asi vám v tuto chvíli není moc jasné, k čemu jsou ukazatele dobré. K tomu se však velice rychle dostanu. Zatím je třeba, aby jste pochopili, k čemu slouží operátor & a * a také pochopili, že ukazatel, ať už ukazuje na jakýkoliv typ, je stále jen ukazatel.
Z toho také vyplývá, že velikost ukazatele je stejná, ať už ukazujete na char, nebo long double:

```
sizeof(char*) == sizeof(long double*).
```

Všimněte si taky, že se hvězdička používá ke dvoum různým účelům. V deklaraci proměnné říká „toto je ukazatel". Jako operátor říká „nepracuj s ukazatelem, ale s proměnnou, na kterou ukazatel ukazuje".

NULL

NULL je speciální hodnota, která označuje velké nic. Používá se v souvislosti s ukazateli. NULL můžete vložit do ukazatele a pak můžete testovat, zda ukazatel obsahuje NULL, nebo adresu někam. NULL se také často používá jako argument funkcí, když tam,

kde je vyžadován jako argument ukazatel, žádný ukazatel vložit nechcete. Praktické využití této hodnoty ukáži později, například v ukázce použití NULL (zatím na to nekoukejte, nebo vám to vypálí oči).

Klíčová slova, konstanty, řetězce

V minulé kapitole jsem probral základní datové typy. Nyní vám ukáži další objekty, které vám jazyk C nabízí. Například **pole**, pomocí nichž se dají vytvářet (mimo jiné) textové řetězce.

- Klíčová slova
- Konstanty
- Řetězce
- Escape sekvence

Klíčová slova

ANSI norma jazyka C určuje následující slova jako klíčová. Tyto slova mají v jazyce C speciální význam a nelze je používat jako uživatelem definované identifikátory (např. jména funkcí, proměnných, konstant atd.). Jejich význam postupně proberu kapitolu po kapitole. Zatím se jejich významem nemusíte zatěžovat.

auto	break	case	const	continue	char	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

Standard C99 přináší tyto nová klíčová slova:

inline	_Bool	_Complex	_Imaginary

Konstanty

Konstanty jsou objekty, jejichž hodnota se za chodu programu nemění. Konstanty se definují za klíčovým slovem **const**. Po něm následuje datový typ konstanty. Pokud ne, překladač jej chápe jako int. Dále je v definici identifikátor konstanty a za rovnítkem hodnota.

```
const float a = 5;
```

Konstanty mají stejné vlastnosti jako proměnné, až na tu maličkost, že jejich hodnota se nedá změnit. Používají se v programu namísto přímého vložení hodnoty (čísla, textu atp.).

Pokud takovou konstantu použijete v programu na několika místech, stačí změnit její hodnotu v definici a tak se změní hodnota na všech místech programu kde byla použita. Pokud byste používali místo konstant přímo hodnoty, museli byste procházet celý program a měnit hodnotu na všech místech, kde je to potřeba. To je nejenom pracné, ale také si to říká o to, že někde zapomenete …

Použití konstant také ulehčuje práci překladači s optimalizací výsledného programu. Pokud překladač ví, že se hodnota nebude měnit, může ji uložit do optimální části paměti. Některé jednoduché výpočty s konstantami může provést už za překladu.

Jako konstantu můžete označit i argument fukce. Tím dáte najevo, že hodnota, kterou funkce dostane, se nebude uvnitř bloku funkce měnit. Pokud byste se konstatní proměnnou pokusili někde v těle funkce změnit, překladač by to nedovolil. Výhodou použití konstant není jen to, že to překladači pomáhá s optimalizací výsledného programu, ale i vám jakožto programátorům to pomůže s čitelností zdrojového kódu. (Stačí se podívat na deklaraci funkce a hned víte, že se argument funkce nebude nijak měnit.)

```
1.  /*----------------------------------------------*/
2.  /* c06/konst.c                                  */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.  const a = 5;            /* konstantu a vyhodnoti prekladac jako int 5 */
9.  const b = 5.9;          /* konstantu b vyhodnoti prakladac jako int 5 */
10. const float c = 5;      /* konstanta c je urcena jako float,tj. 5.0 */
11. int i = 0;
12.
13. printf("a/2 = %f\n", (a / 2) * 1.0);      /* celociselne deleni ... */
14. printf("b/2 = %f\n", (b / 2) * 1.0);      /* celociselne deleni ... */
15. printf("c/2 = %f\n\n", (c / 2) * 1.0);    /* deleni s desetinnou carkou ... */
16.
17. printf("Mala nasobilka pro cislo %i\n\n", a);
18.
19. printf("%i * %i = %i\n", a, i, i * a);
20. i = i + 1;
21. printf("%i * %i = %i\n", a, i, i * a);
22. i = i + 1;
23. printf("%i * %i = %i\n", a, i, i * a);
24. i = i + 1;
25. printf("%i * %i = %i\n", a, i, i * a);
26. i = i + 1;
27. printf("%i * %i = %i\n", a, i, i * a);
28. i = i + 1;
```

```
29.  printf("%i * %i = %i\n", a, i, i * a);
30.  i = i + 1;
31.  printf("%i * %i = %i\n", a, i, i * a);
32.  i = i + 1;
33.  printf("%i * %i = %i\n", a, i, i * a);
34.  i = i + 1;
35.  printf("%i * %i = %i\n", a, i, i * a);
36.  i = i + 1;
37.  printf("%i * %i = %i\n", a, i, i * a);
38.  i = i + 1;
39.  printf("%i * %i = %i\n", a, i, i * a);
40.
41.  return 0;
42.  }
43.
44.  /*----------------------------------------------*/
```

Výstup z programu:

```
a/2 = 2.000000
b/2 = 2.000000
c/2 = 2.500000

Mala nasobilka pro cislo 5
5 * 0 = 0
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Výsledkem programu je malá násobilka čísla 5. Pokud chcete malou násobilku např. čísla 4, stačí změnit konstantu *a*. Asi si říkáte, že kdyby *a* nebyla konstanta ale proměnná, vyšlo by to nastejno. Jenomže pak byste mohli v nějakém velkém programu zapomenout na to, že se hodnota *a* nemá měnit a překladač by vám změnu (na rozdíl od konstanty) umožnil. Až se jednou budete vracet k vlastnímu zdrojovému kódu třeba po půl roce, budete rádi, že máte důsledně oddělené proměnné od konstant (a také, když budete mít zdrojový kód dobře okomentován :-).

Další výhodou konstant je, že za výrazy jako je `(a/2)*1.0` může už překladač dosadit výsledek, protože vše potřebné je v době překladu známo. (Pravda ale je, že v takto jednoduchém programu by překladač dokázal poznat jakou bude mít proměnná *a* hodnotu, i kdyby nebyla označena jako konstanta.)

Používejte konstanty všude kde je to vhodné. Slouží to jako ochrana před nechtěnou změnou konstanty, jako komentář „tato hodnota je konstantní“ a pomáhá to překladači optimalizovat rychlost a velikost výsledného programu!

Řetězce

Kromě konstantních čísel můžete mít také v programu konstantní znaky a řetězce. Definice znakové konstanty není nijak překvapivá:

```
const char pismeno = 'a';
```

V jazyku C je znakový literál ('a') překvapivě typu int. V příkladu výše se automaticky převede na typ char a uloží do proměnné pismeno, takže se o to vlastně nemusíte starat. Jazyk C++ už definuje znakový literál jako typ char.

S konstantními řetězci je to už horší. Nejdříve vám musím říct, co to vlastně textový řetězec je. Jedná se o pole znaků. Co jsou to pole vysvětlím podrobněji později. Zatím stačí, když budete vědět, že **pole** je spojité místo v paměti vyhrazené pro posloupnost objektů.

Textový řetězec je tedy pole znaků (ať už se jedná o konstantní pole nebo ne). Navíc má tu zvláštnost (na rozdíl od ostatních polí), že má na konci tzv. zarážku. Je to znak, který obsahuje nulu (číslo nula). Tak se pozná, kde řetězec v paměti končí. Kde začíná je uloženo v ukazateli.

Takto se jednoznačně určí, kde se textový řetězec nachází a co do něj patří. Zatímco znaky jsou v jednoduchých uvozovkách ('), textové řetězce ve dvojitých ("). Nulový znak na konci textového řetězce za vás vloží překladač sám (za znakem ne – konec konců, znak je jen jeden znak …).

```
char * const veta = "Toto je konstantni retezec.";
```

Ukazatel *veta* je ukazatel na typ char a ukazuje na začátek textového řetězce, v příkladě na místo v paměti, kde je uloženo písmeno T. Pokud bude s řetězcem, na který ukazatel *veta* ukazuje, pracovat například funkce printf(), podívá se do paměti kam *veta* ukazuje, přečte první písmeno, pak se podívá na další bajt (velikost typu char je 8 bitů) a přečte další písmeno atd., až narazí na bajt obsahující nulu a tím skončí.

Klíčové slovo const v předchozí delaraci označuje jako konstantu pole, na které veta ukazuje. Ale pozor! Textový literál (tj. textový řetězec který v programu zapíšete do uvozovek) nelze měnit v žádném případě! Překladač takové literály ukládá do speciální části paměti, kam nesmíte nic zapisovat (pokud to zkusíte, program selže – bude zabit operačním systémem).

Pro označení jako konstanty ukazatele dejte const před hvězdičku.

```
const char *veta const = "Toto je konstantni retezec.";
```

Toto je konstantní ukazatel na konstantí pole. Nejen že nemůžete změnit kam ukazuje ukazatel, ale nemůžete změnit ani to, na co ukazuje. Za domácí úkol zkuste přijít na to, jak definovat konstantní ukazatel na nekonstantí pole :-).

Jak vytvořit „nekonstantí“ pole (znaků), proberu až v kapitole Pole a ukazatele.

V jazyce C neexistuje žádný datový typ „řetězec“. Na řetězec (obecně jakékoliv pole objektů) se můžete odkazovat pouze pomocí ukazatelů.

```
1.  /*---------------------------------------------*/
2.  /* c06/konst2.c                                */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.  const char Kpismeno = 'a';  /* konstanta */
9.  char pismeno = 'b';
10. const char *veta = "Konstantni veta !\n";
11. const char *veta2 = "a";
12. char *ukazatel;           /* ukazatel na typ char */
13.
14. ukazatel = &pismeno;      /* ukazatel ukazuje na promennou pismeno */
15.
16. printf("1: Tisknu pismena: %c %c\n", Kpismeno, pismeno);
17.
18. printf("2: Tisknu vetu: %s", veta);
19. printf("3: Tisknu vetu: %s\n", veta2);
20.
21. printf("4: Tisknu pismeno: %c\n", *ukazatel);
22.
23. printf("5: Tisknu pismeno: %c\n", *veta);
24. printf("6: Tisknu pismeno: %c\n", *veta2);
25.
26. return 0;
27. printf("7: Tisknu vetu: %s\n", ukazatel);
28. printf("8: Tisknu vetu: %s\n", pismeno);
29. /* printf("9: Tisknu: %s\n",*pismeno); je uplna blbost */
30. /* *veta = 'C'; chybne prirazeni */
31. }
32.
33. /*---------------------------------------------*
```

Výstup z programu:

```
1: Tisknu pismena: a b
2: Tisknu vetu: Konstantni veta !
3: Tisknu vetu: a
4: Tisknu pismeno: b
5: Tisknu pismeno: K
6: Tisknu pismeno: a
```

V prvním volání funkce printf() se tisknou znaky tak, jak byste očekávali. Když funkce printf() narazí na sekvenci %c najde si svůj další argument a ten vytiskne jako znak.

Ve druhém volání se tiskne věta, na kterou ukazuje konstantní ukazatel *veta* a ve třetím volání se tiskne věta, na kterou ukazuje ukazatel *veta2*. Funkce printf() ve chvíli, kdy narazí na sekvenci %s si najde odpovídající argument, který bere jako adresu do paměti. Z této paměti začne po znacích číst a znaky tisknout, dokud nenarazí na nulový znak.

Věta "a" obsahuje kromě písmena 'a' také onu zarážku (nulový bajt), je tedy 2 bajty dlouhá (narozdíl od znaku 'a' uloženého v proměnné *Kpismeno*).

Ve čtvrtém volání tiskne funkce hodnotu, na kterou ukazuje proměnná *ukazatel*. Proměnné *veta* a *veta2* jsou také ukazatele na typ char, úplně stejně jako *ukazatel*. Jen mají při své deklaraci přiřazenou hodnotu (odkaz na řetětězec v uvozovkách).

Ukazatele *veta* a *veta2* ukazují na začátky vět, tedy přesněji, obsahují adresu paměti, na které je uloženo první písmeno věty. Proto se v pátém a šestém volání printf() vytiskne první znak věty.

Za příkazem return 0; se nacházejí ještě další tři volání funkce printf(). Tyto volání jsou chybné.

Sedmé volání printf() se pokouší vytisknout větu, na kterou ukazuje ukazatel *ukazatel* (jak šťastně pojmenováno :-).

Jenomže *ukazatel* neukazuje na větu, ale na znak v proměnné *pismeno*. Překladač nemůže kontrolovat kam proměnná *ukazatel* v době spuštění ukazuje (zda na řetězec nebo znak nebo jestli vůbec někam …) a tak nepozná chybu. Pokud by došlo k takovémuto volání funkce *printf()*, vytiskl by se nejdříve znak v proměnné *pismeno* (tj. znak 'b') a pak by se tiskli další a další bajty z paměti počítače, dokud by se nenarazilo někde na nulu. Takže by se vytiskla hromada nesmyslů, nebo by program zkolaboval (neoprávněný přístup do paměti). (Kdybych inicializoval proměnou *ukazatel* např. takto: ukazatel = veta;, bylo by sedmé voláníprintf() naprosto v pořádku.)

Ovšem osmé volání je jednoznačně chyba. Namísto ukazatele jsem předal proměnnou typu char. Na tuto chybu vás již překladač upozorní, ale přesto dokáže program přeložit. Již jsem řekl, že adresa v ukazateli není nic jiného než číslo v bitech, stejně tak písmena (a vůbec všechny data v počítači). V osmém volání funkce printf() vezme znak v proměnné *pismeno* a použije jej jako adresu do paměti, ze které se pokusí přečíst větu. V tomto případě je téměř jisté, že vám program zkolabuje kvůli neoprávněnému přístupu do paměti, nebo se vypíše hromada nesmyslů, stejně jako v sedmém volání printf().

Číselná hodnota písmena 'b' je 98. Poslední volání funkce printf() by se tedy snažilo vytisknout větu začínající na 98. bajtu v paměti (kdyby toto volání překladač umožnil).

Deváté volání je opravdu nesmysl, protože proměnná *pismeno* není ukazatel, nelze použít operátor *(hvězdička, odborně *dereference*; ukazateli se totiž říká reference …). Toto by měl překladač odmítnout přeložit.

Na posledním řádku je ještě zakomentované chybné přiřazení *veta = 'C';. To je syntakticky správně – na adresu, kam ukazuje *veta*, tj. do prvního písmene věty, se snažím uložit znak C (totéž lze zapsat jako veta[0] = 'C';).

Chyba je v tom, že *veta* ukazuje na pole označené jako konstantí. (Nechme teď stranou, že se navíc jedná o řetězcový literál, do kterého se stejně zapisovat nesmí.)

Escape sekvence

Escape sekvence jsou sekvence znaků, které vám umožňují vložit do řetězce některé zvláštní znaky. Přehled escape sekvencí vidíte v tabulce.

Escape sekvence

Escape znak	Význam	Popis
\0	NULL	Nula, ukončení řetězce (má být na konci každého řetězce)
\a	Alert (Bell)	Pípnutí. Na moderních počítačích už bohužel nefunguje.
\b	Backspace	návrat o jeden znak zpět
\f	Formfeed	nová stránka nebo obrazovka
\n	Newline	přesun na začátek nového řádku
\r	Carriage return	přesun na začátek aktuálního řádku
\t	Horizontal tab	přesun na následující tabelační pozici
\v	Vertical tab	stanovený přesun dolů
****	Backslash	obrácené lomítko
\'	Single quote	apostrof
\"	Double quote	uvozovky
\?	Question mark	otazník
\000		ASCII znak zadaný jako osmičková hodnota
\xHHH		ASCII znak zadaný jako šestnáctková hodnota

O tabulce ASCII se zmíním později.

Ještě dodám, že pro vytištění znaku % pomocí funkce printf() je třeba zapsat znak % dvakrát za sebou. Nejedná se o escape znak, ale o vlastnost funkce printf() interpretovat % ve svém prvním argumentu zvláštním způsobem.

Příklad:

```
1.  /*----------------------------------------------*/
2.  /* c06/esc.c                                    */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.  printf("%s %%\n", "%");
9.  printf("%s\r%s\n", "AAAAAAA", "BB");
10. printf("\"\a\"\n");
11. printf("\x6a\n");
12. return 0;
13. }
14.
15. /*----------------------------------------------*/
```

Výstup z programu:

```
% %
BBAAAAA
""
j
```

~~Při třetím volání printf() by se mělo ozvat krátké pípnutí.~~ Neozve se, protože nové počítače nemají pípátko.

Standardní vstup a výstup

Nyní na chvíli odbočím od výkladu syntaxe jazyka C a vysvětlím dvě funkce: printf() a její sestřičku scanf().

- Standardní vstup a výstup
  - Funkce printf()
    - Funkce fflush()

- Funkce scanf()
  - Načítání řetězců

Funkci printf() používám již od prvního programu (a budu ji používat až do posledního) a tak se sluší řádně popsat její možnosti. V souvislosti s těmito funkcemi také objasním pojmy standardní vstup a výstup. Jen připomínám, že s funkcí printf(), resp. s řetězci, které tato funkce tiskne, úzce souvisí tzv. escape sekvenceprobírané v minulé kapitole.

Standardní vstup a výstup

Každý program při svém spuštění dostane od operačního systému standardní vstup (obvykle klávesnice), standardní výstup (obvykle monitor) a standardní chybový výstup (obvykle taky monitor). Standardní vstup se označuje jako **stdin**, standardní výstup jako **stdout** a chybový výstup je **stderr**.

Vstup se řekne anglicky input a výstup output. Z toho vznikla často používáná zkratka I/O.

Funkce, které s těmito vstupy a výstupy pracují jsou definované v hlavičkovém souboru <stdio.h>. Například funkce printf() tiskne vše do standardního výstupu – stdout. V OS máte možnost standardní vstupy a výstupy přesměrovat (například výstup do souboru místo na obrazovku). Pokud používáte Linux, jistě vám je známé přesměrování standardního výstupu pomocí znaku **>** a chybového výstupu pomocí znaků **2>**. Pokud je vstup nebo výstup přesměrován, program to de facto ani nezpozoruje (ne že by to nešlo zjistit).

Vstupní a výstupní zařízení lze rozdělit na **znaková** a **bloková**. Znaková jsou například klávesnice a monitor, bloková např. pevné disky. Rozdíl je především v možnosti získávání či vkládání dat z/do zařízení. Ke znakovým zařízením se přistupuje **sekvenčně** tedy znak po znaku a k blokovým tzv. **náhodným přístupem** (random access).

Znamená to, že z blokových zařízení můžete přistupovat k datům dle libosti (můžete číst z kterékoliv části disku, nebo do kterékoliv části zapisovat), kdežto u sekvenčních zařízení lze číst nebo zapisovat pouze po sekvencích (např. na monitor nebo tiskárnu lze zapisovat pouze řádek za řádkem (alespoň dřív to nebylo možné jinak) a z klávesnice můžete číst jenom ty znaky, které uživatel právě stiskl (ty, které stiskl před minutou nebo stiskne až za minutu nezjistíte)).

Funkce printf()

Funkce **printf()** se používá pro formátovaný standardní výstup (stdout). Deklarace funkce printf() vypadá takto:

```
int printf (const char *format [, arg, ...]);
```

Něco málo o argumentech a návratových hodnotách funkcí jsem již psal v souvislosti s funkcí main(). Ve stejné kapitole jsem mluvil také o funkci printf(). Podrobně je proberu až v kapitole věnované vytváření funkcí.

Funkce printf() má návratovou hodnotu typu int. Tato hodnota je rovna počtu znaků zapsaných do výstupu, nebo speciální hodnota **EOF** v případě chyby zápisu.

Hodnota EOF se používá jako označení konce souboru (End Of File), taktéž konce standardního vstupu a výstupu. EOF je definován v souboru <stdio.h> (obvykle má hodnotu -1) a k jeho využití se ještě vrátím.

Prvním argumentem funkce printf() je konstantní řetězec, který funkce tiskne do stdout. Ten může obsahovat speciální sekvence (začínají znakem %), na jejichž místo dosazuje funkce další argumenty. Z toho vyplývá, že kolik je speciálních sekvencí v tomto řetězci, tolik dalších argumentů – oddělených čárkou – funkci printf()musíte předat. Argumenty by měli mít očekávaný datový typ, který je dán speciální sekvencí.

Formát speciální sekvence prvního argumentu funkce printf() je následující:

```
%[flags][width][.prec][h|l|L]type
```

Vše, co je v deklaraci v hranatých závorkách [ ] v deklaraci může, ale nemusí být. Svislítko | chápejte jako „nebo". Například [h|l|L] znamená, že v deklaraci sekvence může být h nebo l nebo L, nebo ani jedna z těchto možností (protože je to celé v hranatých závorkách).

Jak vidíte, sekvence začíná znakem procento a končí znakem určující typ argumentu, který se bude tisknout (a také jak (!) se bude tisknout). Jsou to jediné dvě povinné části. Například už znáte sekvenci %i která tiskne celé číslo se znaménkem. Pokud chcete vytisknout přímo znak procento, pak jej stačí napsat dvakrát za sebou (%%). V následující tabulce je přehled dalších možných typů.

Přehled typů speciální sekvence řetězce ve funkci printf()

type	význam
d, i	Celé číslo se znaménkem (Zde není mezi d a i rozdíl. Rozdíl viz scanf() níže).
u	Celé číslo bez znaménka.
o	Číslo v osmičkové soustavě.
x, X	Číslo v šestnáctkové soustavě. Písmena ABCDEF se budou tisknout jako malá při použití malého x, nebo velká při použití velkého X.
p	Ukazatel (pointer)
f	Racionální číslo (float, double) bez exponentu.
e, E	Racionální číslo s exponentem, implicitně jedna pozice před desetinnou tečkou a šest za ní. Exponent uvozuje malé nebo velké E.
g, G	Racionální číslo s exponentem nebo bez něj (podle absolutní hodnoty čísla). Neobsahuje desetinnou tečku, pokud nemá desetinnou část.
c	Jeden znak.

Přehled typů speciální sekvence řetězce ve funkci printf()

type	význam
s	Řetězec.

Podrobný popis viz manuálová stránka k funkci printf()

Příklad:

```
1.  /*---------------------------------------------*/
2.  /* c07/print.c                                 */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.  const char *COPYRIGHT = "(C)";
9.  const int ROK = 2003;
10.
11. printf("%i %u %o %x %X %f %e %G\n", -5, -5, 200, 200, 200, 10.0,
12.  10.0, 10.0);
13. printf("%s %i\n", COPYRIGHT, ROK);
14.  return 0;
15.  }
16. /*---------------------------------------------*/
```

Výstup bude následující:

```
-5 4294967291 310 c8 C8 10.000000 1.000000e+01 10
(C) 2003
```

Z čísla -5 se stalo při použití %u číslo 4294967291. To je dáno interpretací bitového zápisu, kdy se první bit zleva nepočítal jako identifikátor znaménka, ale jako součást čísla. Funkce printf() nemá tucha, jaký datový typ má její druhý argument, ona jen ví, na které adrese argument začíná. A podle %u hádá, že se jedná o int beze znaménka.

Položky **width** a **.prec** určují délku výstupu. Délka **width** určuje minimální počet znaků na výstupu. Mohou být uvozeny mezerami nebo nulami. **.prec** určuje maximální počet znaků na výstupu pro řetězce. Pro celá čísla je to minimum zobrazených znaků, pro racionální počet míst za desetinnou tečkou. Má to tedy více významů v závislosti na typu.

Položka **width** může nabývat následujících hodnot:

Hodnoty width

n	Vytiskne se nejméně **n** znaků doplněných mezerami
0n	Vytiskne se nejméně **n** znaků doplněných nulami
*****	Vytiskne se nejméně n znaků, kde n je další argument funkce printf()

Položka **.prec** může nabývat následujících hodnot:

Hodnoty .prec

.0	pro **e**, **E**, **f** nezobrazí desetinnou tečku
	pro **d**, **i**, **o**, **u**, **x** nastaví standardní hodnoty
.n	pro **d**, **i**, **o**, **u**, **x** minimální počet číslic
	pro **e**, **E**, **f** počet desetinných číslic
	pro **g**, **G** počet platných míst
	pro **s** maximální počet znaků
.*	jako přesnost bude použit následující parametr funkce printf()

Příklad:

```
1.  /*---------------------------------------------*/
2.  /* c07/print2.c                                */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
```

```
7.  {
8.    printf("%06i %06u %06x %6x %06f %06E %06G\n\n", -5, -5, 200,
9.    200, 10.0, 10.0, 10.0);
10.
11.  printf("%*s %1s %6s %%%06.2G\n", 10, "%*i", "%06.2f", "%06.0E");
12.
13.  printf("%*i %06.2f %06.0E %06.2G\n", 10, -5, 10.0 / 3, 10.0 / 3,
14.  10.0 / 3);
15.  printf("\n%.8s %0*.*f\n", "Posledni vystup:", 10, 4, 10.0 / 3);
16.  return 0;
17.  }
18.
19.  /*----------------------------------------------*/
```

Výstup z programu:

```
-00005 4294967291 0000c8     c8 10.000000 1.000000E+01 000010

%*i %06.2f %06.0E %06.2G
-5 003.33 03E+00 0003.3


Posledni 00003.3333
```

Všimněte si použití %% při druhém volání funkce printf() (řádek 11) a také rozdílu mezi %06.2f a %06.2G(řádek 13). Také si všimněte, že do délky čísla se započítává i pozice pro znaménko. Čísla, která jsou delší než požadovaná minimální délka se nezkracují, řetězce ano (viz „Posledni vystup“).
Na řádku 15 také vidíte, že do délky čísla (10) se započítává i desetinná tečka a délka desetinné části (4).
Jen pro jistotu ještě dovysvětlím začátek řádku 11: printf("%*s …. Hvězdička znamená „délku vezmi z argumentu funkce“. Takže délka tisknutého řetězce je 10 (druhý argument funkce printf()). Přičemž tisknutý řetězec je další argument, tj. "%*i". Ve výstupu si můžete všimnout, že je řetězec "%*i" odsazený sedmi mezerami.
Příznak **flags** může nabývat hodnot z následující tabulky:

Hodnoty flags	
-	výsledek je zarovnán zleva
+	u čísla bude vždy zobrazeno znaménko
mezera	u kladných čísel bude místo znaménka "+" mezera
#	pro **o**, **x**, **X** výstup jako konstanty jazyka C
	pro **e**, **E**, **f**, **g**, **G** vždy zobrazí desetinnou tečku
	pro **g**, **G** ponechá nevýznamné nuly
	pro **c**, **d**, **i**, **s**, **u** nemá význam.

Znaky **h l** a **L** označují typ čísla. Znak **h** typ short (nemá smysl pro 16bitové překladače), **l** dlouhé celé číslo, **L**long double.
Příklad:

```
1.  /*----------------------------------------------*/
2.  /* c07/print3.c                                 */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.  long int x;
9.  long double y = 25.0L;
10.  x = -25l;              /* je mozno psat jen x = -25, protoze
11.  prekladac vi, ze x je long a tak si
12.  cislo -25 prevede na -25l;
13.  takhle je to vsak "cistci" a citelnejsi */
14.
15.  printf("%10s <%+5i> <% 5ld> <%x>\n", "Cisla:", 25, x, -25);
16.  printf("%-10s <%-+5i> <% 5Lf> <%#x>\n", "Cisla:", 25, y, -25);
17.  return 0;
18.  }
19.
20.  /*----------------------------------------------*/
```

Výstup z programu:

```
Cisla: <  +25> <  -25> <ffffffe7>
```

```
Cisla:     <+25  > < 25.000000> <0xfffffffe7>
```

Funkce fflush()

Pozor! Pokud funkce printf() nevytiskne na konci znak nového řádku \n, může se stát, že výstup zůstane viset v tzv. **vyrovnávací paměti** a na obrazovku se nic nevypíše.

Vyrovnávací paměť (buffer) se vypíše, když funkce printf() vypíše znak nového řádku (\n), nebo když je buffer přeplněn, nebo když zavoláte funkci fflush().

```
int fflush(FILE *stream);
```

O datovém typu FILE budu psát až v souvislosti s prací se soubory. Teď vám bude stačit vědět, že když uvedete jako argument stdout, vyprázdní se buffer standardního výstupního souboru, což je to, co potřebujete :-).

```
#include <stdio.h>

int main(void) {
printf("Rekni neco: ");
    fflush(stdout);
      getchar();
       return 0;
         }
```

PS: to s tím bufferem nemusí fungovat vždy, záleží systém od systému, nebo jaký má váš OS zrovna náladu. Ale abyste měli jistotu, že váš program bude fungovat vždy, měli byste fflush() volat vždy, když hrozí, že se nevypíše ve správný okamžik to, co se vypsat má.

Funkce scanf()

Funkce scanf() se používá pro formátovaný standardní vstup (z **stdin**), což bývá obvykle vstup z klávesnice. Deklarace funkce scanf() vypadá takto:

```
int scanf (const char *format [, address, ...]);
```

Návratová hodnota je rovna počtu bezchybně načtených a do paměti uložených položek, nebo hodnota **EOF**(End Of File) při pokusu číst položky z uzavřeného vstupu. První argument je stejně jako u funkce printf()řetězec, který obsahuje speciální sekvence, určující, co se má načíst. Formát takové sekvence je následující:

```
%[*][width][h|l|L]type
```

Význam položek je následující:

Význam položek sekvence pro funkci scanf()	
*****	přeskočí popsaný vstup (načte, ale nikam nezapisuje)
width	maximální počet vstupních znaků
h|l|L	modifikace typu (jako u printf())
type	typ konverze.

Jak vidíte, sekvence vždy začíná znakem procento a končí typem konverze. Možné typy konverzí jsou v následující tabulce:

význam type	
d	celé číslo
u	celé číslo bez znaménka
o	osmičkové celé číslo
x	šestnáctkové celé číslo
i	celé číslo, zápis odpovídá zápisu konstanty v jazyce C, např. 0x uvozuje číslo v šestnáctkové soustavě
n	počet dosud přečtených znaků aktuálním voláním funkce scanf()
e, f, g	racionální čísla typu float, lze je modifikovat pomocí l a L
s	řetězec; úvodní oddělovače jsou přeskočeny, na konci je přidán znak '\0'
c	vstup znaku; je li určena šířka, je čten řetězec bez přeskočení oddělovačů
[search_set]	jako **s**, ale se specifikací vstupní množiny znaků, je možný i interval, například %[0-9], i negace, například %[^a-c].

Oddělovače jsou tzv. bílé znaky (tabulátor, mezera, konec řádku (ENTER)). Ty se při čtení ze vstupu přeskakují (výjimkou může být typ **c**). Načítání tedy probíhá tak, že se nejdříve přeskočí oddělovače a poté se načte požadovaný typ. Pokud je požadovaný typ například číslo, ale vy místo toho na vstupu zadáte písmeno, pak dojde k chybě.

Pokud se načte požadovaný typ, uloží se na adresu, která je uložena v dalším argumentu funkce scanf(). Volání funkce scanf() může vypadat např. takto:

```
int x;
scanf("%i", &x);
```

Takto se pokusí funkce scanf() načíst číslo a uložit jej do proměnné *x*, jejíž **adresa** je druhým argumentem funkce scanf(). Znovu zdůrazňuji, že je to adresa místa v paměti, kde je proměnná *x*, do které se načtená hodnota uloží. Adresa proměnné se získává pomocí operátoru **&** a může být uložena v ukazateli. Předchozí ukázka tak může vypadat i takto:

```
int x;
int *ui = &x;  /* ui je datoveho typu 'ukazatel na int' */
scanf("%i", ui);
```

Jestli začínáte mít zmatek ve všech těch hvězdičkách, ampersandech a procentech, dejte si kafe.

Pokud se to podaří, vrátí funkce číslo 1 (načtena jedna správná položka). Pokud se to nepodaří (např. místo čísla zadáte nějaké znaky, nebo vstup ukončíte), vrátí se EOF. Návratovou hodnotu vás naučím využívat až v kapitolách věnovaných řízení programu.

Příklad:

```
1.  /*----------------------------------------------*/
2.  /* c07/scan1.c                                  */
3.
4.  #include <stdio.h>
5.  #define _CRT_SECURE_NO_WARNINGS
6.  int main(void)
7.  {
8.  int x = -1;
9.
10. printf("Zadej cislo jako konstantu jazyka C\n"
11. "napr. 10 0x0a nebo 012: ");
12. scanf("%i", &x);
13. printf("Zadal jsi cislo %i\n", x);
14. return 0;
15. }
16.
17. /*----------------------------------------------*/
```

Makro _CRT_SECURE_NO_WARNINGS umožňuje použít funkci scanf() ve VS. VS ji bez tohoto makra považuje za nebezpečnou a nedovolí ji použít. Místo ní vám nabízí funkci scanf_s()(*s* jako secure), což je bezpečná alternativa k scanf(). Funkce scanf_s() není součástí standardu jazyka C, proto ji nedoporučuji používat, pokud nechcete psát pouze a výhradně pro Visual Studio.

Nebezpečnost funkce scanf() tkví v tom, že když její argumenty neodpovídají očekávaným argumentům dle formátovacího řetězce, může funkce scanf() zapsat data do špatné části paměti, což může vést, mimo jiné, ke spuštění záškodnického kódu. Microsoft zakazuje i další nebezpečné funkce a nabízí k nim alternativu s příponou **_s**. Obecně lze říci, že se jedná o všechny funkce, které čtou nebo zapisují někam do paměti, ale nemají parametr, který by omezoval délku čtených / zapisovaných dat. Makro _CRT_SECURE_NO_WARNINGS musí být uvedené před #include <stdio.h>, resp. před každou knihovnou, která deklaruje „nebezpečnou" funkci.

Když program přeložíte a spustíte, bude funkce scanf() čekat na vstup (z klávesnice), dokud nějaký nedostane, nebo dokud nebude vstup uzavřen. Vstup z klávesnice se programu odešle až po stisku klávesy ENTER.

Možné výstupy:

```
Zadej cislo jako konstantu jazyka C
napr. 10 nebo 0x0a nebo 012: -50
Zadal jsi cislo -50
Zadej cislo jako konstantu jazyka C
napr. 10 nebo 0x0a nebo 012: 0xff Tento text už se neneačte
Zadal jsi cislo 255
Zadej cislo jako konstantu jazyka C
napr. 10 nebo 0x0a nebo 012: ff
Zadal jsi cislo -1
```

Nyní již umíte vytvářet skutečně interaktivní programy (-:

Při posledním spuštění programu jsem zadal „ff", což není číslo ale textový řetězec, proto funkce scanf() do proměnné x nic neuložila, tak v ní zůstalo číslo -1.

Vstupní proud (stdin) můžete přerušit ve Windows a v DOSu klávesovou zkratkou CTRL+Z (stiskněte a držte CTRL a k tomu stiskněte písmeno "z"). V Linuxu pomocí klávesové zkratky CTRL+d. Vyzkoušejte.

Ukážu ještě jeden příklad a další si vymyslete sami. Kombinací speciálních sekvencí může být jak u funkce printf() tak scanf() mnoho, takže si s nimi vyhrajte a sledujte jak pracují. Zatím se nepokoušejte načítat řetězce – nejdřív budu muset vysvětlit práci s poli.

```
1.  /*----------------------------------------------*/
2.  /* c07/scan2.c                                  */
3.  #define _CRT_SECURE_NO_WARNINGS
4.  #include <stdio.h>
5.
```

```
6.  int main(void)
7.  {
8.  int x = 0;
9.  double f = 0.0;
10. char znak = 'a';
11.
12. printf("Zadej jeden znak a cislo max. 2 znaky dlouhe: ");
13. scanf("%c %2d", &znak, &x);
14. printf("\t--- zadal jsi %c a %i\n", znak, x);
15.
16. printf("Zadej retezec a pak racionalni cislo: ");
17. scanf("%*s %lf", &f);
18. printf("\t--- zadal jsi %f\n", f);
19.
20. return 0;
21. }
22.
23. /*----------------------------------------------*/
```

Možné výstupy z programu:

```
Zadej jeden znak a cislo max. 2 znaky dlouhe: B 55
--- zadal jsi B a 55
Zadej retezec a pak racionalni cislo: ahoj 15.3
--- zadal jsi 15.300000
```

Všimněte si, jak probíhá načítání ze standardního vstupu. V prvním příkladě jsem po spuštění programu napsal "B 55" a stiskl ENTER. Tím sem předal nějaký vstup a první volání funkce scanf() si načetlo nejdříve požadovaný znak a poté číslo. Pak proběhla funkce printf() a spustilo se druhé volání scanf(). Ta se zastavila a čekala na další vstup. Napsal jsem tedy řetězec "ahoj" a za ním racionální číslo 15.3 a poslal tento vstup stiskem ENTERu. Funkce scanf() řetězec přeskočila (%*s) a načetla číslo za ním.
Mezery, které jsou ve formátovacím řetězci funkce scanf(), nehrají žádnou roli a nemusí tam být. Snad jen, že je formátovací řetězec čitelnější.

```
Zadej jeden znak a cislo max. 2 znaky dlouhe: B 55 ahoj 15.3 nazdar
--- zadal jsi B a 55
Zadej retezec a pak racionalni cislo:   --- zadal jsi 15.300000
```

Při druhém spuštění programu jsem poslal na vstup rovnou znak, číslo, řetězec, další číslo a další řetězec a odentroval jsem to. Program při prvním volání scanf() načetl B a 55, spustil printf() a další volání scanf()pokračovalo ve čtení vstupu tam, kde předchozí scanf() skončilo.
Přeskočilo řetězec „ahoj" a načetlo číslo 15.3.
Mohli byste také zadat znak, stiskem ENTER výstup poslat programu, poté nějaké číslo a zase ENTER, řetězec a ENTER …
Zkoušejte si a sledujte, jak se program chová.
Všimněte si, že když zadáte číslo, jež má být dlouhé jen 2 znaky, pak každý další znak už druhé volání funkce scanf() vyhodnotí jako řetězec.
Také si uvědomte, že číslo může být chápáno též jako řetězec nebo znak. Jde jen o to, jak bude do paměti ukládáno. Pokud načítáte číslo 5, bude uloženo jako číslo 5, pokud načítáte znak 5 bude uložen jako číslo z ASCII tabulky odpovídající znaku 5, což je číslo 53.
Pokud se v prvním argumentu funkce scanf() vyskytne nějaký řetězec (nemluvím teď o bílých znacích), znamená to, že funkce scanf() má právě tento řetězec načíst (a nikam neuložit). Pokud však takový řetězec na vstupu není, dojde k chybě.

```
1.  /*----------------------------------------------*/
2.  /* c07/scan3.c                                  */
3.  #define _CRT_SECURE_NO_WARNINGS
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.  int zamek = 5;
9.  int navrat;
10.
11. printf("Hodnota zamku je nyni %+5i\n", zamek);
12. printf("Pokud chcete hodnotu zmenit, zadejte heslo a novou"
13. "hodnotu\n a stisknete ENTER. Napriklad: heslo 25\n>> ");
14.
15. /* heslo je 1234567890 */
16. navrat = scanf("1234567890 %i", &zamek);
17.
18. printf("Bylo nacteno %i spravnych polozek\n", navrat);
19. printf("Hodnota zamku je nyni %+5i\n", zamek);
20. return 0;
21. }
22.
23. /*----------------------------------------------*/
```

Výstup z programu:

```
Hodnota zamku je nyni    +5
Pokud chcete hodnotu zmenit, zadejte heslo a novou hodnotu
a stisknete ENTER. Napriklad: heslo 25
>> 1234567890 -2345
Bylo nacteno 1 spravnych polozek
Hodnota zamku je nyni -2345
```

Všimněte si, jak byla vložena návratová hodnota funkce scanf() do proměnné navrat. Byla načtena jedna správná položka, tj číslo do proměnné zamek. Řetězec 1234567890 se sice taky správně přečetl, ale nikam neuložil, takže se nepočítá.

```
Hodnota zamku je nyni    +5
Pokud chcete hodnotu zmenit, zadejte heslo a novou hodnotu
a stisknete ENTER. Napriklad: heslo 25
>> heslo 25
Bylo nacteno 0 spravnych polozek
Hodnota zamku je nyni    +5
```

Načítání řetězců

Načítat řetězce znak po znaku není úplně efektivní. Než vám ale ukáži, jak načítat celé řetězce (pomocí sekvence `%s`), musím vám vysvětlit, co to vlastně řetězce jsou (jak jsou reprezentovány v paměti). Takže příklad na načítání řetězců je až v kapitole Pole a ukazatele.

Práce s typy dat

Již od začátku výkladu se potýkáte s bity a bajty, s adresami a ukazateli a datovými typy. Neustále se snažím vysvětlit kdy se jakým způsobem bity reprezentují, kdy jako číslo, kdy jako znak, kdy jako číslo se znaménkem a kdy bez atd. V této kapitole si důležité věci zopakujete a dozvíte se poslední informace, které vám chybí. Po prostudování této kapitoly by vám již mělo být naprosto jasné, jak a proč se bajty interpretují tím či oním způsobem.

- O přetypování a ASCII tabulce
  - ASCII tabulka
  - Přetypování
    - Přetypování návratové hodnoty funkce
    - Chyby přetypování

O přetypování a ASCII tabulce

Přetypování je způsob, jak změnit interpretaci nějakého datového typu, ať již proměnné, či konstanty. Například, když dělíte dvě celá čísla, výsledkem je zase celé číslo. To se vám nemusí vždy hodit. Také víte, že znaky jsou jenom bity a bajty, tudíž je lze reprezentovat jako číslo a stejně tak lze (celé) číslo reprezentovat jako znak. Jakému číslu je přiřazen jaký znak, to určuje tzv. **ASCII tabulka**. Někdy zase můžete chtít reprezentovat číslo typu signed int (celé číslo se znaménkem) jako unsigned int (celé číslo bez znaménka).

ASCII tabulka

(ASCII = American Standard Code for Information Interchange)

Vnitřní interpretace znaku je v jazyku C typu **int** a v jazyku C++ typu **char**. Velikost char je 8 bitů, což je rozsah čísel od 0 do 255. Jakému číslu je přiřazen jaký znak, to určuje právě ASCII tabulka.

Pokud vytvoříte nějaký čistě textový soubor (v 8-bitovém kódování), tedy soubor obsahující jen znaky, pak co jeden znak, to jeden bajt. Bez ohledu na to, že v jazyce C je znak typu int – to je pouze „vnitřní interpretace znaku", tj. hodnota, se kterou pracuje program. Při vstupu/výstupu do/z programu proudí vždy 8 bitů.

Výjimkou může být znak konec řádku, který se v DOSu a Windows reprezentuje dvojicí znaků s kódem (číselnou hodnotou) 10 a 13. Naproti tomu v Linuxu se zapisuje jen jeden znak – 10. (Takže to vlastně žádná vyjímka není, prostě jsou to někdy dva znaky :-))

Některým číslům není přidělen viditelný znak. Například kód 65 odpovídá znaku velké A, ale kód 1 odpovídá stisku kláves CTRL+a a nemá žádný zobrazitelný ekvivalent. **Interpretace číselného kódu se může měnit, v závislosti na použitém kódování** (kódy 128-255). Pod těmito čísly jsou uloženy například znaky s háčky a čárky, které se bohužel kódování od kódování liší. Např. pro češtinu se použají kódování windows-1250 (na Windows), iso-8852-2 (ve starších verzích Linuxu) atp. Většina znaků má v obou kódování stejné číslo, ale některé (s interpunkcí) mají jiná čísla. Proč tomu tak je, na to se zeptejte Billa Gatese.

Zde vidíte tu zajímavější část ASCII tabulky:

ASCII tabulka

kód	znak	pozn.	kód	znak	pozn.
1		CTRL+a	65	A	
2		CTRL+b	66	B	
3		CTRL+c	67	C	
4		CTRL+d	68	D	
5		CTRL+e	69	E	

6		CTRL+f	70	F	
7		CTRL+g	71	G	
8		CTRL+h, BackSpace	72	H	
9		CTRL+i, Tab	73	I	
10		CTRL+j, stránka	74	J	
11		CTRL+k	75	K	
12		CTRL+l	76	L	
13		CTRL+m	77	M	
14		CTRL+n	78	N	
15		CTRL+o	79	O	
16		CTRL+p	80	P	
17		CTRL+q	81	Q	
18		CTRL+r	82	R	
19		CTRL+s	83	S	
20		CTRL+t	84	T	
21		CTRL+u	85	U	
22		CTRL+v	86	V	
23		CTRL+w	87	W	
24		CTRL+x	88	X	
25		CTRL+y	89	Y	
26		CTRL+z	90	Z	
27		CTRL+[,Esc,Ctrl+Esc	91	[	
28		CTRL+\	92	\	
29		CTRL+]	93	]	
30		CTRL+^	94	^	
31		CTRL+_	95	_	podtržítko
32		mezera	96	`	akcent
33	!		97	a	
34	"		98	b	
35	#	sharp	99	c	
36	$		100	d	
37	%		101	e	

38	&	ampersand
39	'	apostrof
40	(	
41	)	
42	*	
43	+	
44	,	čárka
45	-	mínus
46	.	tečka
47	/	
48	0	
49	1	
50	2	
51	3	
52	4	
53	5	
54	6	
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	@	

102	f	
103	g	
104	h	
105	i	
106	j	
107	k	
108	l	
109	m	
110	n	
111	o	
112	p	
113	q	
114	r	
115	s	
116	t	
117	u	
118	v	
119	w	
120	x	
121	y	
122	z	
123	{	
124	|	svislá čára
125	}	
126	~	tilda (vlnovka)
127	□	CTRL+BackSpace

Jiným typem kódování než je ASCII, je například UTF-8. To využívá k interpretaci znaku proměnlivý počet bytů. To je pro jazyk C trochu problém, protože pro jeden znak vyhrazuje vždy stejný počet bitů (obvykle 8). Pokud pracujete s řetězcem jako celkem, je to vcelku jedno, ale pokud chcete pracovat s jednotlivými znaky, už nastává problém – je byt celý znak, nebo jen jeho část? Kolik znaků je v řetězci? (v UTF-8 neplatí, že počet znaků = počet bytů)
Drobnou útěchou může být, že prvních 127 znaků v UTF-8 zabírá jeden bajt a jsou stejné jako v ASCII kódování.
UTF-8 je dnes nejpoužívanější kódování na webu a v Linuxu.
Kvůli problémům s UTF-8 používám v příkladech text bez diakritiky. Nevím taky, jestli budete používat zdrojáky na Windows (kde se používá kódování windows-1250) nebo Linuxu (UTF-8), ačkoliv existuje spousta programů, které dokáží převádět z jednoho kódování do druhého. V Linuxu například máte program recode.

Přetypování

Přetypování slouží ke změně datového typu objektu (proměnné, konstanty,…) Přetypování datového typu se provádí tak, že se před výraz napíše do závorky typ, na který chcete přetypovat. Nelze samozřejmě přetypovat cokoliv na cokoli, například řetězec na číslo. Přetypování je jeden z mocných nástrojů jazyka C. K přetypování výrazů dochází mnohdy automaticky, aniž byste si to uvědomovali. Například když do proměnné typu float přiřadíte celočíselnou konstantu, ta se nejprve (automaticky) převede na typ float a pak se teprve uloží do proměnné. V příkladu ukáži, jak se přetypování může využít.

```
1.  /*----------------------------------------------*/
2.  /* c08/pretyp.c                                 */
3.  #define _CRT_SECURE_NO_WARNINGS
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.   int a, b;
9.   float c, d;
10.  char ch = 'x';
11.
12.  printf("Zadej delenec: ");
13.  scanf("%d", &a);
14.  printf("Zadej delitel: ");
15.  scanf("%d", &b);
16.  c = a / b;
17.  d = (float) a / b;
18.  printf("Celociselne deleni: %+5.2f\n", c);
19.  printf("Racionalni deleni:  %+5.2f\n", d);
20.
21.  printf("ASCII kod znaku %c je %i\n", ch, (int) ch);
22.  return 0;
23.  }
24.
25.  /*----------------------------------------------*/
```

Makro _CRT_SECURE_NO_WARNINGS umožňuje použít funkci scanf() ve Visual Studiu.
Více informací viz první příklad u scanf().

Výstup z programu:

```
Zadej delenec: 50
Zadej delitel: 3
Celociselne deleni: +16.00
Racionalni deleni:  +16.67
ASCII kod znaku x je 120
```

Jak jsem již říkal, teda psal, k přetypování dochází mnohdy automaticky. Například, když do proměnné long uložíte číslo 10. 10 je konstanta, kterou překladač vyhodnocuje jako int (tak je to definováno standardem). Proto se nejdříve musí číslo 10 přetypovat na typ long. Překladač vás v takovém případě může varovat, že zde dochází k přetypování a to není hezké. Řešení je u typu long použít modifikátor l: `long x = 10l;`, u typu float zapsat číslo s desetinnou čárkou: `float x = 10.0;`, nebo můžete použít přetypování: `long x= (long) 10;`.

První a druhý způsob se používá právě u konstant, třetí způsob zvláště u proměnných, protože u nich první způsob použít nelze.

Přetypování návratové hodnoty funkce

Přetypovávat můžete i návratové hodnoty funkcí. Například funkce printf() vrací typ int. Návratovou hodnotu lze získat takto:

```
int navrat;
navrat = printf("Hello World");
```

Pokud návratovou hodnotu funkce chcete ignorovat, můžeme to překladači zdělit přetypováním návratové hodnoty na typ **void** takto:

```
(void) printf("Hello World");
```

Překladač by totiž mohl u některých funkcí (u printf() to standardně nedělá) hlásit varování, že funkce vrací návratovou hodnotu a že se nikam neukládá. Tímto přetypováním překladači dáte jasně najevo, že návratovou hodnotu opravdu, ale opravdu nechcete a on už pak dá pokoj.

Chyby přetypování

Někdy hraje přetypování docela zásadní roli. Přetypování totiž neprobíhá jako zaokrouhlování, desetinná část se usekne. Může tak docházet k problémům, které se těžko odhalují. Podívejte se na následující příklad. Máte nějakou virtuální obrazovku, která obsahuje nejméně 80 sloupců, ale může jich mít i více. Počet sloupců máte uložen v proměnné COLS. Obrazovku chcete rozdělit na dvě části tak, aby jedna část obsahovala 12 sloupců plus cca 25% z toho, co je nad 80 sloupců a druhá část zbytek. Využijete k tomu následující výpočet:

```
(12 + ((COLS - 80) * 0.25));
```

Tedy:

```
okno1 = (12 + ((COLS-80) * 0.25));
okno2 = COLS - (12 + (COLS-80) * 0.25));
```

Na první pohled se může zdát, že `okno1 + okno2 = COLS`. Podívejte se, jak může takový výpočet interpretovat překladač pro COLS = 84 a COLS = 85:

Pro COLS = 84:
okno1 = (12 + ((84-80) *0.25)) = (12 + (4 *0.25)) = (12 + 1.0) = 13.0
okno1 = 13;

okno2 = (84 - (12 + (84-80) * 0.25)) = 84 - 13.0 = 71.0
okno2 = 71;
To je OK. Ale pro COLS = 85:
okno1 = (12 + ((85-80) *0.25)) = (12 + (5 *0.25)) = (12 + 1.25) = 13.25
okno1 = 13;

okno2 = (85 - (12 + (85-80) * 0.25)) = 85 - 13.25 = 71.75
okno2 = 71; /* a chyba je na světě */
Řešení je jednoduché. Stačí výraz správně a včas přetypovat. Podívejte se na výpočet s přetypováním pro COLS = 85;
okno1 = (12 + (int) ((85-80) *0.25))
= (12 + (int) (5 *0.25))
= (12 + (int) 1.25) = 12 + 1
= 13;

okno2 = (85 - (12 + (int) (85-80) * 0.25))
= 85 - (12 + (int) 1.25)
= 85 - (12 + 1) = 85 - 13
= 72;
Chybami se člověk učí, stroj blbne :-).
Přetypování vám samozřejmě nepomůže vždy a ve všem. Celočíselnou polovinu z lichého čísla prostě nevypočítáte ;-).
Pokud překladač narazí při výpočtu na dva rozdílné typy, interpretuje oba jako ten větší (přesnější) z nich. Proto se výpočet 5 * 0.25 automaticky interpretuje jako
((float)5)*0.25 a ne 5*(int)0.25.
Výsledek, který se ukládá do proměnné se do této proměnné musí vejít, proto se nakonec přetypovává vždy na typ proměnné, do které se hodnota ukládá.
Pokud se pokusíte uložit do proměnné hodnotu větší, než kterou je schopna pojmout (například char ch = (char) 256; [1]), výsledkem bude **nedefinovaná hodnota**, tj nikdo vám nijak nedefinuje, jaká nakonec bude skutečná hodnota proměnné *ch*. Rozhodně to bude něco mezi 0 a 255, ale to je asi tak všechno, co se o tom dá říct.

Pole a ukazatele

Datové pole a řetězce

O datových polích jsem se již stručně zmínil v souvislosti s řetězci. K řetězcům snad není nutné říkat nic jiného, než že jsou to vlastně pole typu char zakončené nulovým znakem '\0'.
Datové pole je nějaké spojité místo vyhrazené kdesi v paměti pro několik objektů stejného datového typu. K tomu, abyste mohli k těmto objektům přistupovat, slouží ukazatel na onen datový typ.
Pole může mít různou velikost. Kde začíná je dáno právě ukazatelem, jeho konec si však musíte ohlídat. U řetězců je konec řetězce dán znakem '\0', což není nic jiného než nulový bajt. Je tak ukončen řetězec, nikoliv pole. Konec pole nijak označen není, proto se často stává ta chyba, že se program pokouší číst mimo rozsah pole, což vede ke zhroucení programu.
Rozdíl mezi polem typu char a mezi řetězcem je pouze v tom, že řetezec má na svém konci uložen nulový znak. Vlastně je to jen taková dohoda, na kterou spoléhají všechny funkce pracující s řetězci.
Pole se deklaruje následovným způsobem:
datovy_typ nazev [rozsah];
Například:
char znaky[4]; /* pole pro 4 znaky */
int cisla[1]; /* pole pro jedno cislo */
Proměnné *znaky* a *cisla* jsou proměnné typu array, které se chovají jako **konstantní ukazatele**, které ukazují na první prvek z pole. Jednotlivé položky pole se adresují od nuly pomocí hranatých závorek. Například pole *cisla* obsahuje jen jedno místo pro číslo typu int a s tím lze pracovat následovně (připomínám, že v jazyku C se indexuje od nuly, nikoliv od jedničky, narozdíl od Pascalu):
int x;
cisla[0] = 5;
x = cisla[0];
Chybné by bylo přiřazení cisla=5;, protože tím byste přiřazovali číslo 5 do proměnné *cisla*, čili byste se pokoušeli změnit „konstantní ukazatel", který obsahuje adresu začátku pole.
Konstantní ukazatel na pole obsahuje adresu paměti, kde začíná pole. Tuto hodnotu změnit nelze, můžete měnit jen hodnoty v poli.
Číslo v hranatých závorkách při definici pole určuje jeho velikost. V jazyce C to musí být skutečné číslo, nelze použít proměnnou, ani konstantu (číslo musí překladač znát už v době překladu, aby pro pole dokázal vyhradit dostatek paměti).
V jazyce C++ lze použít i konstantu (její hodnota je překladači známa, protože se nemění).
Číslo v hranatých závorkách za ukazatelem při použití v kódu se nazývá **index pole**. Indexem pole se odkazuje na n-tou položku pole.
Používání indexu pole osvětlím na následujícím příkladu. Definuji pole *znaky*.
char znaky[4];
V tabulce je ukázáno, jak se uloží pole *znaky* do paměti.

ukázka uložení prázdného čtyřznakového pole v paměti

skutečná adresa	1534	1535	1536	1537	1538	1539
pozice (index)	...	0	1	2	3	...
bity	???	????????	????????	????????	????????	???

Momentálně pole *znaky* obsahuje náhodné byty. Ukazatel *znaky* ukazuje na adresu 1535 (to číslo je samozřejmě smyšlené). Pokud chcete vložit do pole znaky a vytvořit řetězec, nesmíte zapomenout na zarážku (nulový byte).

```
znaky[0] = 'j';
znaky[1] = 'e';
znaky[2] = '\0';
```

V tabulce můžete vidět, jak se pole přiřazením hodnot vyplnilo.

ukázka uložení dvouznakového řetězce v paměti

skutečná adresa	1534	1535	1536	1537	1538	1539
pozice (index)	...	0	1	2	3	...
bity	???	01101010	01100101	00000000	????????	???

Všimněte si, že poslední bajt pole je nevyužitý (znak[3] na adrese 1538). Taky si všimněte, že pokud byste se pokusili přiřadit hodnotu do znak[4], odstanete se na adresu 1539, která už zasahuje mimo rozsah pole. To může vést k chybě „neoprávněný přístup do paměti" a následnému odstřelení programu operačním systémem. Ale taky nemusí. Můžete si přepsat nějakou jinou proměnnou, která v této části paměti sídlí a tím vytvořit nějakou jinou, nepředvídatelnou, těžko odhalitelnou chybu.

Operační systém nedovolí vašemu programu sahat do paměti, která mu nebyla vyhrazena. Je to z bezepčnostních důvodů, aby jeden program nemohl škodit druhému.

Pokud pole během jeho definice rovnou inicializujete, nemusíte uvádět jeho délku. Délka pole bude taková, kolik prvků předáte při inicializaci.

```
int cisla[] = { 5, -1, 3, 8};              /* pole pro 4 cisla */
char pole1[] = {'a', 'h', 'o', 'j', '\0'};  /* pole pro 5 znaku */
char pole2[] = "ahoj";                      /* pole pro 5 znaku */
```

Inicializace `pole1` a `pole2` je ekvivalentní. Všimněte si, že v druhém případě je přidán nulový znak na konec automaticky. Nyní můžete přiřadit: `pole2[4] = 'x';`.

Tím se však připravíte o zarážku (nulový znak) a již nemůžete s polem pracovat jako s řetězcem (například jej předat funkci `printf()`, protože by po vytištění ahojx funkce `printf()` tiskla další byty z paměti, dokud by nenarazila na '\0' (nebo ji operační systém nesestřelil).

Podívejte se ještě na tento příklad:

```
char pole1[] = "ahoj";
char *pole2 = "zdar";
```

Proměnné *pole1* i *pole2* jsou ukazatele na typ char. Rozdíl v těchto definicích je ale v tom, že řetězec zdar je řetězcový literál, který nemůžete měnit, zatímco ahoj se uloží do pole 5 znaků, jehož hodnoty můžete libovolně měnit.

Na definici s `pole1[]` prostě pohlížejte jako na `char pole1[5] = {'a', 'h', 'o', 'j', '\0'};` a nebudete si to plést.

Další rozdíl je v tom, že proměnná *pole1* je konstantní (já vím, konstantní proměnná zní divně :D), takže její hodnotu nemůžete měnit, ale hodnotu proměnné *pole2* ano.

```
char pole1[] = "ahoj";
char *pole2 = "zdar";
char pole1[0] = 'A'; // ok, měním hodnotu v poli na které pole1 ukazuje
char pole2[0] = 'Z'; // ale fuj
pole1 = NULL; //syntax error
pole2 = NULL; // ok, no problemo
```

Řetězcový literál `"zdar"` nelze měnit v žádném případě. Pokud byste se pokusili o něco jako `pole2[0] = 'Z';`, program se zhroutí. Na druhou stranu, pro pole1 se alokuje paměť v místě, kde ji lze měnit (a do tohoto místa se `"ahoj"` nakopíruje).

Vícerozměrná pole

Když se pdíváte jakým způsobem je v paměti pole uloženo, pochopíte, že v paměti můžou být jen jednorozměrná pole (prvky pole můžou být řazeny v paměti počítače pouze za sebou). Nic vám však nebrání vytvořit pole, jehož prvky budou jiná pole a tak vytvářet pole libovolných dimenzí (rozměrů). Podívejte se na definici dvourozměrného pole a jeho uložení v paměti.

```
char pole[4][2] = { {0,0}, {1,1}, {3,4}, {0xff,0xff} };
```

Jedná se o čtyřprvkové pole, jehož prvky jsou dvouprvková pole typu char.

Uložení pole 4x2 v paměti

skutečná adresa	1534	1535	1536	1537	1538	1539	1540	1541	1542	1543
index	...	[0][0]	[0][1]	[1][0]	[1][1]	[2][0]	[2][1]	[3][0]	[3][1]	...
hodnota	???	0	0	1	1	3	4	255	255	???

Všimněte si, že při putování pamětí se mění vždy nejdříve poslední index pole a pak ty předešlé.

Můžete vytvářet pole libovolné dimenze a velikosti, ale dejte pozor na to, kolik takové pole zabírá místa v paměti. Např. pole `long double nazev[50][7][2][8]` je velké 10*50*7*2*8 = 56000 bytů (long double je veliký 10 bytů)!

Vícerozměrná pole se využívají poměrně často. Například `kalendar[12][31]`, `sachovnice[8][8]` atp. Práce s takovými poli je daleko snazší a přehlednější než s mnoha jednorozměrnými poli, přestože např. v kalendáři nevyužijete všechny prvky pole (ne každý měsíc má 31 dní).

Ukazatele a pole

Když si zopakujete kapitolu o ukazatelích, pak spolu s předcházejícími odstavci o polích by vám mělo být již vše jasné. Přesto si můžeme ještě ukázat zajímavá „kouzla", která lze s ukazateli provádět. Začnu s řetězci. Víte například to, jaký je rozdíl v následujících definicích?

```
char *retezec1 = "Ahoj";
char retezec2[] = "Ahoj";
```

Rozdíl je v tom, že `retezec2` je konstantní ukazatel (pole), kdežto `retezec1` je ukazatel inicializovaný hodnotou adresy řetězcového literálu "Ahoj".

Při obou definicích se vytvořil řetězec "Ahoj", což je pole znaků (se zarážkou '\0' na konci). První je však uložen v části paměti, která je jen pro čtení, druhý je uložen v poli, se kterým můžete manipulovat.

V následujícím příkladě se podívejte, jakým způsobem lze adresovat jednorozměrné pole. Když si představíte, jakým způsobem je pole v paměti uloženo a uvědomíte si, že ukazatel obsahuje **číselnou hodnotu** která je adresou, neměl by být pro vás problém příklad pochopit.

```
/*----------------------------------------------*/
   /* c09/pole1.c                                */

#include <stdio.h>

int main(void)
   {

float pole[] = { 5.0, 6.0, 7.0, 8.0, 9.0 };
   float *ukazatel;

   ukazatel = pole;

   /* prvni cast */
  printf("%.1f ", pole[1]);
 printf("%.1f ", pole[1] + 10.0);
printf("%.1f\n", (pole + 1)[2]);

   /* druha cast */
   printf("%.1f ", *ukazatel);
 printf("%.1f ", *ukazatel + 10);
printf("%.1f ", *(ukazatel + 1));
printf("%.1f\n", *(ukazatel + 1) + 10);

   return 0;
   }

/*----------------------------------------------*/
```

Výstup z programu:

```
6.0 16.0 8.0
5.0 15.0 6.0 16.0
```

Všimněte si, jakou roli hrají závorky! Také si všimněte, jak chytře pracuje jazyk C s **aritmetikou ukazatelů**. Výraz (ukazatel + 1) ukazuje na další prvek pole, přestože typ float je dlouhý hned 4 bajty a ne jeden. To je jeden z důvodů, proč se při deklaraci ukazatele určuje, jakého je typu. Nemusíte si tak lámat hlavu, o kolik bytů byste měli zvýšit jeho hodnotu, aby se ukazoval na další prvek příslušného datového typu. **Překladač pak zvýší hodnotu ukazatele o správný počet bytů tak**, aby ukazoval na další prvek v poli.

Následující výrazy jsou ekvivalentní:

`&pole[n]` je totéž co `(pole + n)` (tj adresa n-tého prvku v paměti)

a

`pole[n]` je totéž co `*(pole + n)` (tj hodnota n-tého prvku v paměti)

Abych vás ještě trochu potrápil, ukáži vám příklad s dalším možným zápisem adresy pole:

```
/*----------------------------------------------*/
   /* c09/pole2.c                                */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
char array[5] = {'a', 'b', 'c', 'd', 'e'};

// array[2] je totez jako 2[array]
printf("array[2] = '%c'\n", 2[array]);

return EXIT_SUCCESS;
}
```

```
/*-------------------------------------------------*/
```

Tento způsob indexace pole není ukázkou dobrého stylu psaní a měli byste ho používat jen k mučení učitele programování. Ukázka jen ukazuje to, že je jedno, jestli se adresa paměti vypočte jako array + 2, nebo (jako v příkladu) 2 + array.

```
array[2] = 'c'
```

Ukazatele na ukazatele

V jazyku C můžete vytvořit ukazatel na libovolný datový typ (třeba i na vlastní strukturu – o těch později). Ukazatel tedy může ukazovat i na ukazatel. Například můžete vytvořit ukazatel, který může ukazovat na „ukazatel na typ char":

```
char **ukazatel;
```

Můžete taktéž vytvořit pole ukazatelů (včetně vícerozměrných polí ukazatelů…).

```
char * ukazatel[10]; /* 10-prvkove pole ukazatelu na typ char */
```

Takto definované pole má velikost 10*sizeof(char *), tj. 10x velikost ukazatele, tj. např. 40 bajtů (pro 32-bitové adresování).

Příklad:

```
/*-------------------------------------------------*/
/* c09/unau1.c                                     */
#include <stdio.h>

int main(void)
{
int x;
int *ukazatel_na_int;
int **ukazatel_na_ukazatel;

int pole[] = { 5, 7, 9, 11, 13 };

/* inicializace promennych */
x = 25;
ukazatel_na_int = &x;
ukazatel_na_ukazatel = &ukazatel_na_int;

/* pristup k promenne x */
printf("%2d = %2d = %2d\n", x, *ukazatel_na_int,
*(*ukazatel_na_ukazatel));

/* inicializace */
ukazatel_na_int = pole;
/*  ukazatel_na_ukazatel = &ukazatel_na_int; ... toto prirazeni
*  je o nekolik radek vyse a hodnota ukazatel_na_ukazatel ukazuje
*  stale na ukazatel_na_int */

/* pristup k poli */
printf("%2d = %2d = %2d\n", pole[0], *ukazatel_na_int,
**ukazatel_na_ukazatel);
printf("%2d = %2d = %2d\n", pole[1], *(ukazatel_na_int + 1),
*((*ukazatel_na_ukazatel) + 1));
return 0;
}

/*-------------------------------------------------*/
```

Výstup z programu:

```
25 = 25 = 25
 5 =  5 =  5
 7 =  7 =  7
```

Podívejte se na vyhodnocení výrazu *(*ukazatel_na_ukazatel). Závorky v tomto výrazu nejsou podstatné a jsou zde jen pro ilustraci. První, co se provede, je vyhodnocení výrazu *ukazatel_na_ukazatel. Hvězdička je **dereferencí adresy**, jejíž výsledkem je hodnota proměnné, na kterou ukazuje proměnná *ukazatel_na_ukazatel*. V příkladě to znamená, že **ukazatel_na_ukazatel* je hodnota proměnné *ukazatel_na_int* (to je adresa proměnné*x*).

Dejme tomu, že adresa *x* je 123456. Potom *(*ukazatel_na_ukazatel) je *(123456) a to je hodnota v paměti na adrese 123456, tedy hodnota proměnné *x*.

Teď už jen v rychlosti proberu výraz *((*ukazatel_na_ukazatel) + 1).

Dejme tomu, že pole začíná na adrese 12340. Potom ukazatel_na_int bude po druhé inicializaci v příkladu 12340, (*ukazatel_na_ukazatel) je tedy také 12340.

A teď pozor. Výraz (*ukazatel_na_ukazatel + 1) je (12340+**1*n**) a výraz *((*ukazatel_na_ukazatel) + 1)je *(12340+**1*n**). Číslo **n** je počet bajtů, které zabírá typ int, protože jedničku přičítáme k ukazateli na int. Tak tomu se říká **aritmetika ukazatelů**. Výsledkem pak bude hodnota (kvůli hvězdičce) na adrese 12344 (pokud int zabírá 4 bajty). Uff.

Ovšem pozor. Ukazatele a pole nelze libovolně zaměňovat. Tak například **pole1 a pole5[][5] a pole6[][6] jsou tři různé typy. S vědomostmi které máte o **aritmetice ukazatelů** vás jistě napadne, jaké chyby by se mohli stát, kdybyste použití těchto proměnných libovolně zaměňovali. Mám na mysli hlavně aplikaci pole1 na některé z dvourozměrných polí. Správně by se mělo použít např.: int pole5[N][5]; int (*upole5)[5]; upole5 = pole5; atp.

Pokud jste se ve zdraví prokousali až sem, pak vám gratuluji. Váš mozek právě začal mutovat v „mozek programátora jazyka C". Tento proces je, bohužel, nevratný.

Datový typ pro indexaci pole

Standard jazyka C definuje datový typ **size_t** pro určování velikostí oběktů (a indexaci polí). Máte zaručeno, že datový typ size_t je dost velký na to, abyste mohli indexovat „libovolně" velké pole. (Pravděpodobně to bude ve skutečnosti něco jako unsigned int nebo unsigned long.)

Datový typ **ptrdiff_t** se zase používá pro výsledek rozdílu pointerů.

Oba datové typy jsou definovány v knihovně <stddef.h>. Mohou být definovány i v jiných knihovnách (například knihovnách, které <stddef.h> sami využíají), takže není vždy nutné <stddef.h> includovat.

```
/*----------------------------------------------*/
   /* c09/typy.c                                 */

#include <stdio.h>
#include <stddef.h>

int main(void)
   {

char *text = "Hello world!";
   char *u1 = text, *u2;
      size_t i = 0;
      ptrdiff_t diff;

   while(text[i]) i++;
      u2 = text+i;
     diff = u1 - u2;

   /* vypíše délku textu */
   #ifdef _MSC_BUILD
printf("size_t i = %Iu, ptrdiff_t diff = %Ii\n", i, diff);
   #else
printf("size_t i = %zu, ptrdiff_t diff = %ti\n", i, diff);
   #endif

      return 0;
      }

/*----------------------------------------------*/
```

Ve zdrojovém kódu jsou 2 řádky s funkcí printf(). První řádka se použije ve Visual Studiu, druhá všude jinde. To je díky podmíněnému překladu, který budu probírat později.

Důvodem je, jaké "%" sekvence používá printf() pro výpis těchto datových typů. Na druhé řádce je to podle standardu C99, na první řádce je to podle standardu Microsoftu.

Pro datový typ size_t používá printf() dle standardu C99 %zu, ale Microsoft %Iu.

Pokud používáte Dev-C++, musíte si nastavit makro -D__USE_MINGW_ANSI_STDIO=1, jinak vám sekvence %zu a podobné nebudou fungovat.

Výstup z programu:

```
size_t i = 12, ptrdiff_t diff = -12
```

Načtení řetězce pomocí funkce scanf()

V kapitole o Vstupu a výstupu jste se dozvěděli, že funkce scanf() používá pro řetězce typ s. A taky že maximální počet vstupních znků se dá omezit pomocí „width". Z toho vychází následující příklad:

```
/*----------------------------------------------*/
   /* c09/scanstring.c                           */
   #define _CRT_SECURE_NO_WARNINGS
   #include <stdio.h>

   int main(void)
      {
      char znaky[10];
   printf("Zadejte text: ");
      fflush(stdout);
      scanf("%9s", znaky);
printf("Nacteno: '%s'\n", znaky);

      return 0;
      }

/*----------------------------------------------*/
```

Bez `fflush(stdout);` by mohl zůstat text tištěný funkcí `printf()` v bufferu, protože není ukončen novým řádkem `'\n'`.
Možný výstup z programu:

```
Zadejte text: 1234567890
Nacteno: '123456789'
```

Načetlo se 9 znaků (jak jsem chtěl). Desátý znak se využije pro označení konce řetězce '\0'.
A ještě jedna ukázka:

```
Zadejte text: Ahoj světe!
Nacteno: 'Ahoj'
```

Funkce scanf() čte řetězec až k oddělovači (bílým znakům), takže načte jen Ahoj.
Existují i jiné funkce pro načítání vstupu od uživatele, šikovnější než scanf() (třeba v tom, jak se jim předává maximální délka řetězce), které načtou řetězec až ke konci řádku ('\n'). S těmi vás seznámím později.

Operátory a výrazy

V této kapitole popíši operátory, které můžete v jazyku C používat. Některé speciální operátory (jako např. přetypování) už znáte, a naopak použití jiných vysvětlím až v některé z příštích kapitol. Po výčtu všech operátorů bude na konci jeden velký příklad. U většiny operátorů však nechám na vás, abyste si jejich chování vyzkoušeli ve vlastních příkladech.

Operátory lze rozdělit několika způsoby. Mimo jiné na unární, binární, ternární, neboli na operátory s jedním, dvěma nebo třemi operandy. Operandy jsou data (většinou čísla), se kterými operátory (např. plus) pracují. Operátory se také rozdělují na aritmetické, bitové, relační a logické. Čtěte dále.

Unární operátory

Unární operátory

Operátor	Význam
+,-	unární plus a mínus,
&	reference (získání adresy objektu)
*****	dereference (získání hodnoty objektu dle adresy)
!	logická negace
~	bitová negace
++, --	inkrementace a dekrementace hodnoty
(typ)	přetypování
sizeof()	operátor pro získání délky objektu nebo typu!

Unární plus a mínus určuje znaménko čísla. Například ve výrazu 5 + (-4) je plus binární operátor sčítání (má dva operandy) a mínus je unární operátor (vztahuje se jen ke čtyřce).
S operátory reference & a * dereference jste se již setkali při práci s ukazateli.
*~~V jazyce C neexistuje datový typ boolean.~~ Pokud potřebujete někde získat nebo uchovat hodnotu pravda/nepravda (TRUE/FALSE), můžete k tomu využít např. typ int. V jazyce C je totiž vše nenulové považováno za **TRUE** a ostatní (včetně. např. prázdného řetězce, tj řetězce, jehož první znak je nulový znak) za **FALSE**. Nula je tedy FALSE a jiné číslo (nejčastěji se používá jednička) TRUE. Další info viz boolovské datové typy.*
Výsledkem logické negace ! je TRUE nebo FALSE, což jazyk C vyhodnocuje jako 1 nebo 0. Například !5 je 0.
Bitová negace ~ je však něco úplně jiného. Bitová negace obrací jednotlivé bity ve výrazu. Takže například~0x05 je 0xFA (~00000101 je 11111010).
!"" je 0 (FALSE), protože negujete adresu řetězce "", která je nenulová. !"" je už 1 (TRUE), protože negujete hodnotu prázdného řetězce. Což není překvapivé, když si uvědomíte, že prázdný řetězec vlastně obsahuje pouze nulový znak '\0', což je, v bitech zapsáno, nula.*
Operátory inkrementace ++ a dekrementace -- mohou být zapsány jak před výrazem, tak za ním. Zvyšují, resp. snižují hodnotu čísla/ukazatele o jedničku (nezapomeňte ale na aritmetiku ukazatelů).
Pokud leží ++ před proměnnou, nejdříve se zvýší její hodnota a pak se proměnná použije ve výrazu (pokud v nějakém je), leží-li ++ za proměnnou, nejdříve se proměnná použije ve výrazu a pak se teprve zvýší její hodnota. Obdobně je to u dekrementace --.
Například:

```
y = 5;
x = ++y; // y se inkrementuje na 6 a poté se dosadí 6 do x
x = y++; // do x se dosadí y (6) a pak se y inkrementuje (na 7)
```

Výsledkem je, že *x* je 6 a *y* je 7.

Operátor **sizeof()** vrací velikost datového typu nebo objektu v bytech.

```
float y;
int x = sizeof(float);
x = sizeof(y);
sizeof(y) == sizeof(float)
```

Priorita operátorů

Priorita operátorů stanovuje, která část výrazu se vyhodnotí dříve, pokud to není závorkami určeno jinak. Klasickým příkladem je vyhodnocení výrazu `x = 1+1*0;`. Výsledkem bude samozřejmě číslo 1, protože operátor násobení má větší prioritu než operátor sčítání.

Priorita operátorů je v jazyku C velice komplikovaná záležitost a proto se omezím na stručnou radu: **používejte závorky**!

Ovšem pozor! Pořadí vyhodnocení inkrementace (dekrementace) ve výrazu není normou jazyka C nijak dáno. Tudíž není zaručeno, že každý překladač pořadí (rozuměj prioritu) výpočtu vyhodnotí stejně. Vlastně i jeden překladač se na různých místech může v zájmu optimalizace programu rozhodnout k různému pořadí vyhodnocování výrazu (obdobně to platí i pro funkce, viz níže).

Následující konstrukci (a jí podobné) byste se měli vyhýbat jako čert kříži. Odhalit, proč se program nechová jak má kvůli jinému vyhodnocování různými překladači, bývá náročné.

```
y = 5;
x = y+++++y;
```

Po této operaci bude *y* rovno sedmi (dvakrát se inkrementuje). Ale co *x*? Může s to vyhodnotit jako 5+6, nebo jako 5+7? V tomto případě by vám ani závorky nepomohli. Překladač si totiž řekne: první *y* se má zvýšit až po použití ve výrazu, tj první *y* bude 5. A teď si může říct 2 věci:

y už jsem použil, můžu ho inkrementovat. Nebo si to neřekne a přejde na vyhodnocování druhého *y*, které inkrementuje (na 6), obě čísla sečte (5+6) a až teď teprve se rozhodne, že provede postfixovou inkrementaci *y* a zvýší jej na 7.

Taktéž u volání funkcí není jasné, která se ve výrazu zavolá dříve. Podívejte se na následující příklad.

```
x = f1() + f2();
```

Pokud funkce `f1()` a `f2()` vypisují na obrazovku nějaký text (kromě toho, že vracejí nějaké číslo, které se pak sečte), nemůžete si být nikdy jistí, který text se vypíše jako první. A jako vždy, to že si to vyzkoušíte s vaším překladačem neznamená, že všechny ostatní překladače to udělají stejně.

Funkce, která něco vypisuje a zároveň něco počítá, dělá 2 věci, což je obvykle špatně. Správně navržené funkce dělají vždy jen jednu věc. Nejen kvůli výše popsanému problému, ale taky kvůli jednoduššímu použití a případným změnám v programu.

Binární a relační operátory

Binární operátory

Operátor	Význam
=	přiřazení
+,-,*,/	plus,mínus,krát,děleno
%	zbytek po celočíselném dělení (modulo)
<<, >>	bitový posun vlevo, vpravo
&	bitové AND
|	bitové OR
^	bitové XOR
&&	logické AND
||	logické OR
.	tečka, přímý přístup ke členu struktury
->	nepřímý přístup ke členu struktury
,	čárka, oddělení výrazů

Relační operátory

Operátor	Význam
<	menší než

Relační operátory

Operátor	Význam
>	větší než
<=	menší nebo rovno
>=	větší nebo rovno
==	rovnost
!=	nerovnost

Při bitovém posunu vlevo (vpravo) se poslední levý (pravý) bit ztrácí a zprava (zleva) je dosazena nula. Bitové operace je možné provádět jen s celočíselnými hodnotami. Při posunu čísel se znaménkem se však znaménko zachovává (znaménkový bit se nikam neposune). (Na tom vidíte, jak je důležité určovat, zda je datový typ se znaménkem (signed) nebo bez znaménka (unsigned).)
Vlevo od operátoru je objekt, ve kterém se bity posouvají, vpravo od operátoru je číslo určující, kolikrát se bity posunou.
U logického AND a OR, výrazů >= atp. je výsledkem TRUE nebo FALSE, tedy 1 nebo 0.
Operátor přiřazení lze kombinovat s některými výpočty:
`+=, -=, *-, /=, <<=, >>=, &=, |=, ^=`.
Jde jen o zkrácený zápis, tj. např. `a += b;` je zkrácený zápis pro `a = a + b;`.

Ukázka kombinací operátoru přiřazení s jiným operátorem

Výraz	Ekvivalent
x -= 5;	x = x - 5;
x *= x;	x = x * x;
x >>= 2;	x = x >> 2;

K operátorům ->, ==, >, < atp. se ještě dostanu v některých z dalších kapitol.

Podmíněný operátor

Podmíněný operátor **? :** je **ternárním** operátorem (má 3 operandy). Prvním operandem je výraz, který se vyhodnotí jako logický výraz (TRUE nebo FALSE). Pokud se vyhodnotí jako TRUE, výsledkem bude druhý operand (výraz mezi `?` a `:`), jinak třetí operand (výraz za `:`).

```
printf("%s\n", (x > y) ? "x je vetsi nez y" : "x je menci ci roven y");
```

Toto je vlastně první příklad, kde se vyhodnocuje nějaká podmínka (`x > y`) – na základě které se program rozhodne, co vytiskne.

Příklad

```
/*----------------------------------------------*/
  /* c10/operator.c                             */

#include <stdio.h>
#include <stddef.h>
#include <limits.h> /* ziskame konstanty UINT_MAX a INT_MIN */

#ifdef _MSC_VER
    #define ZU "Iu"
      #else
    #define ZU "zu"
      #endif

int main(void)
      {
    int x, y, z;
int pole[] = { 5, -10, 15, -20, 25, -30 };
    size_t delka_pole;
    x = y = z = 10;

    x = y++;
      z++;
  printf("%3d %3d %3d\n", x, y, z);
printf("%3d %3d %3d\n", ++x, y++, ~z);
  printf("%3d %3d %3d\n", x, y, z);

printf("Mate %2" ZU " bitovy prekladac.\n", sizeof(int *) * 8);
printf("Pole pole[] zabira %2" ZU " bytu.\n\n", sizeof(pole));

printf("UINT_MAX = %u = %u\n", UINT_MAX, ~0); /* UINT_MAX = max velikost cisla v typu unsigned int */
printf("INT_MIN = %i = %i \n", INT_MIN, 1 << ((sizeof(int)*8)-1)); /* posunu jednicku az do "nejlevejsiho bitu" */
    printf("-51 >> 1 = %i\n\n", -51 >> 1);
```

```
    delka_pole = sizeof(pole) / sizeof(pole[0]);
    printf("Zadejte index pole (od  0 do %2" ZU "): ", delka_pole - 1);

    scanf("%i", &x);
    printf("Zadal jsi %i\n", x);

    /* musi se zkontrolovat platnost zadaneho indexu pole !! */
    x = (x < 0) ? 0 : x;
    x = (x >= delka_pole) ? delka_pole - 1 : x;
delka_pole = ((size_t)x >= delka_pole) ? delka_pole - 1 : (size_t)x;
    printf("pole[%" ZU "] = %i\n", delka_pole, pole[delka_pole]);

    return 0;
    }

/*--------------------------------------------------*/
```

V příkladu je použitý podmíněný překlad (řádky 8 - 12). Řádek 9 se použije pro Visual Studio, řádek 11 pro všechny ostatní. ZU se dále v kódu nahradí definovanou hodnotou, tj. například řádek 28 bude před překladem nahrazen ve Visual Studiu tímto řádkem:

```
printf("Mate %2" "Iu" " bitovy prekladac.\n", sizeof(int *) * 8);
```

A jak dobře víte, pokud zapíšete zasebou textové literály (to co je v dvojitých uvozovkách), překladač je automaticky spojí do jednoho:

```
printf("Mate %2Iu bitovy prekladac.\n", sizeof(int *) * 8);
```

A to už je snad srozumitelné :) Podmíněný překlad proberu podrobně v další kapitole.

Výstup z programu:

```
10  11  11
11  11 -12
11  12  11
Mate 32 bitovy prekladac.
Pole pole[] zabira 24 bytu.

UINT_MAX = 4294967295 = 4294967295
INT_MIN = -2147483648 = -2147483648

-51 >> 1 = -26

Zadejte index pole (od  0 do  5): -11
Zadal jsi -11
pole[0] = 5
```

Protože mám 32 bitový překladač, ukazatel (int *) zabírá 32 bitů, tj 4 bajty. Pole *pole* zabírá celkem 24 bajtů. To může být i u 64-bitového překladače, protože standard přesně neříká, jak velký má datový typ int být.

Pokud jste zmateni z výpočtu INT_MIN, vzpomeňte si, co jsem psal o dvojkové soustavě a doplňkovém kódu.

Všimněte si, jakým způsobem jsem získal meze pro index pole[]. Pole je vždy indexováno od 0. Počet prvků však může být různý. Kdybyste v kontrole horní meze napsali natvrdo `x = x>5 ? 5 : x;`, museli byste při změně počtu prvků v poli změnit i tento výraz. Ve velkém zdrojovém kódu byste to mohli snadno přehlédnout (nehledě na to, že je to pracné hledání). Takhle se nemusíte o nic starat.

Operátor `? :` lze použít i trochu jinak. Jestlipak správně pochopíte význam následujícího výrazu?

```
z > x ? x : y = z;
```

Jestli si nejste jisti, tak si to vyzkoušejte v programu. Jestli si ste jisti, tak si to raději stejně vyzkoušejte :-). První co vás napadne jako odpověď nemusí být hned správně :-)[1].

Použití NULL

Čistě jenom pro zopakování ukážu jak lze využít hodnotu NULL, společně s podmíněným operátorem.

```
/*--------------------------------------------------*/
    /* c10/null1.c                                  */

    #include <stdio.h>

    int main(void)
    {
    float *uk = NULL;
    float p[] = { 10.2, 5.1, 0.6 };

    printf("%s\n", uk == NULL ? "Ukazatel neni inicializovan" :
                   "Ukazatel lze pouzit");
    printf("%f\n", uk == NULL ? 0.0 : uk[0]);
```

```
    uk = p;

    printf("%s\n", uk == NULL ? "Ukazatel neni inicializovan" :
                                "Ukazatel lze pouzit");
    printf("%f\n", uk == NULL ? 0.0 : uk[0]);
    return 0;
}

/*----------------------------------------------*/
```

Výstup z programu:

```
Ukazatel neni inicializovan
0.000000
Ukazatel lze pouzit
10.200000
```

Jo jo, můžete místo NULL použít nulu, nebo lépěji `(float *)0`. Ale NULL je prostě nulový ukazatel a byl stvořen pro tyto účely, aby byl program čitelnější. S NULL si taky nemusíte dělat starosti s přetypováváním. Překladač ví, co NULL znamená a jak se k němu chovat. V jazyce C++ je NULL definováno jako konstanta `0`, v jazyce C jako `(void *)0`, nebo taky jen `0`.

Preprocesor

Před vlastním překladem zdrojového kódu je kód upraven tzv. **preprocesorem**. Například v Linuxu lze spustit preprocesor příkazem **cpp** (C Pre Processor). Preprocesor si volá překladač sám, takže o tom ani nemusíte vědět. Preprocesor odstraní ze zdrojového kódu komentáře a rozvine makra, které jsou ve zdrojovém kódu. Všechny direktivy preprocesoru začínají znakem # (čti šarp).

Například, když zapíšete do zdrojového kódu `#include <stdio.h>`, tak preprocesor vloží na místo tohoto řádku obsah souboru stdio.h.

Direktiva preprocesoru musí být první na novém řádku, před ní mohou být jen „bílé znaky" (mezera, tabulátor…).

Direktiva končí s koncem řádku. Pokud chcete pokračovat na novém řádku, pak můžete před „konec řádku" napsat zpětné lomítko.

V následující tabulce jsou direktivy preprocesoru. Význam některých z nich proberu.

Direktivy preprocesoru

#define	#elif	#else	#endif	#error	#ident	#if
#ifdef	#ifndef	#include	#line	#pragma	#undef	#warning

Standardní makra

Podle normy ANSI C existují následující makra, které musí každý preprocesor jazyka C znát. Všimněte si, že standardní makra preprocesoru začínají a končí dvěma podtržítky.

Standardní makra

Makro	Význam	datový typ
__DATE__	Datum spuštění preprocesoru	string
__TIME__	Čas spuštění preprocesoru	string
__FILE__	Jméno aktuálního vstupního souboru	string
__LINE__	Pořadové číslo aktuálního řádku	int
__STDC__	Určuje, zda překladač splňuje ANSI C	int (1 nebo 0)
__cplusplus	Určuje, zda byl použit překladač C++	Překladač jazyka C toto makro nedefinuje, v C++ je __cplusplus číslo verze standardu jazyka

Existují i implementačně závislá makra, tedy makra, která jsou definována jen na některých OS a v některých preprocesorech jazyka C. Například makro __linux__ se může hodit ke zjištění, zda program překládáte v linuxu.

Nestandardní makra

Makro	Význam	datový typ
__linux__[1)]	Určuje, zda je zdrojový kód překládán v Linuxu	int (1 nebo 0)
__unix__[1)]	Obdobně jako linux	int (1 nebo 0)
__MSDOS__	Verse DOSu	string
__i386__	Určuje, zda jde o procesor z řady i386	int (1 nebo 0)
__VERSION__	Verze překladače GNU C (gcc). Existuje pochopitelně jen u překladače GNU.	string
_MSC_VER	Verze překladače Visual Studia.	string
_MSC_BUILD	Revision number překladače pro Visual Studio.	string

Makra definovaná pro VS najdete třeba tady: Predefined Macros for Visual Studio.
Příklady použití budou následovat.

Definice maker

Kromě standardních maker si můžete vytvářet makra vlastní. Jejich užívání je velice rozšířené, například je lze použít pro definování počtu prvků pole. Takové makro se pak používá v celém zdrojovém kódu a při změně počtu prvků pole stačí změnit jen makro. Makro se definuje za direktivou #define a lze jej zrušit direktivou #undef.

V C++ je zvykem místo maker používat pro definici rozměru pole konstanty. Jak už jsem psal dříve, v jazyku C nemůžete použít konstantu pro definování rozměru pole, ale v C++ ano.

```
#define NAZEV makro
/* ... odtud dále se ve zdrojovém kód může makro používat ... */
/* kdekoliv použijete NAZEV, preprocesor ho nahradí za makro  */
#undef NAZEV
/* zde už makro zase není definované a nelze používat */
```

Názvy maker je zvykem psát velkými písmeny (stejně jako názvy konstant).
V příkladu ukáži použití standardních i uživatelských maker. Takto vypadá zdrojový kód před zpracováním preprocesorem.

Na řádcích 10 až 12 definuji makro __STDC__ jako 0, pokud není definováno (nemusí být) pomocí podmíněného překladu (který vysvětlím dále).
Řádky 15 až 25 jsou tu kvůli Visual Studiu. Definují makra, která v něm definována nejsou. (Spoléhám se na to, že makro _MSC_VER je definováno jen překladačem Visual Studio.)

```
1.  /*----------------------------------------------*/
2.  /* c11/makra.c                                  */
3.
4.  #include <stdio.h>
5.  #include <stddef.h>
6.
7.  #define N 5
8.  #define VERSE "1.0.0"
9.
10. #ifndef __STDC__
11. #define __STDC__ 0
12. #endif
13.
14. /* Definice pro Visual Studio */
15. #ifdef _MSC_VER
16. #define ZU "Iu"
17. #define SZU "lu"
18. #define __VERSION__ "Visual Studio"
19. #ifdef _M_IX86
20. #define __i386__ 1
21. #else
22. #define __i386__ 0
23. #endif
24. #else
25. #define ZU "zu"
```

```
26.  #define SZU "zu"
27.  #endif
28.
29.  #ifndef __i386__
30.  #define __i386__ 0
31.  #endif
32.
33.  int main(void)
34.  {
35.  int pole[N];
36.  size_t x;
37.
38.  printf("ANSI C:\t%s\t%i\n", __STDC__ ? "ANO" : "NE ", __STDC__);
39.  printf("i386:\t%s\t%s\n", __i386__ ? "ANO" : "NE ", __VERSION__);
40.  printf("Tento program byl prelozen %s v %s.\n"
41.  "Prave jsme na radku %i.\n", __DATE__, __TIME__, __LINE__);
42.
43.  printf("Zadejte index pole <O, %u>: ", N - 1);
44.  scanf("%" SZU, &x);
45.
46.  x = x >= N ? N - 1 : x;
47.
48.  printf("Zadejte hodnotu pole[%" ZU "]: ", x);
49.  scanf("%i", &pole[x]);
50.  printf("verse programu %s\n", VERSE);
51.
52.  return 0;
53.  }
54.
55.  /*----------------------------------------------*/
```

Důvod pro definování ZU SZU viz datový typ pro ukazatel.
A takto po zpracování preprocesorem před vlastním překladem. Direktiva #include <stdio.h> je nahrazena zdrojovým kódem souboru <stdio.h>, ale ten jsem z výpisu vypustil (bylo by to moc dlouhé).

```
int main(void)
{    int pole[5];
     size_t x;

printf("ANSI C:\t%s\t%i\n", 1 ? "ANO" : "NE ", 1);
printf("i386:\t%s\t%s\n", 1 ? "ANO" : "NE ",
    "2.95.4 20011002 (Debian prerelease)");
printf("Tento program byl prelozen %s v %s.\n"
"Prave jsme na radku %i.\n", "Aug 10 2003", "21:43:55", 17);

printf("Zadejte index pole <O,%zu>: ", 5 - 1);
scanf("%zu", &x);

x = x >= 5 ? 5 - 1 : x;

printf("Zadejte hodnotu pole[%zu]: ", x);
scanf("%i", &pole[x]);
printf("verse programu %s\n", "1.0.0");

return 0;
}
```

Teprve tento kód je předán překladači k vytvoření programu.
Výstup z programu:

```
ANSI C: ANO     1
i386:   ANO     2.95.4 20011002 (Debian prerelease)
Tento program byl prelozen Aug 10 2003 v 21:43:55.
Prave jsme na radku 15.
Zadejte index pole <0,4>: 3
Zadejte hodnotu pole[3]: 50
verse programu 1.0.0
```

Makra s argumenty
Makra mohou mít také argumenty. Jsou uzavřeny v kulatých závorkách za jménem makra a pokud je více jak jeden argument, jsou odděleny čárkou.
#define NAZEV(argument1, argument2, ...) makro

Mezi jménem makra a závorkou obsahující argumenty nesmí být mezera. Podívejte se na rozdíl následujících maker (převzato z manuálu k preprocesoru cpp-2.95):

```
#define FOO(x) - 1 / (x)
#define BAR (x) - 1 / (x)
```

Jejich použití …

```
a = FOO(10);
b = BAR;
```

... se vyhodnocení preprocesorem takto:

```
a = - 1 / (10);
b = (x) - 1 / (x);
```

Použití maker s argumenty má své záludnosti se kterými je třeba počítat. Vždy si musíte dát pozor na to, jakým způsobem je makro ve zdrojovém kódu rozvinuto. Podívejte se na definici následujících maker:

```
#define mocnina(X) x*x
#define min(X, Y) ((X) < (Y) ? (X) : (Y))
```

Jejich použití:

```
a = mocnina(3+2);
b = mocnina(++n);
c = min (x+y,foo(z));
```

Vyhodnocení preprocesorem:

```
a = 3+2*3+2;          // tento problém lze vyřešit závorkami v makru
b = ++n*++n;          // proměnná n se inkrementuje dvakrát
c = ((x+y) < (foo(z)) ? (x+y) : foo(z)); // funkce foo() je volána dvakrát
```

Kvůli výše uvedeným problémům se nedoporučuje makry nahrazovat funkce. V jazyce C++ se vůbec od používání maker ustupuje, kde to jen jde.

Operátory # a ##

Operátory `#` a `##` se používají v makrech s parametry. Za operátor `#` se dosadí **řetězec**, který je stejný jako argument makra. Operátor `##` **spojí své dva argumenty** v jeden řetězec.

Definice maker:

```
#define VYPOCET(X,Y) printf("%s = %i\n",#X " + " #Y,(X)+(Y));
#define SPOJENI(X,Y) printf("%i %i\n",X ## Y, cislo ## X);
```

Použití maker:

```
int cislo5 = 10;
VYPOCET(5,6)
SPOJENI(5,6)
```

Vyhodnocení preprocesorem:

```
int cislo5 = 10;
printf("%s = %i\n", "5" " + " "6", (5)+(6));
printf("%i %i\n", 56, cislo5);
```

Podmíněný překlad

Direktiva `#if` vyhodnocuje podmínku, která za ní následuje (tedy preprocesor ji vyhodnocuje). Pokud je tato podmínka vyhodnocena jako FALSE, pak vše mezi `#if` a `#endif` je ze zdrojového kódu vypuštěno.

V podmínce musí být výrazy, které může vyhodnotit preprocesor (tj. nelze tam používat hodnoty proměnných atp.). Pomocí klíčového slova `defined` lze vyhodnocovat, zda existuje nějaké makro (zda bylo definováno). Výraz `#if defined` lze zkrátit výrazem `#ifdef`. Za podmínkou `#if` může být další (a další) podmínka `#elif`. Ta se vyhodnotí v případě, že předchozí podmínka nebyla splněna.

Pokud nejsou splněny žádné podmínky, může být jako poslední direktiva `#else`, která již žádný výraz nevyhodnocuje a provede se právě v případě, že všechny podmínky za `#if` a `#elif` byly vyhodnoceny jako FALSE. Podmíněný překlad se ukončuje direktivou `#endif`. Popsaná syntaxe vypadá následovně:

```
#if PODMINKA1
//... zdrojovy kod pro splnenou podminku1
#elif PODMINKA2
//... zdrojovy kod pro splnenou podminku2 (kdyz podminka1 nebyla splnena)
#elif PODMINKA3
//.... zdrojovy kod pro splnenou podminku3 ...
#else
//... zdrojovy kod, pokud zadna z predchozich podminek nebyla splnena ...
#endif
```

Podmínky lze negovat pomocí operátoru `!` (vykřičník).

Podmínky jsou vyhodnocovány preprocesorem, tedy ještě před tím, než se zdrojový kód předá překladači.

```
/*-----------------------------------------------*/
/* c11/makra2.c                                  */

#include <stdio.h>
#define PT 6

int main(void)
{
#if PT >  5
printf("Makro PT je vetsi jak 5\n");
#endif

#if defined unix && defined linux
printf("Hura, ja bezim na unixovem stroji a snad pod linuxem!\n");
#elif defined unix
printf("Hura, ja bezim pod unixem!\n");
```

```
#else
printf("Hmm, kde jsem se to kruci ocitl??\n");
#endif

return 0;
   }

/*---------------------------------------------*/
```

Užití knihovny

Knihovnami (hlavičkovými soubory) v jazyce C se nazývají soubory obsahující zdrojový kód. Tyto soubory se do jiného zdrojového kódu začleňují pomocí direktivy #include. Tu používáme již od samého začátku pro začlenění standardní knihovny <stdio.h>.

Standardní knihovny hledá překladač na standardních místech. Nemusíte se tedy zajímat o to kde jsou, to už překladač ví. Jména standardních knihoven jsou uzavřeny ve špičatých závorkách <>.

Knihovny, jejichž jména jsou uzavřeny ve dvojitých uvozovkách, se hledají v aktuálním adresáři. Pokud není nalezen, hledá se ve standardních adresářích.

Vyzkoušejte si vytvoření knihovny. Vytvořte nejdříve knihovnu s názvem makra.h a vložte do něj následující kód:

```
/*---------------------------------------------*/
  /* c11/makra.h                               */
        #define POZDRAV "Nazdarek"
      #define COPYRIGHT "(c) 2003"
            #define LICENCE
    #define TISK(a) printf("%s\n",a);
      #define SOUBOR __FILE__
          #define BEGIN {
           #define END }
/*---------------------------------------------*/
```

Poté vytvořte soubor s následujícím obsahem (ve stejném adresáři), do kterého soubor "makra.h" includnete.

```
1.  /*---------------------------------------------*/
2.  /* c11/makra3.c                              */
3.    #include <stdio.h>
4.    #include "makra.h"
5.
6.    int main(void)
7.  BEGIN TISK(POZDRAV) TISK(SOUBOR)
8.    #ifdef LICENCE
9.    TISK(COPYRIGHT)
10.   #endif
11.   return 0;
12.   END
13. /*---------------------------------------------*/
```

Výstup z programu:

```
Nazdarek
makra3.c
(c) 2003
```

Všimněte si, jak jsem definoval makro LICENCE bez toho, že bych mu přiřadil nějakou hodnotu. Pro podmínku#ifdef stačilo, že je definováno. Záměna špičatých závorek za BEGIN a END možná udělalo radost znalcům PASCALu, ale jinak to zdrojový kód značně znepřehledňuje, takže si takové konstrukce raději odpusťte. Leda že byste chtěli potrápit svého učitele programování :-).

Ostatní direktivy

#pragma

Direktiva **#pragma** uvozuje implementačně závislé direktivy. Pokud překladač narazí na tuto direktivu a nerozumí jí, pak ji ignoruje (nezpůsobí to žádnou chybu).

Ukázka (z C++ Builderu):

```
#pragma hdrstop
```

Direktivy #pragma často používají různá vývojová prostředí k uložení informací, které potřebují pro svůj (optimální) běh a se zdrojovým kódem nemají zase až tak moc společného.

#line

Direktiva **#line** nastavuje hodnotu makra __LINE__ (a __FILE__). Není to příliš užitečná direktiva.

```
#line N ["jméno"]
```

N je číslo, kterým se nastaví __LINE__ pro následující řádek a *"jméno"* (nepovinné) je jméno, kterým se nastaví makro __FILE__.

#warning a #error

Tyto direktivy vypíší při překladu programu varování. Direktiva #error navíc překlad ukončí. Používají se spolu s podmíněným překladem.

Jako argument mají text chybového hlášení, který se vypíše.

```
#ifndef __cplusplus
#error Tento program se musi prekladat pomoci prekladace g++ a ne gcc
#endif
```

[1)] Makra __linux__ a __unix__ byla dříve definována jako linux a unix.

Podmínky a cykly

Konečně se dostáváme k té zábavné části programování. Konstrukce které v této kapitole vysvětlím dělají program programem. Díky nim se může program na základě dat rozhodovat, co bude dělat a tím takříkajíc „ožít". Cykly vám zase umožní dělat nějaké akce opakovaně a tak výrazně zefektivní práci při programování. V této kapitole naplno využijete relačních operátorů.

- Podmínka if-else
- Přepínač switch
- Skok goto
- Cyklus for
- Cykly do a while

Podmínka if-else

Podmínka **if-else** je velice podobná podmíněnému operátoru ? : a ještě více podmíněnému překladu preprocesoru.
Větvení programu pomocí podmínky if-else je následující:

```
if (podminka)
    telo bloku
else if (podminka)
    telo bloku
else if (podminka)
    telo bloku
  else
    telo bloku
```

Podmínky musí být v kulatých závorkách. Tělo bloku může být blok (příkazy uzavřené v složených závorkách), nebo jen jeden příkaz (ukončený středníkem).
Větví else if může být konečně mnoho a nejsou povinné, stejně jako není povinná větev else.
Podmínky se vyhodnocují jedna za druhou do té doby, než se narazí na první pravdivou (TRUE). Pak se provedou příkazy v bezprostředně následujícím bloku. Další podmínky se již nevyhodnocují. Pokud se žádná podmínka nevyhodnotí jako TRUE, pak se provede tělo bloku za else (pokud část else existuje).
Else se umisťuje vždy na konec. Je to de facto to samé, jako byste uvedli na konci else if(TRUE) …
V příkladu ukáži několikeré využití podmínky if-else. Připomínám, že funkce scanf() má jako návratovou hodnotu počet správně načtených položek nebo EOF při pokusu číst ze zavřeného vstupu.

```
/*----------------------------------------------*/
    /* c12/ifelse.c                             */
    #define _CRT_SECURE_NO_WARNINGS
    #include <stdio.h>

    #define CISLO 5
    #define MIN 0
    #define MAX 10

    int main(void)
        {
        int x = -1;
        int navrat;
printf("Na jake cislo myslim? Hadej mezi %2i a %2i: ", MIN, MAX);

    navrat = scanf("%i", &x);

    if (navrat == EOF) {
printf("\ascanf nemuze cist. Je uzavren standardni vstup stdin\n");
 return 1;          /* ukoncime funkci main -> konec programu */
      } else if (navrat != 1) {   /* chceme nacist 1 polozku */
            printf("\aNezadal jsi cislo!\n");
                return 1;
                }

    if (x < MIN)
printf("Tak to bylo malo trochu hochu!\n");
        else if (x > MAX)
    printf("Tak to bylo trochu moc!\n");
        else if (x == CISLO)
    printf("UHADNULS :-))\n");
        else
    printf("Smula. Zkus to znovu.\n");

    return 0;
    }

/*----------------------------------------------*/
```

Makro _CRT_SECURE_NO_WARNINGS je tu kvůli funkci scanf(), viz scanf().
Možný výstup:

```
Na jake cislo myslim? Hadej mezi  0 a 10: 5
UHADNULS :-))
```

Zkuste si zadat místo čísla řetězec nebo uzavřít standardní vstup (CTRL+z v DOSu a CTLR+d v Linuxu).

Přepínač switch

Přepínač **switch** obsahuje několik návěstí (case) s **celočíselnou hodnotou**, nebo rozmezí **NIZSI ... VYSSI**(např. 1 ... 5 nebo 'a' ... 'e'). Použití intervalu **NIZSI ... VYSSI** není dle standardu jazyka C, ale rozšíření GNU překladače (gcc), takže vám s jiným překladačem pravděpodobně fungovat nebude (nefunguje např. s Visual Studiem).

Mezi třemi tečkami a hodnotami NIZSI a VYSSI vždy nechávejte mezery.

Na základě argumentu za klíčovým slovem switch přeskočí program na návěstí se stejnou hodnotou jakou má výraz v kulatých závorkách za switch a pokračuje vykonáváním příkazů za ním.

Přepínač switch může obsahovat návěstí default, na které program skočí tehdy, když argument za klíčovým slovem switch neodpovídá hodnotě za žádným návěstím case. Default není povinné a může se použít kdekoliv (první před všemi case, mezi nimi i jako poslední).

```
switch (vyraz)
        {
 case hodnota:
        prikazy
 case hodnota:
        prikazy
   default:
        prikazy
        }
```

Příkaz **break** je speciální příkaz, který způsobí „vyskočení" z těla příkazu switch (a také cyklů, viz dále). Program pak pokračuje ve vykonávání příkazů za blokem switch.

*V příkladu je použita funkce **getchar()** z knihovny <stdio.h>, která vrací hodnotu datového typu int. Proto jsem jí přetypoval na typ char.*

Tato funkce načte ze standardního vstupu znak a vrátí jej jako svou návratovou hodnotu, přičemž vrací speciální hodnotu EOF v případě uzavřeného standardního vstupu.

Hodnota EOF je typu int, proto i getchar() vrací int, ale tuto eventualitu v programu nekontroluju. Vy již jistě víte, jak byste to (pomocí konstrukce if-else) udělali. Tuto a další podobné funkce budu probírat podrobněji později.

```
1.  /*----------------------------------------------*/
2.  /* c12/switch.c                                 */
3.  #include <stdio.h>
4.
5.  int main(void)
6.  {
7.  char znak;
8.
9.  printf("Opravdu chcete smazat vsechny data na disku? [a/n/k]> ");
10. znak = (char) getchar();
11.
12. switch (znak) {
13. default:
14. printf("Mel si zmacknout \'a\',\'n\' nebo \'k\' a ne \'%c\'\n", znak);
15. case 'k':
16. printf("Jako by nekdo zmacknul k\n");
17. return 0;
18. case 'N':
19. printf("Stejne smazu co muzu!\n");
20. break;
21. case 'n':
22. printf("Nechcete? Smula!\n");
23. case 'a':
24. case 'A':
25. printf("Data byla smazana !!!\n");
26. break;
27. #ifdef __unix__
28. case 'b' ... 'e':
29. printf("Trefa do intervalu <b,e>: %c!\n", znak);
30. break;
31. #endif
32. }
33. printf("Ne, nebojte, to byl jenom zertik.\n");
34. return 0;
35. }
36.
37. /*----------------------------------------------*/
```

Možný výstup:

```
Opravdu chcete smazat vsechny data na disku? [a/n/k]> n
Nechcete? Smula!
```

```
Data byla smazana !!!
Ne, nebojte, to byl jenom zertik.
```

nebo

```
Opravdu chcete smazat vsechny data na disku? [a/n/k]> a
Data byla smazana !!!
Ne, nebojte, to byl jenom zertik.
```

nebo

```
Opravdu chcete smazat vsechny data na disku? [a/n/k]> x
Mel si zmacknout 'a','n' nebo 'k' a ne 'x'
Jako by nekdo zmacknul k
```

Všimněte si rozdílu mezi break a return 0. Příkaz break skočí za blok příkazu switch, ale return ukončí funkci main a tím celý program.
Poslední použití break (na konci switch) je zbytečné, protože switch už stejně končí. Je tam jen pro ilustraci.

Skok goto

Předem vás upozorňuji, že použití skoku goto může velice znepřehlednit program a jeho čtení se pak může stát hororem. Proto se vyhýbejte tomuto skoku co můžete a raději jej nepoužívejte vůbec. Ostatně, ani to není nutné. Pomocí ostatních rozhodovacích příkazů se tomu lze **vždy** vyhnout. A teď k příkazu.
Za příkazem goto je název **návěští**, na které má program skočit. Návěští se v programu určuje jednoduše tak, že se zapíše jeho jméno a za ním dvojtečka. Můžete skákat pouze na návěští v rámci jedné funkce. A jak by řekl Spok, z logiky věci vyplývá, že nemohou být dvě návěští stejného jména.

```
1.  /*----------------------------------------------*/
2.  /* c12/goto.c                                   */
3.
4.  #include <stdio.h>
5.
6.  int main(void)
7.  {
8.  char znak;
9.
10. printf("Opravdu chcete smazat vsechny data na disku? [a/n/k]> ");
11. znak = (char) getchar();
12.
13. if ((znak == 'a') || (znak == 'A'))
14.  goto ano;
15.  if (znak == 'n')
16.  goto ne;
17.  if (znak == 'N')
18.  goto ne2;
19.  if (znak == 'k')
20.  goto konec2;
21.
22. printf("Mel si zmacknout \'a\',\'n\' nebo \'k\' a ne \'%c\'\n", znak);
23.  konec2:
24.  return 0;
25.
26.  ne2:
27.  printf("Stejne smazu co muzu!\n");
28.  goto konec;
29.  ne:
30.  printf("Nechcete? Smula!\n");
31.
32.  ano:
33.  printf("Data byla smazana !!!\n");
34.  konec:
35.  printf("Ne, nebojte, to byl jenom zertik\n");
36.  return 0;
37.  }
38.
39. /*----------------------------------------------*/
```

Cyklus for

Pomocí cyklů můžete vykonávat nějakou činnost opakovaně. Můžete cyklit buďto jeden příkaz, nebo nějaký blok příkazů. Pokud chcete cyklus z nějakého důvodu ukončit, můžete v bloku uvést příkaz break. Cyklus se tak ukončí a program pokračuje příkazy za tělem cyklu. Jinou možností je příkaz **continue**. Tento příkaz přeruší vykonávání dalších příkazů v těle cyklu a program skočí na začátek cyklu. Podívejte se na cyklus **for**.

```
for (inicializace; podmínka; výraz)
telo cyklu
```

Cyklus **for** provádí příkazy v těle bloku tak dlouho, dokud platí *podmínka*. Tato podmínka může být jakýkoliv výraz vracející celočíselnou hodnotu. Může obsahovat konstanty, proměnné, relační operátory (==,>,|| atp.) a není povinná. V případě, že žádnou *podmínku* neuvedete, překladač za ní dosadí číslo jedna, což bude mít za následek věčné provádění cyklu. Cyklus ale pořád můžete ukončit pomocí break. Podmínka se vyhodnocuje před každým začátkem cyklu (i po příkazu continue). Pokud se hned na poprvé vyhodnotí jako FALSE, pak tělo cyklu neproběhne ani jednou.
Část *inicializace* také není povinná. Může v ní být libovolný výraz. Tato část proběhne jen jednou (před začátkem cyklu). Klidně byste inicializaci mohli přesunout před příkaz for, ale pokud se inicializace týká cyklu for, je čitelnější ji použít uvnitř for.
Poslední část v příkazu označená jako *výraz* se provádí po každém ukončení cyklu (i příkazem continue), ale už ne po skončení cyklu (třeba příkazem break). Taktéž není povinná.
Typické použití cyklu for:

```
1.  /*--------------------------------------------*/
2.  /* c12/for.c                                  */
3.  #include <stdio.h>
4.
5.  int main(void)
6.  {
7.  int x;
8.
9.  for (x = 1; x < 10; x++) {
10. if ((x % 2) != 0)
11. continue;
12. printf("Sude cislo: %i\n", x);
13. }
14.
15. printf("\nA nebo trochu jinak:\n\n");
16. x = 1;
17. for (;;) {
18. if (x >= 10)
19. break;
20. if ((x % 2) == 0)
21. printf("Sude cislo: %i\n", x);
22. x++;
23. }
24. return 0;
25. }
26.
27. /*--------------------------------------------*/
```

Výstup z programu:

```
Sude cislo: 2
Sude cislo: 4
Sude cislo: 6
Sude cislo: 8

A nebo trochu jinak:

Sude cislo: 2
Sude cislo: 4
Sude cislo: 6
Sude cislo: 8
```

Cyklus for se také často využívá při práci s poli. Například inicializace pole samými nulami (všimněte si indexování pole od 0 do N-1):

```
#define N 100
int x, pole [N];

for (x = 0; x <= N - 1;x++)
    pole[x] = 0;
```

Cykly do a while

Cyklus **do** má tu vlastnost, že proběhne alespoň jednou. Je ukončen klíčovým slovem while, za kterým je*podmínka* v kulatých závorkách a platí o ní totéž, jako u cyklu for. Tato podmínka se vyhodnocuje až po průchodu cyklem.
V cyklu do můžete pro řízení chodu programu využít příkazů break i continue. Tělo cyklu může být jeden příkaz nebo blok (uzavřené příkazy ve složených závorkách).

```
do
    telo cyklu
while ( podminka );
```

Cyklus **while** je podobný, jen s tím rozdílem, že se podmínka vyhodnocuje před provedením cyklu.

```
while ( podminka )
    telo cyklu
```

Zda použijete cyklus for, do nebo while záleží většinou jen na vaší fantazii a preferenci. Máte 3 způsoby, jak udělat totéž.

```
1.  /*--------------------------------------------*/
2.  /* c12/while.c                                */
3.  #define _CRT_SECURE_NO_WARNINGS
4.  #include <stdio.h>
```

```
5.
6.   int main(void)
7.   {
8.   int iterace = -1;
9.   int navrat;
10.  do {
11.  printf("Zadejte pocet iteraci <O,10>: ");
12.  navrat = scanf("%i", &iterace);
13.  if (!navrat)
14.  return 1;          /* uzivatel nezadal cislo; navrat == 0 */
15.  if (navrat == EOF) {
16.  printf("stdin je uzavren.\n");
17.  return 1;
18.  }
19.  } while ((iterace < 0) || (iterace > 10));
20.
21.  while (iterace) {
22.  printf("%i ", iterace--);
23.  }
24.
25.  printf("\n");
26.  return 0;
27.  }
28.
29.  /*-----------------------------------------------*/
```

Možný výstup:

```
Zadejte pocet iteraci <0,10>: -5
Zadejte pocet iteraci <0,10>: 15
Zadejte pocet iteraci <0,10>: 5
5 4 3 2 1
```

Lomené závorky v cyklu while v příkladu jsou sice nadbytečné (protože cyklus obsahuje jen jeden příkaz), ale zlepšují čitelnost kódu a snižuje se riziko chyby při úpravě zdrojového kódu.
Častou chybou je totiž přidání příkazu za příkaz u cyklu (nebo i podmínky if) a zapomenutí uzavření těchto příkazů do bloku.
Jako v následujícím fragmentu kódu, kde se vytiskne 10x písmeno „X" a jen jedenkrát „Y".

```
for (i = 0; i > 10; i++)
    printf("X");
    printf("Y");
```

Tak bacha na to, mí věrní c-čkaři!

Funkce

Funkce jsou základním stavebním kamenem jazyka C. Zatím jste se setkali s funkcí main() a s funkcemi printf() a scanf(). V této kapitole se především naučíte vytvářet vlastní funkce. K popisu funkcí, definovaných normou ANSI C, které již překladač obsahuje, se dostanu později.

- Definice funkce
- Deklarace funkce
- Rekurzívní funkce
- Proměnlivý počet argumentů

Standardní funkce jsou obsaženy ve standardních knihovnách (například printf() a scanf() jsou deklarovány v souboru <stdio.h>).

Překladač je obvykle nainstalován i se spoustou dalších funkcí, jež nejsou v normě ANSI C. Jejich použití si musíte nastudovat v dokumentaci. Jsou to funkce závislé jak na překladači, tak mnohdy na operačním systému, takže jejich použitím se značně snižuje přenositelnost kódu.

Funkce se při spuštění programu usídlí kdesi v paměti. Při její volání program přeskočí do funkce a po jejím provedení se vrátí za místo, kde byla funkce volána. Pokud voláte funkci v programu vícekrát, znamená to jen to, že přeskakujete vícekrát na to samé místo v paměti. Vytvářením funkcí tak šetříte paměť počítače, ale šetříte i sobě práci, protože tělo funkce napíšete jen jednou a v programu použijete kolikrát potřebujete. Na druhou stranu, každé volání funkce stojí nějaký (velmi malý) čas na „odskočení si".
Pokud funkci voláte v cyklu tisíckrát, už je to 1000x malý čas, a to je trochu znát.
Například:

```
for (x = 0; x < 1000; x++)
{
    printf("Cyklus cislo: ");
    printf("%i\n",x);
}
```

… je pomalejší než …

```
for (x = 0; x < 1000; x++)
{
    printf("Cyklus cislo: %i\n",x);
}
```

Úspora místa a zpřehlednění programu je mnohem důležitější, než časová režie spojená s voláním funkce. Navíc platí, že při změně je daleko snazší a bezpečnější (z hlediska možných programátorských chyb) změnit jednu funkci, než několik míst v programu. Pokud však vytváříte malou funkci, kde je její volání v některých místech kritické, je někdy výhodné ji nahradit pomocí maker (ovšem se všemi jejich záludnostmi). Změna makra pak znamená změnu ve všech místech programu, kde je makro použito, stejně jako změna funkce znamená změnu chování programu ve všech místech, kde je funkce volána.

*Jazyk C++ tento problém řeší elegantněji pomocí tzv. **inline** funkcí. Jak už jsem psal, makra se v C++ moc nenosí.*

Definice funkce

Definice funkce vypadá takto:

```
navratovy_typ jmeno ([parametry, ...])
{
telo funkce
}
```

Funkce v C/C++

Jméno funkce slouží k jejímu identifikování. Při volání funkce v programu musíme uvést za jménem funkce kulaté závorky, a to i tehdy, když funkce nemá žádné argumenty. Samotné jméno funkce bez závorek totiž reprezentuje adresu v paměti, kde funkce začíná (kam si program odskočí, když je funkce volána). Toho se dá využít v odkazech na funkci (o tom ale až později).

Parametry funkce popisují očekávaná data, která bude funkce zpracovávat. Každý parametr má své jméno a musí být určen jeho datový typ. Pokud funkce nemá žádné parametry, uvádí se v závorkách slůvko **void** (ale není to povinné). Pokud je jich více než jeden, oddělují se čárkou. Existují i funkce s proměnlivým počtem argumentů.

Parametrem může být celé nebo racionální číslo, znak, struktura, nebo ukazatele. Nemůže to být například pole, které obsahuje x prvků, ale může to být **ukazatel** na začátek pole.

Funkce má jednu **návratovou hodnotu**. Její typ se uvádí před jménem funkce. Například funkce main() má vždy návratovou hodnotu typu int. Pokud nechcete, aby funkce vracela nějaká data, jako návratovou hodnotu uveďte **void**.

Funkci lze kdykoliv ukončit pomocí příkazu **return**, za který se uvádí hodnota nebo výraz, jehož výsledek se stane návratovou hodnotou funkce. Pokud funkce žádnou návratovou hodnotu nemá, uvádí se return bez hodnoty nebo výrazu (ukončený středníkem). Pokud funkce má návratovou hodnotu, je použití return na konci funkce povinné a za return musí být výraz, jenž po vyhodnocení musí mít stejný datový typ, jako je datový typ návratové hodnoty funkce.

Po ukončení funkce, ať již příkazem return, nebo tím že se vykonají všechny příkazy v jejím těle, se pokračuje v provádění kódu za místem kde byla funkce volána (program skočí zpět z těla funkce na místo, odkud byla funkce volána). Pokud však příkazem return ukončíte funkci main(), ukončíte program.

Návratová hodnota funkce main() se vrací operačnímu systému. Zaběhnutá praxe je, že návratová hodnota 0 znamená úspěšné ukončení programu, jakákoliv jiná hodnota určuje číslo chyby (to už je čistě na programátorovi programu, aby vymyslel různým chybám různá čísla).

```
/*----------------------------------------------*/
   /* c13/fce1.c                                 */
                #include <stdio.h>

                void tecka(int pocet)
                {
                  if (pocet <= 0)
 return;       /* kdyby byl pocet unsigned int,
                   bylo by tohle zbytecne */
  /* kdyz se podivate na podminku v cyklu tak
   * zjistite, ze je to zbytecne stejne :-) */
                for (; pocet > 0; pocet--)
                  printf(". ");
                }

                int mocnina(int x)
                {
                return x * x;
                }

                int main(void)
                {
                  tecka(10);
 printf("-5^2 = %i\n", mocnina(-5));
                  return 0;
                }

/*----------------------------------------------*/
```

Výstup programu:

```
. . . . . -5^2 = 25
```

Na řádku 23 se volá funkce tecka(), které se předá do argumentu *pocet* číslo 10. Program si odskočí na řádku 5. Pokud by byl *pocet* záporný, tak na řádce 8 vyskočí z funkce a pokračuje na řádku 24. Protože *pocet* nebyl záporný, funkce pokračovala ve svém těle. Když ukončila cyklus a dostala se na konec svého těla (řádek 14), vrátilo se provádění programu na řádku 24.

*Programu je při spuštěný vyhrazena určitá část paměti, kam si poznamenává, kde v provádění svého kódu skončil, než si odskočí do funkce, aby věděl, kam se zase vrátit. Navíc do této paměti uloží hodnoty argumentů funkce a funkce do této paměti ukládá svůj výsledek (návratovou hodnotu). Této paměti se říká **zásobník** (stack).*

Když ve funkci zavoláte další funkci a v té zase další funkci …, zásobník se pomalu zaplňuje. Když z funkce vyskočíte, zásobník se zase uvolňuje. (Podobně jako když do zásobníku strkáte patrony. Poslední vložená patrona do zásobníku se z něj vystřelí jako první).

Deklarace funkce

Při deklaraci funkce se uvádí pouze datový typ návratové hodnoty, jméno funkce a typy argumentů. To je vše, co potřebuje překladač k tomu, aby její **volání** mohl do programu zapsat. Mohou se uvést i názvy argumentů, ale to není nutné. **Než je funkce v programu volána, musí být deklarována**, ale definována může být až později.

Pokud funkci předem nedeklarujete, ale rovnou definujete, je definice zároveň i deklarací. Pokud se deklarace s pozdější definicí neshodují, oznámí překladač chybu.

V následujícím programu se funkci nejdříve deklaruje, použije se v jiné funkci a pak se teprve definujeme.

```
/*-----------------------------------------------*/
   /* c13/deklar.c                                */
                #include <stdio.h>


float vypocet(int, float, float *);     /* deklarace funkce */

                int main(void)
                    {
                   float a, b;

                a = vypocet(5, (float) 0.3, &b);
printf("Soucet = %5.2f\nNasobek = %5.2f\n", a, b);

                return 0;
                    }

                float vypocet(int a, float b, float *c)
                                    {
                                     float f;
                                   *c = (float) a *b;
                                   f = (float) a + b;
b = 55.5;                               /* menim lokalni promenou b,
                                      to nema s promennou b ve funkci
                                          main nic spolecneho */
                                   return f;
                                    }

/*-----------------------------------------------*/
```

Výstup z programu:

```
Soucet =  5.30
Nasobek =  1.50
```

Tak malý příklad a tolik toho ukazuje. Platnost lokálních proměnných jen uvnitř funkce, využití návratové hodnoty funkce, **dopřednou deklaraci** funkce a použití ukazatele. Funkci jsem jako třetí argument předal adresu proměnné *b* a díky tomu mohla funkce na tuto adresu uložit nějakou hodnotu (v příkladě násobek čísel *a* a *b*).

Přiřazení na řádku 23 je zbytečné, protože se s proměnnou *b* v těle funkce už dál nepracuje a po skončení funkce se hodnota „ztratí" (příští volání funkce vypocet() nastaví zase *b* dle druhého přiřazeného argumentu).

Příklad taky ukazuje, jak je blbé pojmenovávat proměnné *a*,*b*,*c* … Alespoň by stálo za to přemenovat *f* třeba na *soucet* a *c* na *nasobek*. Oč by byl pak program čitelnější.

Rekurzívní funkce

Pokud funkce volá ve svém těle samu sebe, nebo je volána funkcí, kterou volá ve svém těle, hovoříme o **rekurzi**. Data, která jsou funkci předávána, se ukládají do takzvaného **zásobníku**.

To je nějaké vyhrazené místo v paměti. Po skončení funkce se data ze zásobníku zase odstraní. Pokud však funkce během svého vykonávání zavolá samu sebe, pak se do zásobníku umístí další data a tak stále dokola. To je samozřejmě velice náročné na paměť a může vést až ke zhroucení programu (pokud dojde místo vyhrazené pro zásobník). Použití rekurze je sice efektní, ale ne vždy efektivní. Proto je třeba mít opravdu dobrý důvod pro používání rekurzí. Následuje ilustrativní příklad. V něm dobrý důvod pro použití rekurze určitě není :-).

```
1.  /*-----------------------------------------------*/
2.     /* c13/rekurz.c                                */
3.        #include <stdio.h>
4.
5.    float prvni_funkce(int, float); /* deklarace */
6.
7.     float druha_funkce(int a, float f)
8.    {                              /* definice */
9.     printf("Vola se druha funkce\n");
```

```
10.  return prvni_funkce(a, f * (a + 1));
11.  }
12.
13.  float prvni_funkce(int a, float f)
14.  {                              /* definice */
15.  printf("Prvni funkce a= %2i, f=%5.2f\n", a, f);
16.  if (a <= 0)
17.  return f;
18.  return druha_funkce(--a, f);
19.  }
20.
21.  int main(void)
22.  {
23.  printf("Vysledek je %f\n", prvni_funkce(5, 5.0));
24.  return 0;
25.  }
26.
27.  /*-----------------------------------------------*/
```

Výstup z programu:

```
Prvni funkce a=  5, f= 5.00
   Vola se druha funkce
Prvni funkce a=  4, f=25.00
   Vola se druha funkce
Prvni funkce a=  3, f=100.00
   Vola se druha funkce
Prvni funkce a=  2, f=300.00
   Vola se druha funkce
Prvni funkce a=  1, f=600.00
   Vola se druha funkce
Prvni funkce a=  0, f=600.00
  Vysledek je 600.000000
```

V příkladě se počítá jakási číselná řada. Druhá funkce zde slouží jen k nějakému výpisu a k novému zavolání funkce prvni_funkce(). Berte to jenom jako příklad, jistě byste dokázali daný problém vyřešit rozuměji.

Bez dopředné deklarace prvni_funkce() byste tuto funkci nemohli použít v těle druhe_funkce() a naopak.

Mimochodem, napadlo vás, co by se stalo, kdyby se volání return druha_funkce(**--a**, f); změnilo na return druha_funkce(**a--**, f);?

V takovém případě by se volala druha_funkce() s hodnotou proměnné *a*, která by nebyla snížena o jedničku. Ta by se měla snížit až po zavolání funkce. Výsledkem by byl nekonečný cyklus vzájemného volání funkcí. Zkuste si to a uvidíte, jak rychle vám program klekne :-).

Na podobné problémy si musíte dávat pozor. Logické chyby programátora se odhalují mnohem hůře, než syntaktické chyby jazyka C (ty ostatně odhalí již překladač). Při programování v C/C++ prostě nemůžete být nikdy dost obezřetní.

Klasickým příkladem na rekurzi je výpočet faktoriálu (avšak i faktoriál lze vypočítat elegantně bez rekurze).

```
1.  /*-----------------------------------------------*/
2.  /* c13/faktor.c                                  */
3.  #include <stdio.h>
4.
5.  long double faktorial(long double x)
6.  {
7.  if (x == 0L)
8.  return 1L;
9.  return x * faktorial(x - 1L);
10.  }
11.
12.  int main(void)
13.  {
14.  int i;
15.  long double f;
16.  for (i = 0; i <= 40; i++) {
17.  f = faktorial(i);
18.  printf("%2i! = %050.0Lf (%0.3Le)\n", i, f, f);
19.  }
20.  return 0;
21.  }
22.
23.  /*-----------------------------------------------*/
```

Výstup je trochu zkrácen:

```
0! = 00000000000000000000000000000000000000000000000001 (1.000e+00)
1! = 00000000000000000000000000000000000000000000000001 (1.000e+00)
2! = 00000000000000000000000000000000000000000000000002 (2.000e+00)
3! = 00000000000000000000000000000000000000000000000006 (6.000e+00)
4! = 00000000000000000000000000000000000000000000000024 (2.400e+01)
```

```
5! = 00000000000000000000000000000000000000000000000120 (1.200e+02)
6! = 00000000000000000000000000000000000000000000000720 (7.200e+02)
7! = 00000000000000000000000000000000000000000000005040 (5.040e+03)
8! = 00000000000000000000000000000000000000000000040320 (4.032e+04)

...

35! = 00000000010333147966386144929973846968255251480576 (1.033e+40)
36! = 00000000371993326789901217462530208167145295052800 (3.720e+41)
37! = 00000013763753091226345045735828383554804299857920 (1.376e+43)
38! = 00000523022617466601111725872220378936271647539200 (5.230e+44)
39! = 00020397882081197443356844789080046496991166857216 (2.040e+46)
40! = 00815915283247897734263888042887576837447481294848 (8.159e+47)
```

Možná jste si po prohlédnutí výsledků všimli, že na výpočtech není něco v pořádku. Nejsou totiž příliš přesné. Platí přeci, že 40! = 39!*40, ovšem 203978820811974433568447890800464996991166857216*40 = 8159152832478977342737915632018598796466742886 40
a ne
8159152832478977342638880428875768374474812948 48 (a že je to pořádná chyba).
Je to smutné, ale **jazyk C neumí počítat s racionálními čísly moc přesně**.
Je možné, že s vaším překladačem dostanete přesnější výsledky (například s Visual Studiem), zde popsaný problém se ale týká každého překladače - jen se chyba projeví trochu později (s většími čísly).
Je to dáno tím, jak jsou čísla v paměti uložena a jak se s nimi počítá. Když třeba vydělíte 10/3 a výsledek znovu vynásobíte třemi, vyjde vám 9.999 …
Čísla typu float/double mají zaručenou přesnost cca na 14 „nejvýznamnějších" míst. (Záměrně neříkám desetinných míst, neboť to záleží na tom, s jak velkým/malým číslem pracujete). Vzhledem k velikosti čísla je chyba sice zanedbatelná, ovšem pokud budete programovat řekněme pro nějaký bankovní úřad, asi by z vás nikdo neměl radost.
Naštěstí lze přesně počítat s celočíselnými proměnnými, takže lze podobně velká čísla zpracovávat pomocí několika celočíselných proměnných (a složitých algoritmů). Existují knihovny funkcí, které jsou zaměřené na přesné výpočty s velkými čísly.
Za domácí úkol napište funkci faktorial() bez použití rekurze. Bude vám k tomu stačit pár lokálních proměnných a cyklus dle libosti.

Proměnlivý počet argumentů

Zatímco doteď jste při deklaraci nebo definici funkce určovali jaké bude mít argumenty, nyní se naučíte vytvořit funkci bez přesného počtu a typu argumentů. Vzpomeňte si například na funkce printf() a scanf(), které také mají proměnlivý počet argumentů.
Argumenty funkce se ukládají do zásobníku. Aby mohla funkce k těmto argumentům přistupovat, musíte ji předat alespoň místo, kde zásobník začíná. Proto musí mít funkce s proměnlivým počtem argumentů **minimálně jeden argument napevno daný**.
To, že mohou následovat další argumenty, se při definici nebo deklaraci označí pomocí tří teček (tzv. **výpustka**).
Pro práci s nedeklarovanými argumenty existuje standardní knihovna **<stdarg.h>**. V ní jsou definovány tři makra. Zde jsou jejich deklarace:

```
void va_start(va_list ap, last);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Výraz va_list vnímejte jako datový typ proměnné *ap*. Je to proměnná, která reprezentuje nedeklarované argumenty (list všech argumentů). Proměnnou *ap* byste si mohli pojmenovat jakkoliv, ale je dobrým zvykem zůstat u *ap*. Když si na to zvyknete, je pak čtení takového kódu jednodušší.
Makro **va_start** inicializuje proměnnou *ap* pomocí poslední napevno deklarované proměnné ve funkci. Ta je předána jako druhý argument – *last* (díky tomu získá potřebnou adresu do zásobníku s argumenty).
Makro **va_arg** vrací hodnotu dalšího argumentu v řadě. Jakého typu má být se musí určit druhým argumentem **type**. Díky tomu bude makro vědět, kolik bytů načíst (a o kolik se posunout na začátek dalšího argumentu funkce).
Toto makro můžete volat vícekrát, maximálně však tolikrát, kolik je předáno funkci při volání nedeklarovaných argumentů. Jinak byste se pokoušeli přečíst data, která již nepatří k argumentům funkce.
Makro **va_end** zajišťuje bezproblémové ukončení práce s *ap*. Ačkoliv vám program poběží pravděpodobně i bez volání tohoto makra, nikdy na něj nezapomínejte. V určitých kritických situacích by se vám to vymstilo a po půl roce byste takovou chybu v programu jen těžko hledali. (va_end může například uvolňovat paměť o kterou požádalo va_start).
Nyní se nabízí otázka, jak zjistí funkce s proměnlivým počtem argumentů, jaké jí byli argumenty předány (jakého typu) a také kolik. Možností je hned několik. Například známá funkce printf() má vždy jako první argument řetězec (přesněji řečeno ukazatel na řetězec; řetězec je znakové pole a pole nelze předávat jako argument funkce).
V tomto řetězci hledá speciální sekvence (%s,%c) a podle nich zjišťuje, že má mít další argument a také jakého typu. A tak je přesně dáno, o kolik argumentů a jakých typů si funkce řekne.
Další možností, jak určit počet předaných argumentů je, že při deklaraci funkce deklarujete jeden parametr (minimálně jeden stejně vždy musíte mít), který bude obsahovat počet proměnných. Další možností je například definování nějaké zarážky. Třeba že vždy jako poslední argument bude ukazatel s hodnotou NULL, nebo číslo nula (tak se pracuje s textovými řetězci) atp. To už je jen na vás.

```
1.  /*----------------------------------------------*/
2.  /* c13/fce2.c                                   */
3.  #include <stdio.h>
4.  #include <stdarg.h>
5.  #include <stdbool.h>
6.
7.  int maximum(const int pocet, const bool znamenko, ...)
8.  {
9.  va_list ap;
```

```
10.  int i;
11.  int maxS, dalsiS;
12.  unsigned int maxU, dalsiU;
13.
14.  if (pocet <= 0)
15.  return 0;
16.
17.  /* znamenko je posledni deklarovana promenna */
18.  va_start(ap, znamenko);
19.
20.  if (znamenko)
21.  maxS = va_arg(ap, int);
22.  else
23.  maxU = va_arg(ap, unsigned int);
24.
25.  for (i = 0; i < pocet - 1; i++) {
26.  if (znamenko) {
27.  dalsiS = va_arg(ap, int);
28.  maxS = (maxS < dalsiS) ? dalsiS : maxS;
29.  } else {
30.  dalsiU = va_arg(ap, unsigned int);
31.  maxU = (maxU < dalsiU) ? dalsiU : maxU;
32.  }
33.  }
34.
35.  va_end(ap);
36.  if (znamenko)
37.  return maxS;
38.  else
39.  return (signed int) maxU;
40.  }
41.
42.  int main(void)
43.  {
44.  printf("Maximum = %i\n", maximum(3, true, 7, 9, -2));
45.  printf("Maximum = %i\n", maximum(7, true, -5, 8, -2, 5, -6, -1, 4));
46.  printf("Maximum = %u\n", (unsigned int) maximum(7, false, -5, 8, -2, 5, -6,-1, 4));
47.  return 0;
48.  }
49.
50.  /*---------------------------------------------*/
```

Parametry pocet a znamenko jsem deklaroval jako konstantní, aby bylo jasné, že není vhodné jejich hodnotu ve funkci měnit.

Výstup z programu:

```
Maximum = 9
Maximum = 8
Maximum = 4294967295
```

Funkce maximum() je navržená tak, že zvládne najít maximální hodnotu pro čísla se znaménkem i bez. To je ale špatný návrh z několika důvodů. Za prvé, návratová hodnota je definována jako se znaménkem, což je při výpočtu bez znaménka matoucí a výsledek se musí přetypovávat. Za druhé, tělo funkce to dost zesložiťuje a znepřehledňuje. Taky se vám zvětšuje počet parametrů (kvůli parametru *znamenko*).

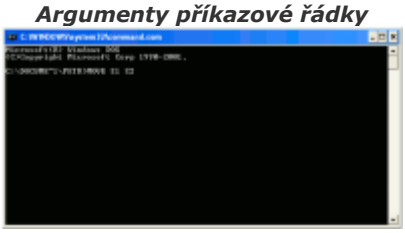
Mnohem lepší by bylo mít dvě funkce - jednu pro int a jednu pro unsinged int.

Funkce by měli, pokud možno, dělat jen jednu, jasně danou, činnost. Důvod je zase zlepšení čitelnosti programu. Jednou napsanou funkci můžete konec konců používat v mnoha programech. Čím je funkce jednodušší, tím je větší šance, že ji budete moci použít i jinde bez dodatečných úprav.

Funkce II

- Argumenty příkazové řádky
- Lokální, globální a statické proměnné
  - Statické funkce a globální proměnné
- Uživatelské knihovny funkcí

Argumenty příkazové řádky



Příkazový řádek ve Windows

Když spouštíte programy z příkazové řádky (ať už v Linuxové konzoli nebo v DOSu), můžete jim předávat nějaké argumenty. Argumenty příkazové řádky jsou všechny texty (oddělené mezerami), které zapíšete za název programu v příkazové řádce.

Například v příkazu MOVE S1 S2 jsou texty S1 a S2 argumenty. Argumenty jsou rozdělené tzv. bílými znaky (mezery, tabulátory) a programu jsou předávány jeden po druhém (za chvíli uvidíte jak). Pokud budete chtít předat více slov jako jeden argument, pak je stačí uzavřít do uvozovek.

Argumenty příkazové řádky jsou předávány funkci main(), která je deklarována následujícím způsobem:

```
int main(int argc, char *argv[]);
```

Parametr argc představuje počet argumentů příkazové řádky a parametr argv je pole ukazatelů na argumenty příkazové řádky. Argumenty příkazové řádky jsou textové řetězce (i když je to jenom jeden znak nebo číslo) a počítá se mezi ně i název programu, který je prvním argumentem.

Například když spustíte program takto:

```
$ program Zvolejte 3 krat hura
```

Pak argc bude mít hodnotu 5, argv[0] bude ukazatel na řetězec "program", argv[1] ukazatel na řetězec "Zvolejte", argv[2] ukazatel na řetězec "3" atd.

Co k tomu dodat. Snad jen to, že jména argc a argv si můžete libovolně změnit. Ovšem doporučuji je nechat taková, jaká jsou. Pokud si na to zvyknete, budou se vám vaše i cizí (třeba moje) programy lépe číst.

Ukáži teď program, který bude pracovat na základě argumentů příkazové řádky. Program převede svůj druhý argument na velká nebo malá písmena podle toho, co mu předáte jako první argument. Aby šlo porovnat jaké argumenty byly předány, napíši funkci, která bude porovnávat dva textové řetězce. Bude vracet 1 (TRUE) v případě, že jsou schodné, jinak 0 (FALSE).

```
1.  /*----------------------------------------------*/
2.  /* c14/argum.c                                  */
3.
4.  #include <stdio.h>
5.  #include <stddef.h>
6.
7.  #define VELKE "velke"
8.  #define MALE "male"
9.
10. #define VAROVANI printf("Zadejte: %s %s | %s ARGUMENT\n",argv[0],VELKE,MALE);
11.
12. int shodne(char *v1, char *v2)
13. {
14.   size_t i = 0;
15.   /* pomoci zarazek '\0' zjistuti konec retezce */
16.   while ((v1[i] != '\0') && (v2[i] != '\0')) {
17.     if (v1[i] != v2[i])
18.       return 0; /* nejsou shodne; */
19.     i++;
20.   }
21.   if (v1[i] || v2[i]) {
22.     return 0; /* jeden retezec jeste neskoncil, nejsou stejne */
23.   }
24.   return 1;    /* vsechny znaky byly shodne */
25. }
26.
27. void velke(char *);
28. void male(char *);
29.
30. int main(int argc, char *argv[])
31. {
32.   if (argc != 3) {
33.     VAROVANI
34.     return 1;
35.   }
36.
37.   if (shodne(VELKE, argv[1]))
38.     velke(argv[2]);
39.   else if (shodne(MALE, argv[1]))
40.     male(argv[2]);
41.   else {
42.     VAROVANI
43.     return 1;
44.   }
45.
46.   printf("Druhy argument byl zmenen na: %s\n", argv[2]);
47.   return 0;
48. }
49.
50. /* ve funkcich velke a male vyuziji vlastnosti ASCII tabulky */
51.
52. void velke(char *v)
53. {
54.   size_t i = 0;
55.   while (v[i]) {          /* '\0' odpovida hodnote FALSE */
56.     if (v[i] >= 'a' && v[i] <= 'z') {
57.       /* rozdil v ASCII tabulce mezi velkymi a malymi pismeny */
58.       /* 'A' == 65, 'a' == 97 */
```

```
59.  v[i] += ('A' - 'a');
60.  }
61.  i++;
62.  }  /* konec while */
63.  }
64.
65.  void male(char *v)
66.  {
67.  size_t i = 0;
68.  while (v[i]) {
69.  if (v[i] >= 'A' && v[i] <= 'Z') {
70.  v[i] += ('a' - 'A');
71.  }
72.  i++;
73.  }
74.  }
75.
76.  /*---------------------------------------------*/
```

Tento příklad taky není zrovna ukázkou programátorského umu. Je to spíše jen ukázka toho, co lze v jazyku C napsat. Tak například, než použít #define VAROVANI by bylo lepší napsat jednoduchou funkci, která by se dala v budoucnu snadněji rozšiřovat. Taky výraz (v1[i] != '\0') mohl být v podmínce if zjednodušen na v1[i], protože jak víte, konec pole je dán nulovým znakem a nulový znak se vyhodnocuje jako FALSE (nepravda). Taky neexistuje žádný rozumný důvod pro to, aby funkce shodne byla definována před funkcí main() a zbylé dvě funkce až za ní (buď si vyberu jedno nebo druhé, ale nepatlám to jak pejsek a kočička).

V příkladu využívám vlastnosti ASCII tabulky, ve které jsou malá i velká písmena uložena abecedně za sebou. Převedení velkého písmene na malé docílím přičtením vzdálenosti mezi velkými a malými písmeny. Typ char, jak víte, je osmibitový typ a jak vidíte, lze jej bez problémů používat jako číslo.

Ne všechny operační systémy využívají tabulku ASCII a národní znaky nemají velká a malá písmena vždy všechny stejně „daleko". Využití vlastností ASCII tabulky jako v tomto příkladě snižuje přenositelnost kódu.

Ukázka použití (/tmp/argum je cesta k programu vytvořenému z argum.c):

```
$ /tmp/argum
Zadejte: /tmp/argum velke | male ARGUMENT
$ /tmp/argum velke "Ahoj Svete"
Druhy argument byl zmenen na: AHOJ SVETE
```

Jak předat argumenty příkazové řádky ve Visual Studiu viz O Visual Studiu.

Lokální, globální a statické proměnné

O lokálních a globálních proměnných jsem již mluvil. Takže jen zopakuji, že lokální proměnné jsou proměnné definované v těle nějakého bloku, kdežto globální proměnné mimo něj. Dále platí, že lokální proměnné mají vyšší prioritu než globální. To znamená, že když vytvoříte lokální a globální proměnnou stejného jména, pak použitím proměnné s tímto jménem používáte lokální proměnou (samozřejmě v bloku, ve kterém byla definována, mimo něj / po skončení bloku nemá lokální proměnná platnost a tudíž mimo blok používáte globální proměnnou). Možná to zní trochu zmateně, ale z příkladu vám to bude jistě jasné. Do lokálních proměnných lze započítat i parametry funkcí. ~~Jejich priorita je mezi globálními a lokálními proměnnými.~~ [1)]

```
/*---------------------------------------------*/
/* c14/promenne.c                              */
#include <stdio.h>

int a = 0, b = 0, c = 0;        /* globalni promenne */

void funkce(int a, int b)
{
/*int a = -5;  // lokalni promenna se nemuze jmenovat jako parametr */
printf("fce: a = %i, b = %i, c = %i\n", a, b, c);
}

int main(void)
{
  int c = 25;

printf("main: a = %i, b = %i, c = %i\n", a, b, c);
funkce(100, 200);
  return 0;
}

/*---------------------------------------------*/
```

Výstup z programu:

```
main: a = 0, b = 0, c = 25
fce: a = 100, b = 200, c = 0
```

Všimněte si, že díky lokální definici proměnné *a* ve funkci funkce() nelze použít ani globální proměnnou *a*, ani parametr *a*. [1)]

O něčem trochu jiném jsou tzv. **statické** proměnné. Statické proměnné se deklarují pomocí klíčového slova**static**. Používají se v těle funkcí a rozdíl oproti „obyčejným" proměnným je v tom, že se jejich hodnota po skončení průběhu funkce zachovává! Můžete tak díky ní třeba zaznamenávat kolikrát byla funkce spuštěna. Obyčejná, nestatická, proměnná se inicializuje při volání funkce vždy znova.

Další výhodou je, že při skončení funkce máte zajištěno, že statická proměnná stále existuje a tudíž ji (pomocí odkazu) můžete upravovat i mimo tělo funkce, zatímco paměť pro „obyčejnou" proměnnou může překladač v zájmu optimalizace programu využívat pro jiné účely – po skončení funkce jsou všechny nestatické proměnné zničeny! Statické proměnné mají v paměti stále stejné (statické) místo, cože je i jejich malá nevýhoda, protože se tím sníží možná optimalizace programu.

Nikdy nepoužívejte jako návratovou hodnotu funkce adresu na nestatickou lokální proměnnou. Platnost nestatické lokální proměnné končí s ukončením těla bloku, ve kterém byla definována. Naopak statická lokální proměnná se do konce běhu programu nezruší.

```
/*-----------------------------------------------*/
    /* c14/static.c                               */
                #include<stdio.h>

                char *funkce(void)
                        {
                static char pole[] = "Ahoj!";
                static int x = 1000;
                    int y = 1000;
                        x++;
                        y++;
                printf("Funkce je volana po %i (%i)\n", x, y);
                return pole;
                        }

                int main(void)
                        {
                    char *uk;
                printf("%s\n", funkce());

            uk = funkce(); /* pomoci ukazatele zmenim statickou promennou */
                uk[0] = 'C';
                uk[1] = 'u';
                uk[2] = 's';
               uk[3] = '\0';

                printf("%s\n", funkce());

                return 0;
                        }

/*-----------------------------------------------*/
```

Výstup z programu:

```
Funkce je volana po 1001 (1001)
Ahoj!
Funkce je volana po 1002 (1001)
Funkce je volana po 1003 (1001)
Cus
```

Statická proměnná se inicializuje jen jednou, ale pokud jí ve funkci přiřadíte hodnotu (mimo inicializaci), přiřadí se při každém volání funkce. Zkuste si místo řádku 8 napsat: `static int x; x = 1000;`

Statické funkce a globální proměnné

Klíčové slovo static můžete napsat i před globální proměnnou. Ta je ale statická už sama o sobě, ne? K čemu je to dobré?

Klíčové slovo static můžete napsat i před funkci. K čemu je to dobé?

Obojí má stejný význam, který je úplně jiný, než jsem psal o static výše. Takto označené funkce a globální proměnné jsou viditelné pouze v rámci svého zdrojového souboru. Z žádného jiného souboru, než ve kterém jsou definovány, se na ně nemůžete odkazovat.

Výhoda je jasná – můžete mít v různých zdrojových soborech stejně pojmenovanou funkci nebo globální proměnnou. Máte taky jistotu, že vám takovou globální proměnnou nezmění nikdo jiný, než jen funkce z vašeho souboru.

Zatím jsem se ještě nedostal k tomu, jak rozdělit zdroják do více souborů, takže vám tato informace momentálně není k užitku. Ale brzo bude :-)

Uživatelské knihovny funkcí

O využívání knihoven jsem již psal při popisování preprocesoru, takže je zde jenom připomenu.

Můžete si vytvářet knihovny funkcí (soubory mají nejčastěji příponou .h, pro jazyk C++ .hpp), v nichž budete mít napsány vlastní funkce, nebo definovaná makra. Při psaní velkého programu se tomu nevyhnete. V jednom souboru by měli být funkce, které spolu logicky souvisí. Například v jednom souboru funkce, které zpracovávají data (třeba něco počítají) a v jiném souboru funkce, která tato data zobrazují. Pokud pak takový program budete chtít přenést z textového režimu do grafického (třeba ve Windows), pak vám bude stačit změnit knihovnu, která zobrazuje výsledky (místo funkce printf() to bude nějaká funkce na zobrazování grafických dialogů atp.). To samé platí pro načítání vstupů atd. Vždy je dobré mít takovéto implementačně závislé funkce programu oddělené od těch nezávislých. Při přenosu programu z jednoho systém u na jiný se vám to tisíckrát vrátí.

V souboru .h můžete taky mít jenom deklarace funkcí a vlastní funkce můžete mít přeložené v nějaké knihovně funkcí (ve Windows jsou to soubory s příponou dll, v Linuxu s příponou so). Například stdio.h obsahuje deklarace funkcí, jejichž těla jsou v Linuxu přeloženy do systémové knihovny /lib/libc.so.6. Jak vytvářet takovéto knihovně budu popisovat až v části věnované programování v Linuxu (protože se to v každém OS dělá trochu jinak).

V ideálním případě by to mohlo vypadat tak, že máte např. soubor "main.c", ve kterém je definována funkce main(). Potom soubor "vypocty.h", který obsahuje všechny funkce pro výpočet a soubor "vystup1.h", který zobrazuje výsledek v textovém

režimu a "vystup2.h", který obsahuje funkce se stejnými jmény a argumenty jako "vystup1.h", ale zobrazuje výsledky v grafickém režimu. Potom stačí v souboru main.c přidat buďto "vystup1.h" nebo "vystup2.h" a podle toho určit chování programu. Výpočty nezávisí na tom, jak chcete mít výsledky vypisovány – jsou implementačně nezávislé.

V jednoduchém příkladu využiji podmíněného překladu. Následující program je implementačně závislý. Přeložíte jej v Linuxu, Windows i v DOSu (např. ve starém dosovském překladači Borland C, který obsahuje knihovnu <dos.h> s funkcí delay()).

Soubor "dos1.h": používá k pozastavení programu funkci delay() z knihovny <dos.h>.

```
/*-----------------------------------------------*/
  /* c14/dos1.h                                  */
#include <dos.h>

void cekej(unsigned int cas)
{
delay(cas);
}

/*-----------------------------------------------*/
```

Soubor "windows1.h": používá k pozastavení programu funkci Sleep() z knihovny .

```
/*-----------------------------------------------*/
  /* c14/windows1.h                              */
#include <windows.h>

void cekej(unsigned int cas)
{
Sleep(cas);
}

/*-----------------------------------------------*/
```

Soubor "unix1.h": používá k pozastavení programu funkci usleep() z knihovny <unistd.h>.

```
1.  /*-----------------------------------------------*/
2.    /* c14/unix1.h                                 */
3.  #include <unistd.h>
4.
5.  void cekej(unsigned int cas)
6.  {
7.  usleep((unsigned long) cas*1000);
8.  }
9.
10. /*-----------------------------------------------*/
```

V DOSu se k pozastavení programu používá funkce delay(), která je definována v knihovně <dos.h> a pozastaví program na zadaný počet milisekund, zatímco v Linuxu funkce usleep(), definována v knihovně <unistd.h>, pozastaví program na zadaný počet mikrosekund. Aby funkce cekej() dělala na obou OS to samé, linuxová verze svůj argument musí násobit 1000.

Funkce usleep má jako argument unsigned long, ale protože jsem chtěl funkci cekej() mít definovanou vždy stejně (s parametrem typu unsigned int), definoval jsem proměnnou cas jako unsigned int. Přetypování při použití s funkcí usleep() není nutné, překladač by to zvládnul i bez této nápovědy.

Soubor "cekej.c": všimněte si, kolik maker je třeba otestovat, abychom zjistili, zda jsme na Windows. Různé překladače totiž používají různá makra (nejsou nikde standardizována). Nejpravděpodobněji však budete mít definované makro __WINDOWS__ nebo __WIN32__ (a také __WIN16__ či __WIN64__ pro 16. a 64. bitové překladače – ty jsem v příkladu pro stručnost vynechal). Ostatní makra, bez dvou podtržítek na začátku a na konci, jsou již zastaralá.

```
*-----------------------------------------------*/
  /* c14/cekej.c                                 */
#include <stdio.h>

#if defined unix || defined __unix__
#include "unix1.h"
#define VERZE "UNIX"
#elif defined __MSDOS__
#define VERZE "MSDOS"
#include "dos1.h"
#elif defined __WINDOWS__ || defined _Windows || defined _WINDOWS || defined __WIN32__ || defined _WIN32 || defined
WIN32
#include "windows1.h"
#define VERZE "WINDOWS"
#endif

#define CEKEJ 100

int main(void)
{
int i, j;

for (i = 0; i <= 100; i++) {
printf("\rZdrzuji: [");
for (j = 0; j < 10; j++) {
printf("%c", (j * 10 <= i) ? '.' : ' ');
}
```

```
            printf("] %3i%%", i);
        fflush(stdout);       /* vyprazdnime standardni vystup */
        cekej(CEKEJ);         /* implementacne zavisla funkce */
                }
            printf(" OK\n");
                return 0;
                }

/*-----------------------------------------------*/
```

Výstup z funkce printf() se obvykle ukládá do tzv. vyrovnávací paměti, než se skutečně vypíše na obrazovku. Pomáhá to zrychlovat vykreslování (protože vykreslit dlouhý text naráz je jednodušší, než vykreslit znak po znaku). Problém ale je, že když funkce cekej() pozastaví provádění programu, může být výstup z pfunkce printf() ještě pořád v bufferu a ne na obrazovce. Takže by to nakonec vypadalo tak, že se program jenom na několik vteřin pozastavil a pak vypsal vše naráz. Abych takovému chování předešel, použil jsem v programu funkci fflush(), která vypíše na obrazovku ihned všechno, co je ve vyrovnávací paměti (vyprázdní vyrovnávací paměť).
Ještě připomínámi, že escape sekvence '\r' přesune kurzor na začátek řádku. Jak bude probíhat výstup z programu si domyslete, nebo, ještě lépe, vyzkoušejte. Vyzkoušejte si taky přeložit a spustit program bez volání fflush().

```
Zdrzuji: [.........] 100% OK
```

A ještě jedna poznámka k příkladu na závěr. Logičtější by bylo umístit makro VERZE do knihoven k funkci cekej(). Tyto knihovny může využívat i jiný program než jen cekej.c, tak by bylo výhodnější mít všechny implementačně závislé věci v knihovnách.

Vytváření typů

Zatím jste pracovali jen s typy, které byly definovány jazykem C (např. typ int, float atp.). V této kapitole se naučíte používat nástroje, pomocí nichž můžete vytvářet složitější datové typy a konstrukce. Přispěje to nejenom k zefektivnění psaní zdrojového kódu, ale také k jeho čitelnosti. Mimoto, takové vytváření struktur a vlastních datových typů může být i zábava :-).

Struktury

Struktura vám umožní spojit několik datových typů do jednoho a pracovat s nimi jako s celkem. Nejdříve se podívejte, jak strukturu definovat.

```
struct [jmeno] {
  typ jmeno_polozky;
  typ jmeno_polozky;
  ...
} [promenne];
```

Jméno struktury je nepovinné. Za definicí struktury mohou být hned definovány proměnné nového typu. Když struktuře nedáte jméno, stanou se jedinými proměnnými tohoto nového (nepojmenovaného) typu.
Podívejte se na příklad struktury polozka. S definicí struktury rovnou vytvořím dvě proměnné typu struct polozka.

```
struct polozka {
  unsigned int rok_narozeni;
  int pohlavi;
  char jmeno[20];
  char prijmeni[20];
} Martin, Pavla;
```

Struktura vždy začíná klíčovým slovem **struct**. Pak může, ale také nemusí, být definováno jméno struktury. Pokud není definováno jméno, není možné později vytvořit další proměnné této struktury, ani ji použít jako parametr funkce.
V těle struktury jsou deklarovány proměnné, které bude struktura obsahovat. Mohou to být i jiné struktury, nebo ukazatel na sebe sama. Struktura však nemůže obsahovat sebe samu (zkuste si představit, že jste překladač a chtěli byste něco takového interpretovat). Struktury nelze sčítat, odčítat atd. U struktur lze použít pouze operátor přiřazení =. Tím se obsah jedné struktury zkopíruje do jiné.
Jazyk C++ umožňuje tzv. přetěžování operátorů, díky čemuž pak můžete definovat sčítání, odčítání atp. vlastních datových typů. Jazyk C nic takového neumí.
Pro přístup ke členům struktury se používají operátory . (tečka) a -> (šipka). Druhý operátor se používá v případě, že pracujete s ukazatelem na strukturu.

```
struct polozka p, *ukazatel;
ukazatel = &p;
/* prirazeni 20 do casti struktury p pojmenovane rok_narozeni */
p.rok_narozeni = 1920;
/* stejne prirazeni, jen pres ukazatel */
(*ukazatel).rok_narozeni = 1940;
/* stejne prirazeni pres ukazatel, jen jiny (hezci) zapis */
ukazatel->rok_narozeni = 1960;
```

Použité závorky jsou nutné, aby bylo jasné že operátor dereference * (hvězdička) pracuje s proměnnou*ukazatel*, a ne s proměnnou *rok_narozeni* ve struktuře (což, mimochodem, není ukazatel, takže by to byla blbost). Přiřazení se šipkou je daleko elegantnější a čitelnější, proto používejte výhradně to.
Vytvořením struktury vlastně nevytváříte nový datový typ. Zkuste se dívat na vytvořenou strukturu podobně jako na vytvořené datové pole. Rozdíl je jen v tom, že datové pole obsahuje několik objektů stejného typu, zatímco struktura obsahuje různé datové typy. Jako není problém vytvořit pole obsahující jiné pole (vícerozměrná pole), není problém vytvořit pole obsahující strukturu.
Struktury se využívají hlavně ve velkých programech pracujících s velkým množstvím dat a také při vytváření datových typů (viz později v této kapitole typedef).

Následující příklad ilustruje možnosti použití struktury a také použití ukazatele typu void a aritmetiky ukazatelů. Prostě toho ukazuje hodně :-). Možná si budete muset přečíst komentář pod programem, než všechno správně pochopíte.

```
1.  /*----------------------------------------------*/
2.  /* c15/strukt.c                                 */
3.  #include <stdio.h>
4.  #include <stddef.h>
5.
6.  #define POCET 40
7.  /**
8.   * funkce zkopiruj kopiruje retezec v2 do retezce v1,
9.   * vcetne zarazky '\0' na konci retezce.
10.  * Predpoklada, ze pole v1 je nejmene tak dlouhe, jako
11.  * retezec v1.
12.  */
13. void zkopiruj(char *v1, char *v2)
14. {
15.  size_t i = 0;
16.  do {
17.  v1[i] = v2[i];
18.  } while (v2[i++]);
19.  }
20.
21.  struct Pisen {
22.  char nazev[POCET];
23.  char zpevak[POCET];
24.  char skladatel[POCET];
25.  float delka, hodnoceni;
26.  } pisne[10], *ukp;
27.
28.  void vytiskni(void *str);  /* deklarujeme funkci, ktera bude
29.  tisknout strukturu Pisen */
30.
31.  int main(void)
32.  {
33.  struct Pisen pisen;
34.
35.  zkopiruj(pisen.nazev, "Twist And Shout");
36.  zkopiruj(pisen.zpevak, "The Beatles");
37.  zkopiruj(pisen.skladatel, "Lennon/McCartney");
38.  pisen.delka = 2.36f;
39.  pisen.hodnoceni = 1.0;
40.
41.  pisne[0] = pisen;
42.
43.  zkopiruj(pisne[1].nazev, "Eleanor Rigby");
44.  zkopiruj(pisne[1].zpevak, "The Beatles");
45.  zkopiruj(pisne[1].skladatel, "Lennon/McCartney");
46.  pisne[1].delka = 2.06;
47.  pisne[1].hodnoceni = 1.5;
48.
49.  ukp = &pisne[2];          /* do ukazatele na strukturu Pisen ukladame adresu
50.  tretiho prvku v poli pisne */
51.  zkopiruj(ukp->nazev, "From Me To You");
52.  zkopiruj(ukp->zpevak, "The Beatles");
53.  zkopiruj(ukp->skladatel, "Lennon/McCartney");
54.  ukp->delka = 1.58f;
55.  ukp->hodnoceni = 1.25;
56.
57.  vytiskni(&pisne[0]);
58.  vytiskni(&pisne[1]);
59.  vytiskni(&pisne[2]);
60.
61.  return 0;
62.  }
63.
64.  void vytiskni(void *str)
65.  {
66.  char *sstr = (char *) str;
67.  /* sstr ukazuje na zacatek struktury */
68.  printf("Nazev pisne:\t%s\n", sstr);
69.
70.  sstr += POCET * sizeof(char);       /* delka pole "nazev" ve strukture */
71.  /* sstr ukazuje za pole "nazev" ve strukture, tj. na
72.  * zacatek pole "zpevak" */
73.  printf("Interperet:\t%s\n", sstr);
```

```
74.
75.  sstr += POCET * sizeof(char);
76.  printf("Skladatel(e):\t%s\n", sstr);
77.
78.  sstr += POCET * sizeof(char);
79.  printf("Delka skladby:\t%.2f\n", *((float *) sstr));
80.
81.  sstr += 1 * sizeof(float);
82.  printf("Hodnoceni:\t%.2f\n\n", *((float *) sstr));
83.  }
84.
85.  /*----------------------------------------------*/
```

*Ukazatel str se z typu void * přetypovává na něco „smysluplnějšího" – na char *.*

Výstup z programu vás asi nepřekvapí:

```
Nazev pisne:    Twist And Shout
Interperet:     The Beatles
Skladatel(e):   Lennon/McCartney
Delka skladby:  2.36
Hodnoceni:      1.00

Nazev pisne:    Eleanor Rigby
Interperet:     The Beatles
Skladatel(e):   Lennon/McCartney
Delka skladby:  2.06
Hodnoceni:      1.50

Nazev pisne:    From Me To You
Interperet:     The Beatles
Skladatel(e):   Lennon/McCartney
Delka skladby:  1.58
Hodnoceni:      1.25
```

Použití struktury, ukazatele na strukturu a operátorů . a -> je, myslím, poměrně jasné.

Podívejte se na funkci vytiskni(void *);, která tiskne strukturu Pisen.

Kdyby struktura Pisen neměla jméno Pisen (prostě byste jej na řádku 21 neuvedli), nešlo by ji použít jako parametr funkce, vytvořit její další instanci (jako na řádku 33) a dokonce ani vytvořit ukazatel na tuto strukturu dále v programu. Existovaly by pouze instance definované při definici struktury (na řádku 26).

Jelikož jsem chtěl ukázat, že i tak lze (pomocí ukazatelů) se strukturou poměrně slušně zacházet, použil jsem ve funkci ukazatel na nedefinovaný typ (void *). Při jeho používání jsem jej vždy nejdříve přetypoval na ukazatel potřebného typu pomocí (char *) nebo (float *). Zatímco textové řetězce ve funkci printf() se předávají právě pomocí ukazatelů, čísla typu float jsem musel předat jako hodnotu (proto ta hvězdička a závorky navíc na řádcích 78 a 81). Aby ukazatel ukazoval do správného místa v paměti, bylo nutné jej postupně posouvat po položkách struktury o příslušný počet bytů (vracených operátorem sizeof).

Programátorsky čistší prací by však bylo, kdybych funkci vytiskni() předával jako argument ukazatel nastruct Pisen, nebo ukazatel typu void přetypoval na ukazatel typu Pisen. Mohl bych i nepředávat žádný argument a používat globální ukazatel ukp, ale používání globálních proměnných většinou programy pouze znepřehledňuje.

Zkuste si za domácí úkol funkci vytiskni() takto přepsat.

Velkou nevýhodou funkce vytiskni() je také to, že při změně struktury Pisen (třeba přidání prvku doprostřed struktury) přestane správně fungovat, ale překladač nic nepozná.

Před názvem struktury je vždy nutné používat klíčové slůvko **struct**. Jak při deklaraci parametrů, tak při definici proměnných struktur.

```
void vytiskni(struct Pisen pisen);
```

Pokud předáváte strukturu jako argument, má to jednu velkou nevýhodu. Tato struktura se totiž celá zkopíruje z proměnné, kterou předáváte jako argument, kamsi do zásobníku v paměti počítače (do „argumentu funkce"). Pokud pracujete s velikými strukturami, pak takové kopírování velice zpomaluje chod programu. Mnohem efektivnější je používat ukazatele na strukturu. Pak se na zásobník volání zkopíruje jen ukazatel, který, jak víte, zabírá jen 16, 32 nebo 64 bitů.

Definování typu pomocí typedef

Pomocí **typedef** se vytvářejí uživatelské datové typy. Syntaxe příkazu je takováto:

```
typedef definice_typu identifikator;
```

Definicí typu může být například struktura, nebo nějaký základní typ jazyka C. **Identifikator** je pojmenování nového datového typu. Podívejte se na následující příklad:

```
1.  /*----------------------------------------------*/
2.  /* c15/typedef1.c                               */
3.  #include <stdio.h>
4.
5.  typedef int integer;
6.
7.  int main(void)
8.  {
9.  int a, b;
10. integer c, d;
11. a = 5;
12. b = a + 4;
13. printf("a = %i, b = %i\n", a, b);
```

```
14.
15.  d = (integer) a;
16.  c = d + 5;
17.  printf("d = %i, b = %i\n", (int) d, (int) c);
18.  return 0;
19.  }
20.
21.  /*----------------------------------------------*/
```

Všimněte si přetypování proměnné *a* při přiřazování její hodnoty do proměnné *d* na typ integer, a také přetypování proměnných *d* a *c* na typ int ve funkci printf(). Funkce printf() totiž nemůže očekávat typ integer, neboť jej norma jazyka C ani nezná. V příkladě je definice typu integer tak jednoduchá, že by se zřejmě dal přeložit i bez přetypování bez jakýchkoliv varování překladače.

Takovéto „přejmenovávání" datových typů je více méně k ničemu a přináší jenom problémy s čitelností programu, proto to raději nedělejte (a nehrajte si se sirkami).

Někdy se takové přejmenování typů ale velice hodí. Představte si, že píšete program, který bude fungovat na různých platformách, třeba na 16-bitové a 32-bitové. Na jedné potřebujete z nějakého důvodu použít datový typ, který má 16 bitů, na druhé 32 bitů. Jedním možným řešením je definovat si v hlavičkovém souboru pro 16-bitovou verzi typedef short int bitcislo; a pro 32-bitovou verzi zase typedef int bitcislo;. Zbytek zdrojového kódu už může pracovat s datovým typem bitcislo a nemusí se starat o to, kolik bitů zabírá. (Kdyby se o to přeci jen zajímal, může použít typedef).

Datový typ ze struktury

Teď se podívejte na **vytváření nového datového typu pomocí struktury**. Jak se to dělá, vidíte v příkladu. Také v něm vidíte, jak lze inicializovat struktury při jejich definici (podobně jako u polí).

```
1.   /*----------------------------------------------*/
2.   /* c15/typedef2.c                               */
3.   #include <stdio.h>
4.
5.   typedef struct pokus {
6.   int x;
7.   int y;
8.   char text[20];
9.   } Pokus;
10.
11.  int main(void)
12.  {
13.  struct pokus a = { 6, 6, "sest" };
14.  Pokus b = { 4, 5, "Ahoj" };
15.
16.  a.x = 0;
17.  b.x = 6;
18.
19.  printf("a.x = %i\n",a.x);
20.  /* nefunguje ve Visual Studiu */
21.  a = (struct pokus) b;
22.  printf("a.x = %i\n", a.x);
23.  return 0;
24.  }
25.
26.  /*----------------------------------------------*/
```

Všiměte si, že na řádku 9 není jméno nové proměnné, ale jméno nového datového typu.

Přetypování datového typu Pokus na struct pokus (viz řádek 21) s Visual Studiem nefunguje. Vyhodí chybu *'type cast': cannot convert from 'Pokus'to 'pokus'*. V normálním „životě" byste ale stejně měli používat buď jen struct pokus, nebo jen Pokus, abyste si neznepřehledňovali kód.

```
a.x = 0
a.x = 6
```

Když už jsem definoval datový typ Pokus, je trochu zbytečné používat strukturu pokus. Používání uživatelského typu je v mnohém příjemnější, než používání struktury jako takové (odpadá psaní slova struct), proto v dalším výkladu budu používat jen verzi s typedef.

Nyní následuje poslední příklad na vytváření datového typu pomocí struktury. Dobře si jej prostudujte.

```
1.   /*----------------------------------------------*/
2.   /* c15/typedef3.c                               */
3.   #include <stdio.h>
4.   #include <stddif.h>
5.
6.   /* funkce zkopiruj kopiruje retezec v2 do retezce v1 */
7.   void zkopiruj(char *v1, char *v2)
8.   {
9.   size_t i = 0;
10.  do {
11.  v1[i] = v2[i];
12.  } while (v2[i++]);
13.  }
14.
```

```
15.  typedef struct {
16.  char nazev[40];
17.  char zpevak[40];
18.  char skladatel[40];
19.  float delka;
20.  } Pisen;
21.
22.  typedef struct {
23.  char nazev[40];
24.  size_t pocet_pisni;
25.  float cenaSK, cenaCZ;
26.  Pisen pisen[20];
27.  } CD;
28.
29.  void tiskni(CD * cd, size_t pocet);
30.
31.  int main(void)
32.  {
33.  CD *ualbum, alba[10], album;
34.  /* inicializuji album */
35.  ualbum = &album; /* pouziti ukazatele ualbum
36.  je tu jen pro ilustraci */
37.  zkopiruj(album.nazev, "The Beatles Songs");
38.  album.cenaSK = 325.50;
39.  ualbum->cenaCZ = 286.50;
40.  album.pocet_pisni = 1;
41.  /* vkladam prvni (a posledni) pisen do alba */
42.  zkopiruj(album.pisen[0].nazev, "Twist And Shout");
43.  zkopiruj(ualbum->pisen[0].zpevak, "The Beatles");
44.  zkopiruj(album.pisen[0].skladatel, "Lennon/McCartney");
45.  ualbum->pisen[0].delka = 2.36;
46.
47.  /* ukazka prace s polem alba */
48.  /* do pole alba vkladam prvni album */
49.  alba[0] = album;
50.  /* to same album vlozim jeste jednou,
51.  tentokrat prez ukazatel */
52.  alba[1] = *ualbum;
53.  /* zmenim nazev a cenu ... */
54.  zkopiruj(alba[1].nazev, "Salute to The Beatles");
55.  alba[1].cenaSK = 350.0;
56.  /* ... a pridam dalsi pisen */
57.  alba[1].pocet_pisni = 2;
58.  alba[1].pisen[1] = alba[1].pisen[0];
59.  zkopiruj(alba[1].pisen[1].nazev, "Eleanor Rigby");
60.  alba[1].pisen[1].delka = 2.06;
61.
62.  tiskni(alba, 2);
63.  return 0;
64.  }
65.
66.  void tiskni(CD * cd, size_t pocet)
67.  {
68.  size_t i, j;
69.
70.  for (i = 0; i < pocet; i++) {
71.  /* tisknu album "i" */
72.  printf("%s\t%5.2fKc (%5.2fSk)\n", cd[i].nazev, cd[i].cenaCZ,
73.  cd[i].cenaSK);
74.  for (j = 0; j < cd[i].pocet_pisni; j++) {
75.  /* tisknu pisne z alba "i" */
76.  printf("\t\tNazev:\t%s\n", cd[i].pisen[j].nazev);
77.  printf("\t\tInterpret:\t%s\n", cd[i].pisen[j].zpevak);
78.  printf("\t\tSkladatel:\t%s\n", cd[i].pisen[j].skladatel);
79.  printf("\t\tDelka:\t%.2f\n", cd[i].pisen[j].delka);
80.  printf("\t\t.....\n");
81.  }
82.  printf("----------------------------------------\n");
83.  }
84.  }
85.
86.  /*---------------------------------------------*/
```

Výstup z programu:

```
The Beatles Songs       286.50Kc (325.50Sk)
```

```
Nazev:  Twist And Shout
Interpret:      The Beatles
Skladatel:      Lennon/McCartney
Delka:  2.36
.....
----------------------------------------
Salute to The Beatles   220.00Kc (350.00Sk)
Nazev:  Twist And Shout
Interpret:      The Beatles
Skladatel:      Lennon/McCartney
Delka:  2.36
.....
Nazev:  Eleanor Rigby
Interpret:      The Beatles
Skladatel:      Lennon/McCartney
Delka:  2.06
.....
----------------------------------------
```

Zkuste si napsat vlastní program, kde použijete vlastní datový typ jako argument funkce, nebo jako návratovou hodnotu funkce.
Něco jako:

```
typedef struct {
...
} struktura;
...
struktura nejaka_funkce(struktura a) {
...
}
...
struktura x,y;
...
x = nejaka_funkce(y);
```

Nebo ještě lépe, napište funkci, která bude mít jako argumenty položky ze struktury nového datového typu, a bude vracet onen datový typ vyplněný těmito argumenty.
Na závěr dodávám, že ani strukturované typy dat vytvořené pomocí typedef nelze sčítat, odčítat atp. Až v jazyce C++ je možné přetěžovat operátory a tak definovat sčítání struktur. V jazyku C si musíte vystačit s funkcemi, které mohou dělat něco jako sčítání. Co budou dělat je čistě na vás, programátorech. Jo jo, už vám říkám programátoři :-).

Vytváření typů II

V této kapitole vám povím o dalších podivuhodných konstrukcích jazyka C. Zde probírané konstrukce patří již k těm pokročilejším a jejich využití není tak časté, ale přesto se vyplatí je znát. Zvláště výčtový typ enum.

Výčtový typ enum

Typ **enum** vám umožní vytvořit proměnnou, která může obsahovat pouze hodnoty konstant určených při deklaraci **výčtového typu**. Proměnná se vytváří pomocí typedef.
Deklarace typu enum má hodně společného s deklarací struktury. Lze vytvořit buďto jen výčtový typ enum (stejně jako jenom strukturu), nebo pomocí typedef definovat nový datový typ. Protože je lepší a častější použití s typedef, ukáži jenom jednou syntaxi výčtového typu a dále v příkladech budu pracovat jen spolu s konstrukcí s typedef.

```
enum [jmeno] {
výčet konstant
...
} [promenne];
```

Příklad vytvoření výčtového typu:

```
enum kalendar {
leden = 1, unor, brezen, duben, kveten, cerven, cervenec, srpen, zari,
rijen, listopad, prosinec
} a, b;
...
a = unor;
...
if (b == duben) {
...
```

V příkladu jsem vytvořil výčtový typ se jménem kalendar a definoval dvě proměnné *a* a *b*. **Konstanty výčtového typu jsou vždy celočíselné**. Pokud konstantě nepřiřadíte hodnotu (jako leden=1), pak má hodnotu o jednotku vyšší, než konstanta předešlá (*unor* tedy odpovídá číslu 2, *brezen* 3 atd). Pokud nepřiřadíte hodnotu ani první konstantě, automaticky je jí přiřazena nula. Hodnoty lze přiřadit kterékoliv konstantě ve výčtovém typu. Opět platí, že následující konstanta, pokud nemá přiřazenou hodnotu, je o jednotku větší, než předcházející konstanta.
Překladač tyto konstanty chápe pouze jako čísla. Je tedy dost dobře možné je srovnávat s čísly (např if(leden == 1)), ale to by zcela postrádalo smysl. Výčtový typ se používá právě proto, aby se v programu nemusela konkrétní čísla používat. Použití výčtového typu se podobá makrům preprocesoru. Jeho výhodou je právě to, že lze vytvářet proměnné výčtového typu, čímž se dá program **zpřehlednit** a jeho čtení je pak o něco snazší. Taktéž překladač může odhalit všelijaké chyby (třeba když do proměnné výčtového typu přiřadíte číslo, které neodpovídá žádné konstantě z výčtového typu.

V následujícím příkladu je již vidět použití typu enum s konstrukcí typedef.

```
/*----------------------------------------------*/
/* c16/enum1.c                                  */

#include <stdio.h>

typedef enum {
vlevo, vpravo, stred, center = stred
} zarovnani;


void tiskni_znak(char ch, zarovnani zr)
{
switch (zr) {
case vlevo:
printf("%c\n", ch);
break;
case vpravo:
printf("%50c\n", ch);
break;
case stred:
printf("%25c\n", ch);
break;
default:
printf("%c%24c%25c\n", '?', '?', '?');
}
}

int main(void)
{
tiskni_znak('A', 50);
tiskni_znak('X', vlevo);
tiskni_znak('Y', center);
tiskni_znak('Z', vpravo);
return 0;
}

/*----------------------------------------------*/
```

Výstup z programu:

```
                                                  ?
                                                  ?
                                                  X
                                                  Y
                                                  Z
```

Typ union

Syntaxe typu union je stejná jako struktury, až na to, že místo klíčového slova struct se použije klíčové slovo **union**. Význam jednotlivých položek je stejný. Jeho použití s konstrukcí typedef také.

```
union [jmeno] {
typ jmeno_polozky;
typ jmeno_polozky;
...
} [promenne];
```

Rozdíl tu však je. A zásadní. Zatímco struktura si vytvoří paměťové místo pro všechny položky, typ **union zabírá v paměti jen tolik místa, kolik největší jeho položka**. Z toho také vyplývá, že lze používat v jeden okamžik jen jednu položku. Kterou, to už závisí na programátorovi. Každá položka začíná v začátku paměti unionu.

Typ union může při programování ušetřit paměť. Navíc můžeme vytvořit funkci, která bude vracet různé datové typy jako typ union (viz příklad níže). Prvkem v union může být i struktura, nebo jiný typ union atp.

Union se dá inicializovat takto (inicializuje se samozřejmě jen jedna položka):

```
union datovy_typ {
char ch;
unsigned int uin;
float fl;
}
union datovy_typ cislo1 = { 'a' }; /* inicializuje ch */
union datovy_typ cislo2 = { uin: 5 }; /* inicializuje uin */
union datovy_typ cislo3 = { .uin = 5 }; /* inicializuje uin */
```

Funkce rand() a srand()

V příkladu vás seznámím, kromě použití výčtového typu **enum** a typu **union**, se dvěma novými funkcemi.

Funkce **rand()** vrací **pseudonáhodné číslo** v rozmezí 0 až RAND_MAX. Pseudonáhodné proto, protože vrací čísla z číselné řady generované na základě nějakých matematických algoritmů.

Aby tato čísla nebyla při spuštění programu generována vždy stejně, nastaví se počátek této řady pomocí funkce **srand()**. Argumentem této funkce je číslo typu unsigned int. Dobré je předat této funkci jako argument aktuální čas. Protože pracovat s

časem ještě neumíte, nastavím jej podle toho, co zadá uživatel programu. Budete-li jako uživatel zadávat stále stejné číslo, bude program vracet stále stejné výsledky. Příklad použití srand() s časem najdete až v kapitole věnující se standardnímu hlavičkovému souboru <time.h>.

Funkce rand() a srand() jsou definovány ve standardním hlavičkovém souboru <stdlib.h>.

```
1.  /*----------------------------------------------*/
2.  /* c16/union.c                                  */
3.  #define _CRT_SECURE_NO_WARNINGS
4.  #include <stdio.h>
5.  #include <stdlib.h>
6.
7.  typedef enum {
8.  znak, cele, racionalni
9.  } typ;
10.
11. typedef union {
12. char ch;
13. unsigned int uin;
14. float fl;
15. } datovy_typ;
16.
17. datovy_typ nahodne_cislo(typ t)
18. {
19. datovy_typ x;
20.
21. switch (t) {
22. case znak:
23. x.ch = (char) (rand() % ('z' - 'a')) + 'a';
24. break;
25. case cele:
26. x.uin = (unsigned int) rand();
27. break;
28. case racionalni:
29. x.fl = (float) rand() / RAND_MAX;
30. break;
31. }
32. return x;
33. }
34.
35. int main(void)
36. {
37. datovy_typ un;
38.
39. printf("Zadej cislo: ");
40. if (scanf("%u", &un.uin) == 0)
41. return 1;
42.
43. srand(un.uin);
44. un = nahodne_cislo(racionalni);
45. printf("float = %f\n", un.fl);
46.
47. un = nahodne_cislo(znak);
48. printf("char = %c\n", un.ch);
49.
50. un = nahodne_cislo(cele);
51. printf("int = %i\n", un.uin);
52.
53. printf("velikost un = %lu == float = %lu bajty\n", sizeof(un), sizeof(float));
54. return 0;
55. }
56.
57. /*----------------------------------------------*/
```

Makro _CRT_SECURE_NO_WARNINGS je tu kvůli funkci scanf(), viz scanf().

Možný výstup z programu:

```
Zadej cislo: 123
float = 0.060051
char = n
int = 436085873
velikost un = 4 == float = 4 bajty
```

Typ union se moc často nepoužívá. Šetření pamětí už není dneska in a přednost se dává psaní srozumitelnějších programů. Lepší řešení předchozího příkladu by bylo napsat 3 funkce pro získání náhodného čísla (pro znak, unsigned int a float). Nepotřebovali byste ani enum, ani union a měli byste 3 krátké, čitelné funkce místo jedné dlouhé nepřehledné.

Typ union se s výhodou používá také jako argument funkcí, které potřebují pro různé situace různé vstupní datové typy, ale vždy právě jen jeden z nich. Ale i pro tyto situace platí předchozí poznámka - raději se unionu vyhýbejte, pokud k němu nemáte opravdu dobrý důvod.

Ukazatele na funkce

V této kapitole zakončím výklad syntaxe jazyka C (no fakt, už je koneec! :D). Vysvětlím vám ukazatele na funkce a pak si zopakujete základní deklarace proměnných a funkcí. V jazyce C existují ještě další konstrukce, které jsem nevysvětloval (například jak použít assembler), ale troufám si tvrdit, že s většinou potřebných konstrukcí jsem vás již seznámil. Příklad na konci bude shrnovat většinu probrané látky. Ale ještě neodcházejte. Výklad jazyka C tím zdaleka nekončí. Jazyk C obsahuje řadu standardních knihoven, jejichž znalost je pro programování v C zásadní. V dalším výkladu se s nimi budete seznamovat.

- Ukazatele na funkce
- Přehled známých deklarací typů
- Opakování – příklad

Ukazatele na funkce

Ukazatele na funkce jsem si nechal schválně až na konec, protože snažit se pochopit jejich zápis je opravdu sado maso. Doufám, že už dobře zvládáte samotné ukazatele a také ukazatele na ukazatele.

Protože i funkce programu leží kdesi v paměti, existuje adresa, která ukazuje na začátek této funkce. Nyní se konečně dozvíte, proč musí být za jménem funkce ve zdrojovém kódu kulaté závorky, i když nečeká funkce žádné argumenty. Je to proto, že samotné jméno funkce (bez kulatých závorek) je překladačem chápáno jako adresa funkce, tedy nějaké číslo! Takové číslo lze i vytisknout, ale to by asi k ničemu nebylo. Pomocí ukazatele na funkci lze funkci zavolat (a to i s příslušnými argumenty). Můžete tak kupříkladu napsat funkci, která bude mít jako argument ukazatel na jinou funkci, kterou pak může spustit. Podívejte se, jak vypadá deklarace ukazatele na funkci. Jde o deklaraci ukazatele (jmeno) na funkci, která má návratovou hodnotu typu typ. V závorce mohou být uvedené typy očekávaných argumentů funkce.

```
typ (*jmeno)([typ_arg,..]);
```

Kulaté závorky kolem jména funkce a hvězdičky jsou **nutné**, jinak by to vypadalo jen jako deklarace funkce, která má jako návratový typ typ *.

V příkladu níže deklaruji ukazatel *uknf*, který ukazuje na funkci bez argumentů s návratovou hodnotou „ukazatel na typ char“. Pro srovnání máte hned za touto deklarací deklaraci oné funkce. Všimněte si, že **závorky při deklaraci ukazatele na funkci jsou vždy nevyhnutelné**. Kdybych je v tomto příkladu nepoužil, vytvořil bych deklaraci funkce s návratovým typem „ukazatel na ukazatel na typ char“.

```
/* deklarace ukazatele na funkci, která vrací ukazatel */
char *(*uknf) (void);
/* deklarace funkce, která vracejí ukazatel */
char *uknf (void);
```

V příkladu použiji novou funkci exit(), která ukončí program v jakémkoliv místě, s návratovou hodnotou, která je argumentem funkce exit().

Tato funkce je definována v souboru <stdlib.h>. Tam se o ní dočtete více.

```
/*-------------------------------------------------*/
/* c17/ukafce.c                                    */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

#ifdef _MSC_VER
  #define SZU "lu"
    #else
  #define SZU "zu"
    #endif

/* nejdrive vytvorim strukturu, ktera bude obsahovat pojmenovani
 * funkce pro uzivatele programu, cislo funkce, bude zaznamenavat
 * pocet spusteni funkce a bude obsahovat ukazatel na funkci */
typedef struct {
  unsigned int cislo;
  char nazev[50];
unsigned int spusteno;
void (*ukfce) (unsigned int *, char *);
} robot;

/* nyni definuji funkce programu, ktere budou uzivateli nabizeny
 * za pomoci predchozi struktury */
/**
* zobrazi pouze velka pismena ASCII tabulky
*/
void velka_pismena(unsigned int *spusteno, char *v)
{
  size_t i = 0;
(*spusteno)++;
  do {
if ((v[i] >= 'A') && (v[i] <= 'Z')) {
  printf("%c", v[i]);
}
} while (v[i++]);
  printf("\n");
```

```
}

/**
 * zobrazi pouze mala pismena ASCII tabulky
 */
void mala_pismena(unsigned int *spusteno, char *v)
{
    size_t i = 0;
    (*spusteno)++;
    do {
        if ((v[i] >= 'a') && (v[i] <= 'z')) {
            printf("%c", v[i]);
        }
    } while (v[i++]);
    printf("\n");
}

/**
 * zobrazi vse, co neni pismeno
 */
void nepismena(unsigned int *spusteno, char *v)
{
    size_t i = 0;
    (*spusteno)++;
    do {
        if (!(((v[i] >= 'a') && (v[i] <= 'z')) ||
            ((v[i] >= 'A') && (v[i] <= 'Z')))) {
            printf("%c", v[i]);
        }
    } while (v[i++]);
    printf("\n");
}

/**
 * zobrazi text pozpatku
 */
void obrat_text(unsigned int *spusteno, char *v)
{
    size_t i = 0;
    if (*spusteno >= 2) {
        printf("Pro dalsi pouziti teto funkce se musite registrovat!\n");
        return;
    }
    (*spusteno)++;

    while (v[++i]);             /* hledam konec retezce */

    for (; i > 0; i--)
        printf("%c", v[i-1]);

    printf("\n");
}

/**
 * vlozi do textu mezery
 */
void roztahni(unsigned int *spusteno, char *v)
{
    size_t i = 0;
    (*spusteno)++;
    while (v[i]) {
        printf("%c ", v[i++]);
    };
    printf("\n");
}
/**
 * Funkce exit() ma jine parametry, nez ktere ocekava nas ukazatel,
 * proto jsem vytvoril funkci konec, ktera simuluje potrebu
 * deklarovanych argumentu
 */
void konec(unsigned int *spusteno, char *v)
{
    /* ukonci program s navratovou hodnotou 0, ktera znaci uspech */
    exit(0);
}
```

```c
/**
 * funkce vlozim do pole struktury robot
 * spolu s nazvem pro uzivatele atd.
 */
robot *inicializuj(void)
{
/* pokud by promenna "r" nebyla staticka, pak by po skonceni
   volani funkce prestala existovat! */
static robot r[] = {
  {1, "Velke pismena",   0, velka_pismena},
  {1, "Mala pismena",    0, mala_pismena},
  {2, "Co neni pismeno", 0, nepismena},
  {3, "Obraceni textu",  0, obrat_text},
  {4, "Roztazeni textu", 0, roztahni},
  {5, "Ukonceni",        0, konec},
{0, "", 0, NULL} /* "zarazka", podle ktere poznam,
                    ze jsem na konci pole */
  };
return r;
  }

int main(void)
{
size_t i; int navrat;
/** ukazatel na pole struktury robot */
robot *rb;
  /**
 * funkce jsou ve strukture ocislovany,
 *  dle tohoto cisla budou vybirany */
size_t cislo;
/* maximalne 50 znaku + '\0' */
char retezec[51];

/* funkce inicializuj() vraci ukazatel na svou
   statickou promennou. At ji zavolate kolikrat
   chcete, bude ukazovat stale na to same pole.
O tom, jak vytvaret nove promenne za behu programu
   bude rec v dalsi kaptile pozdeji. */
rb = inicializuj();
  do {
   i = 0;
    do {
/* zobrazeni menu - snadne a prehledne */
printf("%2i)\t%s\t(%2i)\n", rb[i].cislo, rb[i].nazev, rb[i].spusteno);
} while (rb[++i].ukfce != NULL);

printf("Zadejte cislo z menu a retezec: ");

/* vysledek prirazeni muzeme pouzit jako hodnotu
   (tak ho rovnou porovname s EOF) */
if ((navrat = scanf("%" SZU " %50s", &cislo, retezec)) == EOF)
    break;
else if (navrat != 2) { /* chybne nacteni polozek */
  printf("\n Chyba vstupu (%i)!\n", navrat);
/* Zrejme nebylo zadano jako prvni cislo, ale nejaky retezec.
 * Ten se musi nyni nacist, jinak by se jej predchozi funkce
 * scanf pokousela v cyklu neustale nacitat jako cislo a tim
 * by cyklus nikdy neskoncil.
* Promenna retezec muze obsahnout maximalne 50 znaku, proto
 * funkci scanf v prvnim argumentu urcime max. delku
 * nacitaneho retezce. */
scanf("%50s", retezec); // nacteme "smeti"
  continue;
  }

  i = 0;
/* hleda se sruktura uzivatelem zadaneho cisla */
  while (rb[i].ukfce != NULL) {
    if (rb[i].cislo == cislo) {

/* VOLANI FUNKCE PRES UKAZATEL */
  rb[i].ukfce(&rb[i].spusteno, retezec);
```

```
            /* prochazi se cele pole, takze se spusti vsechny polozky v
             * poli, ktere maji rb[].cislo == cislo */
            /* kdyby zde byl prikaz break, provedla by se prvni nalezena
             * polozka a dale by se jiz nic neprohledavalo */
                }
                i++;
            }
        } while (1); /* nekonecny cyklus, ukonci se jen pomoci return nebo
                        break (nebo exit ukonci program) */

        printf("\nAstalavista baby!\n");
        return 0;
    }

/*----------------------------------------------*/
```

Makro _CRT_SECURE_NO_WARNINGS je tu kvůli funkci scanf(), viz scanf().
Důvod definování SZU viz datový typ pro ukazatel.
Možný výstup z programu:

```
1)   Velke pismena   ( 0)
1)   Mala pismena    ( 0)
2)   Co neni pismeno ( 0)
3)   Obraceni textu  ( 0)
4)   Roztazeni textu ( 0)
5)   Ukonceni        ( 0)
Vyber polozku dle cisla a zadejte retezec: 1 VelkaAMalaPismena
VAMP
elkaalaismena
1)   Velke pismena   ( 1)
1)   Mala pismena    ( 1)
2)   Co neni pismeno ( 0)
3)   Obraceni textu  ( 0)
4)   Roztazeni textu ( 0)
5)   Ukonceni        ( 0)
Vyber polozku dle cisla a zadejte retezec: 5
konec
```

Program má jeden drobný nedostatek, a to, že při volání funkce k ukončení programu (je pod číslem 5) musíte zadat za číslo 5 ještě nějaký zbytečný řetězec. Určitě vás napadne spousta možností, jak se tohoto nedostatku zbavit.
Důležitější je, jak snadno se do takového programu přidá další funkce. Stačí jí jen napsat a ve funkciinicializuj() ji přidat do pole struktury *robot*. Nic víc se na programu měnit nemusí.
Ukazatel na funkci se může s výhodou použít v programech, které načítají funkce z knihoven. Při vytvoření funkce stačí jen změnit nebo vytvořit novou knihovnu pro program (a ne hned překládat celý program). Program si jí načte do paměti a pak s ní může snadno pracovat pomocí ukazatele, jako by ji měl odedávna. (Vytváření knihoven se budu věnovat až v části o programování v Linuxu.)

Přehled známých deklarací

Zde se můžete v přehledu podívat na deklarace, kterým byste měli rozumět. (Zopakujte si alespoň Ukazatele a pole.)

Deklarace základních typů, ukazatelů a funkcí

Deklarace	Význam
typ jmeno;	Proměnná určeného typu
typ *jmeno;	Ukazatel na určitý typ (respektive pole daného typu)
typ jmeno[];	Konstantní ukazatel na pole daného typu (neznámo jak dlouhé)
typ jmeno[10];	Konstantní ukazatel na pole daného typu o velikosti 10-ti proměnných daného typu.
typ **jmeno;	Ukazatel na ukazatel na daný typ
typ *jmeno[];	Konstantní ukazatel na pole ukazatelů daného typu
typ jmeno[][];	Konstantní ukazatel na pole konstantních ukazatelů na pole daného typu.
typ jmeno();	Deklarace funkce vracející typ
typ *(jmeno());	Funkce vracející ukazatel na typ. Zvýrazněné závorky jsou nadbytečné.
typ (*jmeno)()	Ukazatel na funkci bez parametrů vracející typ

Deklarace	Význam
typ *(*jmeno)()	Ukazatel na funkci bez parametrů vracející ukazatel na typ

Lze vytvářet i daleko složitější deklarace. Například:
`unsigned long *(*jmeno[5][4])(char *);`
Toto je dvojrozměrné pole obsahující ukazatele na funkci, jejíž návratová hodnota je ukazatel na typ unsigned long a argumentem této funkce je ukazatel na typ char. Sranda, ne? :D
Rozluštit takovéto zápisy není zrovna legrace. Proto doporučuji využívat typedef pro zjednodušení takovýchto konstrukcí.
Předchozí proměnnou jmeno lze definovat takto:
```
typedef unsigned long *(*ukazatel_na_fci1)(char *);
ukazatel_na_fci1 jmeno[5][4];
```
To už je určitě daleko čitelnější. A myslím, že vám to pomůže i lépe pochopit předchozí zápis.
Pokud si nebudete někdy nějakým zápisem jistí, můžete využít program cdecl:

```
$ cdecl
Type `help' or `?' for help

cdecl> explain unsigned long *(*jmeno[5][4])(char *);
declare jmeno as array 5 of array 4 of pointer to function (pointer to char) returning pointer to unsigned long
```

Opakování – příklad

Je na čase si zopakovat probranou látku. Program, který je zde popsán obsahuje většinu probrané látky. Měli byste být schopni nejenom takovýto zdrojový kód přečíst a pochopit, ale i sami napsat. Programování se naučíte nejlépe tím, že si budete vymýšlet vlastní příklady (stále těžší a těžší) a programovat je. Snad sem vás dokázal během výuky dostatečně inspirovat.
Naprogramujeme si chůzi opilce. Program si přečte z příkazové řádky pravděpodobnosti, s jakými půjde opilec vpřed a vzad. Pravděpodobnost, že zůstane stát si dopočte program sám (zbytek do 100%).
Jelikož na příkazové řádce jsou i čísla chápána jako text, použiji funkci atoi(), ze standardní knihovny, na převod řetězce na číslo typu int. Po spuštění programu bude mít navíc uživatel možnost vybrat si z několika zobrazení opilce. Jakým způsobem se bude opilec zobrazovat se pokusím naprogramovat tak, jako by šlo o*modul* programu. Proto funkce pro zobrazování opilce budou v jiném souboru, než hlavní program a budou se volat pomocí ukazatele na funkce.
Začnu třemi hlavičkovými soubory: "dos1.h", "windows1.h" a "unix1.h". Tyto soubory jsem již použil v kapitole o uživatelských knihovnách. Jsou v nich deklarovány funkce implementačně závislé.

```
/*----------------------------------------------*/
  /* c17/dos1.h                                 */
#include <dos.h>

void cekej(unsigned int cas)
{
delay(cas);
}

/*----------------------------------------------*/
```

```
/*----------------------------------------------*/
 /* c17/windows1.h                              */
#include <windows.h>

void cekej(unsigned int cas)
{
Sleep(cas);
}

/*----------------------------------------------*/
```

```
/*----------------------------------------------*/
  /* c17/unix1.h                                */
#include <unistd.h>

void cekej(unsigned int cas)
{
usleep((unsigned long) cas*1000);
}

/*----------------------------------------------*/
```

V souboru "define1.h" definuji potřebné datové typy a makra. Tyto datové typy a makra se budou používat i v dalších knihovnách, proto je třeba zajistit, aby se nenačítal obsah tohoto soboru vícekrát (pomocí makra _DEFINE1_H, viz zdrojový kód).
Výčtový typ smer bude sloužit k určování směru chůze opilce, struktura pohyb bude obsahovat pravděpodobnosti, s jakou půjde opilec vpřed, vzad nebo zůstane stát a také pozici, kde opilec právě stojí. Všimněte si, že položka *stop* nebude ani využita. Lze jí dopočítat do 100% pomocí položek *vpred* a *vzad*.
Je otázkou, zda je lepší vytvořit takovou položku, do které se hodnota jednou uloží a pak už se jen používá, nebo kdykoliv je v programu potřeba, tak se vypočte výrazem (100-vpred-vzad).
Vzhledem k tomu, že by se tento výraz musel v programu vícekrát počítat (což program zpomaluje) a i tento výraz zabírá v

programu místo (instrukce pro výpočet), je asi výhodnější položku *stop* použít.
V tomto příkladě se však hodnota *stop* nepoužije vůbec, takže je opravdu zbytečná.

```
/*------------------------------------------------*/
/* c17/define1.h                                  */

#ifndef _DEFINE1_H
#define _DEFINE1_H 1

#define DELKACHUZE 40

typedef enum {
dopredu, dozadu, stat
} smer;

typedef struct {
int vpred, vzad, stop;
int pozice;
} pohyb;

#endif
/*------------------------------------------------*/
```

V souboru "zobraz1.h"[1] jsou definovány všechny funkce, kterými se zobrazuje pohyb opilce. (Všimněte si, jak makra TISKNIOPILCE pokračují za zpětným lomítkem na druhé řádce.)
Ukazatele na tyto funkce jsem uložil do pole zobraz. To, která funkce se bude volat, nechám v programu na náhodě. Některé funkce jsem tam tak vložil záměrně vícekrát, aby byla větší pravděpodobnost, že budou vybrány. Naproti tomu jsem tam jednu funkci nevložil vůbec.

```
/*------------------------------------------------*/
/* c17/zobraz1.h                                  */

#include <stdio.h>
#include <stdlib.h>
#include "define1.h"

#define TISKNIOPILCER(ZNK) printf("\r%3i%% (%*s%*c %3i%% (%3i)",p->vzad,\
p->pozice, ZNK, DELKACHUZE - p->pozice,')',p->vpred,pocet);
#define TISKNIOPILCEN(ZNK) printf("\n%3i%% (%*s%*c %3i%% (%3i)",p->vzad,\
p->pozice, ZNK, DELKACHUZE - p->pozice,')',p->vpred,pocet);

void zobraz_opilce1(pohyb *, const smer, const int);
void zobraz_opilce2(pohyb *, const smer, const int);
void zobraz_opilce3(pohyb *, const smer, const int);
void zobraz_opilce4(pohyb *, const smer, const int);

void (*zobraz[]) (pohyb *, const smer, const int) = {
zobraz_opilce1, zobraz_opilce2, zobraz_opilce3,
zobraz_opilce1, zobraz_opilce3
};

void zobraz_opilce1(pohyb * p, const smer s, const int pocet)
{
switch (s) {
case dopredu:
p->pozice++;
TISKNIOPILCER("->")
break;
case dozadu:
p->pozice--;
TISKNIOPILCER("<-")
break;
case stat:
TISKNIOPILCER("<>")
break;
};
}

void zobraz_opilce2(pohyb * p, const smer s, const int pocet)
{
switch (s) {
case dopredu:
p->pozice++;
TISKNIOPILCEN("->")
break;
case dozadu:
p->pozice--;
TISKNIOPILCEN("<-")
```

```
            break;
        default:
    TISKNIOPILCEN("<>")
            break;
        };
    }

void zobraz_opilce3(pohyb * p, const smer s, const int pocet)
{
    switch (s) {
    case dopredu:
        p->pozice++;
    TISKNIOPILCER(">>")
        break;
    case dozadu:
        p->pozice--;
    TISKNIOPILCER("<<")
        break;
    default:
    TISKNIOPILCER("><")
        break;
    };
}

void zobraz_opilce4(pohyb * p, const smer s, const int pocet)
{
    switch (s) {
    case dopredu:
        p->pozice++;
    printf("\rPOZICE: %2i%*c", p->pozice++, DELKACHUZE+10, ' ');
        break;
    case dozadu:
        p->pozice--;
    printf("\rPOZICE: %2i%*c", p->pozice++, DELKACHUZE+10, ' ');
        break;
    default:
    printf("\rPOZICE: %2i%*c", p->pozice++, DELKACHUZE+10, ' ');
        break;
    };
}

/*-------------------------------------------------*/
```

A konečně k srdci programu, souboru opakov1.c.

```
/*-------------------------------------------------*/
/* c17/opakov1.c                                   */

#include <stdio.h>
#include <stdlib.h>

#ifdef unix
 #include "unix1.h"
 #define VERZE "UNIX"
#elif defined __MSDOS__
 #define VERZE "MSDOS"
 #include "dos1.h"
#elif defined __WINDOWS__ || defined __WIN16__ || defined __WIN32__ || defined __WIN64__ || defined _MSC_VER
 #include "windows1.h"
 #define VERZE "WINDOWS"
 #endif

#define CEKEJ 100

#include "define1.h"
#include "zobraz1.h"

/* Pokusim se nacist cisla z prikazove radky a pote zkontroluji,
 * zda maji rozumne hodnoty. Pokud ne, program ukoncim */

pohyb nacti_procenta(int argc, char *argv[])
{
 pohyb p = { 0, 0, 0, DELKACHUZE / 2 };
 if (argc != 3) {
  printf("Spatny pocet argumentu\n"
"Zadejte pravdepodobnost chuze vpred a vzad.\n");
  exit(0);
```

```
33.  }
34.  p.vpred = atoi(argv[1]);
35.  p.vzad = atoi(argv[2]);
36.  p.stop = 100 - p.vpred - p.vzad;
37.
38.  if (p.stop < 0) {
39.  printf("Chybne zadane pravdepodobnosti\n");
40.  printf("Jejich soucet nesmi prekrocit 100\n");
41.  exit(0);
42.  }
43.
44.  if (!(p.vpred || p.vzad)) {
45.  printf("Pravdepodobnosti vpred i vzad musi byt nenulove\n");
46.  exit(0);
47.  }
48.  if ((p.vpred < 0) || (p.vzad < 0)) {
49.  printf("Pravdepodobnosti musi byt kladne!\n");
50.  exit(0);
51.  }
52.  return p;
53.  }
54.
55.  /**
56.  * nahodne vybere smer pohybu opilce
57.  */
58.  smer vyber_smer(const pohyb * p)
59.  {
60.  int x;
61.  x = (rand() % 100);        /* tj. 0 - 99 */
62.  if (x <= p->vpred)
63.  return dopredu;
64.  if (x <= p->vpred + p->vzad)
65.  return dozadu;
66.  return stat;
67.  }
68.
69.
70.  int main(int argc, char *argv[])
71.  {
72.  pohyb opilec;
73.  int pocet_funkci, fce, pocet;
74.  opilec = nacti_procenta(argc, argv);
75.  /* Diky tomu, ze nahodne cisla inicializuji "nenahodne", pak
76.  * pri stejne zadanych procentech se bude program chovat stejne.
77.  * Lepe je funkcisrand inicializovat treba na zaklade aktualniho
78.  * casu */
79.  srand(opilec.vpred * opilec.vzad);
80.  /* NULL je ukazatel "do nikam". Jako takovy ma stejnou velikost
81.  * jako kterykoliv ukazatel, vcetne ukazatele na funkci.
82.  * Tj. sizeof(NULL) = sizeof(char *) atd. */
83.  pocet_funkci = sizeof(zobraz) / sizeof(NULL);
84.
85.  pocet = 0;
86.  do {
87.  pocet++;
88.  /* vybiram nahodne funkci na zobrazeni */
89.  fce = rand() % pocet_funkci;
90.  zobraz[fce] (&opilec, vyber_smer(&opilec), pocet);
91.  /* vyprazdneni standardniho vystupu pred pozastavenim */
92.  fflush(stdout);
93.  cekej(CEKEJ);
94.  } while ((opilec.pozice > 2) && (opilec.pozice < DELKACHUZE-1));
95.
96.  printf("\n");
97.  return 0;
98.  }
99.
100.  /*-----------------------------------------------*/
```

Možný průběh programu:

```
$ opakov1 30 50
50% (          <<                ) 30% (  8)
50% (           >>               ) 30% ( 17)
50% (          <>                ) 30% ( 24)
50% (         <<                 ) 30% ( 27)
```

```
 50% (          ->                ) 30% ( 28)
 50% (          <<                ) 30% ( 38)
 50% (         <>                 ) 30% ( 43)
 50% (         >>                 ) 30% ( 45)
 50% (         <-                 ) 30% ( 49)
 50% (        <-                  ) 30% ( 52)
 50% (   <<                       ) 30% ( 64)
 50% (  <<                        ) 30% ( 70)
 50% (<-                          ) 30% ( 72)
```

[1)] const je zkratka pro const int, viz zobraz1.h

Dynamická alokace paměti

V této kapitole se již nebudete učit nové konstrukce jazyka C, ale ukáži vám některé funkce ze standardní knihovny. Konkrétně funkce pro získávání dynamické paměti.

- Získání a uvolnění paměti pomocí malloc() a free()
- Získání paměti pro struktury pomocí calloc()
- Změna velikosti získané paměti pomocí realloc()

Každý program lze v zásadě rozdělit na dvě části. Na část datovou, kde jsou uloženy data, se kterými program pracuje (tj. proměnné, textové literály, konstanty) a na část programovou, která obsahuje instrukce programu.

Ihned po spuštění programu jsou data která ukládáte do proměnných uložena v paměti počítače. Z této paměti lze data číst a lze do ní i zapisovat.

Takto získaná paměť má své pro i proti. Výhodou je v zásadě rychlost a zaručená dostupnost paměti. Ovšem jsou situace, kdy dopředu nemůžete vědět, kolik budete paměti potřebovat. Například budete chtít program, který načte od uživatele několik čísel a pak je setřídí od nejmenšího do největšího. Jelikož dopředu nevíte, kolik čísel bude uživatel chtít setřídit, můžete to jenom odhadnout a vytvořit si pro tyto čísla pole dlouhé např. 1000 položek typu float. To už je velké pole, které zvětší velikost programu a zpomalí jeho chod, přičemž se může stát, že uživatel bude využívat v průměru třeba jen 10 položek. Na druhou stranu se také může stát, že bude potřebovat 1001 položek.

Naštěstí nejste odkázáni jen na paměť představovanou proměnnými, které definujete ve zdrojovém kódu (taková místa v paměti se nazývají statická). Můžete žádat o paměť pro proměnnou (nebo celá pole proměnných) za chodu programu. A tomu se říká „**dynamická alokace paměti**".

I dynamická alokace paměti má své výhody a nevýhody. Výhodou je, že program využívá právě tolik paměti, kolik potřebuje. Navíc získanou paměť můžete (měli byste) během programu uvolnit. Takže když například jednou potřebujete 10 MiB dat a podruhé 15 MiB dat, nejdříve požádáte o 10 MiB, využijete je, potom uvolníte, pak požádáte o těch potřebných 15 MiB dat … Takže i když jste potřebovali celkem 25 MiB dat, v jeden okamžik jste jich potřebovali maximálně 15.

Jistou nevýhodou je zvýšená pracnost s dynamickými strukturami (a časová režie při získávání a uvolňování paměti). Proto vždy rozvažte, zda se vyplatí (například pro ukládání řetězce čteného od uživatele) vytvářet dynamické data, nebo využít statických polí.

Získání a uvolnění paměti pomocí malloc() a free()

Nejdříve se podívejte na deklarace funkcí malloc() a free() a pak popíši jejich funkce. Obě funkce najdete v hlavičkovém souboru **<stdlib.h>**. Datový typ **size_t**, který se používá k určování velikosti získávané paměti, je definován též v <stddef.h>. Je to datový typ definovaný pomocí typedef. Je to celočíselný typ, něco jako unsigned int. Máte zaručeno, že size_t je dostatečně velký na to, aby se v něm mohla uložit velikost paměti (přesněji řečeno velikost datového typu). Proto používejte size_t a ne unsigned int, který nemusí být na všech platformách dostatečně velký.

```
void *malloc(size_t size);
void free(void *ptr);
```

Funkce **malloc()** má jako jediný argument počet bajtů paměti, které chcete od operačního systému získat (alokovat). Zajímavá je návratová hodnota, která je deklarována jako ukazatel **void ***. Asi vás už napadlo, že s dynamicky alokovanou pamětí se bude pracovat pomocí ukazatelů. Novou paměť vám totiž přidělí operační systém a funkce malloc() vrátí ukazatel na začátek paměti, která byla programu operačním systémem přidělena. Samo sebou se může stát, že již není dostatek volné paměti. V takovém případě funkce malloc()vrací hodnotu NULL.

Vrácený ukazatel je třeba přetypovat na správný datový typ, který budete chtít v alokovaném datovém poli (nebo jedné proměnné) používat.

Paměť, kterou takto získáte, je souvislá oblast dlouhá **size_t** bajtů s nedefinovaným obsahem. Jsou v ní tedy náhodné hodnoty bajtů. Pokud použijete funkci malloc() vícekrát, získáte několik souvislých oblastí, ale vzájemně již souvislé být nemusí a také většinou nejsou. Maximální velikost dynamicky alokované paměti je dána jednak fyzickými možnostmi vašeho počítače a také velikostí adresování paměti. U 32 bitových programů je možné adresovat až 4GB, u 16 bitových je to samozřejmě méně a u 64 bitových více.

Jak jsem již v začátku kapitoly předeslal, dynamicky alokovanou (získanou) paměť je možné operačnímu systému vrátit a tím snižovat nároky programu na paměť. To se provádí funkcí **free()**. Tato funkce má jako argument ukazatel na začátek alokované paměti (tedy hodnotu, kterou vrací funkce malloc(), nebo calloc(), nebo realloc(), viz níže). Pokud funkci free() v programu nepoužijete, je alokovaná paměť vrácena operačnímu systému až po skončení programu. Přesto byste měli funkci free() v programu vždy volat, a to co nejdříve to jde.

Pokud program neuvolňuje nepotřebnou paměť, dochází k takzvanému „**úniku paměti**" (memory leaks). Déle běžící aplikace si tak říká o stále více a více paměti, až nakonec spotřebuje všechnu, která je dostupná. Takovými problémy často trpěla například Mozilla Firefox (ale i mnoho jiných profesionálních aplikací).

Po volání funkce free() již paměť není přidělena programu a pokus o zápis do této paměti (nebo čtení z ní) by byl po zásluze potrestán sejmutím programu nebo dokonce operačního systému (pokud je tak hloupý a nechá si to líbit).

Následující zdrojový kód ukazuje použití funkcí malloc() a free(). Jde o program, který načte od uživatele číslo určující počet následně zadaných čísel, které se poté setřídí. Všimněte si, jak je v programu pomocí NULL kontrolováno, zda byla dynamická paměť skutečně alokována, jak je přetypována návratová hodnota a jak se určuje velikost získávaného datového pole.

```
1.  *-----------------------------------------------*/
2.  /* c18/dynamic1.c                               */
```

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

#ifdef _MSC_VER
#define ZU  "Iu"
#define SZU "Iu"
#else
#define ZU  "zu"
#define SZU "zu"
#endif

/**
 * funkce pro setrideni pole od nejmensiho do nejvetsiho
 */
void setrid(size_t delka, float pole[])
{
size_t i, j;
float pom;
if (delka <= 1)
return;

for (j = 0; j < delka - 1; j++) {
i = 0;
do {
if (pole[i] > pole[i + 1]) {
pom = pole[i];
pole[i] = pole[i + 1];
pole[i + 1] = pom;
}
i++;
} while (i < delka - 1);
}
}

int main(void)
{
unsigned int i;
/*
 ukazatel na dynamicky alokovane cislo, ktere bude urcovat
 delku pole pro tridene cisla. V tomto pripade by bylo
 lepsi pouzit statickou promennou (size_t i);
 je to tu tak udelano jen pro nazorny priklad */
size_t *ui;
/* ukazatel na dynamicky alokovane pole, do ktereho budou
 ulozeny tridene cisla */
float *uf;


ui = (size_t *) malloc(sizeof(size_t));
if (ui == NULL) { /* jeden mozny zpusob kontroly */
printf("Nedostatek pameti!\n");
return 0;
}

printf("Zadejte pocet cisel: ");
/* u funkce scanf by se melo kontrolovat, zda skutecne
 nacetla to, co mela. Pro strucnost prikladu to nedelam */
scanf("%" SZU, ui);

/* pozadam o tolik bytu, kolik ma
 typ float krat pocet prvku pole */
if ((uf = (float *) malloc(sizeof(float) * (*ui))) == NULL)
/* taky zpusob kontroly :-) */
{
printf("Nedostatek pameti!\n");
return 0;
}

for (i = 0; i < *ui; i++) {
printf("Zadejte cislo [%2" ZU "]: ", i + 1);
scanf("%f", uf + i); /* nebo uf[i] */
}

```

```
78.  setrid(*ui, uf);
79.
80.  for (i = 0; i < *ui; i++) {
81.  printf("%5.2f ", uf[i]); /* nebo uf + i */
82.  }
83.  printf("\n");
84.  free(ui);
85.  free(uf);
86.  return 0;
87.  }
88.
89.  /*----------------------------------------------*/
```

Makro _CRT_SECURE_NO_WARNINGS je tu kvůli funkci scanf(), viz scanf().
Popis definic ZU a SZU viz datový typ pro ukazatel.
Výstup z programu:

```
Zadejte pocet cisel: 5
Zadejte cislo [ 1]: 4
Zadejte cislo [ 2]: -3.5
Zadejte cislo [ 3]: 3.8
Zadejte cislo [ 4]: 2
Zadejte cislo [ 5]: 0.5
-3.50  0.50  2.00  3.80  4.00
```

V druhém příkladě uvidíte, jak lze dynamicky vytvořit dvourozměrné pole. Jeho velikost bude 100*200*sizeof(int) bajtů.
Především si všimněte, jak je paměť uvolňována.

```
1.   /*----------------------------------------------*/
2.   /* c18/dynamic2.c                               */
3.
4.   #include <stdio.h>
5.   #include <stdlib.h>
6.   #include <stddef.h>
7.
8.   #ifdef _MSC_VER
9.   #define ZU  "Iu"
10.  #else
11.  #define ZU  "zu"
12.  #endif
13.
14.  #define X  100              /* pocet radku */
15.  #define Xv1 10              /* vypis radek Xv1 az Xv2 */
16.  #define Xv2 20
17.
18.  #define Y  200              /* sloupce */
19.  #define Yv1 50              /* vypis sloupcu Yv1 az Yv2 */
20.  #define Yv2 60
21.
22.  int main(void)
23.  {
24.  unsigned int **ui;
25.  size_t a, b;
26.
27.  ui = (unsigned int **) malloc(X * (sizeof(unsigned int *)));
28.  if (ui == NULL)
29.  return 1;
30.
31.  for (a = 0; a < X; a++) {
32.  ui[a] = (unsigned int *) malloc(Y * (sizeof(unsigned int)));
33.  if (ui[a] == NULL)
34.  return 1;
35.  }
36.
37.  for (a = 0; a < X; a++)
38.  for (b = 0; b < Y; b++)
39.  ui[a][b] = (unsigned int) a * b;
40.
41.  printf("     ");
42.  for (b = Yv1; b <= Yv2; b++)
43.  printf("%5" ZU " ", b);
44.
45.  printf("\n");
46.  for (a = Xv1; a <= Xv2; a++) {
47.  printf("%3" ZU ": ", a);
48.  for (b = Yv1; b <= Yv2; b++)
```

```
49.  printf("%5i ", ui[a][b]);
50.  printf("\n");
51.  }
52.
53.  /* jak byla pamet alokovana, tak musi byt dealokovana.
54.   * Jednoduseji to opravdu nejde. */
55.  for (a = 0; a < X; a++) {
56.  free(ui[a]);
57.  }
58.  free(ui);
59.
60.  return 0;
61.  }
62.
63.  /*------------------------------------------------*/
```

Pokud bych zavolal jen free(ui);, uvolnila by se jen paměť alokovaná prvním zavoláním funkce malloc(). Navíc by se tak ztratili všechny ukazatele na paměť získávanou v cyklu na řádku 32. To by byla krásná ukázka úniku paměti.

Získání paměti pro struktury pomocí calloc()

Funkce calloc() je deklarována následovně:

void *calloc(size_t nmemb, size_t size);

Funkce calloc() alokuje paměť pro *nmemb* položek velikosti *size*. Velikost položky může být například sizeof(int). Ovšem častěji se využívá pro vytvoření dynamického pole struktur.

Tato funkce, na rozdíl od funkce malloc(), **vyplní alokovanou paměť nulami**. Všechno ostatní je stejné jako u funkce malloc() (návratová hodnota, uvolňování pomocí free() atd.). Musíte samozřejmě počítat s nějakou časovou režijí, kterou zabere nulování paměti.

Na příkladu ukáži, jak lze vytvářet **seznam**. Seznam obsahuje datové objekty (struktury), které jsou mezi sebou provázány ukazateli. První prvek seznamu ukazuje na druhý, druhý na třetí atd. V příkladě bude prvek obsahovat (kromě ukazatele na další prvek v seznamu) pole typu char, do kterého bude ukládat slova načtené od uživatele. Po skončení načítání se slova lexikograficky seřadí. Struktura bude také obsahovat číslo určující kolikáté bylo slovo načteno.

K setřídění použiji funkci strcmp(), kterou popíši v kapitole věnované standardnímu souboru <string.h>. Tato funkce porovnává lexikograficky dva řetězce a vrací nulu, pokud jsou si rovny.

```
1.  /*------------------------------------------------*/
2.  /* c18/dynamic3.c                                 */
3.  #define _CRT_SECURE_NO_WARNINGS
4.  #include <stdio.h>
5.  #include <stdlib.h>
6.  #include <stddef.h>
7.  #include <string.h>
8.  #include <stdbool.h>
9.
10. #define MAXSLOVO 20
11. #define NACTISLOVO "%20s"
12.
13. struct veta {
14. unsigned int poradi;
15. char slovo[MAXSLOVO + 1];
16. struct veta *dalsi;        /* ukazatel na dalsi strukturu */
17. };
18.
19.
20. /**
21.  * pomocna funkce na kopirovani retezce
22.  */
23. void strcopyruj(char *cil, const char *zdroj)
24. {
25. size_t i = 0;
26. do {
27. cil[i] = zdroj[i];
28. } while (zdroj[i++]);
29. }
30.
31.
32. /**
33.  * funkce vypisujici seznam
34.  */
35. void vypis_veta(struct veta *zacatek)
36. {
37. struct veta *dalsi;
38.
39. dalsi = zacatek;
40. printf("\n");
41. do {
42. if (strcmp(dalsi->slovo, "")) { /* prazdne slova ignoruj */
43. printf("%3i: %s\n", dalsi->poradi, dalsi->slovo);
44. }
```

```
45.  dalsi = dalsi->dalsi;   /* prejdi na dalsi slovo */
46.  }
47.  while (dalsi != NULL);
48.  }
49.
50.  void setrid_seznam(struct veta *zacatek)
51.  {
52.  bool zmena = false;
53.  struct veta *dalsi, pom;
54.
55.  dalsi = zacatek;
56.  if (dalsi == NULL)
57.  return;
58.
59.  do {
60.  /* dosli jsme na konec a nic se nezmenilo?
61.  pak mame setrideno */
62.  if ((dalsi->dalsi == NULL) && (!zmena))
63.  break;
64.  else if (dalsi->dalsi == NULL) {
65.  dalsi = zacatek;
66.  zmena = false;
67.  }
68.  if (strcmp(dalsi->slovo, dalsi->dalsi->slovo) > 0) {
69.  pom = *dalsi;
70.  strcopyruj(dalsi->slovo, dalsi->dalsi->slovo);
71.  dalsi->poradi = dalsi->dalsi->poradi;
72.  strcopyruj(dalsi->dalsi->slovo, pom.slovo);
73.  dalsi->dalsi->poradi = pom.poradi;
74.  zmena = true;
75.  }
76.  dalsi = dalsi->dalsi;
77.  } while (1);
78.  }
79.
80.
81.  /**
82.  * Funkce pro uvolneni pameti. Prvni polozka je staticka, proto se
83.  * pomoci free() neuvolnuje a take se musi jeji atribut "dalsi"
84.  * nastavit na NULL */
85.  void zabij(struct veta *zacatek)
86.  {
87.  struct veta *v, *pom;
88.
89.  v = zacatek->dalsi;
90.  while (v != NULL) {
91.  pom = v->dalsi;
92.  free(v);
93.  v = pom;
94.  }
95.  zacatek->dalsi = NULL;
96.  }
97.
98.  int main(void)
99.  {
100.  unsigned int poradi;
101.  int navrat;
102.  struct veta prvni, *posledni;
103.
104.  printf("Zadavejte slova ne delsi nez %i znaku.\n"
105.  "Zadavani vet ukoncete pomoci znaku konec souboru.\n"
106.  "(tj. CTRL+D v unixech nebo CTRL+Z + Enter v DOSu a Windows)\n\n",
107.  MAXSLOVO);
108.
109.  poradi = 0;
110.  /* Struktura "prvni" je prvni prvek ze seznamu. Ukazatel
111.  "posledni" ukazuje na posledni prvek v seznamu. */
112.  posledni = &prvni;
113.  do {
114.  poradi++;
115.  navrat = scanf(NACTISLOVO, posledni->slovo);
116.  posledni->poradi = poradi;
117.  if ((navrat == EOF) || (navrat != 1)) {
118.  /* EOF * -> konec vstupu,
119.  * navrat  !=1 -> nebyl nacten spravne retezec */
```

```
120.          /* dalsi prvek v seznamu jiz nebude. seznam ukoncime
121.             zarazkou NULL */
122.          posledni->dalsi = NULL;
123.          break;
124.          }
125.       posledni->dalsi = (struct veta *) calloc(1, sizeof(struct veta));
126.          if(posledni->dalsi == NULL) {
127.          printf("Nedostatek pameti!\n");
128.          exit(1);
129.          }
130.          posledni = posledni->dalsi;
131.          } while (1);
132.
133.          setrid_seznam(&prvni);
134.          vypis_veta(&prvni);
135.          zabij(&prvni);
136.
137.          return 0;
138.          }
139.
140.       /*---------------------------------------------*/
```

Možný výstup z programu:

```
Zadavejte slova ne delsi nez 20 znaku.
Zadavani vet ukoncete pomoci znaku konec souboru.
(tj. CTRL+D v unixech nebo CTRL+Z + Enter v DOSu a Windows)

Kdyz nejde hora k Moha-Medovi, musi Moha-Med k hore.
1: Kdyz
7: Moha-Med
5: Moha-Medovi,
3: hora
9: hore.
4: k
8: k
6: musi
2: nejde
```

Lepším příkladem na použití calloc() by byl asi příklad s řídkou maticí (tj matice, která obsahuje spoustu nul). Tam by se ukázala výhoda inicializace paměti nulami funkcí calloc(), oproti neinicalizované paměti vrácené funkcímalloc() (paměť alokované funkcí malloc() může obsahovat všelijaké „smetí").

Změna velikosti získané paměti pomocí realloc()

Funkce **realloc()** umožňuje změnit velikost dříve alokované paměti s tím, že data v již alokované paměti zůstanou nezměněna.

Deklarace funkce je následující:

```
void *realloc(void *ptr, size_t size);
```

První argument funkce, *ptr* je adresa paměti (anglicky pointer), která byla alokována pomocí funkcí malloc(),calloc() nebo realloc(). Velikost tohoto paměťového bloku se zvětší (či zmenší) na *size* bajtů.

Návratovou hodnotou je ukazatel na „přealokovanou" paměť. To ovšem nemusí být stejné místo v paměti jako původní paměť (na kterou ukazoval argument ptr)! Proto po volání funkce realloc() se již nepokoušejte přistupovat do paměti pomocí hodnoty v ukazateli *ptr*, ani ji uvolňovat, ale používejte jen paměť na kterou ukazuje hodnota vrácená funkcí realloc().

Pokud by byl argument ptr NULL, pak by bylo volání realloc() obdobné jako malloc(). Pokud byste velikost pole *size* nastavili na nulu, bylo by to stejné, jako byste volali pro *ptr* funkci free().

Nově alokovaná paměť má nedefinované hodnoty (není vynulována jako u calloc()). Původní části paměti, kterou jste přealokovali, bude obsahovat stejná data, jako před alokací. (Jen mohou být na jiném místě v paměti, takže pozor na používání odkazů inicializovaných před realokací).

Pokud se funkci realloc() nepodaří získat dostatek nové paměti, vrátí ukazatel NULL. Proto byste neměli v programu používat následující konstrukci:

```
ukazatel = realloc(ukazatel, SIZE);
```

Takto by se totiž do ukazatele uložila hodnota NULL a vy byste přišli o ukazatel na původní alokovanou paměť, která by zůstala alokovaná, ale už by jste se k ní neměli jak dostat.

V příkladu bude uživatel zadávat programu požadovanou velkost v MiB alokované paměti a program jí bude alokovat.

```
/*---------------------------------------------*/
/* c18/dynamic4.c                              */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

#define MIB 1024*1024

int main(void)
{
unsigned long x; size_t a;
char *pom, *v = NULL;
```

```
                do {
        printf("Zadejte pocet megabajtu, 0 pro konec: ");
                scanf("%lu", &x);
            pom = (char *) realloc((void *) v, x * MIB);
                if ((pom == NULL) && (x))
                printf("Nedostatek pameti!!\n");
                else {
                v = pom;
            /* vyplneni pameti jednickami.
         * To pocitac trosilinku zamestna :-) */
            for (a = 0; a < x * MIB; a++) {
                v[a] = '1';
                }
            }
        } while (x);

        return 0;
        }

/*-------------------------------------------------*/
```

*Pokud spustíte program v Linuxu, můžete na konzoli pomocí příkazu **free** sledovat, jak vám roste a klesá velikost využité paměti počítače podle toho, kolik si jí zrovna žádáte.*
V DOSu a Windows 3.11 bude 1 MiB souvislé paměti příliš velký požadavek[1]. Pokud překládáte program tam, žádejte o paměť raději po kibibytech :-).