Jazyk

C

Předmluva

C je univerzální programovací jazyk, který charakterizují úsporné výrazy, moderní řízení běhu a struktura údajů, bohatství operátorů. C není jazyk „na vysoké úrovni“ ani není „velký“ a není specializován na použití pouze v určité oblasti. Tato obecnost jazyka C jej činí vhodnějším a efektivnějším pro mnoho úloh než jiné „mocnější“ jazyky.

Autorem jazyka C je Denis Ritchie, který původně koncipoval tento jazyk pro operační systém UNIX na počítači DEC PDP-11. Operační systém, překladač jazyka C a vlastně všechny aplikační programy systému UNIX (i veškeré programové vybavení připravené pro tutu knihu), jsou napsány v jazyku C. Účinné překladače existují pro další počítače, např. IBM System 1370, Honeywell 6000 a Interdata 8/32. Jazyk C není svázán s určitým konstrukčním řešením počítačů.

Tato kniha je koncipována tak, aby pomohla naučit se čtenáři programovat v jazyku C. Obsahuje učební úvod, umožňující uživatelům co nejrychlejší začátek, samostatné kapitoly týkající se nejdůležitějších rysů jazyka a přehled literatury. Úspěch při studiu bude zajištěn zejména čtením, psaním a opakováním uvedených příkladů, než pouhým učením zákonitostí. Všechny uvedené příklady tvoří úplné, skutečné programy (nikoliv pouze části). Jsou spojeny s průvodním textem a jsou napsány ve vhodné počítačové formě. Kromě ukázek, jak účinně tento jazyk používat, jsme se pokusili na vhodných místech ukázat užitečné algoritmy a vhodný programovací styl.

Tato kniha není úvodní programovací příručkou, přesto i nováček bude po prostudování schopný se v jazyku C vyznat a orientovat, zvláště když mu pomůže zkušenější kolega.

V našich zkušenostech se jazyk C projevil jako příjemný, výrazný a mnohostranný jazyk se širokým použitím v různých problémech. Snadno se učí a čím více porostou vaše zkušenosti s ním, tím lépe vám bude sloužit. Doufáme, že kniha vám pomůže užívat C dobře a prospěšně.

Brian W. Kernighan
Denis M. Ritchie

Úvod

C je univerzální programovací jazyk. Je úzce spjat s operačním systémem UNIX, protože byl vyvinut v tomto systému a také operační systém UNIX a jeho programové vybavení je napsáno v jazyku C. Jazyk C však není vázán k určitému operačnímu systému nebo určitému typu počítače, přestože je nazýván „systémovým programovacím jazykem“ a je použitelný pro psaní operačních systémů. Je stejně dobře použitelný i pro tvoření programů numerického charakteru, programů pro zpracování textů a hromadných dat. Jazyk C je jazyk relativně nízké úrovně. Tento jeho rys nesnižuje však jeho význam; je tím řečeno, že C pracuje se stejnou třídou objektů jako většina počítačů tj. se znaky, čísly a adresami. To může být kombinováno s obvyklými aritmetickými a logickými operátory implementovanými na konkrétním počítači.

Jazyk C nemá operace zpracovávající přímo složené objekty jako jsou řetězce znaků, seznamy nebo pole uvažované jako celek. Není zde např. analogie s operacemi jazyka PL/1, které zpracovávají celá pole znaků. Jazyk umožňuje pouze statické definice obsazení paměti, není zde možnost dynamického obsazení paměti a obsazení volných míst jako v jazyku ALGOL 68. Konečně, C nemá vybavení pro vstupní a výstupní operace. Nemá příkazy READ a WRITE a přímý přístup k souborům. Všechny tyto mechanizmy známé z vyšších programovacích jazyků musí být vykonány explicitně voláním funkcí.

Jazyk C umožňuje pouze přímé jednoduché řízení běhu programu: testy, cykly, podprogramy. V tomto jazyce není možné uvažovat o multiprogramování, paralelních operacích nebo synchronizaci. Přesto, že nepřítomnost těchto možností může vypadat jako vážný nedostatek („To mi chcete říci, že musím zavolat funkci, když chci porovnat dva řetězce znaků?“), tak udržení jazyka na nižší úrovni přináší opravdu značné výhody. Protože jazyk C je relativně malý, může být popsán na malé ploše a je snadné se mu naučit. Překladač jazyka C proto může být jednoduchý a kompaktní a může být snadno vytvořen. Použitím současných postupů může tvorba překladače pro nový počítač trvat pouze několik měsíců, protože 80% překladače je shodných s již existujícími překladači. To je umožněno vysokým stupněm přenositelnosti jazyka. Protože typy dat a struktury, jaké jsou v jazyku C používány, jsou zajišťovány většinou počítačů, tak implementace knihoven je jednoduchá. Např. na PDP 11 obsahuje pouze podprogramy pro násobení a dělení 32-bitových slov a podprogramy pro volání funkcí a návrat z nich. Ovšem každá implementace počítá s kompaktní knihovnou pro vstupy a výstupy, obsluhování řetězců a operace s pamětí. Protože jsou však volány pouze explicitně, mohou být připojeny jen když je třeba. Rovněž mohou být napsány v jazyku C.

Programy v jazyku C jsou dostatečně efektivní a není třeba místo nich psát programy v assembleru. Jedním z takových příkladů je operační systém UNIX, který je téměř celý napsán v jazyku C. Navíc veškerý aplikační software systému UNIX je psán v jazyku C. Převážná většina uživatelů systému UNIX (včetně autorů této knihy) nezná assembler počítače PDP-11.

Přestože jazyk C pracuje na mnoha počítačích, je nezávislý na architektuře daného počítače, a tak je možné s trochou péče psát „přenositelné“ programy. V našem oddělení je software, který je vyvinut pod systémem UNIX, přenášen na počítače Honeywell, IBM a Interdata system. Ve skutečnosti jsou překladače a prostředky jazyka C na těchto čtyřech počítačích podstatně kompatibilnější než např. ANSI standard FORTRAN. Pro programátory, kteří pracují s jinými jazyky, může být pro srovnání užitečné se dozvědět o historických, technických a filozofických aspektech jazyka C.

Většina nejdůležitějších myšlenek jazyka C pochází z dosti starého, ale stále živého jazyka BCPL, který byl vyvinut Martinem Richardsem. Vliv BCPL na C se uskutečnil nepřímo jazykem B, který napsal Ken Thompson v r. 1970 pro první systém UNIX na PDP-11.

Přestože jazyk C má s BCPL mnoho společných znaků, není v žádném případě jeho kopií. Jazyky BCPL a B jsou jazyky bez „typu“. Jediný typ dat je slovo počítače a přístup k ostatním druhům je pomocí speciálních operátorů nebo funkcí. V jazyku C jsou základními datovými typy znaky, celá čísla různých délek a čísla s pohyblivou desetinou čárkou. Navíc je zde hierarchie odvozených datových typů vytvořených pointry, poli, strukturami, uniony a funkcemi. C umožňuje základní konstrukce pro řízení běhu, požadované pro dobře strukturované programy: shlukování příkazů, rozhodování (if), cykly s testem na ukončení nahoře (while, for) nebo dole (do), klíč výběru (switch). Vše již bylo implementováno v jazyku BCPL, ale s poněkud jinou syntaxí; BCPL dlouhá léta očekával příchod „strukturovaného programování“.

Jazyk C umožňuje používání pointru a aritmetické adresy. Argumenty funkcí jsou při předání kopírovány a není možné, aby volaná funkce změnila hodnotu aktuálního parametru ve volající funkci. Pokud požadujeme „předávání adresou“, může být předán pointer, a volaná funkce může změnit objekt, na který pointer ukazuje. Pole jsou předávána adresou počátku pole.

Funkce mohou být volány rekurzívně a lokální proměnné jsou převážně „automatické“, tj. jsou vytvořeny znovu při každém vyvolání. Definice funkcí nemohou být vkládány do sebe, ale proměnné mohou být deklarovány blokovou strukturou. Funkce jazyka C mohou být překládány samostatně. Proměnné mohou být interní dané funkcí, externí a známé jen v jednom zdrojovém souboru anebo úplně globální. Interní vnitřní proměnné mohou být automatické nebo statické. Automatické proměnné mohou být ukládány pro zvýšení efektivity do registru počítače, ale příkaz register je pouze pokyn pro překladač a ne přímo pro počítačový registr.

Jazyk C není tak jednoznačný jazyk ve smyslu Pascalu nebo Algolu 68. Umožňuje datové verze, ale neprovádí automatické konverze dat s velkou přehlíživostí jazyka PL/1. Dosud vytvořené překladače nekontrolují při výpočtu meze polí, typy argumentů atd.

Pro situace, kdy je nezbytné kontrolovat typy dat, se používá specialní verze překladače. Tento program se nazývá LINT. LINT negeneruje kód, ale místo toho přísně kontroluje vše, co je možné kontrolovat při běhu a zavádění programu. Nalezne nesouhlas typu, nekonzistentní použití argumentu, nepoužité nebo očividně neinicializované proměnné atd. Program, který projde přes LINT si může užívat (s několika výjimkami) svobodu hlášení chyb tak jako např. programy v Algolu 68. O dalších vlastnostech programu LINT se zmíníme, až k tomu bude příležitost.

Jazyk C má stejně jako ostatní jazyky také svoje nedostatky. Některé operátory mají nesprávnou prioritu, některá část syntaxe by mohla být lepší; existuje mnoho verzí jazyka, lišících se od sebe. Nicméně se ukázalo, že jazyk C je velice efektivní a výrazný pro celou škálu aplikací.

Kniha je organizována následujícím způsobem: kapitola 1 tvoří úvod do výuky střední partie jazyka C. Účelem je začít co nejrychleji, protože pevně věříme tomu, že nový jazyk se nejlépe naučíme, budeme-li v něm psát programy. Předpokládáme základní znalost programovaní. Není zde vysvětlováno, co je to počítač, překladač ani co znamená výraz $n = n + 1$. Přestože jsme se pokoušeli ukázat užitečnou techniku programování všude, kde to jen bylo možné, tak si nemyslíme, že tato kniha bude příručkou datových struktur a algoritmů. Vždy, když jsme si mohli vybrat, jsme se soustředili na jazyk.

V kapitolách 2 až 6 jsou probrány detailněji vlastnosti jazyka C. Důraz je stále kladen na tvorbu kompletních užitečných programů. Kapitola 2 pojednává o základních typech dat, operátorů a výrazů. V kapitole 3 se hovoří o řízení běhu: if-else, while, for atd. Kapitola 4 popisuje funkce a strukturu programu – externí proměnné, oblast platnosti proměnných atd. V kapitole 5 je diskutováno použití pointrů a aritmetických adres. Kapitola 6 obsahuje popis struktur a unionů. Kapitola 7 popisuje standardní knihovnu vstupů a výstupů, která zajišťuje styk s operačním systémem. Tato kniha je dodávána na všechny typy počítačů, kde pracuje jazyk C, takže programy mohou být snadno převáděny z jednoho systému do druhého téměř beze změny.

Obsah

Kapitola 1

Úvod k výuce

Začneme stručným úvodem do jazyka C. Naším cílem je ukázat základní prvky jazyka na skutečných programech s tím, že nebudou vynechávány detaily, formální pravidla ani výjimky. V tomto okamžiku se nesnažíme o kompletnost. Snažíme se umožnit čtenářům psát užitečné programy tak rychle, jak je jen možné. Soustředíme se na tyto základní pojmy: proměnné a konstanty, aritmetika, větvení programu, funkce a základy vstupu a výstupu. Úmyslně v této kapitole vynecháváme vlastnosti jazyka C, které jsou důležité pro psaní větších programů. Jedná se o pointry, struktury, většinu z bohatého rejstříku operátorů jazyka C, většinu příkazů pro podmíněné větvení programu a další podrobnosti.

Tento přístup má ovšem své nevýhody. Zvláštností je to, že kompletní popis určité vlastnosti jazyka není na jednom místě. Vzhledem k tomu, že není možné používat od začátku všechny možnosti jazyka C, příklady nejsou tak stručné a elegantní jak by mohly být. Snažili jsme se tuto nevýhodu minimalizovat, ale přesto na to čtenáře upozorňujeme.

Další nevýhodou je to, že se budeme v několika článcích opakovat. Myslíme si však, že opakování vám spíše pomůže v učení, než že vás bude obtěžovat.

Zkušení programátoři by měli být v každém případě schopni si z této kapitoly vybrat to, co potřebují. Začátečníkům doporučujeme, aby si studium této kapitoly doplnili napsáním malých programů, ktere jsou modifikací programů uvedených.

1.1 Začínáme

Jediný způsob, jak se naučit nový programovací jazyk, je psát programy v tomto jazyku. První program, který napíšeme, je stejný pro všechny jazyky:

Vytiskni slova

```
h e l l o,   w o r l d
```

To je základní překážka; k tomu, abychom ji překonali, musíme být schopni někde vytvořit text programu, úspěšně ho přeložit, sestavit a spustit, a potom zkontrolovat, zda vytvořil požadovaný výstup. Vše ostatní je snadné. Program pro vytištění textu „hello, world“ vypadá v jazyku C takto:

```
main ()
{
   printf ("hello, world\n");

}
```

Jak naložit s tímto programem záleží na tom, jaký systém používáte. Např. v operačním systému UNIX je třeba uložit zdrojový text programu do souboru, jehož jméno má příponu „.c“ (např. hello.c) a přeložit ho použitím příkazu

```
cc hello.c
```

Pokud jste něco nepokazili, překlad proběhne bez chyb a vytvoří se soubor a.out, který je možno spustit příkazem

```
a.out
```

Výstupem bude text

```
hello, world
```

Na jiných systémech budou ovšem platit jiná pravidla a ty je třeba konzultovat s místním odborníkem.

Cvičení 1-1. Spusťte tento program na vašem systému. Zkoušejte vynechávat některé jeho části a pozorujte chybová hlášení.

Nyní něco o programu samém. Program v jazyce C, ať je jeho velikost jakákoliv, sestává vždy z jedné nebo více „funkcí“, které mají být vykonány. Funkce v jazyce C jsou obdobné funkcím a podprogramům v jazyce FORTRAN nebo procedurám v jazyce PASCAL, PL/1 atd. V našem případě je takovou funkcí main. Obyčejně můžeme dávat funkci libovolné názvy, avšak název main je speciální název – váš program zahajuje svoji činnost vždy na začátku funkce main. To znamená, že každý program musí obsahovat jednotku main. main se obvykle odvolává na ostatní funkce; některé jsou přímo obsaženy v programu, některé se volají z knihoven již dříve vytvořených funkcí.

Jednou z metod vzájemné komunikace mezi funkcemi je předávání dat argumenty. Závorky následující za názvem funkce ohraničují seznam argumentů. V našem příkladu je main funkcí, která nemá parametry. To je znázorněno symboly ( ) – prázdným seznamem argumentů. Složené závorky { } sdružují příkazy, které vytvářejí tělo funkce. Jsou analogické příkazům DO-END v jazyku PL/1 nebo příkazům begin-end v ALGOLU, PASCALU atd. Funkce je vyvolávána názvem, za kterým následuje seznam argumentů v kulatých závorkách. Nepoužívá se příkaz CALL jako ve FORTRANU nebo v PL/1. Závorky musí být uvedeny, i když neobsahují seznam argumentů.

Programová řádka

```
printf("hello, world\n");
```

je voláním funkce, která se jmenuje printf a jediným argumentem je řetězec znaků "hello, world \n". printf je knihovní funkce, která zobrazuje výstup na terminál (jestliže není specifikováno jiné médium). V tomto případě zobrazí řetězec znaků, který je jejím argumentem.

Souvislá řada libovolného množství znaků vložená do uvozovek "..." je nazývána znakový řetězec nebo řetězcová konstanta. V této chvíli budou znakové řetězce používány jako argumenty funkcí jako je např. funkce printf.

Dvojice znaků \n v řetězci je v jazyku C symbolem pro znak nové řádky, který když je nalezen, posune kurzor na levý okraj nové řádky. Kdybychom znaky \n neuvedli (což můžeme udělat jako pokus), uvidíme, že výstup není ukončen přesunem na novou řádku. Uvedení dvojice znaků \n je jediný způsob přechodu na novou řádku. Když budete zkoušet něco podobného jako

```
printf("hello, world
");
```

tak bude překladač jazyka C hlásit chybu (chybějící pravé uvozovky).

Funkce printf nikdy nevykonává přesun na novou řádku automaticky, proto pro vytvoření požadovaného výstupu může být použito vícenásobné vyvolání funkce printf. Náš první program může tedy vypadat takto

```
main()
{
  printf ("hello,");
  printf (" world");
  printf ("\n");
}
```

Výstup bude stejný jako v předešlém příkladu.

Je třeba si uvědomit, že dvojice znaků \n reprezentuje pouze jeden znak; znak změny, (escape) označený znakem \, umožňuje obecný a rozšiřitelný mechanizmus pro reprezentaci těžko zobrazitelných nebo neviditelných znaků. Např. \t je symbol pro tabelátor, \b pro zpětný posun, \" uvozovky a \\ pro obrácené lomítko samo.

Cvičení 1-2. Pokuste se zjistit co se stane, když řetězec znaků, který je argumentem funkce printf, obsahuje \x, kde x je nějaký znak, který jsme výše neuvedli.

1.2 Proměnné a aritmetika

Následující program tiskne převodní tabulku mezi stupni Fahrenheita a stupni Celsia za použití vztahu

$$C = \frac{5}{9}(F - 32)$$

t_C [℃]	t_F [℉]
0	-17,8
20	-6.7
40	4.4
60	15.6
...	...
260	126.7
280	137.8
300	148.9

Program vypadá takto:

```
/*tisk převodní tabulky Fahrenheit - Celsius
pro f=0, 20, ..., 300*/
int lower, upper, step;
main ();
{
      float fahr, celsius;
      lower = 0;      /* dolní mez tabulky teplot */
      upper = 300;    /* horní mez */
      step = 20;      /* hodnota kroku */
      fahr = lower;
      while(fahr <= upper)
            {
            celsius = (5.0/9.0) * (fahr - 32.0);
            printf ("%4.0f %6.1f\n",fahr,celsius);
            fahr = fahr + step;
            }
}
```

První dva řádky programu

```
/*tisk převodní tabulky Fahrenheit - Celsius
pro f=0,20, ..., 300 */
```

jsou komentář, který v tomto případě ve stručnosti vysvětluje činnost programu. Libovolné znaky mezi /* a */ jsou překladačem ignorovány. Komentáře mohou a mají být používány pro zpřehlednění programu. Je dovoleno je používat pro zpřehlednění programu všude tam, kde se jinak může objevit mezera nebo nový řádek.

V jazyku C musí být všechny proměnné deklarovány před prvním použitím, obvykle na začátku funkce před prvním výkonným příkazem. Když zapomenete nějaké proměnné deklarovat, překladač to bude hlásit jako chybu. Deklarace sestává z určení typu a seznamu proměnných.

```
int lower, upper, step;
float fahr, celsius;
```

Typ int znamená, že všechny uvedené proměnné budou celočíselné; float znamená, že se jedná o proměnné s pohyblivou řádovou čárkou. Přesnost a rozsah obou typů je závislý na použitém typu počítače. Např. na počítači PDP-11 je int 16-ti bitové číslo se znaménkem a leží v rozsahu - 32768 a +32767. Číslo typu float je 32 bitové, má 7 významových číslic a leží v rozsahu od 10E-38 do 10E+38. V kapitole 2 je uveden rozsah pro ostatní vybrané typy počítačů.

Další z typů proměnných v jazyce C jsou např.:

char — znak – jeden byte

short — krátké celé číslo

long — dlouhé celé číslo

double — číslo s pohyblivou čárkou a dvojitou přesností

Rozsah těchto typů závisí na použitém typu počítače. Podrobnosti jsou uvedeny v kapitole 2. Základní typy jsou rovněž pole, struktury, uniony, ukazatele na ne, a funkce, které s nimi pracují. Se všemi těmito typy se postupně v textu shledáme.

Skutečný výpočet v programu pro výpočet převodní tabulky mezi stupni Fahrenheita a stupni Celsia začínáme přiřazením

```
lower = 0;
upper = 300;
step = 20;
fahr = lower;
```

které nastaví počáteční hodnoty proměnných. Jednotlivé příkazy jsou odděleny středníkem. Protože každý řádek tabulky je počítán ze stejného výrazu, můžeme použít cyklus, který je definován příkazem while:

```
while (fahr <= upper)
{
        ...
}
```

Podmínka v kulatých závorkách je vyhodnocena. Jestliže je pravdivá (tj. proměnná *fahr* je menší nebo rovna proměnné *upper*), tělo cyklu (tj. příkazy ohraničené složenými závorkami { }) je vykonáno. Potom je podmínka znovu vyhodnocena a jestliže je opět pravdivá, tělo cyklu je opět vykonáno. Jestliže je podmínka nepravdivá (*fahr* je větší než *upper*) cyklus je ukončen. Protože již nejsou v našem programu žádné příkazy, tak je program ukončen.

Tělo příkazu while může být tvořeno jedním nebo více příkazy vloženými do složených závorek, jako je tomu v našem programu, nebo jedním příkazem bez složených závorek, např.

```
while (i < j)
        i = 2 * i;
```

V obou příkladech příkazy, které jsou podmíněny příkazem while, začínají jedním tabelátorem, takže je možné na první pohled určit, které příkazy jsou uvnitř těla cyklu. „Zubová“ struktura textu programu zdůrazňuje logickou strukturu programu. Přestože v jazyku C příliš nezáleží na pozici příkazu v textu, je vhodné zdůrazňovat logickou strukturu a užívat mezery, abychom zdůraznili členění programu. Doporučujeme psát pouze jeden příkaz na řádku a nechávat mezery okolo operátoru. Poloha závorek není již tak důležitá; my jsme si vybrali jeden z mnoha populárních způsobů. Vyberte si i vy svůj způsob, který vám vyhovuje, a ten pak stále používejte.

Většina činností našeho programu je vykonána v těle smyčky while. Teplota ve stupních Celsia je vypočítána a přiřazena proměnné *celsius* příkazem

```
celsius = (5.0/9.0) * (fahr - 32.0);
```

Důvod pro použití 5.0/9.0 místo jednoduššího 5/9 je ten, že v jazyku C, stejně jako v mnoha dalších jazycích, je výsledek celočíselného dělení **ořezán**, takže desetinná část je ztracena. Proto výsledek operace 5/9 je nula a všechny teploty by byly také nula. Desetinná tečka v konstantě indikuje, že se jedná o číslo s pohyblivou řádovou čárkou, a proto $5.0/9.0 = 0.555\ldots$, což je to, co jsme potřebovali.

Rovněž píšeme 32.0 místo 32, přestože *fahr* je typu float a 32 by bylo automaticky zkonvertováno na float (na 32.0) před odečítáním. Je to vlastně jen otázka stylu, přesto je však lepší psát konstanty typu float s desetinnou tečkou i když mají celou hodnotu. Je tím zdůrazněn jejich typ i pro čtenáře programu a zajišťuje se i stejné chápání věci překladačem.

Podrobná pravidla pro to, zda budou celočíselné konstanty konvertovány na typ float jsou uvedeny v kapitole 2.

Nyní si všimněme, že přiřazení

```
fahr = lower;
```

a podmínka

```
while (fahr <= upper)
```

funguje tak, jak očekáváme, tj. že proměnné typu int jsou konvertovány na typ float před uskutečněním operací. Tento příklad také podrobněji ukazuje, jak pracuje funkce printf. printf je vlastně obecně použitelná funkce pro formátovaný výstup. Podrobně bude popsána v kapitole 7. Jejím prvním argumentem je řetězec znaků, který má být zobrazen a znak % určuje, kam mají být další argumenty (druhý, třetí...) umístěny a jakým způsobem mají být tištěny. Např. v příkazu:

```
printf ("%4.0f %6.1f\n" , fahr, celsius);
```

specifikace %4.0f znamená, že číslo typu float má být zobrazeno s délkou čtyř znaků a nemá mít desetinnou část. %6.1f určuje, že další číslo bude v délce 6 znaků s jednou číslicí za desetinnou tečkou: tato specifikace je obdobou formátové specifikace f6.1 ve FORTRANU nebo specifikace f(6,1) v jazyku PL/1. Některé části specifikace nemusí být uvedeny, např. %6f určuje, že číslo má být alespoň 6 znaků dlouhé; %.2f požaduje dvě místa za desetinnou tečkou a celkový počet znaků není omezen; %f specifikuje pouze tisk čísla typu float. printf rovněž rozpoznává %d pro desetinné celé číslo, %o pro oktalovou reprezentaci, %x pro hexadecimální reprezentaci, %c pro znak, %s pro řetězec znaků a %% pro % samo.

Každý znak % v prvním argumentu funkce printf je spojen jen s odpovídajícím druhým, třetím, ... argumentem. Specifikace musí být přesně popsána čísly a typem, jinak program dává nesmyslné výsledky.

Mimochodem funkce printf není částí jazyka C. Sám jazyk C nemá definované operace vstupu nebo výstupu. Funkce printf není obestřena nějakými kouzly, je to jen užitečná funkce, která je částí standardní knihovny funkcí jazyka C. Abychom se mohli soustředit pouze na jazyk C nebudeme až do kapitoly 7 mnoho uvádět o vstupně – výstupních operacích. Odložíme také do té doby pojednání o formátovaném vstupu. Když budete chtít zadávat čísla jako vstup, přečtěte si o funkci scanf v kap. 7 odstavec 7.4. scanf je funkce obdobná jako printf, až nato, že čte vstup místo psaní výstupu.

Cvičení 1-3. Upravte program pro konverzi teplot tak, aby vytiskl záhlaví tabulky.
Cvičení 1-4. Napište program, který bude počítat stupně Celsia v závislosti na stupních Fahrenheita.

1.3 Příkaz for

Jak je možno očekávat, je mnoho způsobů, jak napsat program; uvedeme jinou variantu našeho programu pro konverzi teplot:

```
main ()     /* tabulka převodu Fahrenheit-Celsius/
{
  int fahr;
  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
  printf("%4d %6.1f\n", fahr, (5.0/9.0) * (fahr - 32));
}
```

Výsledek bude stejný, ale program vypadá jinak. Jednou z hlavních změn je zmenšení počtu proměnných. Jediná proměnná, která zůstala, je celočíselná proměnná *fahr* (celočíselná proto, abychom mohli ukázat, jak pracuje konverze %d v printf). Dolní a horní mez a hodnota kroku *step* se objevují jenom jako konstanty příkazu for, který je pro nás novinkou. Výraz pro výpočet převodu se nyní vyskytuje na místě třetího argumentu funkce printf.

Poslední změna je příkladem obecného pravidla jazyka C; v kterémkoliv místě, kde se může vyskytnout proměnná, je možné použít výraz stejného typu. Protože třetí argument funkce printf má být typu float, aby jej bylo možno zobrazit konverzí %6.1f, tak na jeho pozici se může použít výraz v pohyblivé řádové čárce.

Příkaz for je podmíněný příkaz, který je zobecněním příkazu while. Jestliže jej porovnáme s příkazem while, tak by nám měla být jeho činnost jasná. Sestává ze tří částí, které jsou od sebe odděleny středníky.

První část, inicializace

```
fahr = 0;
```

je provedena jednou na začátku. Druhá část je podmínka, která řídí cyklus

```
fahr <= 300;
```

Podmínka je vyhodnocena; jestliže je pravdivá, tak příkazy těla cyklu for jsou vykonány (v našem případě je to pouze jedno vyvolání funkce printf). Potom je vykonána třetí část příkazu for, reinicializace,

```
fahr = fahr + 20
```

a znovu je vyhodnocena podmínka v druhé části příkazu for. Cyklus je ukončen, jestliže je podmínka nepravdivá. Stejně jako u příkazu while může tělo cyklu tvořit jeden nebo skupina příkazů, které jsou uvnitř složených závorek. Inicializace a reinicializace mohou být jednoduché výrazy.

Je lhostejné, použijeme-li v programu příkaz for nebo while, ale vždy se snažíme volit tu variantu, která vypadá jasněji. Příkaz for je vhodný obyčejně v takových příkladech, kdy inicializace a reinicializace jsou jednoduché, logicky svázané příkazy. Příkaz for je tomto případě kompaktnější, protože příkazy pro řízení cyklu jsou zde pohromadě.

Cvičení 1-5. Modifikujte program pro konverzi teplot tak aby, tiskl tabulku v opačném pořadí tj. od 300° F do 0° F.

1.4 Symbolické konstanty

Předtím, než opustíme náš program pro konverzi teplot, učiňme ještě závěrečnou poznámku. Je špatné „pohřbívat magická čísla“ jako jsou čísla 300 a 20 v programu, protože neposkytují těm, kteří budou program číst nebo upravovat, systematickou informaci. Naštěstí existuje v jazyku C způsob, jak se těmto „magickým číslům“ v programu vyhnout. Na začátku programu je totiž možné popisem #define definovat řetězce znaků jako symbolická jména nebo symbolické konstanty. Potom překladač nahradí tato jména odpovídajícími řetězci znaků všude tam, kde se objeví. Náhradou může být skutečně libovolný text, nejen čísla.

```
      #define LOWER 0      /*dolní mez tabulky*/
      #define UPPER 300    /*horní mez*/
      #define STEP 20      /*hodnota kroku*/
      main ()              /*tabulka převodu Fahrenheit -
Celsius/
      {
        int fahr
        for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
          printf("%4d %6.1f\n", fahr, (5.0/9.0) * (fahr - 32));
      }
```

Položky *LOWER, UPPER, STEP* jsou konstanty a proto se neobjevují v deklaracích. Symbolická jména je vhodné psát velkými písmeny, aby je bylo možno jednoduše odlišit od názvů proměnných, které jsou psány písmeny malými. Uvědomme si, že na konci popisu #define není středník, protože vše, co se objeví za symbolickým jménem je za něj v programu dosazováno.

1.5 Výběr užitečných programů

Uvažujme nyní o skupině programů, které budou vykonávat jednoduché operace se znaky. Zjistíme potom, že mnohé z programů jsou jen rozšířením těchto základních programů.

1.5.1 Vstup a výstup znaku

Standardní knihovna obsahuje funkce pro čtení a vypsání znaku.

Funkce getchar načítá vždy další znak pokaždé, když je vyvolána a jako hodnotu vrací tento znak. Takže po příkazu

```
c = getchar()
```

obsahuje proměnná c další znak ze vstupu. Znaky jsou obvykle zadávány z klávesnice, ale to nás až do 7. kapitoly nemusí zajímat.

Funkce putchar je doplňkem funkce getchar

```
putchar (c)
```

Tato funkce vytiskne obsah proměnné *c* na nějaké médium – obyčejně na obrazovku. Volání funkce putchar a printf můžeme kombinovat; výstup se objeví v tom pořadí, jak byly funkce volány. Stejně jako v případě funkce printf není na funkcích getchar a putchar nic magického. Nejsou součástí jazyka C, ale jsou z něho dosažitelné.

1.5.2 Kopírování souboru

Znáte-li funkce getchar a putchar, můžete napsat mnoho užitečných programů, aniž budete vědět něco dalšího o operacích vstupu a výstupu. Nejjednodušším příkladem je program, který kopíruje vstup do výstupu po jednom znaku.

Vývojové schéma vypadá takto:

```
přečti znak
while (znak není symbol pro konec souboru)
   vypiš znak
   přečti další znak
```

Program v jazyku C bude vypadat takto:

```
main ()      /*kopírování vstupu na výstup: 1.verze*/
{
      int c;
      c = getchar();
      while (c != EOF)
      {
            putchar (c);
            c = getchar();
      }
}
```

Operátor != znamená „nerovná se“.

Hlavním problémem je zjistit, byl-li načten konec vstupu. Obvykle je dáno konvencí, že když funkce getchar narazí na konec vstupu, tak vrací hodnotu, která není normálním platným znakem. Jediný, ale zanedbatelný problém je to, ze existují dvě konvence pro indikaci konce souboru. My jsme se této nepříjemnosti vyhnuli tím, že používáme symbolické jméno EOF pro hodnotu „konec souboru“, ať už je jakákoliv. V praxi je EOF rovna buď -1 nebo 0, a proto musí být na začátku programu definice

```
#define EOF -1
```

nebo

```
#define EOF 0
```

Použitím symbolického jména EOF, které reprezentuje hodnotu funkce getchar pro načtení konce vstupu, jsme si zajistili, že jen jediná řádka v programu závisí na konkrétní číselné hodnotě.

Současně musíme deklarovat proměnou *c* typu int, ne char, aby mohla obsahovat hodnotu, kterou funkce getchar vrací. Jak potom uvidíme v kapitole 2, tato funkce je skutečně typu int, protože musí být schopna vracet nejen znaky, ale také reprezentaci symbolu EOF.

Program pro kopírování může být zkušenějšími programátory v jazyku C napsán stručněji. V tomto jazyku může být každý příkaz, jako např.

```
c = getchar()
```

použit ve výrazu, jehož hodnota je prostě rovna hodnotě, která je přiřazována levé straně výrazu. Jestliže je přiřazení znaku proměnné *c* vloženo na pozici podmínky v příkazu while, tak program pro kopírování souboru může být napsán takto:

```
main      /* kopírování vstupu na výstup 2.verze */
{
      int c;
      while((c = getchar()) != EOF)
            putchar (c);
}
```

Program přečte znak, přiřadí ho proměnné *c* a testuje, zda tento znak byl příznakem konce souboru. Jestliže nebyl, tak tělo cyklu while je vykonáno. Testování se znovu opakuje. Když se narazí na konec souboru, příkaz while, stejně jako funkce main, ukončí činnost.

Tato verze programu soustřeďuje vstup na jedno místo – nyní je už jen jedno volání funkce getchar – a zkracuje text programu. Vložení přiřazovacího příkazu do testovacího výrazu je jednou z možností, kdy jazyk C umožňuje významné zkrácení textu programu. (Je možné se o tuto možnost nestarat a vytvářet „nevnořující se“ text programu, ale tomu se budeme snažit vyhýbat.)

Je důležité si uvědomit, že vložení přiřazovacího příkazu do závorek je opravdu nezbytné. Priorita operátoru != je vyšší než přiřazení = , z čehož vyplývá, že při nepřítomnosti závorek bude příkaz != vykonán dříve než přiřazení = .

Proto příkaz

```
c = getchar() != EOF
```

je shodný s příkazem

```
c = (getchar() != EOF)
```

To má za následek, že proměnné *c* bude přiřazena hodnota 0 nebo 1 podle toho, zda funkce getchar narazila na konec souboru nebo ne (více o této problematice bude uvedeno v kapitole 2).

1.5.3 Počítání znaků

Následující program počítá znaky. Je to úprava programu pro kopírování.

```
main() /*počítání znaků na vstupu*/
{
      long nc;
      nc = 0;
      while (getchar() = EOF)
            ++nc;
      printf("%1d\n",nc);
}
```

Příkazem

```
++nc
```

uvádíme nový operátor ++, který provádí zvětšení o jedničku. Můžeme ovšem rovněž napsat $nc = nc + 1$, ale $++nc$ je stručnější a mnohem efektivnější z výpočetního hlediska. $--$ je obdobný operátor pro odečítání jedničky. Operátory ++ a $--$ se mohou objevit buď před proměnnou ($++nc$) nebo za ní ($nc++$). Tyto dva tvary mají ve výrazech různý význam, jak uvidíme v kap. 2, ale jak $++nc$ tak i $nc++$ zvětšuje proměnnou nc o jedničku. V této chvíli budeme psát tyto operátory před proměnnými.

Program pro počítání znaků sčítá znaky v proměnné nc, která je typu long int (proměnná s dvojnásobným počtem bytů). Maximální možná hodnota, kterou může nabýt proměnná typu int je na počítači PDP-11 32767 a může se proto snadno při počítání znaků přeplnit; na počítačích IBM jsou proměnné typu long a int identické a mnohem větší. Výstupní konverze %1d umožňuje tisk proměnných deklarovaných jako long int.

Chceme-li počítat s mnohem většími čísly, můžeme použít proměnné typu double (proměnných s pohyblivou čárkou, které mají proti proměnným typu float dvojnásobnou délku). Také použijeme příkaz for místo příkazu while, abychom ukázali jiný způsob tvorby cyklu.

```
main()     /*počítání znaků na vstupu*/
{
      double nc;
      for(nc = 0; getchar() != EOF; ++nc)
            ;
      printf("%.0f\n", nc);
}
```

Specifikace %f ve funkci printf je používána jak pro proměnné typu float, tak i typu double. %.0f potlačuje tisk neexistující desetinné části čísla nc.

V tomto programu je tělo cyklu for prázdné, protože veškerá činnost je soustředěna na testování a reinicializaci. Ale gramatická pravidla jazyka C vyžadují, aby tělo cyklu for existovalo. Izolovaný středník zde představuje prázdný příkaz, a tak je pravidlo splněno. V programu jej píšeme na samostatnou řádku, abychom si ho snadněji všimli.

Předtím, než program pro počítání znaku opustíme, všimneme si, že když vstup neobsahuje žádné znaky, tak je podmínka cyklu while nebo for nepravdivá hned při prvním vyhodnocení a proměnná nc je rovna nule, což je správný výsledek. Jednou z příjemných vlastností příkazu while a for je to, že podmínka je testována na začátku cyklu, předtím než je vykonáno tělo cyklu (na rozdíl od jazyka FORTRAN – pozn. překl.). Program počítá správně, i když vstup nenabízí „žádné znaky“.

1.5.4 Počítání řádek

Následující program počítá řádky na vstupu. Předpokládáme, že řádky jsou od sebe odděleny znakem \n.

```
main()          /*počítání řádek*/
{
      int c, nl;
      nl = 0;
      while((c = getchar()) != EOF)
            if(c == '\n')
               ++nl;
      printf("%d\n", nl);
}
```

Tělo cyklu while nyní obsahuje příkaz if, který řídí přírůstek $++nl$. Příkaz if testuje podmínku v závorkách a jestliže je pravdivá, tak vykoná příkaz (nebo skupinu příkazů), které následují. Znovu můžeme snadno ukázat, co je čím řízeno.

Dvojnásobný znak == je v jazyku C zápis podmínky „je rovno“ (jako .EQ. ve FORTRANU). Dvojitý symbol je použit proto, aby jej bylo možno odlišit od přiřazovacího symbolu = . Protože se přiřazovací příkaz v typických programech jazyka C používá zhruba dvakrát častěji než relační operátor pro rovnost, je logické, že má poloviční počet znaků (na rozdíl od ALGOLU nebo PASCALU).

Je-li některý znak psán mezi apostrofy, výsledkem je číselná hodnota tohoto znaku. To se nazývá znakovou konstantou. Tak např. 'A' je znaková konstanta; v ASCII je její hodnota 65, což je vlastně vnitřní reprezentace znaku A. Vždy ale dáváme přednost psaní 'A' před 65: význam je jasnější a nezáleží na místní reprezentaci znaku.

Znaky uvedené za obráceným lomítkem jsou rovněž dovolené znakové konstanty a mohou se vyskytnout v podmínkách i aritmetických výrazech. '\n' se používá místo hodnoty znaku pro novou řádku. Uvědomte si, že '\n' je jeden znak a ve výrazu je roven číslu typu int; na druhé straně "\n" je řetězec znaků, který obsahuje jeden znak. Tato problematika je podrobně popsána v kap. 2.

Cvičení 1-6. Napište program, který počítá mezery, tabelátory a znaky pro novou řádku.
Cvičení 1-7. Napište program, který kopíruje text ze vstupu na výstup a nahrazuje jednu nebo více mezer vždy jen jednou mezerou.
Cvičení 1-8. Napište program, který nahrazuje každý tabelátor řadou tří znaků > tiskne jako a každý zpětný znak obdobnou řadou ←. To nám zviditelní znaky pro tabelátor a backspace.

1.5.5 Počítání slov

Čtvrtý ze série užitečných programů počítá řádky, slova a znaky. Slovo je zde definováno jako skupina znaků, která neobsahuje mezeru, tabelátor nebo znak pro novou řádku. (Tento program je kostrou obslužného programu wc v systému UNIX.)

```
#define  YES   1
#define  NO    0
main()      /*zjištění počtu řádek, slov a znaků
            ze vstupu*/
```

```
{
        int c, nl, nw, nc, inword;
        inword = NO;
        nl = nw = nc = 0;
        while((c = getchar()) != EOF)
        {
                ++nc;
                if(c == '\n')
                        ++nl;
                if(c == ' ' || c == '\n' || c == '\t')
                        inword = NO;
                else if(inword == NO)
                        {
                                inword = YES;
                                ++nw;
                        }
        }
printf( "%d %d %d\n", nl, nw, nc);
}
```

Pokaždé, když program narazí na první znak slova, započítá slovo. Proměnná *inword* indikuje, zda je program právě uprostřed slova nebo ne; na začátku „není uprostřed slova", což je vyjádřeno hodnotou *NO*. Dáváme přednost symbolickým konstantám *YES* a *NO* před hodnotami 1 a 0, protože je program srozumitelnější. Ovšem v tak malém programu, jako je náš, to přináší spíše potíže. Ve větších programech se nám však toto úsilí, jež od začátku vynakládáme, bohatě vrátí. Poznáte také, že lze snadněji dělat rozsáhlé změny v programu, kde se čísla vyskytují pouze jako symbolické konstanty. Řádka

```
nl = nw = nc = 0;
```

vynuluje všechny proměnné. Není to speciální případ, ale výsledek toho, že přiřazování probíhá zprava doleva. Je to jako bychom napsali

```
nc = ( nl = ( nw = 0 ));
```

Operátor || znamená bf nebo a tak řádka

```
if (c == ' ' || c == '\n' || c == '\t')
```

znamená: „jestliže *c* je mezera nebo *c* je znak pro nový řádek nebo *c* je tabelátor …". && je obdobný operátor pro logický součin. Výrazy obsahující operátory || nebo && jsou vyhodnocovány zleva doprava a je zajištěno, že vyhodnocování se ukončí tehdy, jestliže je už známo, že výraz je pravdivý nebo nepravdivý. Proto obsahuje-li proměnná *c* mezeru, není třeba dále testovat zda obsahuje tabelátor či znak pro novou řádku. V krátkém programu to není důležité, ale na významu to nabývá v komplikovanějších situacích, jak brzy uvidíme.

Náš program také ukazuje činnost příkazu else, který určuje alternativní činnost, která má být konána, když podmínka příkazu if není splněna. Obecný tvar je

```
if (podmínka)
      příkaz_1
else
      příkaz_2
```

Za této situace bude vykonán právě jeden ze dvou příkazů spojených s příkazem if. Jestliže je podmínka pravdivá, bude vykonán *příkaz_1*; jestliže ne, bude vykonán *příkaz_2*. Každý příkaz může být ve skutečnosti složen zase z příkazů. V programu pro počítání slov je za else další příkaz if, který podmiňuje další dva příkazy ve složených závorkách.

Cvičení 1-9. Jak budete testovat program pro počítání slov? Jaká jsou omezení?
Cvičení 1-10. Napište program, který tiskne slova ze vstupu vždy na novou řádku.
Cvičení 1-11. Upravte definici slova v programu pro počítání slov. Např. slovo je posloupností písmen, číslic a apostrofů, která začíná písmenem.

1.6 Pole

Napišme program, který bude zjišťovat počet výskytů každého čísla, oddělovačů (tj. mezer tabelátoru a znaku nové řádky) a všech ostatních znaků. Je to uměle vykonstruovaný příklad, ale umožní nám ilustrovat různé vlastnosti jazyka C v jednom programu.

Na vstupu se může objevit dvanáct druhů znaků, a proto spíše než jednotlivé proměnné použijeme s výhodou pole, které bude obsahovat počet výskytů každého čísla. Zde je jedna z verzí programu:

```
main()     /*počítání čísel, oddělovačů, ostatních*/
{
      int c, i, nwhite, nother;
      int ndigit[10];
      nwhite = nother = 0;
      for(i = 0; i < 10; ++i)
            ndigit[i] = 0;
      while((c = getchar()) != EOF)
         if (c >= '0' && c <= '9')
            ++ndigit [c-'0'];
         else if(c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
            else
            ++nother;
      printf ("digits =");
      for (i = 0; i < 10; ++i)
            printf (" %d", ndigit[i]);
      printf ( "\nwhite space = %d, other = %d\n",
            nwhite, nother);

}
```

Popis

```
int ndigit[10];
```

deklaruje pole *ndigit* o délce 10 celočíselných proměných. Indexy pole začínají v jazyku C vždy nulou (oproti FORTRANU nebo PL/1, kde pole začínají indexem 1), a tak prvky pole jsou *ndigit*[0], *ndigit*[1] , ..., *ndigit*[9]. To se projevuje v cyklu for, který inicializuje a tiskne pole *ndigit*.

Indexem pole může být libovolný celočíselný výraz, ve kterém mohou být jen celočíselné proměnné nebo celočíselné konstanty. Tento zvláštní program se spoléhá na znakovou reprezentaci čísel. Např. podmínka

```
if(c >= '0' && c <= '9')
```

určuje, zda je znak *c* číslicí. Jestliže číslicí je, tak numerická hodnota této číslice je

```
c - '0'
```

Tento postup funguje jen tehdy, když '0', '1', ... jsou kladná čísla uspořádaná vzestupně a jestliže mezi '0' a '9' jsou jen číslice. Naštěstí je to splněno pro většinu konvenčních souborů znaků. V aritmetických operacích, kde se vyskytují proměnné typu char a int, jsou před výpočtem proměnné typu char převedeny na celočíselné proměnné a jsou tedy shodné co do obsahu s proměnnými typu int. Je to výhodné a zcela přirozené; např. *c*−'0' je celočíselný výraz, jehož hodnota leží mezi 0 a 9 v odpovídajícím pořadí jako znaky '0' a '9' a je tedy platným indexem pole *ndigit*.

Rozhodnutí, zda je znak číslicí, oddělovačem nebo něčím jiným je dáno posloupností

```
if (c >= '0' && c <= '9')
      ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
      ++nwhite;
else
      ++nother;
```

Konstrukce

```
if (podmínka)
      příkaz
else if (podmínka)
      příkaz
     else
      příkaz
```

se často v programech objevuje jako způsob vícenásobného rozhodování. Příkazy jsou jednoduše vykonávány odshora dolů. Když je některá podmínka splněna, tak je odpovídající příkaz vykonán a sekvence příkazu je ukončena. Příkazem může být samozřejmě několik příkazů ve složených závorkách. Jestliže není žádná z podmínek splněna, příkaz následující poslední příkaz else je vykonán, pokud je ovšem uveden. Jestliže je závěrečný příkaz else a příkaz vynechán (jako je tomu v našem programu počítání slov), neprovede se nic. V této konstrukci může být mezi prvním if a posledním else libovolný počet skupin

```
else if (podmínka)
      příkaz
```

Je moudřejší formovat tyto konstrukce tak, jak jsme ukázali, aby se dlouhý rozhodovací příkaz příliš neblížil pravé straně papíru (používáme-li zvýrazněné logické struktury odsazováním příkazu).

Příkaz switch, který bude popsán v kapitole 3, umožňuje další způsob mnohonásobného rozhodování. Je vhodný pro zjišťování, ze kterého souboru je dané číslo nebo znakový výraz.

Cvičení 1-12. Napište program, který bude tisknout histogram délek načtených slov. Nejsnadnější je nakreslit histogram horizontálně, vertikální uspořádání je složitější.

1.7 Funkce

Funkce jazyka C jsou obdobou podprogramů a funkcí jazyka FORTRAN a procedur v PL/1, PASCALU atd. Funkce umožňuje vhodným způsobem soustředit určité výpočty do „černé skříňky“, kterou můžeme použít, aniž se staráme o to, co je uvnitř. Užití funkcí je jediným způsobem, jak překlenout složitost velkého programu. Máme-li správně nadefinované funkce, tak potom nás nemusí zajímat, jak jsou udělány; stačí vědět co dělají. Jazyk C je navržen tak, aby používání funkcí bylo snadné, výhodné a účinné. Často se setkáme s funkcí, která má jen několik řádek, a je volána jen jednou. Je použita proto, že zpřehledňuje program.

Zatím jsme používali funkce printf, getchar a putchar, které již dříve někdo vytvořil; nyní je čas napsat několik vlastních funkcí. Protože v jazyku C vůbec není operátor mocnění (jako je $**$ ve FORTRANU nebo PL/1), vysvětleme si způsob vytvoření funkce a napišme funkci power(m, n), která umocňuje celé číslo m na celé číslo n. Např. hodnota power(2,5) je 32. Tato funkce nenahrazuje operátor $**$ zcela, protože pracuje pouze s malými celými čísly, ale je vhodné začínat jednoduchou ukázkou.

Následuje funkce power a hlavní program main, který ji prověří, takže je možné vidět celou strukturu programu najednou.

```
      main()          /*testování funkce power*/
      {
            int i;
            for (i = 0; i = 10; ++i)
                  printf("%d %d %d\n", i, power(2,i), power(-
3,i));
      }

      power(x, n)     /*funkce power*/
      int x, n;
      {
            int i, p;
            p = 1;
            for(i = 1; i <= n; ++i)
                  p = p * x;
            return (p);
      }
```

Všechny funkce mají stejnou strukturu:

```
jméno (seznam argumentů, pokud nějaké jsou)
deklarace argumentů, pokud nějaké jsou;
{
      deklarace
      příkazy
}
```

Funkce se mohou umístit v textu programu libovolně, a to jak v jednom, tak ve dvou souborech. Ovšem když je text programu ve dvou souborech, je třeba změnit příkazy pro kompilaci a sestavení, ale to je již záležitost operačního systému a ne jazyka C. Pro začátek budeme předpokládat, že obě funkce jsou v jednom souboru, takže platí vše, co jste se zatím o spouštění programu v jazyku C naučili.

Funkce power je vyvolána v jednom řádku dvakrát

```
printf ("%d %d %d\n", i, power (2,i), power (-3,i));
```

Při každém vyvolání funkce power jí jsou předány dva argumenty a hodnotou funkce je celé číslo, které je zformátováno a vytištěno. Ve výrazu je power$(2, i)$ celé číslo zrovna tak, jako 2 a i (ne všechny funkce vrací celé číslo – blíže o tom v kapitole 4).

Ve funkci power je třeba argumenty nadeklarovat. To je provedeno v řádce

```
int x, n;
```

která následuje za řádkou se jménem funkce. Deklarace argumentů je umístěna mezi seznam argumentů a první složenou závorku; každá deklarace je zakončena středníkem. Jména, která se pro argumenty používají ve funkci power jsou čistě místní a ostatní funkce k nim nemají přístup. To znamená, že ostatní funkce mohou používat stejná jména pro jiné proměnné. To platí i pro proměnné i a p: proměnná i ve funkci power nemá žádný vztah k proměnné i v hlavním programu main.

Hodnota funkce power je vrácena programu main příkazem return stejným způsobem jako v jazyku PL/1. V závorkách se zde může objevit libovolný výraz. Funkce nemusí ale vracet hodnotu; samotný příkaz return předává pouze řízení volající jednotce a nepředává hodnotu.

Cvičení 1-13. Napište program, který převádí písmena ze vstupu na malá písmena a používá při tom funkci lower. Funkce lower vrací c, když c není písmeno, a hodnotu malého písmena c, jestliže je c písmeno.

1.8 Argumenty - volání hodnotou

Jedna z vlastností jazyka C může být některým programátorům, kteří používají FORTRAN, PL/1 atd., neznámá. V jazyce C jsou všechny argumenty (vyjma polí) předávány „hodnotou“. To znamená, že volaná funkce pracuje s dočasnými proměnnými, které jsou kopiemi skutečných argumentů (ve skutečnosti jsou v zásobníku). Tento způsob je odlišný oproti FORTRANU a PL/1, kde jsou argumenty předávány adresou a nikoli hodnotou.

Hlavní rozdíl je v tom, že funkce v jazyku C nemůže měnit hodnoty proměnných volající funkce; může pouze měnit hodnoty vlastních, dočasných proměnných. Volání hodnotou je kladnou stránkou a nikoliv omezením. Obvykle je možné vytvářet kompaktnější programy s menším množstvím přídavných proměnných, protože s argumenty je možno ve volání funkcí nakládat jako s běžnými místními proměnnymi.

Jako příklad uvedeme variantu funkce power, která tohoto faktu využívá.

```
power(x, n)      /*umocnění x na n; n>0; verze 2*/
int x, n;
{
      int p;
      for (p = 1; n > 0; --n)
            p = p * x;
      return (p);
}
```

Argument n je použit jako dočasná proměnná a je zmenšován až do nuly. Již nemusíme používat proměnnou i. Je lhostejné jaké hodnoty proměnná n ve funkci power nabude – nebude to mít žádný vliv na argument, s nímž byla funkce power volána.

Když je potřeba, tak funkce může měnit hodnoty proměnných ve volající jednotce. Volající funkce musí předat adresu proměnné (což se nazývá ukazatel, pointer na proměnnou) a volaná funkce musí tento argument deklarovat jako ukazatel a odkazovat se na proměnnou tímto ukazatelem. Tato problematika bude diskutována v kapitole 5.

Je-li jako argument uvedeno jméno pole, tak předávaná hodnota je ve skutečnosti pozice nebo adresa začátku tohoto pole. (Prvky pole se nekopírují!) Funkce má tedy přístup ke kterémukoliv prvku pole. O tomto budeme hovořit v následujícím odstavci.

1.9 Znaková pole

Pravděpodobně nejrozšířenějšími typy polí v jazyku C jsou znaková pole. Abychom si ilustrovali použití těchto polí a činnost funkcí, které s nimi pracují, napíšeme program, který načítá řádky a vytiskne tu nejdelší. Hrubý diagram je velmi jednoduchý:

```
while   /existuje další řádka/
      if(je delší než předchozí nejdelší)
          uchovej ji a její délku
vytiskni nejdelší řádku
```

Tento diagram objasňuje činnost programu a umožňuje nám rozdělit ho na jednotlivé části. Jedna část načítá řádku, druhá část ji testuje, další uchovává a zbytek řídí činnost.

Protože je diagram úplně rozdělen, bude třeba napsat program stejným způsobem. Napišme tedy nejprve funkci getline, která načítá ze vstupu další řádku; getline je zobecnění funkce getchar. Aby byla funkce getline užitečná i v dalších programech, napišme ji tak univerzálně, jak jen to půjde. Přinejmenším musí funkce getline signalizovat konec souboru; dalším zobecněním bude, aby její hodnota byla rovna délce načtené řádky, nebo nule, byl-li načten konec souboru. Každá řádka má minimálně jeden znak a tak nula není platná hodnota pro délku řádky ani tehdy, obsahuje-li řádka jen znak pro novou řádku (taková řádka má délku 1).

Když narazíme na řádku, která je delší než předcházející nejdelší řádka, tak ji musíme někde uchovat. To bude vykonávat druhá funkce, copy, která bude nejdelší řádku kopírovat na bezpečné místo. A na závěr potřebujeme hlavní program, který bude činnost funkcí getline a copy řídit. Zde je výsledek:

```
#define   MAXLINE 1000  /*max. velikost vstupu řádky*/
main()                      /*nalezení nejdelší řádky*/
{
      int len;                      /*délka běžné řádky*/
      int max;          /*maximální dosažená velikost*/
      char line[MAXLINE];                 /*běžná řádka*/
      char save[MAXLINE];  /*nejdelší uchovaná řádka*/
      max = 0;
      while ((len = getline (line, MAXLINE)) > 0)
      if (len > max)
      {
            max = len;
            copy(line, save);
      }
      if (max>0)                    /*byla načtena řádka*/
            printf("%s", save);
}

getline(s, lim)       /*načti řádku do s, vrať délku*/
char s[];
int lim;
{

      int c, i;
      for(i = 0; i < lim-1 && (c=getchar()) != EOF
                    && c!= '\n'; ++i)
            s[i] = c;
      if (c == '\n')
      {
            s[i]= c;
            ++i;
      }
      s[i] = '\0';
      return (i);
}

copy(s1, s2)      /*kopíruje s1 do s2*/
char s1[], s2[];
{
      int i;
      i = 0;
      while ((s2[i] = s1[i]) != '\0')
            ++i;
}
```

Funkce main a getline komunikují pomocí dvojice argumentů a vrácené hodnoty. V getline jsou argumenty deklarovány na řádcích

```
char s[];
```

```
int lim;
```

které specifikují první argument jako pole a druhý jako celočíselnou proměnnou. Délka pole s není specifikována ve funkci getline, protože je určena v hlavním programu main. getline vrací hodnotu pomocí příkazu return volající jednotce stejným způsobem jako funkce power. Některé funkce vracejí užitečné hodnoty, jiné (jako např. copy) jsou užívány pouze pro to, co dělají a hodnotu nevracejí.

getline vkládá znak \0 (znak, jehož hodnota je nula) na konec pole, které vytváří a označuje tím konec tohoto znakového pole. Stejnou konvenci používá překladač jazyka C; napíšeme-li nějaký znakový řetězec jako

```
"hello \n "
```

překladač vytvoří znakové pole, které obsahuje znaky řetězce a ukončí jej znakem \0, takže fukce jako je např. printf může zjistit konec řetězce

```
h e l l o~\n \0
```

Formátovaná specifikace %s v printf očekává řetězec znaků v tomto tvaru. Když si prostudujete funkci copy, tak objevíte, že se rovněž opírá o fakt, že vstupní řetězec je ukončen znakem \0. Znak \0 se rovněž zkopíruje do výstupního argumentu $s2$. (Předpokládá se, že řetězec sám neobsahuje znak \0).

Stojí za zmínku, že i tak malý program jako je tento, přináší obtíže. Např. co by měl program main dělat, když narazí na řádku, která je delší než zvolené maximum? Funkce getline pracuje správně. Přestane číst znaky m je-li pole plné i když nebyl načten znak nové řádky. Kontrolou délky řádky a testováním posledního znaku může main určit, zda řádka nebyla příliš dlouhá a zachovat se podle toho. Pro jednoduchost jsme se tímto problémem nezabývali.

Neexistuje způsob, jak se při používání funkce getline dozvědět dopředu, jak budou vstupní řádky dlouhé, a proto funkce getline kontroluje přeplnění pole. Na druhé straně uživatel funkce getline ví (nebo může zjistit) jak dlouhé jsou řetězce, a tak jsme kontrolu délky do programu nedávali.

Cvičení 1-14. Upravte hlavní program main v programu pro tisk nejdelší řádky tak, aby správně vytiskl délku libovolně dlouhé vstupní řádky a co nejvíce jejího obsahu.
Cvičení 1-15. Napište program, který bude tisknout všechny řádky delší než 80 znaků.
Cvičení 1-16. Napište program, který vyjme úvodní mezery a tabelátory z každé řádky na vstupu a vymaže prázdné řádky.
Cvičení 1-17. Napište funkci reverse, která převrací znakový řetězec s. Použijte ji v programu, který převrací vstupní řádky.

1.10 Externí proměnné

Proměnné v programu main (*line*, *save*, atd.) jsou soukromé, místní jednotky main, protože v ní jsou deklarovány a žádná jiná funkce k nim přímo nemá přístup. Totéž platí pro proměnné v ostatních funkcích: např. proměnná i ve funkci getline nemá vztah k proměnné i ve funkci copy. Každá místní proměnná funkce je aktivována v okamžiku vyvolání této funkce, a zmizí, když je činnost této funkce ukončena. Je to proto, že tyto proměnné jsou nazývány (jako v ostatních jazycích) automatické proměnné. Od nynějška budeme používat termín automatická proměnná pro tyto dynamické místní proměnné. (V kapitole 4 poznáme statické oblasti paměti, do kterých místní proměnné ukládají své hodnoty mezi vyvoláním funkce.)

Protože automatické proměnné přicházejí a odcházejí spolu s aktivováním funkcí, neuchovávají své hodnoty od jednoho vyvolání do druhého a musí být při každém vyvolání explicitně nastaveny. Jestliže nastaveny nebudou, budou nedefinované a budou obsahovat náhodná čísla.

Na rozdíl od automatických proměnných je možné definovat externí proměnné, které jsou přístupny všem funkcím, tzn. že to jsou globální, veřejné proměnné (což připomíná příkaz COMMON ve FORTRANU nebo EXTERNAL v PL/1). Protože externí proměnné jsou veřejně přístupné, mohou být používány místo seznamu argumentů pro komunikaci mezi funkcemi. Navíc externí proměnné nezanikají s vyvoláním a ukončením funkcí, takže jejich hodnota, která byla nějakou funkcí nastavena, zůstává v paměti.

Externí proměnné musí být definovány mimo funkce; to pro ně zajistí místo v paměti. Navíc ještě musí být v každé funkci, která k nim má mít přístup, deklarovány. To může být učiněno explicitním příkazem extern nebo implicitně dle kontextu. Abychom byli konkrétní, přepišme program pro nalezení nejdelší řádky s proměnnými *line, save* a *max* definovanými jako externí proměnné. To vyžaduje změnit volání funkce, změnit deklarace a upravit všechny funkce.

```
#define MAXLINE 1000       /*max. délka vstupní věty*/
char line[MAXLINE];                    /*vstupní řádka*/
char save[MAXLINE]; /*nejdelší řádka je uchována zde*/
int max;          /*předposlední nejdelší délka řádky*/
main()     /*nalezení nejdelší řádky; zvláštní verze*/
{
      int len;
      extern int max;
      extern char save[];
      max = 0;
      while ((len = getline()) > 0)
            if (len > max)
            {
                  max = len;
                  copy();
            }
      if (max> 0)                  /*byla načtena věta*/
            printf("%s", save);
}

getline()                           /*zvláštní verze*/
{
      int c, i;
      extern char line[];
      for(i = 0; i < MAXLINE-1
      && (c=getchar()) != EOF && c != '/n'; ++i)
            line[i] = c;
      if (c == '/n')
```

```
        {
              line[i] = c;
              ++i;
        }
        line[i] = '/0';
        return (i);
  }

  copy()                              /*zvláštní verze*/
  {
        int i;
        extern char line[], save[];
        i = 0;
        while ((save[i] = line[i]) != '\0')
              ++i;
  }
```

Externí proměnné v jednotkách main, getline a copy jsou definovány prvním řádkem programu. Zde je určen typ těchto proměnných a vyhrazeno místo v paměti. Po syntaktické stránce jsou definice externích proměnných shodné s deklaracemi, které jsme již dříve užívali, ale protože se objevují mimo funkce, tak jsou chápány jako externí proměnné. Předtím, než funkce může používat externí proměnnou, musí znát její jméno. Jeden ze způsobů je napsat deklaraci extern uvnitř funkce; deklarace je stejná s tím, že na začátku je slovo extern.

Za jistých okolností může být deklarace extern vynechána. Jestliže se objeví ve zdrojovém textu před použitím určité funkce, tak není nutné použít deklarace extern uvnitř této funkce. V našem příkladu jsou tedy deklarace extern v jednotkách main, getline, copy nadbytečné. Je rozšířenou praxí umístit definice všech externích proměnných na začátek souboru a nikde již nepoužívat příkaz extern.

Jestliže je ale program rozdělen do několika souborů a proměnná je definována řekněme v souboru file1 a používána v souboru file2, potom je deklarace extern ve file2 nezbytná. To bude ukázáno v kapitole 4.

Všimněte si, jak opatrně používáme výrazy deklarace a definice, když mluvíme v této sekci o externích proměnných. „Definice“ určuje místo, kde bude externí proměnná vytvořena (tj. je jí přiděleno místo v paměti). „Deklarace“ poukazuje na místo, kde je proměnná uložena, ale není jí přitom přidělena paměť.

Zdálo by se, že je výhodné veškerou komunikaci mezi funkcemi omezit na používání externích proměnných – seznamy argumentů jsou krátké a proměnné jsou vždy tam, kde je očekáváme. Ale externí proměnné jsou tam i tehdy, když je nepotřebujeme. Tento způsob programování je ale plný nebezpečí, protože vede k programům, kde je vzájemná komunikace nejasná. Za prvé proměnné mohou být neočekávaně z nedopatření změněny a za druhé není snadné program modifikovat. Druhá verze programu je horší než první částečně kvůli smyslu a částečně proto, že zničila univerzálnost dvou užitečných funkcí, když je připoutala k určitým názvům proměnných.

Cvičení 1-18. Podmínka v příkazu for ve funkci getline je poněkud neohrabaná. Přepište program tak, aby byla jasnější a přitom zachovávejte činnost funkce, narazí-li na konec dat nebo při přeplnění pole.

1.11 Shrnutí

V této chvíli jsme probrali to, co může být chápáno jako základy jazyka C. Se stavebními bloky, které jsme sestrojili, můžeme nyní psát užitečné programy rozumné velikosti a bude jistě dobře, když si je procvičíte před dalším studiem. Následující příklady by vám měly dát náměty pro složitější programy, než jsou popsány v této kapitole.

Cvičení 1-19. Napište program detab, který nahrazuje tabelátory ze vstupu odpovídajícím počtem mezer. Předpokládejte, že tabelátory jsou na každé n-té pozici.
Cvičení 1-20. Napište program entab, který nahrazuje řetězec mezer minimálním počtem tabelátorů a mezer. Užívejte stejné pozice tabelátoru jako ve funkci detab.
Cvičení 1-21. Napište program, který rozděluje dlouhé řádky ze vstupu po posledním nemezerovém znaku, který se objeví před n-tým sloupcem vstupu, kde n je parametr. Ujistěte se, že váš program funguje s velmi dlouhým řádkem, nebo jestliže nejsou žádné mezery nebo tabelátory před n-tou pozicí.
Cvičení 1-22. Napište program, který vyjímá všechny komentáře z programu v jazyce C. Neopomeňte správně používat znakové řetězce a znakové konstanty.
Cvičení 1-23. Napište program, který kontroluje základní symboly programu jako jsou neuzavřené kulaté, hranaté a složené závorky, nezapomeňte na apostrofy, uvozovky a komentáře. Tento program je složitý, jestliže jej plně zobecníte.

Kapitola 2

Typy, operátory a výrazy

Proměnné a konstanty jsou základní datové objekty, se kterými se v programu pracuje. V deklaracích jsou vyjmenovány proměnné, které budou používány a je určen jejich typ a někdy i jejich počáteční hodnoty. Operátory určují, jak s nimi bude naloženo. Výrazy obsahují proměnné a operátory a výsledkem je nová hodnota. To je předmětem této kapitoly.

2.1 Názvy proměnných

Přestože jsme si to ještě neřekli, tak pro názvy proměnných a symbolických konstant platí jistá omezení. Názvy se skládají z písmen a číslic, první znak musí být písmeno. Podtržení „_“ je chápáno jako písmeno: je užitečné pro zpřehlednění dlouhých jmen některých proměnných. Velká a malá písmena se od sebe liší: v jazyku C je tradicí používat malá písmena pro jména proměnných a velká písmena pro symbolické konstanty.

Jen prvních osm znaků jmen je významných, ačkoliv jich může být použito více. Pro externí názvy, jako jsou jména funkcí a proměnných, musí být někdy počet znaků menší, protože jsou tato jména používána různými assemblery a sestavovacími programy. Podrobnosti jsou v dodatku A. Klíčová slova, jako jsou `if`, `else`, `int`, `float`, atd. jsou rezervována a nelze je používat pro jména proměnných (musí být psána malými písmeny).

Je přirozeně výhodné volit jména proměnných tak, aby byl jasný jejich význam a aby nebylo snadné je navzájem zaměnit.

2.2 Typy dat a jejich délka

V jazyku C je jen několik základních typů dat:

char — jeden byte, může obsahovat jeden znak místního souboru znaků

int — celočíselná proměnná, délka odpovídá délce celých čísel na daném počítači

float — číslo s pohyblivou desetinou čárkou a jednoduchou přesností

double — číslo s pohyblivou desetinou čárkou a dvojnásobnou přesností

Navíc je ještě několik kvalifikátorů, které mohou být aplikovány na typ `int`: `short`, `long` a `unsigned`. `short` a `long` odpovídají různým délkám celočíselných proměnných. Čísla unsigned splňují pravidla aritmetické funkce modulo $2**n$, kde n je počet bitů proměnné int. Tato čísla jsou vždy kladná. Deklarační příkazy vypadají takto

```
short int x;
long int y;
unsigned int z;
```

Slovo int může být vynecháno a obyčejně v takových případech nebývá. Přesnost těchto typů proměnných záleží na konkrétním typu počítače. V tabulce jsou uvedeny některé typy.

Typ	DEC PDP-11 ASCII	Honeywell 6000 ASCII	IBM 370 EBCDIC	Interdata 8/32 ASCII
char	8 bitů	9 bitů	8 bitů	8 bitů
int	16	36	32	32
short	16	36	16	16
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

short a long by měly být různě dlouhé tam, kde je to výhodné. int obyčejně představuje „přirozenou“ délku pro daný typ počítače. Jak vidíte, každý překladač jazyka C interpretuje délku long a short různě v závislosti na typu počítače. Jediné, s čím můžete počítat je, že short není delší než long.

2.3 Konstanty

Konstanty typu int a float jsme již používali, takže upozorněme na nezvyklé tvary

```
123.456e-7
```

nebo

```
0.12E3
```

Tento tvar konstant se nazývá „vědecká“ notace. Každá konstanta s pohyblivou řádovou čárkou je chápána jako double, takže notace „e“ slouží jak pro float, tak pro double.

Dlouhá celá čísla se píší s příponou L. Obyčejná celočíselná konstanta, která je delší než short je rovněž chápána jako long.

Existuje také možnost zápisu oktalových a hexadecimálních konstant: je-li na první pozici konstanty 0 (nula), tak je konstanta brána jako oktalová; jsou-li na začátku znaky 0x nebo 0X, tak se jedná o hexadecimální proměnnou. Např. dekadické číslo 31 může být psáno oktalově jako 037 a hexadecimálně 0xlf nebo 0X1F. Oktalové a hexadecimální konstanty mohou mít rovněž příponu L pro označení typu long.

Znaková konstanta je jeden znak napsaný v apostrofech, např. 'x'. Hodnota znakové konstanty je numerická hodnota tohoto znaku v souboru znaků daného počítače. Např. ve znakovém souboru ASCII má znak nula, nebo '0' hodnotu 48 a v EBCDIC '0' je 240. Obojí je velice odlišné od číselné hodnoty 0. Napíšeme-li v programu '0' místo 48 nebo 240, tak učiníme program nezávislým na konkrétní reprezentaci. Znaková konstanta se může zúčastnit číselných operací jako ostatní čísla, přestože se používá spíše pro porovnávání s ostatními znaky. V následující části se hovoří o konverzních pravidlech.

Určité netisknutelné znaky mohou být reprezentovány sekvencí, která začíná obráceným lomítkem (např. \n (znak nové řádky), \t (tabelátor), \0 (prázdný znak), \\ (obrácené lomítko), \' (apostrof) atd.) a vypadá jako dva znaky, ale ve skutečnosti to je jen jeden znak. Navíc může být libovolný vzor bitů v bytu generován sekvencí:

```
'\ddd'
```

kde ddd jsou jedno až tři oktalová čísla, jako např.

```
#define   FORMFEED   '\014' (# ASCII Form Feed)
```

Znaková konstanta '\0' představuje znak s nulovou hodnotou, kterou často píšeme místo 0, abychom zdůraznili vlastnost určitého výrazu.

Konstantní výraz je výraz, který obsahuje pouze konstanty a operátory. Takový výraz je vyhodnocován již ve fázi překladu a ne při běhu programu a může být použit všude tam, kde se může objevit konstanta, jako např.

```
#define   MAXLINE   1000
char line  [MAXLINE + 1];
```

nebo v

```
seconds = 60*60*hours;
```

Znaková konstanta je sekvence několika znaků jako např.

```
"I am a string"
""            /*prázdný řetězec*/
```

Uvozovky nejsou součástí řetězce, slouží jenom k jeho omezení. Sekvence s obráceným lomítkem může být součástí řetězce: např. \" reprezentuje znak uvozovky.

Z technického hlediska je řetězec pole, jehož prvky jsou jednotlivé znaky. Překladač automaticky ukončuje každý řetězec prázdným znakem \0, takže program může snadno detekovat konec řetězce. Tato reprezentace tudíž nijak neomezuje délku řetězce, ale program musí celé pole projít, aby určil jeho délku. Skutečná paměť potřebná pro řetězec je o jednu pozici delší, což je faktická délka řetězce. Následující funkce strlen (s) určuje délku řetězce s a nebere při tom v úvahu znak \0.

```
strlen(s)      /*zjištění délky řetězce*/
char s[];
{
      int i;
      i = 0;
      while (s[i] != '\0')
            ++i;
      return (i);
}
```

Buďte opatrní při rozlišování znakové konstanty a řetězce, který obsahuje jeden znak: 'x' není totéž jako "x". 'x' je jeden znak, který je v tomto tvaru číselnou reprezentací znaku x v souboru znaků počítače. "x" je znakový řetězec, který obsahuje znak x (písmeno x) a znak \0.

2.4 Deklarace

Všechny proměnné musí být před použitím deklarovány, ačkoliv některé deklarace mohou být podle kontextu implicitní. Deklarace specifikuje typ a za ním následuje seznam jedné nebo více proměnných tohoto typu, jako např.:

```
int lower, upper, step;
char c, line[1000];
```

Proměnné mohou být umístěny v deklaracích v libovolném pořadí, takže uspořádání může být i následovné:

```
int lower;
int upper;
int step;
char c;
char line[1000];
```

Tento způsob zápisu zabírá sice více místa textu programu, ale je výhodný, chceme-li ke každé proměnné přidat komentář nebo chceme-li provádět modifikace.

Proměnné mohou být rovněž v deklaracích inicializovány. Inicializace se provádí tímto způsobem:

```
char backslash = '\\';
int i = 0;
float eps = 1.0e-5;
```

Jestliže je inicializovaná proměnná externí nebo statická, inicializace je provedena jen jednou, a to před začátkem činnosti programu. Automatické proměnné jsou inicializovány vždy, když jsou funkce vyvolány. Automatické proměnné, které nejsou explicitně definovány, mají nedefinovanou hodnotu. Externí a statické proměnné mají implicitně nulovou hodnotu, ale je dobrým stylem je stejně inicializovat. Podrobněji se problémem inicializace budeme zabývat dále.

2.5 Aritmetické operátory

Binární aritmetické operátory jsou +, -, $*$, / a operátor modulo %. Existuje unární $-$ ale neexistuje unární +. Celočíselné dělení x/y odřezává desetinnou část. Výraz

```
x % y
```

produkuje zbytek po dělení x%y a je nula tehdy, když x je dělitelné y. Např. rok je přechodný, když je dělitelný 4 a není dělitelný 100. Roky dělitelné 400 jsou také přechodné. Proto

```
if(year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
      je přechodný rok
else
      není přechodný rok
```

Operátor % nemůže být použit pro proměnné float nebo double.

Operátory + a − mají stejnou prioritu, která je nižší než priorita operátoru ∗, / a %. Unární minus má prioritu nejvyšší. Aritmetické operátory stejné priority se vykonávají zleva doprava (na konci kapitoly je uvedena tabulka, která shrnuje prioritu a asociativnost všech operátorů). Sled vyčíslování není určen pro asociativní a komutativní operátory jako jsou + a -. Překladač proto může výrazy s těmito operátory přeskupit. Proto výraz $a+(b+c)$ může být vyčíslen ve tvaru $(a+b)+c$. Druhý způsob je výhodnější, protože není potřeba používat pomocnou proměnnou.

2.6 Relační operátory

Relační operátory jsou

```
> >= < <=
```

Všechny mají stejnou prioritu. Nižší a navzájem shodnou prioritu mají operátory rovnosti a nerovnosti.

```
== !=
```

Relační operátory mají nižší prioritu než aritmetické operátory, takže výrazy jako $i < \lim - 1$ jsou chápány jako $i < (\lim - 1)$, jak může být nakonec očekáváno.

Dalšími operátory jsou && a ||. Výrazy s těmito operátory jsou vyhodnocovány zleva doprava a vyhodnocování skončí tehdy, je-li zřejmé, že výraz je pravdivý nebo nepravdivý. Tuto vlastnost si je třeba uvědomit při psaní programu. Podíváme-li se na příkaz cyklu funkce getline, kterou jsme vytvořili v kap. 1

```
for(i = 0; i < lim-1 && (c = getchar()) != '\n' &&
       c != EOF; ++i)  s[i] = c;
```

je zřejmé, že test $i < \lim - 1$ **musí** být vykonán dříve než načteme další znak. A ne jenom to. Když totiž není tato podmínka splněna, **nesmíme** číst další znak.

Obdobně budeme v nesnázích, pokusíme-li se porovnávat znak c s EOF předtím, než zavoláme getchar. Volání se musí objevit předtím, než testujeme konec souboru.

Priorita operátoru && je vyšší než priorita operátoru ||. Oba tyto oparátory mají nižší prioritu než relační operátory a operátory rovnosti a nerovnosti, takže výraz:

```
i < lim-1 && (c = getchar()) != '\n' && c != EOF
```

nepotřebuje další závorky, protože priorita != je vyšší než přiřazení, závorky jsou nutné v

```
(c=getchar()) != '\n'
```

abychom dosáhli požadovaného efektu.

Unární operátor negace ! převádí nenulové, neboli pravdivé operátory, na 0, a nulové, nebo-li nepravdivé operátory, na 1. Rozšířeným použitím operátoru ! je výraz jako

```
if (!inword)
```

spíše než

```
if (inword == 0)
```

Je těžké stanovit obecná pravidla, která forma je lepší. Výraz jako !*inword*, lze snadno číst („jestliže není slovo“), ale komplikovanějším výrazům je někdy těžké porozumět.

Cvičení 2-1. Napište příkaz cyklu ekvivalentní cyklu for v předchozí ukázce bez užití &&.

2.7 Konverze typů

Jestliže se ve výrazech objevují proměnné různých typů, tak jsou konvertovány na společný typ, podle několika pravidel. Automaticky jsou prováděny pouze konverze, které mají smysl, jako je konvertování čísla typu int na typ float ve výrazu $f+i$. Výrazy, které nemají smysl, jako např. užití float jako indexu, nejsou povoleny. Typy char a int mohou být v aritmetických výrazech libovolně promíchány; každá proměnná char je konvertována na int. To umožňuje dostatečnou flexibilitu v určitých druzích znakové transformace. To je užito ve funkci atoi, která konvertuje řetězec číslic na číselný ekvivalent:

```
atoi(s)     /*konvertování s na celá čísla*/
char s[];
{
  int i, n;
  n = 0;
  for(i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
     n = 10 * n + s[i] - '0';
  return(n);
}
```

Jak bylo ukázáno v kapitole 1, výraz:

```
s[i] - '0'
```

je numerickou hodnotou znaku uloženého v $s[i]$, protože hodnoty znaků '0', '1', ... tvoří vzestupnou, nepřerušenou řadu kladných čísel.

Jiným příkladem konverze char na int je funkce lower, která převádí velká písmena na malá (platí pouze pro ASCII!). Jestliže znak není velké písmeno, vrací jej funkce lower nezměněn.

```
lower(c) /*převod c na malé písmeno, pouze pro ASCII*/
int c;
{
      if(c >= 'A' && c <= 'Z')
            return(c + 'a' - 'A');
      else
            return(c);
}
```

Tato funkce může pracovat pouze se znakovým souborem ASCII, protože velká a malá písmena mají od sebe konstantní vzdálenost, číselné hodnoty obou abeced jsou ve vzestupném pořadí a mezi 'A' a 'Z' není nic jiného než písmena. Poslední podmínka neplatí pro EBCDIC (IBM 360/370).

Jeden problém je ale s konverzí velkých písmen na malá spojen. V jazyku C není definováno, zda je proměnná char se znaménkem nebo bez něho. Konvertujeme-li char na int, bude výsledek **kladný** nebo **záporný**? Naneštěstí se to liší podle typu počítače. Na některých (např. PDP-11) je proměnná char, jejíž levý bit je 1, konvertována na záporné celé číslo (rozšíření znaménka). Na jiných počítačích je char doplněn zleva nulami a výsledek je proto kladný.

Definice jazyka C garantuje, že libovolný znak v souboru znaků počítače nebude nikdy záporný, takže tyto znaky mohou být volně použity ve výrazech jako kladná čísla. Naproti tomu proměnná, které jsme přiřadili nějaký bitový obrazec, může být na některých typech počítačů záporná, na jiných kladná.

Často se tato situace vyskytuje v případě je-li pro EOF použita hodnota -1. Uvažme případ:

```
char c;
c = getchar();
if (c == EOF)
  ...
```

Na počítačích, které nemají rozšíření znaménka, je c vždy kladné, protože je char zatímco EOF je záporné. Výsledkem je to, že test $c ==$ EOF nemůže být nikdy pravdivý. Abychom se tomuto problému vyhnuli musíme pro proměnnou, která obsahuje hodnotu danou funkcí getchar, deklarovat místo char typem int.

Pravý důvod pro užití int místo char nemá co dělat s otázkou pravděpodobného rozšíření znaménka. Je to proto, že funkce getchar musí vracet všechny možné znaky (aby mohla být použita pro čtení libovolného vstupu) a navíc odlišné hodnoty pro EOF. Proto proměnná, které je funkce přiřazena, **nesmí** být typu char, ale musí být definována jako int.

Jinou užitečnou formou automatické konverze typu je to, že relace jako $i > j$ a logické výrazy s operátory && a || mají hodnotu 1 jsou-li pravdivé a 0 nejsou-li pravdivé; proto přiřazení:

```
isdigit = c >= '0' && c <= '9';
```

nastavuje proměnnou *isdigit* na 1, je-li c číslice, a na 0, není-li číslicí. (V podmínkách příkazů if, while, for atd. „pravdivý“ znamená „nenulový“.)

Implicitní aritmetické konverze fungují převážně tak, jak očekáváme. Obecně řečeno, v operacích s binárními operátory ($+$, $*$ atd.) je operand „nižšího“ typu konvertován na „vyšší“ předtím, než dojde k vyčíslení. Výsledek je rovněž vyššího typu. Pro každý aritmetický operátor platí následující soubor pravidel:

— char a short jsou konvertovány na int, float je konvertováno na double,

— jestliže je nějaký operand double, jsou ostatní operandy konvertovány na double a výsledek je double. Jinak, jestliže je nějaký operand long, ostatní jsou konvertovány na long a výsledek je long,

— jestliže je nějaký operand unsigned, ostatní jsou konvertovány na unsigned a výsledek je unsigned. Jinak operandy musí být int a výsledek je int.

Uvědomte si, že všechny proměnné float jsou konvertovány na double; v jazyku C se uplatňuje jedině aritmetika s dvojnásobnou přesností.

Konverze se rovněž uplatňují při pořizování; hodnota pravé strany je konvertována na typ proměnné na levé straně, což je typem výsledku. Znak je konvertován na celé číslo, ať už platí pravidlo rozšíření znaménka nebo ne, jak už bylo psáno dříve. Opačné přiřazení — int na char je bez problému. Přebytečné bity vyšších řádů jsou vypuštěny. Proto v

```
int i;
char c;
i = c;
c = i;
```

je hodnota c nezměněná, ať již platí pravidlo rozšíření znaménka nebo ne. Jestliže je x float a i je int, potom jsou výrazy

```
x = i
```

a

```
i = x
```

konvertovány; float na int odřezává desetinnou část, double na float je konvertováno zaokrouhlením, long jsou konvertovány na int nebo char odříznutím horních bitů.

Protože argument funkce je výraz, je rovněž prováděna konverze při předávání: char a short na int, float na double. Proto tedy deklarujeme argumenty funkce jako int a double, i když potom funkci voláme s char nebo float.

Na závěr budiž řečeno, že konverze lze provádět rovněž explicitně způsobem zvaným **vlastnost**. V konstrukci

```
(název typu) výraz
```

je **výraz** konvertován na vyjmenovaný typ podle výše uvedených pravidel. Přesný význam vlastnosti je v tom, že hodnota výrazu je přiřazena proměnné specifikovaného typu a ta je potom použita místo celého výrazu. Např. argument knihovní funkce sqrt je proměnná typu double a jestliže vyvoláme tuto funkci s argumentem jiného typu, dostaneme nesmyslný výsledek. Proto, je-li n celé číslo, tak

```
sqrt((double)n)
```

převádí n na double předtím, než je argument předán funkci sqrt (uvědomte si, že vlastnost vytváří **hodnotu** n správného typu; skutečný obsah proměnné n je nezměněn). Operátor vlastnosti má stejnou prioritu jako ostatní známé operátory, jak je shrnuto v tabulce na konci této kapitoly.

2.8 Operátory přičtení a odečtení jedničky

V jazyce C jsou dva nezvyklé operátory pro přírůstek a zmenšení proměnných. Operátor přírůstku ++ přidává ke svému operandu 1; operátor zmenšení ubírá 1. Operátor ++ jsme často užívali pro zvětšování proměnných, jako např.

```
if(c == '\n')
   ++nl;
```

Neobvyklým rysem je to, že ++ a −− mohou být použity buď před operandem ($++n$) nebo za operandem ($n++$). V obou případech je operand zvětšen o 1. Výraz $++n$ zvětšuje n **před** použitím jeho hodnoty a $n++$ zvětšuje n **potom**, co jeho hodnota byla použita. To znamená, že v kontextu, kde má být použita hodnota proměnné, jsou konstrukce $++n$ a $n++$ odlišné. Jestliže n je 5 potom po

```
x = n++;
```

je x=5, zatímco po

```
x = ++n;
```

je x=6. Operátory ++ a −− mohou být aplikovány pouze na proměnné.

V případech, že nepotřebujeme ve výrazech používat hodnotu proměnné, jako v

```
if(c == '\n')
   n1++;
```

si můžeme zvolit buď $++nl$, nebo $nl++$. Výrazy jako $x=(i+j)++$ jsou nepřípustné.

Jsou ovšem situace, kdy potřebujeme používat právě jednu z těchto možností. Podívejme se např. na funkci squeeze(s, c), která vymazává znak c z řetězce s.

```
sgueeze(s, c)  /*vymaže všechna c ze s*/
char s[];
int c;
{
   int i, j,
   for(i = j = 0; s[i] != '\0'; i++)
      if (s[i] != c)
            s[j++] = s[i];
   s[j] = '\0';
}
```

Každý znak, který není písmenem c je kopírován na pozici j, a tehdy je také proměnná zvětšena o jedničku a je připravena na další znak. To je ekvivalentní příkazům

```
if(s[i] != c)
{
   s[j] = s[i];
   j++;
}
```

Jiným příkladem je část funkce getline, kterou jsme vytvořili v kapitole 1, kde můžeme příkazy

```
if (c == '\n')
{
   s[i] = c;
   ++i;
}
```

přepsat kompaktněji takto:

```
if (c == '\n')
   s[i++] = c;
```

Ve třetím příkladě funkce strcat(s, t) připojuje řetězec t na konec řetězce s. Předpokládáme, že v řetězci s je dostatek místa pro výsledek:

```
strcat(s,t)             /*připojení t na konec s*/
char s[], t[];     /*s musí být dostatečně velké*/
{
   int i, j;
   i = j = 0;
   while(s[i] != '\0')        /*nalezení konce s*/
      i++;
   while((s[i++] = t[j++]) != '\0')  /*kopíruj t*/
      ;
}
```

Když jsou znaky kopírovány z t do s, tak operátor ++ za proměnnými i, j připravuje bezprostředně kopírování dalšího znaku.

Cvičení 2-3. Napište modifikaci funkce squeeze($s1$, $s2$), která zruší všechny znaky v $s1$, které jsou shodné se znaky v řetězci $s2$.
Cvičení 2-4. Napište funkci any($s1$, $s2$), která vrátí pozici prvního výskytu řetězce $s2$ v řetězci $s1$; nebo -1, jestliže $s1$ neobsahuje $s2$.

2.9 Bitové logické operátory

V jazyku C jsou k dispozici operátory pro manipulaci s bity. Tyto operátory nemohou být aplikovány na proměnné float nebo double.

&	bitové AND
|	bitové OR
ˆ	bitově exkluzívní OR
<<	posun doleva
>>	posun doprava
‘	jedničkový doplněk

Bitové AND, &, je často používáno pro vynulování určitých bitů. Např.:

```
c = n & 0177;
```

vynuluje všechny bity proměnné n kromě nejnižších 7 bitů. Bitové OR, |, je používáno pro nastavení bitu

```
x = x | MASK;
```

nastaví v x na jedničku ty bity, které jsou jednotkové v $MASK$.

Z opatrnosti rozlišujte bitové operátory & a | od logických operátorů && a ||, které provádějí vyhodnocování pravdivosti zleva doprava. Např. jestliže x je 1 a y je 2, potom x & y je nula, zatímco x && y je jedna (proč?).

Operátory posunu << a >> provádějí posun bitů doleva nebo doprava v levém operandu o počet bitů, udaných operandem vpravo. Proto x « 2 posouvá bity proměnné x o dvě pozice doleva a vyplňuje prázdná místa nulami – to odpovídá násobení číslem 4. Posun doprava proměnné bez znaménka plní prázdná místa nulami. Posun doprava proměnné se znaménkem vyplní prázdná místa na některých počítačích (jako je PDP-11) jedničkovým bitem (aritmetický posun), na jiných nulami (logický posun).

Výsledkem unární operace operátorem ‘ je doplněk celého čísla; to znamená, že převádí bity 0 na 1 a naopak. Tyto operátory se obvykle používají ve výrazech jako

```
x & ‘077
```

který nuluje posledních šest bitů. Uvědomte si, že operace x & ‘077 je nezávislá na délce slova a proto je lepší ji používat místo operace x & 0177700, která předpokládá, že x je šestnáctibitové. Výraz ‘077 nestojí při výpočtu žádný čas navíc, protože je jako konstantní vyčíslován již při překladu.

Abychom si ilustrovali použití některých bitových operátorů, napíšeme funkci getbits (x, p, n), která vrací pole n-bitů proměnné x, začínajících na pozici p. Předpokládejme, že bitová pozice 0 je vpravo a že n a p mají rozumné kladné hodnoty. Např. getbits(x, 4, 3) vrací tři bity na pozicích 4, 3 a 2 zarovnané vpravo.

```
getbits( x, p, n)   /*získání n bitů od pozice p*/
unsigned x
{
      return((x >> (p+1-n)) & ‘(‘0 << n));
}
```

x >>(p+1-n) přesouvá požadované pole bitů na pravý konec slova. Protože jsme deklarovali x jako unsigned (bez znaménka), tak prázdná místa jsou vyplněna nulami a nezáleží na typu počítače, který používáme. ‘0 je proměnná, která má všechny bity 1; posun o n bitů doleva příkazem ‘0<< n vytváří masku s nulami na místě pravých n bitů a jedničkami všude jinde; komplement této masky má n pravých bitů rovných 1.

Cvičení 2-5. Modifikujte getbits pro číslování bitů zleva doprava.
Cvičení 2-6. Napište funkci wordlength(), která zjišťuje délku slova použitého počítače, tj. počet bitů proměnné int. Funkce by měla být přenosná, tzn. že by měla být použitelná na všech druzích počítačů.
Cvičení 2-7. Napište funkci rightrot(n, b), která rotuje celé číslo n o b bitů doprava.
Cvičení 2-8. Napište funkci invert(x,p,n), která invertuje (tzn. mění 0 na 1 a opačně) n bitů proměnné x od pozice p a ostatní nechává beze změny.

2.10 Operátory přiřazení a výrazy

Výrazy

```
i = i + 2
```

ve kterých se levá strana opakuje na straně pravé, mohou být napsány zhuštěným způsobem

```
i += 2
```

použitím **přiřazovacího operátoru** jako += . Většina binárních operátorů má odpovídající přiřazovací operátor *op*=, kde *op* je jeden z operátorů

```
+  -  *  /  %  <<  >>  &  ^  |
```

Jestliže (*e1*) a (*e2*) jsou výrazy, potom

```
e1 op= e2
```

je ekvivalentní

```
e1 = (e1) op (e2)
```

s tou výjimkou, že (*e1*) je vyčíslováno jen jednou. Uvědomte si, že *e2* je v závorkách. To znamená, že

```
x *= y + 1
```

je

```
x = x * (y + 1)
```

a ne

```
x = x * y + 1
```

Jako příklad uvedeme funkci bitcount, která počítá jedničkové bity:

```
bitcount(n)   /*počet jedničkových bitů v n*/
unsigned n;
{
   int b;
   for(b = 0; n != 0; n >>= 1)
      if  (n & 01)
             b++;
   return(b);
}
```

Přiřazovací operátory mají výhodu hlavně v tom, že jsou bližší myšlení člověka. Říkáme spíše „přidej 2 k i“ nebo „zvětši i o 2“ než „vezmi i, přidej 2 a výsledek vlož do i“. Proto i += 2. Navíc při složitých výrazech jako

```
yyal[yypv[p3+p4]+yypv[p1+p2]] += 2
```

je tomuto zápisu lépe rozumět, protože čtenář nemusí otrocky kontrolovat, zda výrazy na obou stranách jsou shodné nebo ne. Navíc přiřazovací operátor umožňuje překladači generovat efektivnější kód.

Již dříve jsme využívali skutečnosti, že přiřazovací příkaz má hodnotu, kterou lze používat ve výrazech. Typickým příkladem je

```
while (c=getchar()) != EOF)
```

Přiřazení užívající operátor přiřazení (+=, − = , adt.) se rovněž může objevit ve výrazech, přestože je to méně obvyklé.

Typ operátoru přiřazení je shodný s typem na levé straně.

Cvičení 2-9. V systému s dvojkovým doplňkem výraz x &(x-1) nuluje pravý bit proměnné x (proč?). Použijte této skutečnosti a napište rychlejši verzi funkce bitcount.

2.11 Podmíněné výrazy

Příkaz

```
if(a > b)
  z = a;
else
  z = b;
```

počítá zajisté maximum z čísel a a b. **Podmíněný výraz** psaný s operátorem "?:", umožňuje jiným způsobem napsat podobnou konstrukci. Ve výrazu

```
e1 ? e2 : e3
```

je výraz $e1$ spočítán nejdříve. Jestliže je nulový (pravdivý), potom je vyčíslen výraz $e2$, který je hodnotou podmíněného výrazu. Jinak je vyčíslen výraz $e3$, který je potom hodnotou výrazu. Jenom jeden z výrazů $e2$ a $e3$ je vyhodnocen. Proto výpočet maxima z a a b můžeme napsat

```
z = (a > b) ? a : b;      /*z=max(a,b)*/
```

Dlužno poznamenat, že podmíněný výraz je skutečný výraz a může být použit jako jiné výrazy. Jestliže jsou $e2$ a $e3$ odlišného typu, potom je výsledek zkonvertován podle pravidel uvedených v této kapitole. Je-li např. f typu float a n je int, potom výraz

```
(n > 0) ? f : n
```

je typem double ať již je n kladné nebo ne. Závorky kolem prvního výrazu nejsou nezbytné, protože priorita operátoru ?: je velmi malá, právě vyšší než přiřazení. Je však vhodné je pro přehlednost psát.

Podmíněné výrazy vedou k velmi hutnému kódu. Např. tento cyklus tiskne N prvků pole, 10 na řádku, oddělené jednou mezerou a každá řádka (i poslední) je ukončena přesně jedním znakem pro novou řádku.

```
for(i = 0; i < N; i++)
   printf("6d%c", a[i], (i%10==9 || i==N-1) ? '\n' : '
');
```

Znak pro novou řádku '\n' je tištěn vždy po desátém prvku a po N – tém prvku. Ostatní prvky jsou následovány mezerou. Ačkoliv tento příklad může vypadat nepoctivě, je vhodné zkusit si jej napsat bez podmíněného výrazu.

Cvičení 2-10. Přepište funkci lower, která konvertuje velká písmena na malá s užitím podmíněného výrazu místo if- else.

2.12 Priorita a pořadí vyhodnocování

Níže uvedená tabulka shrnuje pravidla priority a asociativnosti všech operátorů včetně těch, se kterými jsme se ještě neseznámili. Operátory na jedné řádce mají stejnou prioritu; řádky jsou seřazeny sestupně, tak např. $*$, $/$ a $\%$ mají stejnou prioritu vyšší než $+$ a $-$.

```
Operátor                          Asociativnost

() [] -> .                        zleva doprava
| - ++ -- ‘ (typ) * &             zprava doleva
* / %                             zleva doprava
+ -                               zleva doprava
<< >>                             zleva doprava
< <= > >=                         zleva doprava
== !=                             zleva doprava
&                                 zleva doprava
^                                 zleva doprava
!                                 zleva doprava
&&                                zleva doprava
||                                zleva doprava
? :                               zprava doleva
= += -= atd.                      zprava doleva
, /kapitola 3/                    zleva doprava
```

Operátory $\to$ a . jsou používány pro přístup ke členům struktur; budou popsány v kapitole 6 spolu se sizeof (velikost objektu). V kapitole 5 jsou uvedeny operátory $*$ a &.

Uvědomte si, že priorita bitových operátorů &, ˆ a | je nižší než == a != . Z toho vyplývá, že při testování bitů ve výrazu jako:

```
if((x & MASK) == 0) ...
```

jsou závorky nezbytné.

Jak už jsme se zmínili, výrazy obsahující jeden z asociativních a komutativních operátorů ($*$, +, &, ˆ, |) mohou být překladačem přeskupeny i když jsou použity závorky. Ve většině případů to nevadí; v situacích, kdy by to vadit mohlo, musí být explicitně použity pomocné proměnné, aby pořadí vyčíslování probíhalo tak, jak si přejeme.

Jazyk C, ostatně jako většina ostatních jazyků, nespecifikuje, v jakém pořadí budou operandy operátoru vyčísleny. Např. v příkazu

```
x = f() + g();
```

může být f vyčíslena dříve než g nebo také naopak; proto jestli jedna z funkcí f nebo g mění externí proměnné, na kterých závisí druhá z funkcí, může hodnota x záležet na pořadí vyčíslování. Znovu opakujeme, že mezivýsledek může být uložen v pomocné proměnné, abychom si zajistili správné pořadí vyčíslení.

Obdobně není stanoveno pořadí, v jakém jsou vyčíslovány argumenty funkcí, takže příkaz

```
pritf("%d %d \n", ++n, power(2,n));      /*chyba*/
```

může produkovat (a také produkuje) různé výsledky na různých počítačích, což záleží na tom, je-li n zvětšeno před voláním funkce power nebo ne. Správný zápis vypadá takto:

```
++n;
printf("%d\n", n, power(2,n));
```

Volání funkcí a operátory ++ a −− v přiřazovacích příkazech způsobují „postranní efekty“ – některá proměnná je změněna jako „vedlejší produkt“ při vyčíslování výrazu. Ve výrazech, kde se projevují vedlejší efekty, záleží na tom, jak jsou uloženy proměnné, které vystupují ve výrazu. Jedna nešťastná situace vypadá takto:

```
a[i] = i++;
```

Otázkou je, zda je index stará nebo nová hodnota proměnné i. Překladač může s tímto výrazem naložit různým způsobem. Jestliže se projevují vedlejší efekty, jsme vydáni na milost a nemilost překladači, protože nejlepší způsob překladu záleží vždy na architektuře počítače.

Morálním závěrem této diskuze je to, že psaní programu, kde záleží na pořadí vyhodnocování, je špatným programátorským stylem ve všech jazycích. Je přirozeně nutné vědět, kterým věcem se máme vyhnout, ale nevíte-li, jak jsou na kterých počítačích implementovány, pak vás tato nevědomost může i zachránit.

Kapitola 3

Větvení programu

Příkazy pro větvení programu specifikují, v jakém pořadí budou instrukce programem vykonány. S nejobvyklejšími příkazy pro větvení programu jsme se už setkali v dřívějších příkazech, zde uvedeme kompletní sestavu těchto příkazů a upřesníme předchozí.

3.1 Příkazy a bloky

Výraz jako je $x = 0$, nebo $i{+}{+}$, nebo printf($\ldots$) se stává **příkazem**, je-li následován středníkem, jako např.:

```
x = 0;
i++;
printf(...);
```

V jazyku C je středník spíše koncový znak příkazu než oddělovač, jako je tomu např. v jazyku ALGOL.

Složené závorky { } se používají pro združování deklarací a příkazů ve **složený příkaz** neboli **blok**, který je syntakticky ekvivalentní jednomu příkazu. Jedním příkladem jsou složené závorky kolem jednoho příkazu. Dalším příkladem jsou tyto závorky kolem několika příkazů za příkazem if, else, while nebo for. (Proměnné mohou být ve skutečnosti definovány uprostřed **libovolného bloku**; o tom budeme mluvit v kapitole 4). Za pravou závorkou, která ukončuje blok, není nikdy středník.

3.2 if – else

Pro rozhodování se používá kostrukce if-else. Formální syntaxe je následující

```
if(výraz)
      příkaz_1
else
      příkaz_2
```

kde část else není povinná. **Výraz** je vyhodnocen; jestliže je „pravdivý“ (tj. má nenulovou hodnotu), *příkaz_1* je vykonán; jestliže je nepravdivý (tj. má nulovou hodnotu) a jestliže je uvedeno else, je vykonán *příkaz_2*.

Protože příkaz if pouze testuje hodnotu výrazu, je možné psát zkráceně

```
if(výraz)
```

místo

```
if(výraz != 0)
```

Někdy je tento způsob přirozený a jasný, někdy je ale tajemný.

Protože část else příkazu if-else je nepovinná, tak jestliže else vynecháme v sekvenci příkazu if, vzniká nejednoznačnost. else je logicky spojeno s nejbližším předchozím výrazem if, který nemá část else. Např. v

```
if(n > 0)
      if(a > b)
            z = a;
      else
            z = b;
```

část else je spojena s vnitřním příkazem if, jak jsme ukázali strukturou zápisu. Jestliže tento způsob není ten, který chcete, je třeba užít závorek pro správné sdružení.

```
if(n > 0)
{
      if(a > b)
            z = a;
}
else
   z = b;
```

Dvojznačnost je zvláště nebezpečná v situacích jako:

```
if(n > 0)
      for(i = 0; i < n; i++)
            if(s[i] > 0)
            {
                  printf("...");
                  return(i);
            }
else            /*chyba*/
      printf("error - n is zero\n");
```

Logická struktura zápisu ukazuje přesně co chcete, ale překladač přesto spojí else s vnitřním příkazem if. Tento druh chyb se velmi těžko odstraňuje.

Uvědomte si, že za $z = a$ v

```
if(a > b)
      z = a;
else
      z = b;
```

je středník. Je to kvůli gramatickým pravidlům: příkaz následující if je vždy ukončen středníkem.

3.3 else – if

Konstrukce

```
if(výraz)
    příkaz
else if(výraz)
         příkaz
else if(výraz)
         příkaz
else
    příkaz
```

se objevují tak často, že stojí zato se o nich podrobněji zmínit. Sekvence příkazu if je nejobecnějším způsobem mnohonásobného rozhodování. **Výrazy** jsou vyhodnocovány v tomto pořadí: jestliže je výraz pravdivý, tak příkaz s ním spojený je vykonán, čímž je celá sekvence ukončena. **Příkaz** znamená buď jeden příkaz nebo skupinu příkazů v závorkách.

Příkaz spojený s posledním else je vykonán, jestliže žádná předchozí podmínka nebyla splněna. V některých případech může být část

```
else
   příkaz
```

vypuštěna.

Abychom ilustrovali třícestné rozhodování, uvedeme binární funkci rozhodování, která určuje, zda se určitá hodnota objevuje v roztříděném poli v. Prvky pole musí být uspořádány vzestupně. Funkce vrací pozici (číslo v rozsahu 0 až n-1) jestliže x je obsaženo ve v a -1 jestliže není.

```
binary(x, v, n)      /*nalezení x ve v[0]...v[n-1]*/
int x, v[], n;
{
      int low, high, mid;
      low = 0;
      high = n - 1;
      while(low <= high)
      {
            mid = (low + high) / 2;
            if(x < v[mid])
                  high = mid - 1;
            else if(x > v[mid])
                  low = mid + 1;
            else      /*found match*/
                  return(mid);
      }
      return(-1)
}
```

Základním rozhodnutím je, zda je x menší, větší, nebo rovno střednímu prvku $v[mid]$ v každém kroku; to je přirozené pro else-if.

3.4 Přepínač

Přepínač je speciální, mnohonásobný rozhodovací příkaz, který testuje, zda se výraz v závorkách rovná některé z **konstantních** hodnot a podle toho pokračuje v činnosti. V kapitole 1 jsme zjišťovali výskyt každé číslice, mezery a ostatních znaků použitím sekvence if ... else, if ... else. Zde uvedeme stejný program s příkazem switch.

```
main()      /*počítání čísel, oddělovačů, ostatních*/
{
      int c, i, nwhite, nother, ndigit[10];
      nwhite = nother = 0;
      for(i = 0; i < 10; i++)
            ndigit[i] = 0;
      while((c = getchar ()) != EOF)
            switch(c)
            {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
                  ndigit[c-'0']++;
                  break;
            case ' ':
            case '\n':
            case '\t':
                  nwhite++;
                  break;
            default:
                  nother++;
                  break;
            }
      printf("digits = ");
      for(i = 0; i < 10; i++)
            printf(" %d ", ndigit[i]);
      printf("\nwhite space = %d, other = %d\n", nwhite,
nother);
}
```

Příkaz switch vyhodnocuje celočíselný výraz v závorkách (v našem příkladu je to znak c) a porovnává jeho hodnotu se všemi case. Každý case musí být označen celočíselnou nebo znakovou konstantou, nebo konstantním výrazem. Jestliže některý case odpovídá hodnotě výrazu, tak zde začíná činnost programu. Položka označená default je vykonána, jestliže nebyl uspokojen žádný case. Položka default nemusí být uvedena; když není uvedena a žádný case není uspokojen, tak není provedena žádná činnost. Položky case a default se mohou objevit v libovolném pořadí. case musí být od sebe odlišné.

Příkaz break způsobuje okamžité ukončení příkazu switch. Protože case slouží pouze jako návěští, tak potom, co jsou vykonány příkazy odpovídající návěští case, program **prochází** další návěští case, dokud není nějakým explicitním příkazem ukončen. Základními příkazy pro ukončení příkazu switch jsou return a break. Příkaz break může být také použit pro okamžité vystoupení z cyklu while, for a do, jak bude ukázáno v dalších kapitolách.

Postupné procházení návěští case je dvojsečné. Na jedné straně dovoluje pro jednu činnost více návěští case, na druhé straně ale musí být normálně každá činnost po návěští case ukončena příkazem break pro zamezení procházení dalších návěští. Procházení návěští case není čitelné. S výjimkou použití více návěští pro jednu činnost by mělo být postupné procházení užíváno zřídka.

Je dobrým zvykem ukončovat case příkazem break (v našem příkladu default), i když není z logického hlediska potřebný. Někdy třeba budeme přidávat další návěští case a break se bude hodit.

Cvičení 3-1. Napište funkci expand(s, t), která konvertuje znaky jako je znak pro novou řádku a tabelátor do viditelného tvaru \n a \t při kopírování řetězce s na t. Použijte switch.

3.5 Cykly while a for

S příkazy cyklů while a for jsme se již setkali. Ve

```
while (výraz)
příkaz
```

je výraz vyhodnocen. Pokud je nenulový, tak je příkaz vykonán a výraz je znovu vyhodnocen. Cyklus pokračuje do té doby, než je výraz nulový. Potom program pokračuje za příkazem while.

Příkaz for

```
for(výraz_1;  výraz_2;  výraz_3)
   příkaz
```

je ekvivalentní

```
výraz_1;
while( výraz_2 )
{
   příkaz
   výraz_3
}
```

Z gramatického hlediska jsou všechny tři položky příkazu for výrazy. Obyčejně výraz_1 a výraz_3 jsou přiřazovací příkazy nebo volání funkce a výraz_2 je relační výraz. Kterákoliv položka může být z příkazu for vypuštěna, ale středníky musí být uvedeny. Jestliže vypustíme výraz_1 a výraz_3, proměnná i se přestane měnit. Jestliže vypustíme relační výraz_2, je tento výraz brán jako stále pravdivý:

```
for(; ;)
{
   ...
}
```

tvoří nekonečný cyklus, který může být ukončen jinými způsoby (např. příkazy break nebo return).

Zda použít příkaz while nebo for je čistě záležitost vkusu. Např. ve

```
while((c = getchar()) == ' ' || c == '/n' || c == '\t')
      ;      /*přeskočení oddělovačů*/
```

není žádná inicializace ani reinicializace, takže použití příkazu while je přirozené.

Příkaz for má jasné přednosti tehdy, je-li v cyklu jednoduchá inicializace a reinicializace, protože soustředí příkazy cyklu blízko sebe a viditelně na vrcholu cyklu. Je to zcela zřejmé v

```
for(i = 0; i < N; i++)
```

což je v jazyku C příkaz obdobný příkazu cyklu DO v jazyku FORTRAN nebo PL/1. Analogie to ale není úplná, protože limity cyklu for mohou být uvnitř měněny a proměnné i si uchovávají svoji hodnotu, i když je cyklus for z jakýchkoliv důvodů ukončen. Protože složky cyklu for jsou libovolné výrazy, tak cyklus není omezen jen na aritmetické posloupnosti. Nicméně není dobrým stylem „vnutit" do příkazu for výrazy, které spolu nemají nic společného.

Jako větší příklad uvedeme jinou verzi funkce atoi pro konvertování řetězce na číselný ekvivalent. Tato verze je univerzálnější, počítá totiž s úvodním oddělovačem a znaménkem + a − (v kapitole 4 ukážeme funkci atof, která provádí stejnou konverzi pro čísla s pohyblivou řádovou čárkou).

Základní struktura programu vypadá takto:

```
ignoruj oddělovače, jestliže nějaké jsou

vezmi znaménko, jestliže nějaké je

vezmi číselnou část a konvertuj ji
```

Každý krok vykoná určitou činnost a ponechá věci v jasném stavu pro další kroky. Celý proces je ukončen při nalezení prvního znaku, který není součástí čísla.

```
atoi(s)             /*konvertování s na číslo*/
char s[];
{
int i, n; sign;
for(i = 0; s[i] == ' ' || s[i] == '\n' || s[i] =='\t';
i++)
      ;      /*přeskočení oddělovačů*/
sign = 1;
if(s[i] == '+' || s[i] == '-')              /*znak*/
      sign = (s[i++] == '+') ? 1 : -1;
for(n = 0; s[i] >= '0' && s[i] <= '9'; i++)
```

```
        n = 10 * n + s[i] - '0';
    return(sign * n);
    }
```

Výhoda soustředění cyklu na jedno místo je mnohem jasnější, než několik vnořených cyklů. Následující funkce shell třídí celočíselného pole. Základní myšlenka tohoto třídění je, že v počátku jsou spíše než sousední prvky porovnávány prvky vzdálené, zrovna tak jako v normálním třídění. Tím je ale v první fázi vykonáno hodně práce, takže dále není třeba už moc dělat. Interval mezi porovnávanými prvky je postupně snižován až na jedna, kdy tato metoda efektivně přechází na normální přemísťování sousedních prvků.

```
    shell(v, n,)        /*třídění v[0]...v[n-1] vzestupně*/
    int v[], n;
    {
    int gap, i, j, temp;
          for(gap = n/2; gap > 0; gap /= 2)
             for(i = gap; i < n; i++)
                for(j = i-gap; j >= 0 && v[j] > v[j+gap]; j -
= gap)
                   {
                      temp = v[j];
                      v[j] = v[j + gap];
                      v[j + gap] = temp;
                   }
    }
```

V přehledu jsou tři vnořené cykly. Vnější cyklus určuje vzdálenost mezi porovnávanými prvky a postupně ji zmenšuje od $n/2$ dělením dvěma až do nuly. Prostřední cyklus porovnává každé dva prvky, jejichž vzdálenost je rovna *gap*, vnitřní cyklus zaměňuje ty prvky, které jsou nesprávně uspořádány. Protože proměnná *gap* je nakonec zmenšena na jedničku, tak jsou všechny prvky správně uspořádány. Uvědomte si, že obecnost příkazu for dovoluje vnějšímu cyklu mít stejnou formu jako ostatní, i když to není aritmetická řada.

Jedním z posledních operátorů jazyka C je čárka „ , “, kterou nejčastěji nalezneme v příkazu for. Dvojice výrazů oddělených čárkou je vykonávána zleva doprava, a typ i hodnota výsledku je shodná s typem a hodnotou pravého operandu. Proto je možné v příkazu for používat na různých místech vícenásobné výrazy, např. měnit dva indexy pole současně. To je ilustrováno ve funkci reverse(s), která invertuje řetězec s.

```
    reverse(s)       /*invertován řetězec s*/
    char s[];
    {
          int c, i, j;
          for(i = 0, j = strlen(s) - 1; s < j; i++, j--)
          {
                c = s[i];
                s[i] = s[j];
                s[j] = c;
          }
    }
```

Čárky, které oddělují argumenty funkcí, proměnné v deklaracích atd., **nejsou** operátory a **nezaručují** vyhodnocování zleva doprava.

Cvičení 3-2. Napište funkci expand($s1$, $s2$), která rozepisuje zkratky jako $a-z$ v řetězci $s1$ v kompletní seznam $abc...xyz$ v $s2$. Připusťte velká i malá písmena, číslice a buďte připraveni zacházet s případy typu $a-b-c$ a $a-z$, $0-9$ a $a-z$. (Užitečnou konvencí je to, že znak $-$ je brán doslovně.)

3.6 Cyklus do – while

Jak jsme již uvedli v kapitole 1, cykly while a for mají rozhodovací podmínku umístěnou na začátku cyklu. Třetí typ cyklu v jazyce C, příkazy do-while, mají rozhodování umístěno na konci po vykonání příkazu těla cyklu; tělo cyklu je tady vykonáno minimálně jednou. Syntaxe vypadá takto

```
do
      příkaz
while (výraz)
```

Příkaz je vykonán a potom je vyhodnocen výraz. Když je pravdivý, *příkaz* je znovu vykonán atd. Jestliže je výraz nepravdivý, cyklus je ukončen. Jak můžeme očekávat, do-while se používá méně než while a for (odhadem je to asi 5% všech cyklů) nicméně je čas od času užitečný, jako např. v následující funkci itoa, která konvertuje číslo na znakový řetězec (inverzní funkce k atoi). Úloha je poněkud komplikovanější, než by se mohlo na první pohled zdát, protože jednoduchá metoda pro generování číslic je generuje ve špatném pořadí. Zvolili jsme způsob generování pozpátku a potom invertování řetězce.

```
itoa(n, s)      /*konverze n na znaky v s*/
char s[];
int n;
{
      int i, sign;
      if((sign = n) < 0)          /*zjištění znaménka*/
            n = -n;                   /*n bude kladné*/
      i = 0;
      do        /*generování číslic v opačném pořadí*/
      {
            s[i++] = n % 10 + '0';   /*další číslice*/
      }
      while((n /= 10) > 0);         /*smazání číslice*/
      if(sign < 0)
            s[i++] = '-';
      s[i] = '\0';
      reverse(s);
}
```

Cyklus do-while je zde nezbytný, nebo alespoň výhodný, protože pole s musí obsahovat alespoň jeden znak nehledě na hodnotu čísla n. Užili jsme rovněž závorky okolo jednoho příkazu, který tvoří tělo cyklu do-while, i když jsou zbytečné, takže ukvapený čtenář si nesplete část cyklu do s příkazem cyklu while.

Cvičení 3-3. Naše verze funkce itoa při reprezentaci čísel ve dvojkovém doplňku neumí zacházet s největším záporným číslem, tj. $n = -(2**(délka\_slova - 1))$. Vysvětlete proč. Upravte ji tak, aby tisklo i tuto hodnotu správně a nezávisle na typu počítače.
Cvičení 3-4. Napište obdobnou funkci itob(n, s), která konvertuje celé číslo n bez znaménka na binární reprezentaci do řetězce s. Napište také funkci itoh, která konvertuje celé číslo na hexadecimální reprezentaci.
Cvičení 3-5. Napište verzi funkce itoa, která má tři argumenty místo dvou. Třetím argumentem bude minimální délka pole; konvertované číslo musí být doplněno zleva mezerami, aby bylo dostatečně široké.

3.7 Break

Někdy je výhodné mít možnost vycházet z cyklu jinde než na začátku nebo na konci. Příkaz break umožňuje předčasný výstup z cyklu for, while, do a stejně tak z přepínače switch. Příkaz break působí na nejvnitřnější vložený cyklus (nebo switch).

Následující program odstraňuje koncové mezery a tabelátory z každé řádky na vstupu a používá příkazu break k výstupu z cyklu tehdy, když je nalezen zprava znak, který není tabelátor ani mezera.

```
#define      MAXLINE      1000
main()      /*odstranění koncových mezer a tabelátorů*/
{
      int n;
      char line[MAXLINE];
      while((n = getline(line, MAXLINE)) > 0)
      {
            while(--n >= 0)
                  if(line[n] != ' ' && line[n] != '\t')
                        break;
            line[n + 1] = '\0';
            printf("%s\n", line);
      }
}
```

Funkce getline vrací délku řádky. Vnitřní cyklus while začíná na posledním znaku řádku (uvědomte si, že $--n$ zmenšuje n **před** použitím jeho hodnoty), prohledává ji odzadu a hledá první znak, který není mezera, tabelátor nebo znak pro novou řádku. Cyklus je přerušen tehdy, je-li takový znak nalezen nebo když se n stane záporným (tzn. celá řádka byla prohledána). Měli byste si ověřit, že program funguje správně, i když řádka obsahuje jen mezery nebo tabelátory.

Namísto příkazu break je možné položit testovací relaci na začátek cyklu while:

```
while((n = getline(line, MAXLINE)) > 0)
{
      while(--n > = 0 && (line [n] == ' '
      || line[n] == '\t' || line[n] == '\n'))
      ;
      ...
}
```

Tato varianta není lepší než předchozí, protože testovací podmínka není již tak jasná. Podmínkám, které obsahují směs &&, ||, ! nebo závorky, je lepší se vyhnout.

3.8 Continue

Příkaz continue je obdobou příkazu break, ale méně se ho užívá; způsobuje skok na začátek **další iterace** nejvnitřnějšího cyklu (for, while, do). V cyklech while a do je okamžitě vyhodnocena podmínka, v cyklu for je vykonána reinicializace. (continue může být použito pouze v cyklech, nikoliv v přepínačích switch, který je uvnitř cyklu, neboť způsobuje vykonání další iterace cyklu.)

V následující ukázce jsou vybírána z pole a pouze kladná čísla, záporná jsou přeskočena.

```
for(i = 0; i < N; i++)
{
      if(a[i] < 0)      /*přeskoč záporné prvky*/
            continue;
      ...                         /*kladné prvky*/
}
```

Příkaz continue je často používán, když část cyklu, která následuje, je složitá, takže obrácení podmínky a vložení další by mohlo být příliš složité.

Cvičení 3-6. Napište program, který kopíruje vstup na výstup s tou výjimkou, že kopíruje pouze jednu řádku ze skupiny totožných řádek. (Toto je jednoduchá verze obslužného programu uniq v systému UNIX.)

3.9 Příkaz goto a návěští

Jazyk C obsahuje také nekonečně zatracovaný příkaz goto a návěští, na které směřuje. Obecně vzato, příkaz goto není nikdy nepostradatelný a prakticky lze vždy napsat program bez něj. V této knize příkaz goto nepoužíváme.

Nicméně ukážeme několik situací, kde se příkaz goto může uplatnit. Obecně použitelný je při výstupu z mnohonásobně vnořených struktur, jako např. výstup ze dvou cyklů najednou. Příkaz break nemůže být přímo použit, protože vystupuje pouze z vnitřního cyklu.

```
for(...)
      for(...)
      {
      ...
      if(katastrofa)
            goto error;
      }
...
error:
      chybové hlášení
```

Tato organizace je výhodná, jestliže program není triviální a když se chyba objevuje na různých místech. Návěští má stejnou formu jako jméno proměnné a je následováno dvojtečkou. Může být připojeno k libovolnému příkazu ve stejné funkci, jako je goto.

Jako jiný příklad uvažujme problém nalezení prvního záporného čísla ve dvojdimenzionálním poli. (Vícedimenzionální pole jsou probrána v kap.5.) Jedna možnost je

```
      for(i = 0; i < N; i++)
            for(j = 0; j < M; j++)
                  if(v[i][j] < 0)
                        goto found;
      /*nenalezeno*/
      ...
found:
      /*nalezeno na pozici i, j*/
      ...
```

Program, který je napsán s příkazem goto může být vždy napsán bez něj, možná za cenu opakovaných testů nebo extra proměnné. Náš příklad bez goto bude vypadat takto

```
found = 0;,
for(i = 0; i < N && !found; i++)
      for(j = 0; j < M && !found; j++)
            found = v[i][j] < 0;
if(found)
      /*nalezeno na pozici i-1, j-1*/
      ...
else
      /*nenalezeno*/
      ...
```

Přestože nejsme dogmatici, vypadá to, že příkaz goto může být užíván řídce, ne-li vůbec.

Kapitola 4

Funkce a struktura programu

Funkce drobí velké výpočetní úlohy na menší a umožňují lidem stavět na tom, co ostatní již udělali, místo toho, aby začínali od začátku. Vhodné funkce mohou často skrýt detaily části programu, kde o nich není třeba nic vědět a tak vyjasnit celek a usnadnit případné změny v programu.

Jazyk C byl navržen tak, aby bylo možné snadno užívat funkce. Programy v jazyku C sestávají obecně z množství malých funkcí spíše než z několika velkých. Program může být obsažen v jednom nebo více zdrojových souborech libovolným způsobem. Zdrojové soubory mohou být překládány odděleně a linkovány dohromady spolu s dříve přeloženými funkcemi z knihoven. Nebudeme zde o těchto problémech hovořit, protože jsou závislé na použitém operačním systému.

Většina programátorů je již seznámena s „knihovními" funkcemi pro vstup a výstup (getchar, putchar) a pro numerické výpočty (sin, cos, sqrt). V této kapitole ukážeme více o používání nových funkcí.

4.1 Základy

Pro začátek napíšeme program, který tiskne každou řádku ze vstupu, která má jistou vlastnost nebo obsahuje určitý řetězec znaků. (To je speciálním případem obslužného programu grep v systému UNIX.) Např. hledejme řádky, které obsahují řetězec „the" v následujícím souboru:

```
Now is the time
for all good
men to come to the aid
of their party.
```

Výsledkem bude:

```
Now is the time
men to come to the aid
of their party.
```

Základní strukturu můžeme popsat ve třech částech

```
while (je ještě další řádka)
     if (řádka obsahuje daný řetězec)
        vytiskni ji
```

Přestože by bylo jistě možné napsat tento program jako celek, lepší cestou je využít přirozené struktury a použít pro každou část oddělenou funkci. Se třemi menšími úseky se lépe zachází než s jedním velkým celkem, protože nepodstatné detaily mohou být do nich „utopeny“ a mohou být minimalizovány nežádoucí interakce. Jednotlivé úseky také mohou být užitečné pro svou činnost.

„Je ještě další řádka“ je funkce getline, která je popsána v kap. 1 a *„vytiskni ji“* je funkce printf, kterou již pro nás někdo napsal. To znamená, že musíme pouze napsat funkci, která rozhoduje, zda řádka obsahuje určitý řetězec znaků. Tento problém můžeme vyřešit tím, že ukradneme ideu z jazyka PL/1; funkce index(s,t) vrací pozici v řetězci s, kde začíná řetězec t, nebo -1, když t není obsaženo v s. Pro začátek v řetězci s použijeme spíše 0 než 1, protože pole v jazyce C začínají rovněž nulou. Když později budeme potřebovat dokonalejší vyhledávání, stačí pouze změnit funkci index; zbytek programu zůstane nezměněn.

Rozhodneme-li se pro tento postup, lze program přímo napsat. V programu je zřetelně vidět, jak spolu jeho části souvisí. Pro tentokrát bude řetězec, který hledáme, znakovou konstantou v argumentu funkce index, což není úplně obecný způsob. Krátce se vrátíme k tomu, jak inicializovat znaková pole a v kap. 5 ukážeme, jak můžeme nastavit požadovaný řetězec pro výběr, když je program vyvolán. Popsána je rovněž nová verze funkce getline. Porovnejte ji s funkcí z kap. 1.

```
      # define       MAXLINE   1000
      main()  /*najdi všechny řádky obsahující řetězec */
      {
            char line [MAXLINE];
            while (getline(line, MAXLINE) > 0)
               if (index(line, "the") >= 0)
                  printf ("%s",line);
      }
      getline(s,lim)   /*načti řádku do s, vrať délku*/
      char s[];
      int lim;
      {
         int c, i;
         i=0;
         while(--lim > 0 && (c = getchar()) != EOF && c !=
'\n')
              s[i++] = c;
         if (c == '\n')
              s[i++] = c;
         s[i] = '\0';
         return(i);
      }

      index(s,t) /* vrať index výskytu t v s */
      char s[], t[];
      {
         int i,j,k;
         for (i = 0; s[i] != '\0'; i++)
```

```
        {
            for(j = i, k = 0; t[k] != '\0' && s[j] == t[k];
j++,k++)
              ;
            if(t[k] == '\0')
              return(i);
        }
        return(-1);
    }
```

Každá funkce má strukturu

```
název  (seznam argumentů, jsou-li nějaké)
   deklarace argumentů, jsou-li nějaké
   {
      deklarace a příkazy, jsou-li nějaké
   }
```

Jak je naznačeno, různé části mohou být vynechány; minimální funkce vypadá takto:

```
dummy() {}
```

a nedělá nic. (Funkce, která nedělá nic je užitečná proto, že „drží místo“ nové funkci při vývoji programu.) Před jménem funkce může být rovněž uveden typ, když funkce vrací něco jiného než celočíselnou hodnotu; o tom budeme hovořit v této sekci.

Program je jen soubor definic jednotlivých funkcí. Komunikace mezi funkcemi je (v tomto případě) pomocí argumentů a hodnot jimi vrácenými; může být rovněž prováděna externími proměnnými. Funkce se ve zdrojovém souboru mohou objevit v libovolném pořadí. Zdrojový program může být rozdělen do mnoha souborů, pokud není některá funkce roztržena.

Příkazem return vrací volaná funkce hodnotu jednotce volající. Za příkazem return se může objevit libovolný výraz:

```
return (výraz)
```

Volající jednotka může dle libosti hodnotu ignorovat. Navíc po příkazu return nemusí následovat žádná hodnota: v tomto případě není žádná hodnota vrácena. Řízení se rovněž volající jednotce vrací tehdy, dospěla-li volaná funkce „na konec“, tj. k právě uzavírající složené závorce. Obecně není zakázáno, aby funkce z jednoho místa hodnotu vracela a z jiného ne, ale může to způsobit potíže. V případě, že funkce žádnou hodnotu nevrací, je „hodnota“ funkce náhodná, nedefinovaná, program LINT jazyka C lokalizuje takovéto chyby.

Způsob, jak překládat a sestavovat program v jazyku C, který je v několika zdrojových souborech, se liší systém od systému. Např. v systému UNIX je prováděno příkazem cc, jak jsme ukázali v kap. 1. Předpokládejme, že tři funkce jsou ve třech souborech nazvaných main.c, getline.c, index.c. Potom příkaz

```
cc main.c getline.c index.c
```

přeloží tyto tři soubory a uloží přemístitelný kód do souboru main.o, getline.o, index.o a sestaví je všechny do proveditelného programu nazvaného a.out.

Jestliže se např. v main.c vyskytne chyba, tak může být později přeložen sám a výsledek sestav spolu s dříve přeloženými soubory

```
cc main.c getline.o index.o
```

Příkaz cc užívá „.c" a „.o" k odlišení zdrojových a přemístitelných souborů.

Cvičení 4-1. Napište funkci rindex(*s*,*t*), která vrací pozici posledního výskytu řetězce *t* v *s*, nebo - 1, když *s* neobsahuje *t*.

4.2 Funkce, které nevrací celá čísla

Doposud v žádném našem programu nebyla použita deklarace funkce. To proto, že každá funkce je implicitně deklarována podle toho, v jakém výrazu nebo příkazu se vyskytuje. Jako např.

```
while (getline(line, MAXLINE)>0)
```

Jestliže se jméno, které nebylo dosud deklarováno, objeví ve výrazu a bezprostředně za ním následuje levá závorka, tak je podle kontextu definováno jako jméno funkce. Navíc implicitně se předpokládá, že funkce vrací hodnotu int. Protože char je ve výrazech bráno jako int, tak není potřeba deklarovat funkci jako char. Tyto předpoklady pokrývají většinu případů a zahrnují rovněž naše příklady.

Co se ale stane, má-li funkce vracet hodnoty jiného typu? Mnoho numerických funkcí jako jsou sqrt, sin a cos vrací hodnotu typu double; jiné speciální funkce vrací jiné typy. Abychom si to ilustrovali napišme funkci atof(*s*), která konvertuje řetězec *s* na ekvivalentní číslo typu double. Funkce atof je rozšířením funkce atoi, kterou jsme v kapitolách 2 a 3 vytvořili v několika verzích. Pracuje se znaménkem a desetinnou tečkou, celá a desetinná část může nebo nemusí být uvedena. (Není to konverzní program vysoké kvality.)

Za prvé: funkce atof musí sama deklarovat typ, který vrací, protože to není typ int. Protože float je konvertováno ve výrazech na double, není námitek proti tomu, že vrací float; stejně ale můžeme využít zvýšené přesnosti a proto deklarovat funkci jako double. Příkaz typu je uveden před jménem funkce, jako

```
   double atof(s)   /*konvertován řetězec s na double */
   char s[];
   {
      double val, power;
      int i, sign;
      for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t';
i++)
            ;  /*přeskoč oddělovač*/
      sign = 1;
      if (s[i] == '+' || s[i] == '-')   /*znak*/
           sign = (s[i++] == '+') ? 1 : -1;
      for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
              val = 10 * val + s[i] - '0';
      if (s[i] == '.')
              i++;
      for (power = 1; s[i] >= '0' && s[i] <= '9';i++)
      {
            val = 10 * val s[i] - '0';
```

```
        power *= 10;
    }
    return (sign * val / power);
}
```

Za druhé: ve volající jednotce je nutno určit, že atof vrací neceločíselnou hodnotu. Tato deklarace je ukázána v následujícím programu simulujícím primitivní stolní kalkulátor, který čte jedno číslo z řádky a sečítá je a tiskne předběžně součet před každým vstupem.

```
#define MAXLINE  100
main()     /*primitivní stolní kalkulátor*/
{
      double sum, atof ();
      char line[MAXLINE];
      sum = 0;
      while (getline(line, MAXLINE) > 0)
             printf ("\t%.2f\n", sum += atof(line));
}
```

Deklarace

```
double sum, atof();
```

říká, že *sum* je proměnná s dvojnásobnou délkou a atof je funkce, která vrací hodnotu double. Jako mnemoniku doporučujeme, aby *sum* a atof byly oboje typu double.

Jestliže je atof explicitně deklarována na obou místech, překladač jazyka C předpokládá, že vrací hodnotu integer, a výsledek bude nesmyslný. Jestliže funkce atof a program main jsou ve stejném souboru a typ funkce samé a volání v jednotce main sobě neodpovídají, tak předkladač bude hlásit chybu. Ale jestliže (což je častější případ) atof byla předkládána odděleně, nesoulad nebude detekován, atof bude vracet hodnotu double, se kterou program main bude pracovat jako s proměnnou integer a dostaneme nesmyslný výsledek (obslužný program LINT tuto chybu odhalí).

Máme-li vytvořenou funkci atof, můžeme napsat funkci atoi, která konvertuje řetězec znaků na číslo typu int:

```
atoi(s)        /*konvertuje řetězec s na integer*/
  char s[];
{
      double atof ();
      return (atof(s));
}
```

Všimněte si struktury deklarací a příkazu return. Hodnota výrazu v

```
return (výraz)
```

je vždy konvertována na typ funkce předtím, než je příkaz return vykonán. Proto hodnota atof, která je typu double, je automaticky konvertována na int, když se objeví v příkazu return, protože funkce atoi vrací int. (Konverze čísla s pohyblivou řádovou čárkou na celé číslo ořezává desetinnou část, jak jsme ukázali v kap. 2.)

Cvičení 4-2. Rozšiřte funkci atof tak, aby uměla zpracovávat čísla ve vědecké notaci, jako např.

$$123.45e - 6$$

kde číslo s pohyblivou desetinnou tečkou může být následováno písmenem e nebo E a exponentem se znaménkem (které nemusí být uvedeno).

4.3 Více o argumentech funkcí

V kapitole 1 jsme uvedli skutečnost, že argumenty funkcí jsou předávány hodnotou, což znamená, že volaná funkce obdrží „soukromou“, přechodnou kopii každého argumentu a ne adresu. To znamená, že funkce nemůže změnit původní argument ve volající jednotce. Ve funkci je argument ve skutečnosti lokální proměnná, která je inicializována na hodnotu, kterou je funkce vyvolána.

Když se jako argument funkce objeví identifikátor pole, je předáno místo začátku tohoto pole, prvky pole nejsou kopírovány. Funkce může měnit hodnoty prvku pole. Pole jsou tedy předávány adresou. V kapitole 5 budeme hovořit o použití pointrů, které umožňují funkcím měnit také jednotlivé proměnné.

Mimochodem, neexistuje uspokojivý způsob, jak napsat obecnou funkci, která by mohla mít proměnný počet argumentů, protože volaná funkce nemůže určit, kolik argumentů bylo skutečně při vyvolání předáno. Proto nemůžete napsat obecnou funkci, která bude počítat maximum libovolného množství argumentů jako to je u funkcí MAX ve FORTRANU nebo PL/1.

Bezpečně můžeme nakládat s proměnným množstvím parametrů, když volaná funkce nepoužívá ty argumenty, které jí nebyly ve skutečnosti dodány, a když typy sobě odpovídají. Funkce printf, která je nejobecnější funkcí jazyka C používající proměnný počet argumentů, užívá první parametr k určení počtu a typu argumentů. Skončí to ovšem špatně, když volající jednotka nedodá dostatečné množství argumentů, nebo když neodpovídají jejich typy. Je také nepřenosná a musí být modifikována pro různé typy počítačů.

Jestliže jsou typy argumentů známé, je možné označit konec seznamu argumentů nějakým dohodnutým způsobem, např. použít speciální hodnotu (často se užívá nuly), která ukončuje seznam argumentů.

4.4 Externí proměnné

Program v jazyku C je složen z množiny externích objektů, které jsou buď proměnné nebo funkce. Přídavné jméno „externí, vnější“ je užito v kontrastu se slovem „interní, vnitřní“, které popisuje argumenty a automatické proměnné uvnitř funkcí. Externí proměnné jsou definovány mimo funkce a jsou potenciálně dosažitelné mnoha funkcím. Samotné funkce jsou **vždy** externí, protože jazyk C nedovoluje definovat funkci uprostřed jiné funkce. Implicitně jsou externí proměnné také „globální“, takže odkazy na tyto proměnné jsou možné i z funkcí, které jsou přeloženy odděleně. V tomto smyslu jsou externí proměnné analogií bloku COMMON ve FORTRANu nebo EXTERNAL v PL/1. Uvidíme později, jak je možné definovat externí proměnné, které nejsou obecně dosažitelné, ale jsou dosažitelné pouze z jednoho zdrojového souboru.

Protože externí proměnné jsou všeobecně dosažitelné, je možné je používat pro komunikaci mezi funkcemi namísto seznamu argumentů. Libovolná funkce může dosáhnout externí proměnnou odkazem na její identifikátor, jestliže tento identifikátor byl nějak deklarován.

Jestliže musí být funkcemi sdílen velký počet proměnných, tak použití externích proměnných je výhodnější a efektivnější než dlouhé seznamy argumentů. Avšak jak jsme poznamenali v kap. 1 musí být tento způsob používán opatrně, protože působí špatně na strukturu programu a vede k programům, které mají mnoho datových spojení mezi jednotlivými funkcemi.

Druhý důvod, proč používat externí proměnné je problém inicializace. Externí pole mohou být na rozdíl od automatických (lokálních) polí inicializovány. O tom budeme hovořit na konci této kapitoly.

Třetím důvodem pro používání externích proměnných je jejich obor působnosti a životnost. Automatické proměnné jsou vnitřní, interní, dané funkcí; začnou existovat při vyvolání funkce a zmizí, když skončí činnost funkce. Externí proměnné jsou stálé. Nevznikají, nekončí a udržují hodnotu od volání jedné funkce do volání funkce další. Proto, jestliže dvě funkce musí sdílet určitá data a jedna nevolá druhou, je často pohodlnější, když jsou tyto proměnné uvedeny jako externí než předávat tato data sem a tam argumenty.

Vyzkoušejme tyto návrhy na větším příkladu. Napíšeme jiný programový kalkulátor, který bude lepší než předchozí. Bude dovolovat operátory $+$, $-$, $*$, $/$ a $=$ (pro vytištění výsledku). Kalkulátor bude používat obrácenou polskou notaci (dále RPN) namísto notace infix, protože je snadněji implementovatelná. (RPN používají např. kalkulátory firmy Hewlett-Packard.) V RPN každý operátor následuje za operandy; výraz v infixové notaci jako

```
(1-2) * (4+5) =
```

je v RPN napsán takto

```
1 2  -  4 5 + * =
```

není nutné používat závorky.

Implementace je opravdu jednoduchá. Každý operand je uložen do zásobníku; když se ve výrazu vyskytne operátor, tak odpovídající počet operandů (dva pro binární operace) je vytažen ze zásobníku, operátor je na ně aplikován a výsledek je uložen do zásobníku. Např. v předchozím příkladě jsou 1 a 2 uloženy do zásobníku a potom jsou nahrazeny jejich rozdílem. Dále 4 a 5 jsou uloženy a potom nahrazeny jejich součtem. Násobek -1 a 9, který je -9 je v zásobníku nahradí. Operand $=$ zobrazí vrchní prvek zásobníku, aniž ho vyjme (takto můžeme kontrolovat mezivýsledky).

Operace pro ukládání a vyjímání ze zásobníku jsou jednoduché, ale když jsou přidána chybová hlášení, tak jsou dostatečně dlouhá na to, aby byly soustředěny do funkcí namísto opakování v programu. Rovněž by měla být použita oddělená funkce pro načtení dalšího vstupního operátoru nebo operandu. Proto bude struktura programu následující:

```
while (další vstup není konec souboru)
if (číslo)
      ulož je
 else if (operátor)
      vyjmi operandy
```

```
            proveď operaci
            ulož výsledek
                  else
            chyba
```

Hlavním rozhodnutím při návratu, což jsme zatím nediskutovali, je to, kde je zásobník uložen, a která funkce k němu má přímý přístup. Jednou z možností je držet jej v jednotce main a předávat jeho pozici funkcím, které s ním pracují. Ale v jednotce main není třeba určovat proměnné, které řídí činnost ukládání a vyjímání ze zásobníku, měly by se použít pouze operace „*ulož*“ a „*vyjmi*“. Takže se rozhodneme, že zásobník a jemu přidružené proměnné budou externí a dosažitelné jen funkcím push — pop.

Tento postup je snadno naprogramovatelný. Jednotka main bude velký příkaz switch, který pracuje s operátory a operandy. To bude typičtější příklad pro použití příkazu switch, než jaký byl uveden v kap. 3.

```
      #define MAXOP    20  /*maximální množství operandů,
operátorů*/
      #define NUMBER  '0'  /*symbol pro číslo*/
      #define TOOBIG  '9'  /*symbol pro příliš dlouhý řetězec*/

      main ()            /* RPN kalkulátor*/
      {
            int type;
            char s[MAXOP];
            double op2(), pop(), push();
            while ((type = getop(s,MAXOP)) != EOF)
                switch (type)
                {
                  case NUMBER:
                             push (atof(s));
                             break;
                  case '+':
                             push (pop() + pop());
                             break;
                  case '-':
                             op2 = pop();
                             push (pop() - op2);
                             break;
                  case '/':
                             op2 = pop();
                             if (op2 != 0.0)
                              push (pop() / op2);
                               else
                              printf("zero divisor popped\n");
                             break;
                  case '=':
                             printf("\t%\n", push (pop()));
                             break;
                  case 'c':
```

```
                              clear();
                              break;
                  case TOOBIG:
                              printf("%.20s ... is too long\n",
s);
                              break;
                  default:
                              printf("unknown command %c\n",
type);
                              break;
                }
      }

      #define MAXVAL 100    /*maximální velikost zásobníku*/
      int sp = 0;           /*ukazatel zásobníku*/
      double val[MAXVAL];   /*zásobník čísel*/
      double push(f)        /*ulož f do zásobníku*/
      double f;
      {
            if (sp < MAXVAL)
                return (val[sp++] = f);
            else
            {
                 printf ("error: stack full\n");
                 clear();
                 return(0);
            }
      }

      double pop()       /*vyjmi vršek zásobníku*/
      {
            if (sp > 0)
                return (val[--sp];
            else
            {
                printf ("error: stack empty\n");
                clear();
                return(0);
            }
      }

      clear()            /*vymaž zásobník*/
      {
            sp = 0;
      }
```

Příkaz c vymaže zásobník. Užívá k tomu funkci clear, kterou rovněž používají funkce push a pop v případě chyby. Za chvíli se vrátíme k funkci getop.

Jak bylo uvedeno v kap. 1, proměnná je externí, je-li definována mimo těla funkce. Proto musí být ukazatel zásobníku, který je sdílen funkcemi push, pop a clear, definován mimo tyto funkce. Ale jednotka main se na tento ukazatel zásobníku neodkazuje — jeho reprezentace je skryta. Proto část programu pro operátor = musí být napsána takto

```
push(pop());
```

abychom dostali pouze hodnotu vrcholu zásobníku bez jejího vyjmutí.

Uvědomte si rovněž, že + a − jsou komutativní operátory a nezáleží na pořadí, v jakém jsou operandy vyjímány. Avšak pro operandy − a / musí být operandy rozlišeny.

Cvičení 4-3. Máme-li napsán základní rámec kalkulátoru, je snadné jej rozšířit. Přidejte operátory % pro zbytek po dělení a unární minus. Přidejte příkaz „erase“, který vymaže vrchol zásobníku. Přidejte příkazy pro používání proměnných. (26 jednoduchých proměnných můžete snadno přidat.)

4.5 Pravidla pole působnosti

Funkce a externí proměnné, které tvoří program v jazyku C nemusí být překládány najednou; zdrojový text programu může být v několika souborech a dříve přeložené funkce mohou být připojovány z knihoven. Dvě hlavní otázky, které klademe, jsou:

— Jakým způsobem jsou napsány deklarace, aby proměnné byly řádně deklarovány v průběhu překladu?
— Jak jsou deklarace udělány, aby všechny části programu byly řádně spojeny, když je program sestavován?

Pole působnosti identifikátoru proměnné je část programu, ve které je identifikátor definován. Pro automatické proměnné deklarované na začátku funkce je pole působnosti funkce, ve které je tato proměnná deklarována. Proměnné stejného jména v různých funkcích nemají nic společného. Totéž platí pro argumenty funkcí.

Pole působnosti externích proměnných začíná v místě, ve kterém jsou deklarovány ve zdrojovém souboru a končí na konci tohoto souboru. Jestliže např. *val*, *sp*, push, pop a clear jsou definovány v jednom souboru v tomto pořadí tj.

```
int sp = 0;
double val[MAXVAL];
double push (f) (...)
double pop ()  ( ...)
clear ()  (...)
```

potom proměnné *val* a *sp* mohou být použity ve funkcích push, pop a clear jednoduše jménem; není potřeba je nějak deklarovat.

Je-li na druhé straně potřeba použít externí proměnnou předtím, než je definována, nebo když je definována v jiném zdrojovém souboru než v tom, kde je používána, potom je nezbytná deklarace extern.

Je důležité rozlišovat mezi **deklarací** externí proměnné a její **definicí**. Deklarace popisuje vlastnosti proměnné (tj. typ, rozměr, atd.); definice této proměnné vyhrazuje ale také místo v paměti. Jestliže se řádky

```
int sp;
double val[MAXVAL];
```

uvedou mimo funkce, potom **definují** externí proměnné *sp* a *val* a vyhrazují jim místa v paměti a zároveň slouží jako deklarace až do konce zdrojového souboru. Na druhé straně řádky

```
extern int sp;
extern double val[];
```

deklarují pro zbytek zdrojového textu proměnnou *sp* jako int a *val* jako pole typu double (jehož velikost je někde jinde definována), ale nevytvářejí proměnné ani pro ně nevyhrazují místa v paměti.

Ve všech souborech, které tvoří daný program, musí být jen jedna definice externí proměnné; ostatní soubory mohou obsahovat pouze deklaraci extern. (Deklarace extern může být rovněž v souboru, který obsahuje definici externí proměnné. Inicializace externích proměnných je možná jen v definici. Velikost pole musí být specifikována v definici a v deklaraci extern ji uvést můžeme nebo nemusíme.

Přestože takováto organizace není pro tento program pravděpodobná, představme si, že *val* a *sp* jsou definovány a inicializovány v jednom souboru a funkce push, pop a clear jsou definovány v jiném souboru. Potom následující definice a deklarace jsou nezbytné:

```
V souboru 1
      int sp = 0;   /*ukazatel zásobníku*/
      double val[MAXVAL];    /*zásobník*/
V souboru 2
      extern int sp;
      extern double val[];
      double push(f) (...)
      double pop() (...)
      clear() (...)
```

Protože deklarace extern je v souboru 2 uvedena dříve a mimo funkce, tak platí pro tyto funkce; tedy v souboru 2 stačí jen jedna deklarace.

Pro větší program je možné použít příkazu *#include* (pro vkládání souboru, podrobně o tom později v této kapitole) a tak mít jen jednu deklaraci extern pro celý program a tu vkládat tímto příkazem jen při překladu.

Nyní obraťme pozornost k implementaci funkce getop, která načítá další operátor nebo operand. Základní funkce je jasná: přeskoč mezery, tabelátory a symboly pro novou řádku. Jestliže načtený znak není ani číslo ani desetinná tečka, vrať jej. Jinak načti řetězec číslic (kde rovněž může být desetinná tečka) a vrať NUMBER, což je signál pro to, že bylo načteno číslo.

Podprogram je poněkud komplikovaný, protože musí správně fungovat, je-li načtené číslo příliš dlouhé. Funkce getop čte číslice (i s desetinnou tečkou). Jestliže nedošlo k přečtení vrací NUMBER a řetězec číslic. Jestliže číslo bylo příliš dlouhé, getop ignoruje zbytek vstupu, takže uživatel může přepsat řádku od místa chyby; bylo-li číslo příliš dlouhé, vrací TOOBIG.

```
getop(s, lim)     /*načten operátor nebo operand*/
char s[];
```

```
      int lim;
      {
            int i, c;
            while ((c = getch()) == ' ' || c == '\t' || c ==
'\n')
                  ;
            if (c != '.' && (c < '0' || c > '9'))
                return (c);
            s[0] = c;
            for (i = 1; (c = getch()) >= '0' && c <= '9'; i++)
                  if (i < lim)
                      s[i] = c;
            if (c == '.')            /*desetinná část*/
            {
                  if (i < lim)
                      s[i] = c;
                  for(i++; (c = getch()) >= '0' && c <= '9';
i++)
                      if (i < lim)
                        s[s] = c;
            }
            if (i< lim)              /*číslo je ok*/
            {
                ungetch(c);
                s[i] = '\';
                return (NUMBER);
            }
            else   /*je příliš dlouhé, přeskoč zbytek řádku*/
            {
                  while (c != EOF)
                       c = getch();
                  s[lim-1] = '\0';
                  return (TOOBIG);
            }
      }
```

Co dělají funkce getch a ungetch? Často se vyskytne situace, že program načítající vstup nemůže rozhodnout, zdali již načetl dost a přitom nenašel více než je potřeba. Jedním z takových případů je čtení znaků, které tvoří číslo: dokud nebyl načten znak, který není číslicí, tak číslo není kompletní. Potom ale program přečetl jeden znak navíc. Problém by byl snadno vyřešen, kdyby bylo možno tento znak „vrátit zpátky". Tedy vždy když program načte o znak více, může tento znak vrátit zpátky do vstupu, takže se tento znak pro zbytek programu jeví tak, jako kdyby nebyl nikdy načten. Naštěstí je tento problém snadno řešitelný pomocí dvojice doplňujících se funkcí. Funkce getch načítá další znak ze vstupu; ungetch jej vrací zpět do vstupu, takže při dalším volání funkce getch vezme opět tento znak.

Jejich spolupráce je jednoduchá a ungetch vrátí znak do společně sdíleného bufferu

— znakového pole. getch čte z tohoto pole tehdy, když v něm něco je, v opačném případě zavolá getchar. Musí rovněž existovat index, který určuje pozici znaku v bufferu.

Protože buffer a index jsou sdíleny funkcemi getch a ungetch a musí uchovávat svoji hodnotu, mezi vyvoláním těchto funkcí musí být definovány jako externí pro obě funkce. Potom můžeme napsat getch a ungetch takto:

```
# define  BUFSIZE   100
char buf[BUFSIZE];              /*definice bufferu*/
int bufp = 0;                 /*volná pozice v buf*/
getch()                  /*načtení vráceného znaku*/
{
      return((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch(c)              /*vrať znak zpět do vstupu*/
int c;
{
      if ( bufp > BUFSIZE)
           printf("ungetch: too many characters\n");
      else
           buf[bufp++] = c;
}
```

Pro vrácené znaky jsme použili pole místo jednoho znaku pro obecnost a pozdější použití.

Cvičení 4-4. Napište funkci ungets(s), která vrací celý řetězec zpátky do vstupu. Musí ungets něco vědět o poli *buf* a indexu bufp nebo stačí použít pouze ungetch?
Cvičení 4-5. Předpokládejme, že nikdy nebude potřeba vracet více než jeden znak. Modifikujte odpovídajícím způsobem funkce getch a ungetch.
Cvičení 4-6. Naše funkce getch a ungetch nefungují správně, je-li načteno EOF. Rozhodněte, co by se mělo stát, je-li EOF vráceno zpět a zdůvodněte a implementujte svůj názor.

4.6 Statické proměnné

Statické proměnné jsou třetím druhem proměnných vedle externích a automatických, se kterými jsme se již seznámili. Statické proměnné mohou být buď **interní** nebo **externí**. Interní statické proměnné jsou lokální, místní, dané funkcí tak jako automatické proměnné, ale na rozdíl od nich existují trvale. To znamená, že interní statické proměnné tvoří stálou a privátní „paměť“ funkce. Znakové řetězce, které se vyskytují uvnitř funkce jsou interní a statické (např. argument funkce printf).

Externí statické proměnné mají platnost ve zbytku zdrojového souboru, ve kterém jsou deklarovány, ale v jiných souborech již platnost nemají. Externí statické proměnné slouží tedy k uschování jmen jako *buf* a *bufp* ve funkcích getch a ungetch. Tato jména musí být externí, aby je bylo možno sdílet, ale neměla by být přístupná uživateli funkcí getch a ungetch, aby nevznikl konflikt. Jestliže jsou tyto dvě funkce a tyto dvě proměnné překládány současně v jednom souboru jako

```
static char      buf[BUFSIZE];   /*guffer pro ungetch*/
```

```
static int      bufp = 0;  /*další volná pozice v buf*/
getch()  (...)
ungetch(c)  (...)
```

potom žádná další funkce nemůže pracovat s proměnnými *buf* a *bufp*; tato jména tedy vlastně nebudou kolidovat se jmény stejnými v jiných souborech téhož programu.

Statická paměť, ať již interní nebo externí, je specializována slovem static, které se uvede před normální deklaraci. Tyto proměnné jsou externí, jsou-li definovány mimo funkce a interní, jsou-li definovány uvnitř nějaké funkce.

Funkce samy jsou vlastně externí objekty: jejich jména mají obecnou platnost. Je ovšem také možné, aby funkce byla definována jako statická. Potom má platnost pouze v tom souboru, kde je deklarována.

„Statické“ neznamená v C pouze stálost, ale také vlastnost, která může být nazvána „privátnost“. Interní statické proměnné patří jen jedné funkci, externí statické objekty (tj. proměnné nebo funkce) mají platnost pouze ve zdrojovém souboru, kde jsou definovány a jejich jména nemají žádnou souvislost se stejnými jmény v jiných souborech.

Externí statické proměnné a funkce umožňují uschovávat data a statické funkce, které s nimi manipulují tak, že nemůže dojít ke kolizi s jinými funkcemi. Např. getch a ungetch tvoří „modul“ pro vstup a navrácení znaku; proměnné *buf* a *bufp* by měly být statické, aby nebyly dosažitelné z venku. Funkce push, pop a clear vytvářejí stejným způsobem „modul“ pro operace se zásobníkem a proto jsou proměnné *val* a *sp* definovány jako externí statické.

4.7 Proměnné typu registr

Čtvrtý a poslední typ proměnné je nazýván **registr**. Deklarace register říká překladači, že proměnná bude hodně využívána. Když je to možné, tak jsou tyto proměnné uloženy přímo do registru počítače, což se může projevit zmenšením a zrychlením programu. Deklarace register má následující formu

```
register int   x;
register char  c;
atd.
```

Slovo int může být vynecháno. Typ register může být použit pouze pro automatické proměnné a pro formální parametry funkcí. V druhém případě vypadá deklarace takto

```
f(c,n)
register int c, n;
{
      register int i;
      ...
}
```

Ve skutečnosti pro tyto proměnné platí určitá omezení, která závisí na použitém počítači. Pouze několik proměnných ve funkci může být tohoto typu a rovněž platí omezení pro typ proměnné. Slovo register je ignorováno pokud bylo nesprávně aplikováno nebo pokud počet těchto proměnných překračuje určitou mez. Rovněž není možné určit adresu proměnné typu register (o tom více v kapitole 5). Omezení se mění s typem počítače. Např. na PDP-11 jsou brány v úvahu pouze první tři deklarace register. Typ proměnných musí být int, char nebo pointer.

4.8 Blokové struktury

Jazyk C není jazykem blokových struktur v tom slova smyslu jako ALGOL nebo PL/1, v němž funkce nemohou být definovány uvnitř jiných funkcí. Na druhé straně ale mohou být proměnné definovány do blokových struktur. Za deklaracemi proměnných (včetně inicializace) může následovat levá závorka, která uvádí **libovolný** složený příkaz, ne pouze ten příkaz kterým začíná funkce. Proměnné deklarované tímto způsobem překrývají stejně nazvané proměnné ve vnějším bloku a zůstávají v platnosti až do výskytu pravé závorky. Např. v

```
if (n > 0)
{
   int i;       /*definice nové proměnné*/
   for (i = 0; i < n; i++)
      ...
}
```

je platnost proměnné i v příkazu for a nemá vztah k žádné jiné proměnné i v programu. Blokovou strukturu je možno aplikovat i na externí proměnné. Máme-li danou deklaraci

```
int x;
f()
{
   double x;
   ...
}
```

tak potom uvnitř funkce f je x proměnnou typu double a vně je x externí proměnnou typu int. Totéž platí i pro formální parametry

```
int z;
f(z)
double z;
{
   ...
}
```

Uvnitř funkce f je z formálním parametrem a nikoli externí proměnnou.

4.9 Inicializace

O inicializaci jsme se již mnohokrát zmiňovali, ale vždy jen okrajově. V tomto odstavci shrneme některá pravidla zvláště potom, co jsme si uvedli nové typy proměnných.

Externí a statické proměnné mají na začátku **nulové hodnoty**. Automatické proměnné a proměnné typu register jsou **nedefinované**. Jednoduché proměnné (ne pole ani struktury) mohou být inicializovány při deklaraci použitím znaménka = a hodnoty:

```
int x = i;
char squote = '\'; long day = 60 * 24;   /*minuty dne*/
```

Externí a statické proměnné jsou inicilizovány jednou — v době překladu. Automatické proměnné a proměnné typu register jsou inicializovány při každém volání funkce nebo vyvolání bloku.

Pro inicializaci automatických proměnných a proměnných typu register nemusí být použity jenom konstanty. Mohou to být libovolné výrazy, v nichž vystupují předem definované proměnné nebo dokonce i funkce. Např. inicializace v programu z kapitoly 3 může být napsána takto

```
binary (x,v,n)
int x, v[], n;
{
      int low = 0;
      int high = n-1;
      int mid;
      ...
}
```

místo

```
binary (x,v,n)
int x, v[], n;
{
      int low, high, mid;
      low = 0;
      high = n-1;
      ...
}
```

Ve skutečnosti je inicializace automatických proměnných pouze zkrácením přiřazovacího příkazu. Je pouze věcí vkusu, který způsob použijeme. My používáme převážně explicitní přiřazení, protože přiřazení v deklaraci není tak zřetelné.

Automatická pole nemohou být inicializována. Externí a statická pole mohou být inicializována seznamem proměnných uvedených ve složených závorkách. Např. program pro počítání znaků z kapitoly 1, který začíná

```
main ()       /*počítej čísla, mezery, jiné*/
{
      int c, i, nwhite, nother;
      int ndigit[10];
      nwhite = nother = 0;
      for (i = 0; i < 10; i++)
            ndigit[i] = 0;
      ...
}
```

může být napsán takto

```
int nwhite = 0;
int nother = 0;
int ndigit[10] = {0,0,0,0,0,0,0,0,0,0};
```

```
main ()      /*počítej čísla, mezery, jiné*/
{
      int c, i;
      ...
}
```

Tato inicializace není ve skutečnosti nutná, protože pole je automaticky nulováno, ale je dobré to udělat explicitně. Jestliže je počet čísel v seznamu kratší než je rozměr pole, tak ostatní prvky budou vynulovány. Na druhé straně větší počet způsobuje chybové hlášení a v seznamu není možné používat opakovače a ani není možné inicializovat pole od poloviny.

Znakové pole je možné inicializovat i jinak; může být použit přímo řetězec

```
char pattern[] = "the";
```

To je kratší forma ekvivalentního zápisu

```
char pattern[] = {'t', 'h', 'e', '\0'};
```

Když není uveden rozměr pole, počítač zjistí délku z inicializačního příkazu. V tomto našem příkladě je délka 4 (3 znaky a ukončující znak \0).

4.10 Rekurze

Funkce v jazyku C mohou být používány rekurzívně: to znamená, že funkce může volat sama **sebe** buď přímo nebo nepřímo. Klasickým příkladem je tisk čísla jako znakového řetězce. Jak už jsme se zmínili dříve, čísla jsou generována v opačném pořadí, ale tisknuta musí být správně. Tento problém je možno řešit dvojím způsobem. Jeden způsob je uložit znaky do pole a tisknout je v opačném pořadí, tak jak jsme to udělali v kapitole 3 ve funkci itoa. První verze je udělána tímto způsobem.

```
printd(n)   /* tisk n */
int n;
{
      char s[10];
      int i;
      if (n < 0)
      {
          putchar ('-');
          n = -n;
      }
      i = 0;
      do
      {
           s[i++] = n % 10 + '0'; /*další znak*/
      } while ((n /= 10) > 0);        /*dělení*/
      while (--i >= 0)
            putchar (s[i]);
}
```

Druhá verze používá rekurzi. Funkce printd při každém vyvolání volá nejprve sama sebe.

```
printd(n)       /* tisk n (rekurzivně)*/
int n;
{
        int i;
        if (n < 0)
        {
                putchar ('-');
                n = -n;
        }
        if ((i = n / 10) != 0)
                printd(i);
        putchar (n % 10 + '0');
}
```

Když funkce volá sebe sama, tak pokaždé má „čerstvou" sadu automatických proměnných, které jsou úplně nezávislé na předchozí sadě. Tak při printd(123) má první printd $n = 123$. Ta volá druhou printd s $n = 12$ a potom tiskne 3. Stejným způsobem druhá printd předává 1 třetí printd (ta ji vytiskne) a potom sama tiskne 2.

Obecně vzato, rekurze nešetří paměť, protože někde musí existovat zásobník, kam se proměnné ukládají. Navíc není ani rychlejší. Rekurze je ale zato mnohem kompaktnější a umožňuje snažší zápis a lepší porozumění. Rekurze je speciálně výhodná pro rekuzívně definované struktury dat jako jsou stromy. O tom více v kapitole 6.

Cvičení 4.7. Použijte ideí z printd a napište novou funkci itoa, tj. převeďte integer na řetězec znaků použitím rekurze.
Cvičení 4.8. Napište rekurzivní verzi funkce reverse(s), která obrací řetězec s.

4.11 Preprocesor jazyka C

Jazyk C umožňuje rozšíření jazyka užitím jednoduchých makroinstrukcí. Příkaz *#define* je jednou z nejrozšířenějších. Další makroinstrukcí je vkládání obsahu jiných souborů v průběhu překladu.

4.11.1 Vkládání souboru

Každá řádka, která má následující tvar

```
#include "filename"
```

je nahrazena obsahem souboru *filename*. (Uvozovky jsou povinné). V souboru se na začátku často objevují jeden nebo dva takové řádky. Je tím vkládán common, příkazy *#define* a deklarace extern pro globální proměnné. Vkládaný soubor může obsahovat další *#include*.

Příkaz *#include* je doporučován pro propojení deklarací velkého programu. Zaručuje totiž, že všechny zdrojové soubory budou obsahovat shodné definice a deklarace proměnných a tak se eliminuje možnost ošklivých chyb. Změní-li se ovšem obsah vkládaného souboru, všechny soubory na něm závislé musí být znovu přeloženy.

4.11.2 Makroinstrukce

Definice ve tvaru

```
#define  YES  1
```

je tou nejjednodušší formou makroinstrukce — nahrazuje jméno řetězcem znaků. Jména v definici *#define* mají stejnou formu jako identifikátory v jazyku C. Text, kterým jsou nahrazovány je libovolný. Normálně je text celý zbytek řádky. Dlouhé definice mohou pokračovat na dalším řádku, mají-li jako poslední znak \. „Obor působnosti“ jména definovaného v *#define* je od bodu definice do konce souboru. Jména mohou být definována znovu a definice mohou používat definice předchozí. Jména nejsou nahrazována jestliže se vyskytují v uvozovkách. Např. je-li YES definované jméno, tak v příkazu *printf*("YES") nebude nahrazeno. Protože implementace příkazu *#define* je makroinstrukce a není součástí překladače, není mnoho gramatických omezení. Např. vyznavači ALGOLU mohou definovat:

```
#define then
#define begin {
#define end   ; }
```

a psát

```
if (i > 0) then
   begin
      a = 1;
      b = 2
   end
```

Rovněž je možné definovat makra s argumenty, takže náhrada je závislá na způsobu volání. Jako příklad uvedeme makro max:

```
#define    max(A, B)  ((A) > (B) ? (A) : (B))
```

Potom řádka

```
x = max(p+q, r+s);
```

bude nahrazena řádkou

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Je to funkce, která ze dvou proměnných vybírá větší. Nakládáme- li s argumenty konstantně, tak funkce může mít argumenty nejrůznějšího typu. Není nutne mít různé funkce max pro různé typy dat, jak by tomu bylo při volání funkce.

Zamyslíme-li se nad funkcí max nahoře, uvědomíme si jistě nedokonalost. Výraz je vyčíslován dvakrát. To je nepříjemné používáme-li vedlejší efekty jako je volání funkcí a přírůstkové operátory (++, −−). Rovněž je třeba správně používat závorky, aby pořadí vyčíslení bylo správné. Uvažujte, co se stane, je- li makro

```
#define   square(x)   x * x
```

vyvoláno takto: square($z+1$). Existují také čistě lexikální problémy: **nesmí** být mezera mezi názvem makra a levou závorkou, která obsahuje seznam argumentů.

Přes všechno je používání makra výhodné. Jedním z praktických příkladů je standardní knihovna vstupu a výstupu, která bude popsána v kapitole 7, kde jsou getchar a putchar definovány jako makra. Další možnosti jsou popsány v příloze A.

Cvičení 4-9. Definujte makro swap(x, y), které vyjmenuje své dva argumenty. (Pomůže vám bloková struktura).

Kapitola 5

Pointery a pole

Pointr (ukazatel) je proměnná, která obsahuje adresu jiné proměnné. V jazyku C se pointrů hojně používá. Částečně proto, že je to často jediná možnost pro vykonání výpočtu a částečně proto, že vedou ke kompaktnějšímu kódu.

Pointry jsou házeny do jednoho pytle spolu s příkazy `goto` jako příkazy, které dokážou nádherným způsobem vytvářet programy, kterým není vůbec rozumět. To je částečně pravda tehdy, nejsou-li využívány rozumně a opatrně. Je totiž velice snadné vytvořit pointer, který ukazuje někam, kam to neočekáváme. Naopak jsou-li pointry využívány disciplinovaně, můžeme napsat programy jasně a jednoduše. Tuto vlastnost pointrů se budeme snažit popsat.

5.1 Pointry a adresy

Protože pointry obsahují adresy objektů, je možné dosáhnout objektu „nepřímo“ — právě pomocí pointru. Předpokládejme, že x je proměnná typu `int` a px je pointr, vytvořený zatím nespecifikovaným způsobem. Unární operátor & udává adresu objektu, a tak příkaz

```
px = &x;
```

přiřazuje adresu proměnné x do proměnné px; říkáme, že px „ukazuje“ na x. Operátor & může být aplikován pouze na proměnné a prvky pole; výraz ve tvaru $\&(x+1)$ nebo $\&3$ je nedovolený. Není rovněž možné získat adresu proměnné typu `register`.

Unární operátor $*$ nakládá se svým operandem jako s adresou a z dané adresy vybírá obsah. Proto je-li y proměnná `int`, tak

```
y = *px;
```

přiřazuje proměnné y to, na co ukazuje pointr px. Sekvence

```
px = &x;
y = *px;
```

je ekvivalentní s příkazem

```
y = x;
```

Rovněž je nutné deklarovat všechny proměnné takto:

```
int x, y;
int *px;
```

S deklarací proměnných x a y jsme se již setkali dříve. Deklarace pointru je ale novinkou.

```
int *px;
```

je mnemonikem: říká, že kombinace $*px$ je typu int, což znamená, že objeví-li se px v kontextu, $*px$ je ekvivalentní proměnné typu int. Ve skutečnosti syntaxe deklarace tohoto typu proměnné simuluje syntaxi výrazu, ve kterém se daná proměnná vyskytuje. To je užitečné ve všech komplikovaných deklaracích. Např. deklarace

```
double atof(), *dp;
```

určuje, že ve výrazu mají funkce atof() a $*dp$ hodnotu typu double.

Měli byste si uvědomit, že v deklaraci je pointr svázán s určitým objektem, na který ukazuje. Pointry se mohou objevit ve výrazech. Např. ukazuje-li px na celočíselnou proměnnou x, potom se $*px$ může objevit všude tam, kde x.

```
y = *px + 1;
```

přiřazuje proměnné y hodnotu o jednu větší než x.

```
printf("%d\n", *px);
```

tiskne obsah proměnné x.

```
d = sqrt((double) *px);
```

přiřazuje proměnné d odmocninu proměnné x, která je předtím převedena na typ double (viz kap. 2).

Ve výrazech typu

```
y = *px + 1;
```

mají unární operátory $*$ a & větší prioritu než operátory aritmetické. Brzy se vrátíme k tomu, co znamená

```
y = *(px + 1);
```

Odkaz na pointr se může rovněž objevit na levé straně přiřazovacího příkazu. Ukazuje-li px na proměnnou x, potom

```
*px = 0;
```

nuluje proměnnou x, a

```
*px += 1;
```

ji zvětšuje o jednotku zrovna tak jako

```
(*px)++;
```

V tomto příkazu jsou závorky nezbytné. Bez nich by byla jednička přičtena k px a ne k tomu, na co px ukazuje, protože unární operátory jako $*$ a $++$ jsou prováděny zprava doleva.

Protože jsou pointry proměnné, může s nimi být nakládáno jako s normálními proměnnými. Jestliže je py pointr na proměnnou int, potom

```
py = px;
```

zkopíruje obsah proměnné px do py. Potom py ukazuje na stejné místo jako px.

5.2 Pointry a argumenty funkcí

Protože proměnné jsou předávány funkcím „hodnotou“, tak funkce nemůže přímo změnit hodnoty těchto proměnných ve volající funkci. Co musíte udělat, chcete-li opravdu změnit obyčejný argument? Např. podprogram pro třídění může vyměnit dva prvky pomocí funkce swap. Nestačí ale napsat pouze

```
swap(a, b);
```

je-li funkce swap nadefinována takto

```
swap(x, y)      /*chybně!*/
int x, y;
{
      int temp;
      temp = x;
      x = y;
      y = temp;
}
```

Funkce swap nemůže ovlivnit argumenty ve volající jednotce, protože jsou předávány hodnotou.

Naštěstí existuje způsob, jak může dosáhnout zadaného efektu. Volající program bude předávat **pointry** proměnných:

```
swap(&a, &b);
```

Protože operátor & udává adresu proměnné, tak $\&a$ je pointr na a. Ve funkci swap musí být argumenty deklarovány jako pointry a skutečné parametry jsou jejich prostřednictvím ovlivňovány.

```
swap(px, py)      /*výměna *px a *py*/
int *px, *py;
{
      int temp;
      temp = *px;
      *px = *py;
      *py = temp;
}
```

Pointrů jako argumentů funkcí se používá hojně tehdy, vyžadujeme-li, aby funkce vracela více než jednu hodnotu (můžeme říci, že funkce swap vrací dvě hodnoty — nové hodnoty svých argumentů). Jako příklad uvažujme funkci getint, která zajišťuje čtení čísel ve volném formátu rozdělením vstupní sekvence na celočíselné hodnoty; při jednom vyvolání jedno číslo. getint vrací hodnotu, která byla načtena, nebo EOF, když narazila na konec vstupu. Tyto hodnoty musí být vráceny v různých proměnných, protože ať už pro EOF zvolíme jakoukoli hodnotu, mohlo by dojít ke kolizi.

Jedno z řešení je založeno na funkci scanf, která bude popsána v kapitole 7. Tato funkce využívá toho, že getint vrací jako hodnotu EOF, když je nalezen konec souboru. Každá jiná hodnota znamená, že bylo načteno číslo. Numerická hodnota načteného čísla je předávána argumentem, který musí být pointr. Tento způsob odděluje určování konce souboru od předání číselných hodnot. Následující cyklus vyplňuje pole celými čísly, načtenými funkcí getint:

```
int n, v, array[SIZE];
for(n = 0; n < SIZE && getint(&v) != EOF; n++)
      array[n] = v;
```

Při každém volání je proměnné v přiřazeno načtené číslo. Uvědomte si, že je nezbytné uvést $\&v$ jako argument funkce getint. Použijeme-li jako argument jenom v, bude ohlášena adresová chyba, protože getint počítá s tím, že argument je pointr.

Funkce getint je modifikací funkce atoi, kterou jsme tvořili již dříve:

```
getint(pn);      /*načti číslo ze vstupu*/
int *pn;
{
      intc, sign;
      while((c = getch()) == ' ' || c == '\n' || c
=='\t')
            ;           /*ignoruj oddělovače*/
      sign = 1;
      if(c == '+' || c == '-')
      {           /*znaménko*/
            sign = (c == '+') ? 1 : -1;
            c = getch();
      }
      for(*pn = 0; c >= '0' && c <= '9'; c = getch())
            *pn = 10 * *pn + c - '0';
      *pn *= sign;
      if(c != EOF)
            ungetch(c);
      return(c);
}
```

Uvnitř funkce getint je $*pn$ použita jako normální proměnná typu int. Rovněž jsme použili funkci getch a ungetch (popsané v kapitole 4), takže znak, který byl načten navíc, může být vrácen zpátky na vstup.

Cvičení 5-1. Napište funkci getfloat, která načítá číslo float a je analogickou k funkci getint. Jaký typ bude funkce getfloat vracet jako hodnotu?

5.3 Pointry a pole

V jazyce C je úzký vztah mezi pointry a poli. Dokonce tak úzký, že s poitry a poli může být nakládáno stejně. Každá operace s prvkem pole může být provedena s pointry. Obecně je verze s pointry rychlejší, ale někdy je těžší k pochopení.

Deklarace

```
int a[10]
```

definuje pole o délce 10 jako blok deseti po sobě následujících prvků $a[0]$, $a[1]$, …, $a[9]$. Notace $a[i]$ znamená, že tento prvek je vzdálen i pozic od počátku. Jestliže je pa pointr na proměnnou typu integer definovaný takto

```
int *pa;
```

potom přiřazení

```
pa = &a[0];
```

nastavuje pa na nultý prvek pole a. pa obsahuje adresu prvku $a[0]$. Nyní příkaz

```
x = *pa;
```

zkopíruje obsah $a[0]$ do x.

Jestliže pa ukazuje na určitý prvek pole a, potom $pa + 1$ ukazuje na další prvek. Obecně $pa - i$ ukazuje na i-tý prvek vlevo a $pa + i$ na i-tý prvek vpravo. Proto když pa ukazuje na $a[0]$, tak

```
*(pa + 1);
```

ukazuje na prvek $a[1]$ $pa + i$ je adresa $a[i]$ a $*(pa + i)$ je obsah prvku $a[i]$.

Tyto poznatky platí nehledě na to, jakého typu jsou proměnné pole a. Definice „přičti 1 k pointru“ a obecně celá aritmetika pointrů je založena na tom, že přírůstek je vynásoben velikostí objektu, na který pointer ukazuje. Proto v $pa + i$ je i vynásobeno rozměrem objektu, na který ukazuje pointr pa.

Souhlas mezi indexováním a aritmetikou pointrů je zřejmý. Ve skutečnosti je odkaz na pole překladačem převeden na pointr na začátek pole. Z toho plyne, že název pole je vlastně pointr. To je velice užitečný závěr. Protože jméno pole je synonymem pro umístění nultého prvku, pak přiřazení

```
pa = &a[0];
```

může být zrovna tak napsáno

```
pa = a;
```

Ještě více překvapivý, alespoň na první pohled, je fakt, že odkaz na $a[i]$ může být napsán jako $*(a+i)$. Překladač jazyka C totiž výraz $a[i]$ převádí vždy na $*(a+i)$. Tyto formy jsou naprosto ekvivalentní. Aplikujeme-li operátor & na obě části této ekvivalence dostaneme, že $\&a[i]$ a $a + i$ jsou rovněž identické: $a + i$ je adresa i-tého prvku pole a. Druhou stranou této mince je to, že je-li pa pointr, tak může být používán s indexem: $pa[i]$ je totožné s $*(pa + i)$. Krátce řečeno indexový výraz z libovolného pole může být napsán jako pointr a naopak; dokonce ve stejném příkazu.

Je jenom jeden rozdíl mezi jménem pole a pointrem, který si musíme uvědomit. Pointr je proměnná a proto $pa = a$ a $pa++$ jsou dovolené operace. Název pole je naproti tomu **konstanta** a ne proměnná. Proto jsou výrazy typu $a = pa$ nebo $a++$ nebo $p = \&a$ neplatné.

Když je funkci předáváno jméno pole, tak je předána adresa počátku pole. Uvnitř volané funkce je ale argument již normální proměnnou a tak je jméno pole opravdu pointr, tj. proměnná obsahující adresu. Tento fakt můžeme použít k napsání nové verze funkce strlen, která počítá délku řetězce

```
strlen(s)      /*zjištění délky řetězce*/
char *s;
{
```

```
        int n;
        for(n = 0; *s != n '\0'; s++)
              n++;
        return(n);
}
```

Zvětšování proměnné s je dovolené, protože to je pointr. $s++$ nijak neovlivňuje řetězec ve funkci, která strlen volá, ale jenom zvětšuje privátní kopii adresy tohoto pole.

Formální parametr ve funkci může být definován buď takto

```
char s[];
```

nebo takto

```
char *s;
```

Obě definice jsou totožné. Která z nich má být použita záleží výhradně na tom, v jakém tvaru budou psány výrazy této funkce. Jestliže je funkci předáno pole, funkce předpokládá, že ji bylo předáno pole nebo pointr a podle toho s ním také nakládá. Funkci je také možno předat pouze část pole předáním pointru začátku tohoto podřetězce. Např. je-li a pole, potom

```
f(&a[2])
```

a

```
f(a+2)
```

obojí předává adresu prvku $a[2]$, protože $\&a[2]$ a $a + 2$ jsou výrazy, které ukazují na třetí prvek pole a.

Ve funkci f může být provedena deklarace takto

```
f(arr)
int arr[];
{
...
}
```

nebo takto

```
f(arr)
int *arr;
{
...
}
```

Co se týče funkce f samé, tak vůbec nezáleží na tom, že argument ukazuje pouze na část nějakého většího pole.

5.4 Adresová aritmetika

Jestliže je p pointr, potom operace p++ zvětšuje p a p potom ukazuje na další prvek a p++ $= i$ zvětšuje p a i tak, že ukazuje o i prvků dál. Tyto a jim podobné operace jsou nejjednodušší a nejvíce používanou formou adresové aritmetiky.

C je konzistentní a regulární v přístupu k adresové aritmetice. Systém pointrů polí a adresové aritmetiky je hlavní silou jazyka. Ilustrujme některé vlastnosti tím, že napíšeme jednoduchý program pro alokaci paměti. Budou to dva podprogramy: alloc(n) vrací pointr p na n po sobě jdoucích znakových pozic, které mohou být použity pro uložení znaku. Funkce free(p) uvolňuje alokovanou paměť. Funkce jsou opravdu „základní“, protože funkce free musí být vyvolána, je-li vyvolána funkce alloc. To znamená, že paměť obhospodařovaná těmito funkcemi je zásobník neboli LIFO (last in first out). Ve standardní knihovně jsou obdobné funkce, které nemají daná omezení a v kap. 8 ukážeme zdokonalenou verzi. Zatím ale potřebujeme pouze triviální funkci alloc, abychom mohli alokovat paměť neznámé velikosti v různých chvílích.

Funkce alloc bude „podávat kousky pole“, nazvaného *allocbuf*. Toto pole bude soukromé pro funkce free a alloc. Protože tyto funkce pracují s pointry a nikoliv s idexy, jiné funkce nemusí o tomto poli nic vědět. Toto pole tedy může být deklarováno jako extern static, což znamená, že je místní ve zdrojovém souboru obsahující funkce alloc a free a mimo ně je „neviditelné“. Toto pole nemusí mít ani jméno. Může být získano tak, že požádáme operační systém o nějaký nepojmenovaný blok paměti.

Další informace, kterou potřebujeme znát je to, jak často je *allocbuf* využíváno. Budeme používat pointr nazvaný *allocp* na další volný prvek. Jestliže požádáme funkci alloc o n znaků, tak alloc zjistí, jestli je ještě místo v poli *allocbuf*. Jestliže je, tak alloc vrátí stávající hodnotu pointru *allocp* (tzn. začátek volného bloku) a zvýší hodnotu *allocp* o n. free(p) nastaví *allocp* na p, když p je uvnitř *allocbuf*.

```
      #define      NULL 0               /*pointr pro chybové
hlášení*/
      #define      ALLOCSIZE 1000      /*maximální dosažitelná
paměť*/
      static char allocbuf[ALLOCSIZE];             /*paměť pro
alloc*/
      static char *allocp = allocbuf;           /*další volná
pozice*/
      char *alloc(n)                       /*vrať ukazatel na n
znaků*/
      int n;
      {
            if(allocp + n <= allocbuf + ALLOCSIZE)
            {            /*sedí to*/
                  allocp += n;
                  return(allocp - n);      /*staré p*/
            }
            else                         /*málo místa*/
                  return(NULL);
      }

      free(p)                           /*volná paměť*/
```

```
char *p;
{
      if(p >= allocbuf && p < allocbuf + ALLOCSIZE)
            allocp = p;
}
```

Obecně může být pointr inicializován zrovna tak jako každá jiná proměnná, přestože normálně má význam jedině NULL, nebo výraz zahrnující adresy dříve definovaných dat shodného typu. Deklarace

```
static char *allocp = allocbuf;
```

definuje *allocp* jako znakový pointr a inicializuje jej tak, že ukazuje na *allocbuf*, což je vlastně první volné místo v paměti, když program začíná činnost. To by ale také stejně dobře mohlo být napsáno ve tvaru

```
static char *allocp = &allocbuf[0];
```

protože jméno pole je adresa jeho nultého prvku.

Podmínka

```
if(allocp + n <= allocbuf + ALLOCSIZE)
```

testuje, zda je ještě dostatek paměti pro n znaků. Jestliže je, tak *allocp* bude maximálně o jednu za koncem pole *allocbuf*. Jestliže požadavek může být splněn, tak alloc vrací normální pointr (všimněte si vlastní definice funkce). Jestliže požadavek nemůže být splněn, tak alloc musí nějak tuto skutečnost signalizovat. V jazyku C je zaručeno, že žádný pointr nebude obsahovat nulu jako hodnotu, a proto může být nula použita pro tuto signalizaci. Píšeme raději NULL než nula, protože to je jasnější. Pointrům obecně vzato nemohou být přiřazena čísla integer. Nula je ale speciální případ. Podmínky jako

```
if(allocp + n < = allocbuf + ALLOCSIZE)
```

a

```
if(p > = allocbuf && p < allocbuf + ALLOCSIZE)
```

ukazují další užitečné vlastnosti aritmetiky pointrů. Pointry mohou být za určitých okolností porovnávány. Jestliže p a q ukazují na prvky téhož pole, tak relace $<$, $>=$ atd. mají význam

```
p < q
```

může být pravda, např. jestliže p ukazuje na dřívější člen pole než q. Relace $==$ a $!=$ je rovněž možno použít. Libovolný pointr může být porovnán s NULL. Ale všechny výhody jsou pryč, porovnáváte-li pointry, které ukazují každý na něco jiného. Jestliže máte štěstí, tak program nebude pracovat na žádném počítači. Jestliže ale štěstí nemáte, tak program bude na jednom počítači řádně pracovat a na druhém zkolabuje.

Dále jsme si mohli všimnout, že pointr a číslo integer mohou být sečteny nebo odečteny. Konstrukce

```
p + n
```

znamená n-tý prvek za místem, kam ukazuje pointr p. Počítač vynásobí n odpovídajícím rozměrem objektu, na který pointr ukazuje. Např. na počítači PDP-11 je pro char násobný faktor 1, pro int a short 2, pro long a float 4 a pro double 8.

Odečítání pointrů má také význam; jestliže p a q ukazují do stejného pole, pak $p - q$ je počet prvků mezi p a q. Tohoto faktu může být použito pro novou variantu funkce strlen

```
strlen(s)      /*výpočet délky řetězce s*/
char *s;
{
       char *p = s;
       while(*p != '\0')
              p++;
       return(p-s);
}
```

V deklaraci je p inicializováno na s, to znamená, že ukazuje na jeho první znak. V cyklu while jsou znaky testovány na \0. Protože \0 je nula a protože while testuje, zda je výraz nulový, můžeme vynechat explicitní text a cyklus můžeme psát

```
while(*p)
       p++;
```

Protože p ukazuje na znak, p++ posouvá p na další znak a $p - s$ udává počet znaků, o který je p posunuto — tj. délka řetězce. Aritmetika pointrů je konzistentní. Jestliže pracujeme s float, p++ se posune na další float. Tak můžeme napsat další funkci alloc, která bude pracovat s float místo s char. Toho docílíme tím, že všude ve funkcích alloc a free nahradíme deklaraci char deklarací float. Operace s pointry budou prováděny opět správně.

Jiné operace s pointry, než o kterých jsme se zde zmínili, jsou nedovolené. Nemůžeme sčítat dva pointry, násobit je, nebo k nim přičítat čísla float a double.

5.5 Znakové pointry a funkce

Znaková konstanta psaná jako

```
"I am a string"
```

je znakovým polem. Ve vnitřní interpretaci překladač toto pole zakončuje znakem \0, takže program snadno nalezne konec. Požadavek na paměť je tady o jednotku vyšší než skutečná délka řetězce.

Pravděpodobně se řetězce nejčastěji vyskytují jako parametry funkcí

```
printf("hello, world\n");
```

Jestliže se takovýto řetězec znaků objeví v programu, tak přístup k němu je zprostředkován pointry. Funkce printf ve skutečnosti obdrží pointr na tento řetězec znaků.

Znaková pole nemusí ale být pouze argumenty funkcí. Jestliže je *message* deklarováno takto

```
char *message;
```

potom příkaz

```
message = "now is the time";
```

přiřadí *message* pointr na skutečný řetězec. Není to **kopie** řetězce. C neumožňuje práci s řetězci jako s jednotkami.

Další vlastnosti pointrů a polí budeme ilustrovat dvěma užitečnými funkcemi ze standardní knihovny vstupu a výstupu, které budou probrány v kapitole 7.

První funkcí je strcpy(s, t), která kopíruje řetězec t do řetězce s. Argumenty jsou v tomto pořadí podle analogie a přiřazovacím příkazem je

```
s = t;
```

První verze s použitím polí

```
strcpy(s, t)        /*kopíruj t do s*/
char s[], t[];
{
        int i;
        i = 0;
        while((s[i] = t[i]) != '\0')
                i++;
}
```

Pro srovnání nyní strcpy s pointry

```
strcpy(s, t)        /*kopíruj t do s; verze s pointry*/
char *s, *t;
{
        while((*s = *t) != '\0')
        {
                s++;
                t++;
        }
}
```

Protože argumenty jsou předávány hodnotou, strcpy může použít s a t jak chce.

Prakticky nebude strcpy napsána tak, jak jsme uvedli. Další možností je:

```
strcpy(s, t)        /*kopíruj t do s; 2. verze s pointry*/
char *s, *t;
{
        while((*s++ = *t++) != '\0')
                ;
}
```

Tato verze zvětšuje s i t v testovací části. Hodnota $*t{+}{+}$ má hodnotu znaku, na který t ukazuje ještě před zvětšením. Postfix ++ nemění t dokud nebyl vybrán prvek. Podobným způsobem je znak uložen na starou hodnotu s. Tento znak je také porovnán s \0. Výsledkem je, že jsou kopírovány znaky řetězce až po znak \0 včetně.

Znovu si uvědomíme, že porovnávání s \0 je zbytečné a napíšeme konečnou verzi

```
strcpy(s, t)      /*kopíruj t do s; 3. verze s pointry*/
char *s; *t;
{
      while(*s++ = *t++)
            ;
}
```

Přestože se to na první pohled může zdát nesrozumitelné, výhoda je zřejmá a tento tvar se nám musí vžít už jen proto, že se často v C programech užívá.

Druhou funkcí je funkce strcmp(s, t), která porovnává znakové řetězce s a t a vrací buď zápornou hodnotu, nulu nebo kladnou hodnotu podle toho, je-li řetězec s lexikálně menší, roven nebo větší než t. Vrácená hodnota je rozdílem prvních dvou znaků, ve kterých se řetězec s a t liší.

```
      strcpm(s, t) /*vrať<0 když s<t, 0 když s==t, >0 když
s>t*/
      char s[], t[];
      {
            int i;
            i = 0;
            while(s[i] == t[i])
                  if(s[i++] == '\0')
                        return(0);
            return(s[i] - t[i]);
      }
```

Verze s použitím pointrů:

```
strcmp(s, t)      /*dtto*/
char *s, *t;
{
      for( ; *s == *t; s++, t++)
            if(*s == '\0')
                  return(0);
      return(*s - *t);
}
```

Protože $++$ a $--$ mohou být buď před nebo za proměnnou, mohou se objevit i jiné kombinace $++$ a $--$. Např.

```
*++p
```

zvětšuje p **před** vybíráním znaku, na který ukazuje.

Cvičení 5-2. Napište pointerovou verzi funkce strcat, kterou jsme uvedli v kapitole 2. strcat(s, t) kopíruje řetězec t na konec řetězce s.
Cvičení 5-3. Napište makro pro strcpy.
Cvičení 5-4. Přepište programy z dřívějších kapitol s použitím pointrů.

5.6 Pointry nejsou celá čísla

V předchozích programech jste si mohli všimnout kavalírského přístupu k pointrům. Obecně platí, že na mnoha počítačích je pointrem přiřazeno celé číslo a naopak. To ale vedlo k přílišné svobodě. V podprogramech, které vrací pointry, které jsou předávány dále, jsou vynechány deklarace pointru. Uvažujme např. funkci strsave(s), která ukládá řetězec s na bezpečné místo, které získá pomocí funkce alloc. Správně má být napsána takto:

```
char *strsave(s)      /*ulož někam řetězec*/
char *s;
{
      char *p, *alloc();
      if((p = alloc(strlen(s) + 1)) != NULL)
            strcpy(p, s);
      return(p);
}
```

Prakticky se ale vynechává deklarace

```
strsave(s)      /*ulož někam řetězec*/
{
      char *p;
      if((p = alloc(strlen(s) + 1)) != NULL)
            strcpy(p, s);
      return(p);
}
```

Tato verze bude fungovat na mnoha typech počítačů, protože implicitní hodnota pro funkce a argumenty je int a pointr a int je možno často zaměnit navzájem. Nicméně je tento způsob zápisu riskantní, protože příliš závisí na použitém počítači. Moudřejší je psát **řádně** všechny deklarace. (Program lint nás bude při takových konstrukcích varovat).

5.7 Vícerozměrná pole

C umožňuje používání vícerozměrných polí, přestože jsou v praxi mnohem méně používány než pole nebo pointry. V tomto odstavci si ukážeme některé jejich vlastnosti.

Uvažujme o problému konverze data ze dne a měsíce na den v roce a naopak. Např. 1. březen je 60. den nepřestupného roku a 61. den roku přestupného. Budeme definovat dvě funkce: day_of_year bude konvertovat měsíc a den na den v roce a month_day bude konvertovat den v roce na měsíc a den. Protože tato funkce vrací dvě hodnoty, tak *month* a *day* budou pointry

```
month_day(1977, 60, &m, &d)
```

nastaví m na 3 a d na 1 (1.březen).

Obě funkce potřebují shodné informace: tabulku počtu dní každého měsíce. Protože se počet dní liší podle toho, je-li rok přestupný nebo ne, je jednodušší použít dvourozměrné pole. Funkce bude vypadat takto:

```
static int day_tab[2][13] =
{
(0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31),
(0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31),
}
;
day_of_year(year, month, day)
int year, month, day;
{
int i, leap;
leap = year % 4 == 0 && year % 100 != 0 || year % 400 ==
0;
   for(i = 1; i < month; i++)
         day += day_tab[leap][i];
   return(day);
}
month_day(year, yearday, pmonh, pday)
int year, yearday, *pmonth, *pday;
{
int i, leap;
leap = year % 4 == 0 && year % 100 != 0 || year % 400 ==
0;
   for(i = 1; yeardy > day_tab[leap][i]; i++)
         yearday -= day_tab[leap][i];
   *pmonth = i;
   *pday = yearday;
}
```

Pole day_tab je externí oběma funkcím a je prvním vícerozměrným polem, se kterým jsme se setkali. V jazyku C je dvojrozměrné pole — pole, jehož prvky jsou zase jednorozměrná pole. Proto je psáno

```
day_tab[i][j]
```

spíše než

```
day_tab[i, j]
```

jako je tomu v jiných jazycích. Jinak se s dvojrozměrnými poli nakládá úplně stejně. Prvky jsou skládány po sloupcích, což znamená, že pravý index se mění nejrychleji. Pole je inicializováno seznamem hodnot v závorkách. Každá řádka dvojrozměrného pole je inicializována odpovídajícím podseznamem. První prvek pole *day_tab* jsme inicializovali na nulu, takže můžeme pracovat s celými čísly 1–12 namísto 0–11. Jeden prvek navíc zde není rozhodující a tak se vyhneme složitostem s přepisem indexu pole.

Jestliže je funkci předáváno dvojrozměrné pole, tak v deklaraci argumentu **musí** být uveden počet sloupců pole. Deklarace počtu řádek není rozhodující, protože je předáván pointr. V tomto případě je to pointr na objekty, které jsou 13-ti dimenzionálními poli. Je-li tedy předáváno pole *day_tab* funkci, tak deklarace ve funkci f musí vypadat takto

```
f(day_tab)
```

```
int day_tab[2][13];
{
...
}
```

Deklarace argumentů může stejně dobře vypadat takto

```
int day_tab[][13];
```

protože počet řádek není důležitý, nebo takto

```
int *(day_tab[13]);
```

V této deklaraci je uvedeno, že argument je pointer na pole 13 celých čísel. Kulaté závorky jsou nezbytné, protože hranaté závorky mají vyšší prioritu než $*$. Bez závorek bude

```
int(*day_tab)[13];
```

deklarováno pole 13 pointrů. To uvidíme v dalším odstavci.

5.8 Pole pointrů. Pointry na pointry

Protože jsou pointry proměnné, tak lze předpokládat, že můžeme využít pole pointrů. Ilustrujeme to na programu, který bude třídit soubor řádek podle abecedy. (Bude to zjednodušená forma utility `sort` systému UNIX.)

V kapitole 3 jsme uvedli funkci `shell`, která třídí pole celých čísel. Použijeme stejný algoritmus s tím, že nyní musíme nakládat s řetězci čísel neznámé délky, které nemohou být porovnávány nebo přemístěny jednou operací. Potřebujeme vhodnou a dostatečně efektivní datovou reprezentaci řádek proměnné délky.

Nyní vstoupí na scénu pole pointrů. Jestliže jsou řádky, které mají být tříděny, uloženy v jednom poli bez mezer mezi sebou, tak každá řádka může být reprezentována pointrem na její první znak. Pointry mohou být uloženy do pole. Dvě řádky mohou být potom porovnávány funkcí `strcmp`. Jestliže chceme prohodit dvě řádky, potom stačí prohodit pouze pointry na ně. To značně zjednodušuje celou operaci.

Třídící postup sestává ze tří částí:

— načtení všech řádek ze vstupu

— třídění

— vytištění seřazených řádek

Jako obvykle rozdělíme program na funkce, které budou vykonávat jednotlivé kroky a hlavní program, který vše bude řídit.

Odložme na chvíli krok třídění a věnujme se datovým strukturám vstupu a výstupu. Vstupní funkce musí číst a uchovávat znaky každé řádky a sestavit pole pointrů na tyto řádky. Protože vstupní funkce může nakládat pouze s konečným počtem řádek, mohla by vrátit nesmyslnou hodnotu, pokud by řádků bylo příliš mnoho. Výstupní funkce pouze řádky tiskne podle pořadí pole pointrů.

```
#define NULL 0
#define LINES 100           /*maximální počet řádků*/
main()                     /*třídění vstupních řádků*/
{
      char *lineptr[LINES];      /*pointry na řádky*/
      int nlines:           /*počet načtených řádků*/
      if((nlines = readlines(lineptr, LINES)) >= 0)
      {
            sort(lineptr, nlines);
            writelines(lineptr, nlines);
      }
      else
            printf("input too big to sort\n");
}


#define MAXLEN      1000
readlines(lineptr, maxlines)    /*čti vstupní řádky*/
char *lineptr[];
int maxlines;
{
      int len, nlines;
      char *p, *alloc(), line[MAXLEN];
      nlines = 0;
      while((len = getline(line,MAXLEN)) > 0)
            if(nlines >= maxlines)
                  return(-1);
            else if((p = alloc(len)) == NULL)
                  return(-1);
            else
            {
                  line[len-1] = '\0';  /*nový řádek*/
                  strcpy(p, line);
                  lineptr[nlines++] = p;
            }
      return(nlines);
}
```

Znak pro novou řádku je z konce řádek vymazán, aby neovlivnil pořadí pro třídění.

```
writelines(lineptr, nlines)           /*výpis řádku*/
char *lineptr[];
int nlines;
{
      int i;
      for(i = 0; i < nlines; i++)
            printf("%s\n", lineptr[i]);
}
```

Hlavní novinkou je deklarace *lineptr*:

```
char *lineptr[LINES];
```

která říká, že *lineptr* je pole prvků, z nichž každý je pointrem na proměnnou char. To znamená, že *lineptr*[*i*] je znakový pointr a ∗*lineptr*[*i*] je daný znak.

Protože je *lineptr* pole, které je předáváno funkci writelines tak s ním může být nakládáno jako s pointrem stejným způsobem, jako v našich dřívějších ukázkách. Funkce může být napsána takto:

```
writelines(lineptr, nlines)    /*piš vystupní řádky*/
char *lineptr[];
int nlines;
{
      while(--nlines >= 0)
            printf("%s\n", *lineptr++);
}
```

lineptr na začátku ukazuje na první řádku. Každý inkrement jej posouvá na další řádku a přitom se proměnná *nlines* zmenšuje.

Když jsme se postarali o vstup a výstup, můžeme přejít ke třídění. Program shell z kapitoly 3 potřebuje ale jisté změny: musí být modifikovány deklarace a srovnání musí být převedeno do speciální funkce. Základ algoritmu se nezměnil, což nám dokazuje, že je stále dobrý.

```
sort(v, n)          /*roztřiď řetězec v[0]...v[n-1]*/
char *v[];          /*vzestupně*/
int n;
{
      int gap, i, j;
      char *temp;
      for(gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i < n; i++)
                  for(j = i-gap; j >= 0; j -= gap)
                  {
                  if(strcmp(v[j], v[j+gap]) <=0)
                        break;
                  temp = v[j];
                  v[j] = v[j+gap];
                  v[j+gap] = temp;
                  }
}
```

Protože libovolný prvek *v* (neboli *lineptr*) je znakový pointr, také *temp* musí být znakový pointr.

Napsali jsme tento program tak, aby pracoval co nejrychleji. Mohl by být ovšem rychlejší. Mohl by totiž kopírovat řádky ze vstupu přímo do pole *array* a ne do pole *line* a potom dále. Je ale lepší udělat první verzi jasně a o „účinnost“ se starat později. Tato úprava by ale program nijak podstatně nezrychlila. Rozdíl by ale byl, pokud bychom použili nějaký lepší třídící algoritmus (např. QUICKSORT).

V kap. 1 jsme si ukázali, že příkazy while a for provádějí testování **před** prvním vykonáním těla cyklu. To nám zaručuje, že program bude fungovt správně, i když na vstupu nejsou žádné řádky. Je dobré si program pro třídění projít a zjišťovat, co se stane, nebyl-li načten žádný vstup.

Cvičení 5-5. Přepište readlines tak, aby řádky ukládal v jednotce main a ne ve funkci alloc. O kolik bude program rychlejší?

5.9 Inicializace pole pointrů

Napišme funkci month_name(n), která vrací pointer na řetězec, obsahující jméno n-tého měsíce. To je ideální aplikace interního statického pole. Funkce month_day obsahuje svoje vnitřní pole řetězce znaků a je-li vyvolána, vrací správný pointr na patřičné místo. V této části se budeme zabývat problémem inicializace pole jmen.

Syntaxe je obdobná jako dříve:

```
char *month_name(n)      /*vrať jméno n-tého měsíce*/
int n;
{
      static char *name[] =
      {
            "illegal month",
            "January",
            "February",
            "March",
            "April",
            "May",
            "June",
            "July",
            "August",
            "September",
            "October",
            "November",
            "December",
      }
      ;
      return((n < 1 || n > 12) ? name[0] : name[n]);
}
```

Deklarace proměnné *name*, která je polem znakových pointrů, je obdobná deklaraci *lineptr* v třídícím programu. Inicializace se provádí jednoduše seznamem znakových řetězců. Každý řetězec má v poli odpovídající místo. Přesněji řečeno znaky i-tého řetězce jsou někde uloženy a pointr na ně je uložen v $name[i]$. Protože není specifikována délka pole *name*, překladač ji sám zjistí z počtu inicializací.

5.10 Pointry a vícedimenzionální pole

Začátečníci jsou občas zmateni rozdílem mezi dvojdimenzionálním polem a polem pointrů, jako je např. pole jmen měsíců v předchozím příkladě. Máme-li dány deklarace

```
int a[10][10];
int *b[10];
```

tak je použití a a b obdobné v tom, že $a[5][5]$ a $b[5][5]$ jsou dovolené reference na jeden prvek int. a je opravdové pole. Bylo pro něj alokováno 100 prvků a používání normálního postupu při výpočtu indexu. V případě pole b je ale alokováno pouze 10 pointrů. Každý musí být nastaven tak, aby ukazoval na pole typu int. Když předpokládáme, že každý pointr ukazuje na pole int o rozměru 10, tak bude alokováno celkem 100 buněk plus 10 buněk pro pointry. Pole pointrů tedy potřebuje více paměti a může také požadovat explicitní inicializaci. Má ale také dvě výhody: adresování prvku je prováděno nepřímo pointrem a ne násobením a sčítáním jako u pole normálního a navíc řádky mohou mít proměnnou délku. To znamená, že ne každý prvek b musí ukazovat na pole o rozměru 10. Některý může ukazovat na 2 pole, druhý na 20 a další na žádné.

Přestože jsme se zde omezili na prvky typu int, tak největší použití pro pole pointrů je takové, jako v případě pole month_name: tj. ukládání znakových řetězců různé délky.

Cvičení 5-6. Přepište funkce day_of_year a month_day s použitím pointrů.

5.11 Argumenty ve tvaru příkazové řádky

Existuje způsob, jak předávat argumenty nebo parametry programu, který začíná svou činnost. Když je spouštěn program main, tak má dva argumenty. První z nich (obyčejně nazývaný *argc*) udává počet argumentů příkazové řádky, kterou byl program vyvolán. Druhý parametr (it argv) je pointr na pole znakových řetězců, které obsahují argumenty — vždy jeden na řetězec.

Nejjednodušší ilustrace použití nezbytných deklarací je program echo, který prostě tiskne argumenty. Potom příkaz

```
echo hello, world
```

bude mít výstup

```
hello, world
```

$argv[0]$ je jméno programu, který byl vyvolán a tak je $argc$ přinejmenším 1. V předchozím příkladě je $argc$ 3 a $argv[0]$ je „echo“, $argv[1]$ je „hello“, a $argv[2]$ je „world“. Prvním skutečným argumentem je $argv[1]$ a posledním $argv[argc\text{-}1]$. Je-li $argc$=1, potom nebyly zadány parametry. Zde je program echo:

```
main(argc, argv)      /*opakuj argumenty, 1.verze*/
int argc;
char *argv[];
{
      int i;
      for(i = 1; i < argc; i++)
```

```
                printf("%s%c", argv[i], (i < argc-1) ? ' ' :
'\n');
      }
```

Protože *argv* je pointr na pole pointrů, tak tento program můžeme napsat mnoha způsoby. Ukážeme 2 varianty.

```
main(argc, argv)      /*opakuj argumenty, 2. verze*/
int argc;
char *argv[];
{
      while(--argc > 0)
      printf("%s%c, *++argv, (argc > 1) ? ' ' : '\n');
}
```

Protože je *argv* pointrem na začátek pole řetězců argumentů, tak zvětšením o 1 (++*argv*) bude ukazovat na *argv*[1] a ne na *argv*[0]. Každé zvětšení ho posouvá na další argument. Ve stejném čase je *argc* zmenšováno. Když je nulové, tak již nejsou žádné další agrumenty.

```
main(argc, argv)      /*opakuj argumenty 3. verze*/
int argc;
char *argv[];
{
      while(--argc > 0)
      printf((argc > 1) ? "%s" : "%s\n", *++argv);
}
```

V této verzi je ukázáno, že argumenty funkce printf mohou být výrazy. Není to příliš časté, ale stojí to za zapamatování.

Ve druhém příkladě provedeme vylepšení programu pro vyhledávání řetězců z kap. 4. Tam jsme řetězec, který má být vyhledán začlenili do programu. Nyní tento program změníme tak, že řetězec bude zadáván jako argument (obdoba systémové funkce grep v UNIX).

```
#define MAXLINE 1000
main(argc, argv)    /*nalezni první výskyt řetězce s*/
int argc;
char *argv[];
{
      char line[MAXLINE]
      if(argc != 2)
            printf("Usage: find pattern\n");
      else
            while(getline(line, MAXLINE) > 0)
                  if(index(line, argv[1] >= 0)
                        printf("%s, line);
}
```

Na tomto základním modelu budeme ilustrovat další konstrukce s pointry. Předpokládejme, že chceme používat dva argumenty. Jeden říká: „vytiskni všechny řádky kromě těch, v nichž je daný řetězec“ a druhý „vytiskni každou řádku s číslem řádky“.

Začíná-li v C nějaký argument znakem minus, tak je to parametr. Vybereme si tuto konvenci: $-x$ (except) pro inverzi a $-n$ (number) pro číslování. Potom

```
find -x -n the
```

se vstupem ve tvaru

```
now is the time
for all good men
to come to the eid
of their party
```

vytiskne

```
2: for all good men
```

Parametry se mohou vyskytovat v libovolném pořadí a program by neměl být citlivý na to, kolik argumentů bylo zadáno. Konkrétně, volání programu index by nemělo referovat na *argv*[2], když tam byl zadán parametr a na *argv*[1], když tam parametr zadán nebyl. Navíc je pro uživatele výhodné, mohou-li být parametry slučovány. Tj.

```
find -nx the
```

Zde je program:

```
#define MAXLINE 1000
main(argv, argc)      /*nalezni řetězec*/
int argc;
char *argv[];
{
      char line[MAXLINE], *s;
      long lineno = 0;
      int except = 0, number = 0;
      while(--argc > 0 && (*++argv)[0] == '-')
            for(s = argv[0]+1; *s != '0'; s++)
                  switch(*s)
                  {
                  case 'x':
                        except = 1;
                        break;
                  case 'n':
                        numer = 1;
                        break;
                  default:
                        printf("find: illeagal
option%c\n", *s);
                        argc = 0;
                        break;
                  }
      if(argc != 1)
            printf("Usage: find -x -n pattern\n");
```

```
        else

                while(getline(line, MAXLINE) > 0)
                {
                        lineno++;
                        if((index(line, *argv) >= 0) != except)
                        {
                                if(number)
                                        printf("%ld: ", lineno);
                                printf("%s", line);
                        }
                }
}
```

argv je zvětšeno před každým parametrem a *argc* je zmenšeno. Jestliže nenastaly chyby, tak na konci musí být $argc{=}1$ a *argv* ukazovat na řetězec, který má být vyhledán. Uvědomte si, že $*{+}{+}argv$ je pointr na řetězec argumentů; $(*{+}{+}argv)[0]$ je jeho první znak. Závorky jsou zde nezbytné, protože jinak by byl tento výraz chápán takto: $*{+}{+}(argv[0])$, což je nesprávné. Další vhodnou formou zápisu je $**{+}{+}argv$.

Cvičení 5-7. Napište program add, který vyčísluje v obrácené polské notaci výraz, zadaný jako argument. Např. add 2 3 4 + $*$ vypočte $2 * (3{+}4)$.
Cvičení 5-8. Modifikujte program entab a detab (z kap.1) aby tabelátory byly zadávány jako argumenty. Použijte normální tabelátor, nebyl-li zadán žádný argument.
Cvičení 5-9. Rozšiřte entab a detab tak, aby umožňoval zadání ve tvaru entab $m + n$ což jest: tabelátor staví každý n-tý sloupec, počínaje na sloupci m.
Cvičení 5-10. Napište program tail, který tiskne posledních n řádek ze vstupu. Implicitně bude $n = 10$, ale může být změněno argumentem tail $-n$. Program by se měl chovat normálně bez ohledu na to, jak nesmyslná hodnota n je. Napište program tak, aby co nejlépe využíval paměť. Řádky by měly být ukládány stejným způsobem jako ve funkci sort a ne jako v dvoudimenzionálním poli konstantní délky.

5.12 Pointry funkcí

Funkce v C není proměnná, ale je možno definovat **pointr funkce**, se kterým může být manipulováno (může být předáván jako argument funkcím, ukládán do pole atd.). Budeme to ilustrovat tím, že zmodifikujeme třídící program tak, že bude- li uveden parametr $-n$, tak řady budou setříděny číselně a ne podle abecedy.

Třídění obvykle sestává ze tří částí: **porovnávání**, které určuje pořadí porovnávaného páru, **výměny**, která mění pořadí páru a **třídícího algoritmu**, který provádí porovnání a výměnu tak dlouho, dokud není vše utříděno. Třídící algoritmus je nezávislý na porovnávacích operacích a operacích výměny, takže předáním různých porovnávacích funkcí a funkcí výměny můžeme provádět třídění podle libovolných kritérií. Tento postup bude použit v novém třídícím programu.

Porovnání podle abecedy provádí funkce strcmp, výměnu funkce swap. Dále budeme potřebovat funkci numcmp, která řádky porovnává na základě číselné hodnoty a vrací stejnou indikaci jako funkce strcmp. Tyto tři funkce jsou deklarovány v jednotce main a jejich pointry jsou předávány funkci sort. Funkce sort volá funkce pomocí pointru.

```
#define LINES 100            /*maximální počet řádků*/
main(argc, argv)         /*setřídění vstupních řádků*/
int argc;
char *argv[];
{
      char *lineptr[LINES]; /*pointry na text. řádky*/
      int nlines;           /*počet načtených řádků*/
      int strcmp(),numcmp();    /*porovnávací funkce*/
      int swap();                     /*funkce výměny*/
      int numeric = 0;     /*1 pro numerickou výměnu*/
      if(argc > 1 && argv[1][0] == '-' && argv[1][1] ==
'n')
            numeric = 1;
            if((nlines = readlines(lineptr, LINES)) >= 0)
            {
            if(numeric)
               sort(lineptr,nlines,numcmp,swap);
            else
               sort(lineptr,nlines,strcmp,swap);
            writelines(lineptr,nlines);
            }
            else
                printf("input too big to sort\n");
}
```

strcmp, numcmp a swap jsou adresy funkcí. Protože víme, že jde o funkce, tak operátor & není nezbytný. Proto ho také není třeba uvádět před názvem pole. Překladač sám připraví adresy funkcí nebo polí. Funkce sort vypadá takto:

```
sort(v, n, comp, exch)     /*setřídění v[0]...v[n-1]*/
char *v[];
int n;
int (*comp)(),(*exch)();
{
      int gap, i, j;
      for(gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i<n; i++)
                  for(j = i-gap; j >= 0; j -= gap)
                  {
                      if((*comp)(v[j],v[j+gap]) <= 0)
                              break;
                       (*exch)(&v[j],&v[j+gap]);
                  }
}
```

Musíme dát pozor na deklarace. Deklarace

```
int (*comp)()
```

říká, že comp je pointr funkce, která vrací int. První pár závorek je nezbytný, protože bez nich

```
int *comp()
```

znamená, že comp je funkce, která vrací pointr, což jak vidíme je něco zcela odlišného.
Použití funkce comp v řádce

```
if((*comp) (v[j], v[j+gap]) <= 0)
```

je v souhlase s deklarací: *comp* je pointr na funkci, $*$comp je funkce a

```
(*comp)(v[j], v[j+gap])
```

je volání funkce. Závorky jsou nezbytné.
Ještě uvedeme funkci numcmp:

```
numcmp(s1, s2)      /*porovnávání s1 a s2 číselně*/
char *s1, *s2;
{
      double atof(), v1, v2;
      v1 = atof(s1);
      v2 = atof(s2);
      if(v1 < v2)
            return(-1);
      else if(v1 > v2)
            return(1);
      else
            return(0);
}
```

Nakonec uvedeme funkci swap, která mění dva pointry:

```
swap(px, py)      /*výměna *px a *py*/
char *px[], *py[];
{
      char *temp;
      temp = *px;
      *px = *py;
      *py = temp;
}
```

Cvičení 5-11. Upravte sort tak, aby bylo možno zadat parametr $-r$, který požaduje třídění sestupně. $-r$ musí rovněž pracovat s $-n$.
Cvičení 5-12. Přidejte parametr $-f$, který dává naroveň malá a velká písmena.
Cvičení 5-13. Přidejte parametr $-d$ („slovníkové srovnávání“). Potom budou porovnávány pouze písmena, čísla a mezery. Zkontrolujte, zda bude pracovat spolu s $-f$.

Kapitola 6

Struktury

Struktura je sestava jedné nebo více proměnných stejného nebo různého typu. Je označena jedním jménem, aby se s ní dalo dobře zacházet. (Struktury jsou někdy nazývány „rekordy“ např. v PASCALU.)

Klasickým příkladem jsou osobní data zaměstnanců na výplatní listině. „Zaměstnanec“ je popsán množinou atributů, jako je jméno, adresa, rodné číslo, plat atd. Některé z položek mohou být opět struktury: jméno má více složek, adresa a plat také. Struktury pomáhají organizovat složitá data speciálně v rozsáhlých programech, protože v mnoha situacích umožňují, aby s celou soustavou dat bylo nakládáno jako s jednou proměnnou. V této kapitole popíšeme, jak se používají struktury. Programy budou větší než ty, které jsme dosud poznali, ale budou ještě stále v rozumné velikosti.

6.1 Základy

Podívejme se znova na program z kapitoly 5 zabývající se datem. Datum je složeno z různých částí: den, měsíc, rok a den v roce a popř. jméno měsíce. Těchto pět údajů může být uloženo do jedné struktury:

```
struct date
{
    int day;
    int month;
    int year;
    int yearday;
    char mon_name[4];
};
```

Klíčové slovo `struct` uvádí deklaraci struktury, která je seznamem deklarací, vložených do závorek. Nepovinným paramatrem může být jméno, nazývané označení struktury (`tag`). Je uvedeno za slovem `struct`. V našem příkladu je to jméno `date`. Označení struktury označuje strukturu a může být použito jako zkratka pro podobnou deklaraci. Prvky nebo proměnné uvedené ve struktuře jsou nazývány jejími **členy**. Člen struktury nebo její označení může být shodné se jménem obyčejné proměnné, protože může být vždy odlišeno podle kontextu. Ovšem vhodné je používat shodné názvy pouze pro úzce svázané veličiny.

Pravá závorka ukončuje seznam členů a za ní může následovat seznam obyčejných proměnných. Tj.

```
struct (...) x, y, z;
```

je z hlediska syntaxe identické s

```
int x, y, z;
```

Deklarace struktury, za kterou nenásleduje seznam členů, nealokuje žádnou paměť. Popisuje pouze „šablonu“ struktury. Jestliže je struktura označena jménem, tak jméno může být použito pro definici aktuálních objektů struktury. Např. máme- li dánu deklaraci struktury date, potom

```
struct date d;
```

definuje proměnnou *d*, která je strukturou typu date. Struktura typu static nebo external může být inicializována.

```
struct date d = {4, 7, 1776, 186, "Jul"};
```

Člen struktury může být dosažen takto:

```
označení_struktury.jméno
```

Operátor členu struktury „.“ spojuje jméno struktury s jejím členem. Např. nastavení proměnné *leap* z data struktury může být provedeno takto:

```
leap = d.year % 4 == 0 && d.year % 100 != 0 || d.year %
400 == 0;
```

Kontrola názvu měsíce

```
if (strcmp(d.mon_name, "Aug") == 0) ...
```

nebo konvertování prvního znaku měsíce na malé písmeno

```
d.mon_name[0] = lower (d.mon_name[0]);
```

Struktury mohou být vkládány do sebe. Data zaměstnance mohou vypadat takto

```
struct person
{
   char name[NAMESIZE];
   char address[ADRSIZE];
   long zipcode;
   long ss number;
   double salary;
   struct date birthdate;
   struct date hiredate;
};
```

Struktura person obsahuje dvě data. Jestliže budeme deklarovat *emp* jako

```
struct person emp;
```

potom

```
emp.birthdate.month
```

udává měsíc narození. Operátor „.“ pracuje zleva doprava.

6.2 Struktury a funkce

V jazyku C existují omezení pro struktury. Základním pravidlem je to, že jediné operace, které můžeme se strukturami provádět, je zjištění adresy operátorem & a přístup k jejím členům. Z toho plyne, že struktury nemohou být přiřazovány nebo kopírovány jako obyčejné proměnné a nemohou být předávány jako parametry. (Tato omezení budou v další verzi jazyka C odstraněna.) Pointry na struktury nemají tato omezení, takže struktury a funkce mohou dobře spolupracovat. Automatické struktury stejně jako automatická pole nemohou být inicializovány.

Některé z těchto závěrů si ověříme při přepsání funkce pro konverzi data z předchozí kapitoly. Protože není možné předávat struktury funkcím přímo, musíme buď předávat jednotlivé prvky nebo pointry na struktury. První varianta používá funkcí day_of_year z kapitoly 5:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

Druhý způsob je předání pointru. Jestliže jsme deklarovali *hiredate* jako

```
struct date hiredate;
```

a přepsali funkci day_of_year, můžeme napsat

```
hiredate.yearday = day_of_year (&hiredate);
```

Funkce musí být modifikována, protože jejím argumentem je nyní pointr a ne seznam proměnných

```
      day_of_year(pd)      /* výpočet dne roku z měsíce a dne
*/
      struct date *pd;
      {
            int i, day, leap;
            day = pd -> day;
            leap = pd -> year % 4 == 0 && pd -> year % 100 != 0
                      || pd -> year % 400 == 0;
            for (i=1; i < pd -> month; i++)
                  day += day_tab[leap][i];
            return (day);
      }
```

Deklarace

```
struct date *pd
```

určuje, že *pd* je pointr na strukturu typu date. Notace

```
pd -> year
```

je novinkou. Jestliže je *p* pointr struktury potom

```
p -> clen_struktury
```

ukazuje na určitý člen. Operátor sestává ze znaménka minus a >.

Protože je *pd* pointr na strukturu, tak člen *year* může být dosažen také takto

```
(*pd).year
```

Pointry na struktury jsou často používány a tak operátor $->$ je výhodnou zkratkou. Ve výrazu *(*pd).year* jsou závorky nezbytné, protože operátor „.“ má vyšší prioritu než *. „$->$“ a „.“ pracují zleva doprava

```
p -> q -> memb
```

je *emp.birthdate.month*

```
(p -> q) -> memb
```

je *(emp.birthdate).month*

```
month_day (pd)       /*výpočet měsíce a dne ze dne roku*/
struct date *pd;
{
      int i, leap;
      leap = pd -> year % 4 == 0 && pd -> year % 100 != 0
                || pd -> year % 400 == 0;
      pd -> day = pd -> yearday;
      for (i = 1; pd -> day_tab[leap][i]; i++)
            pd -> day -= day_tab[leap][i];
      pd -> month = i;
}
```

Operátor $->$ a ., spolu s ( ) pro seznam argumentů a [ ] pro indexy mají ze všech operátorů nejvyšší prioritu. Např. máme-li dánu deklaraci

```
struct
{
      int x;
      int *y;
}  *p;
```

potom

```
++p -> x
```

zvětšuje x a ne p, protože příkaz je vykonán takto: $++(p->x)$. Pro změnu priority mohou být použity závorky: $(++p)->x$ zvětšuje p před dosazením x a $(p++)->x$ zvětšuje p potom. (Tyto poslední závorky jsou zbytečné. Proč?)

Stejným způsobem $*p->y$ vybírá to, na co ukazuje y: $*p->y++$ zvětšuje y po dosazení objektu (právě tak jako $*s++$). $(*p->y)++$ zvětšuje to, na co ukazuje y. $*p++->y$ zvětšuje p po dosazení toho, na co ukazuje y.

6.3 Pole struktur

Struktury jsou výhodné hlavně pro operace s poli proměnných. Např. uvažujme program, který zjišťuje výskyt klíčových slov jazyka C. Potřebujeme pole znakových řetězců, které bude obsahovat jména a pole čísel `int`, kde bude ukládán počet. Jednou z možností je použití dvou paralelních polí – *keyword* a *keycount*:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Právě fakt, že jsou pole paralelní indikuje to, že by byla možná jiná organizace. Každý prvek je vlastně pár:

```
char *keyword;
int keycount;
```

Deklarace struktury

```
struct key
{
      char *keyword;
      int keycount;
}
keytab[NKEYS];
```

Pole *keytab* alokuje paměť. Každý prvek pole je strukturou. To může být napsáno takto

```
struct key
{
      char *keyword;
      int keycount;
};
struct key keytab[NKEYS];
```

Protože struktura keytab obsahuje ve skutečnosti konstantní soubor jmen, je usnadněna inicializace. Inicializace struktury je podobná předchozím – definici následuje seznam vložený do závorek:

```
struct key
{
      char *keyword;
      int keycount;
}
keytab[] =
{
      "break", 0,
      "case", 0,
      "char", 0,
      "continue", 0,
      "default", 0,
      /* ... */
      "unsigned", 0,
      "while", 0
};
```

Inicializace je provedena páry odpovídajícími struktuře členu. Přesnější by bylo vkládat patřičné páry do závorek

```
        ("break", 0)
        ("case", 0)
        ...
```

ale není to nutné, protože v našem případě jsou to jednoduché proměnné a znakové řetězce. Jako obvykle překladač zjistí počet členů z inicializace a závorky [ ] proto mohou zůstat prázdné.

Program pro zjišťování počtu klíčových slov začíná deklarací *keytab*. Hlavní program čte vstup opakovaným voláním funkce getword, která načítá ze vstupu vždy jedno slovo. Pro každé slovo je prohledána tabulka *keytab* pomocí funkce pro binární hledání, kterou jsme uvedli v kapitole 3. (Seznam klíčových slov může být uspořádán vzestupně.)

```
#define MAXWORD 20
main()                  /* počítej klíčová slova */
{
      int n, t;
      char word [MAXWORD];
      while ((t = getword(word, MAXWORD)) != EOF)
         if(t == LETTER)
            if((n = binary(word, keytab, NKEYS)) >= 0)
                keytab[n].keycount++;
      for (n = 0; n < NKEYS; n++)
          if (keytab[n].keycount > 0)
               printf ("%4d %s\n",
                keytab[n].keycount,
                keytab[n].keyword);
}

binary (word, tab, n) /*najdi slovo v tab[0]...tab[n-1]*/
char *word;
struct key tab[];
int n;
{
      int low, high, mid, cond;
      low = 0;
      high = n - 1;
          while(low <= high)
      {
              mid = (low + high) / 2;
            if((cond = strcmp(word, tab[mid].keyword)) <
0)
            high = mid - 1;
          else if(cond > 0)
               low = mid + 1;
          else
             return(mid);
      }
      return(-1);
}
```

Na chvíli se zastavíme u funkce getword. Pro začátek stačí říci, že vrací *LETTER* pokaždé, když nalezne slovo. Toto slovo kopíruje do prvního argumentu.

NKEYS udává počet klíčových slov v tabulce *keytab*. Přestože bychom ho mohli snadno zjistit sami, je lepší to nechat na počítači, zvláště bude-li se program ještě měnit. Rozsah pole je už znám v době překladu. Počet prvků je

```
rozmer keytab / rozmer struct key
```

Existuje unární operátor sizeof, který může být použit pro určení délky objektu. Výraz

```
sizeof (objekt)
```

je celé číslo, které je rovno velikosti specifikovaného objektu (délka je udána v „bytech“, které mají stejnou velikost jako char).

Objekt může být obyčejná proměnná, pole, struktura, jméno typu int nebo double, nebo jméno odvozené od struct jako v našem případě. Tohoto výpočtu je použito při výpočtu *NKEYS*:

```
#define NKEYS (sizeof(keytab) / sizeof(struct key))
```

Nyní k funkci getword. Již jsme napsali obecnější funkci getword, než je v našem příkladě potřeba. getword vrací „slovo“ ze vstupu. Jméno je buď řetězcem znaků a čísel začínající písmenem, nebo jeden znak.

Typ objektu je určen hodnotou, kterou funkce vrací: *LETTER* bylo-li načteno slovo, *EOF* pro konec souboru nebo načtený nealfanumerický znak.

Funkce getword používá funkce getch a ungetch, které jsme napsali v kapitole 4.

Funkce getword volá type pro určení typu znaku ze vstupu. Následující verze je pouze pro ASCII znaky.

```
type (c)
int c;
{
      if(c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
            return(LETTER);
      else if(c >= '0' && c <= '9')
            return(DIGIT);
      else
            return(c);
}
```

Symbolické konstanty *LETTER* a *DIGIT* mohou mít libovolnou hodnotu, která není v rozporu s nealfanumerickými znaky a *EOF*. Obvykle se volí

```
#define LETTER  'a'
#define DIGIT   '0'
```

Funkce getword by mohla být rychlejší, kdyby bylo volání funkce type nahrazeno odkazem na odpovídající pole *type*[ ]. Standardní knihovna jazyka C obsahuje makro nazvané isalpha a isdigit, které funguje tímto způsobem.

Cvičení 6-1. Proveďte tyto změny v getword a změřte rozdíl v rychlosti.

Cvičení 6-2. Napište funkci type, která je nezávislá na souboru znaků.

Cvičení 6-3. Napište verzi programu pro počítání klíčových slov, který nepočítá výskyt v řetězcích v uvozovkách.

6.4 Pointry na struktury

Abychom ilustrovali některé vlastnosti pointrů a polí struktur, napišme znovu program pro zjištění počtu výskytu jednotlivých klíčových slov. Použijme ale pointry a ne indexy pole.

Deklarace extern pole *keytab* může zůstat nezměněna. Musíme změnit main a binary.

```
    main()  /*počítej klíčová slova, verze s pointry*/
    {
        int t;
        char word[MAXWORD];
        struct key *binary(), *p;
        while ((t = getword(word, MAXWORD)) != EOF)
            if (t == LETTER)
                if ((p = binary(word, keytab, NKEYS)) != NULL)
                    p -> keycount++;
        for (p = keytab; p < keytab + NKEYS; p++)
            if (p -> keycount > 0)
                 printf("%4d %s\n", p -> keycount, p ->
keyword);
    }


    struct key *binary(word, tab, n)  /*najdi slovo*/
    char *word;,
    struct key tab[];
    int n;
    {
          int cond;
          struct key *low = &tab[0];
          struct key *high = &tab[n-1];
          struct key *mid;
          while (low <= high)
          {
                mid = low + (high - low) / 2;
                if ((cond = strcmp(word, mid -> keyword))<0)
                    high = mid - 1;
                else if(cond > 0)
                        low = mid + 1;
                     else
                        return(mid);
          }
          return(NULL);
    }
```

Za zmínku zde stojí více věcí. Za prvé deklarace funkce binary musí indikovat, že funkce vrací pointr na strukturu typu key. To musí být deklarováno jak v jednotce main, tak ve funkci binary. Jestliže funkce binary slovo najde, tak vrací pointr na něj. Když slovo nenajde vrací *NULL*.

Za druhé – přístup k prvkům pole *keytab* je přes pointry. Proto musíme změnit funkci binary. Prostřední prvek už nemůže být jednoduše zjišťován výrazem

```
mid = (low + high) / 2
```

protože součet pointrů produkuje nesmyslný výsledek (dokonce i když je vydělen 2). To musí být změneno na

```
mid = low + (high - low) / 2
```

což nastavuje *mid* na prvek na polovině cesty mezi *low* a *high*.

Všimněte si rovněž inicializace *low* a *high*. Je totiž možné inicializovat pointr na adresu dříve definovaného objektu.

V main jsme napsali

```
for (p = keytab; p > keytab + NKEYS; p++)
```

p je pointr na strukturu a tak každá operace s *p* bere v potaz rovněž struktury. *p++* zvětšuje *p* odpovídajícím způsobem na další pole struktur. Nepředpokládejte ale, že rozměr struktury je dán pouze součtem rozměru jejich členů.

Nakonec se podívejme na formát programu. Jestliže funkce vrací komplikovaný typ jako

```
struct key *binary(word, tab, n)
```

tak můžeme v deklaraci jméno funkce přehlédnout. Proto někdy používáme zápis ve tvaru

```
struct key *
binary(word, tab, n)
```

To je věcí vkusu programátora. Vyberte si jeden způsob a držte se ho.

6.5 Struktury odkazující se samy na sebe

Předpokládejme, že chceme řešit obecnější problém: počítat množství výskytu **všech** slov ze vstupu. Protože ale seznam slov není na začátku znám, nemůžeme použít binárního prohledávání. Ani lineárního prohledávání nemůžeme použít, protože by program spotřebovával mnoho času (požadovaný čas by rostl kvadraticky s množstvím načtených slov). Jak tedy musíme organizovat data?

Jedním z řešení je neustále načítaná slova třídit. To znamená ukládat právě načtené slovo tam, kam patří. To ale nemůžeme dělat v lineárním poli, protože to by rovněž trvalo dlouho. Místo toho použijeme datovou strukturu nazývanou **binární strom**.

Tento strom obsahuje vždy jeden uzel pro každé slovo. Každý uzel obsahuje:

– pointr na další slovo
– počet výskytů daného slova
– pointer na levý poduzel
– pointer na pravý poduzel.

Žádný uzel nemůže mít víc než dva poduzly; může mít jeden nebo nemusí mít žádný.

Uzly jsou uspořádány tak, že levý podstrom obsahuje slova, která jsou menší než slovo v daném uzlu, a pravý podstrom obsahuje slova větší. Abychom zjistili, zda je právě načtené slovo ve stromu, začneme u kořenu a porovnáme načtené slovo se slovem v uzlu. Jestliže slovo souhlasí, jsme hotovi. Jestliže je načtené slovo menší než slovo v uzlu, pokračujeme do levého poduzlu; je-li větší, tak pokračujeme vpravo. Jestliže ale již v daném směru není žádný uzel, znamená to, že načtené slovo není ve stromu obsazeno a místo pro ně je právě chybějící uzel. Proces vyhledávání je rekurzivní, protože se při prohledávání z jednoho uzlu využívá prohledávání z jednoho poduzlu. Obdobně proces ukládání a tisku je rekurzivní.

Vraťme se zpět k popisu uzlu. Je to struktura se čtyřmi položkami:

```
struct tnode                    /*základní uzel*/
{
      char *word;               /*pointr na text*/
      int count;                /*počet výskytů*/
      struct tnode *left;       /*levý poduzel*/
      struct tnode *right;      /*pravý poduzel*/
};
```

„Rekurzivní“ definice uzlu může vypadat zmateně, ale je úplně správná. Je zakázáno, aby struktura obsahovala sebe samu jako položku, ale

```
struct tnode *left;
```

deklaruje text jako **pointr** na uzel a neobsahuje uzel.

Text programu je velice krátký a využívá funkce, které jsme napsali dříve. Je to getword pro načtená slova ze vstupu a alloc, která zajišťuje místo v paměti pro jednotlivá slova.

Hlavní program jednoduše čte slova a ukládá je do stromu tree

```
#define MAXWORD 20
main()              /*počítání frekvence slov*/
{
      struct tnode *root, *tree();
      char word[MAXWORD];
      int t;
      root = NULL;
      while ((t = getword(word, MAXWORD)) != EOF)
            if (t == LETTER)
                  root = tree(root, word);
      treeprint(root);
}
```

Funkce tree je jednoduchá. Slovo *word* je uloženo na vrcholek stromu (*root*). V každé fázi je porovnáno se slovem uloženým v uzlu. Potom se pokračuje do levého nebo pravého poduzlu rekurzivním voláním funkce tree. Slovo je buď ve stromu nalezeno (a je přičtena jednička k množství výskytu), nebo je výsledkem nulový pointr, který indikuje, že uzel musí být ve stromu vytvořen. Jestliže je vytvořen nový uzel, tak tree vrací pointr na něj a tento pointr je zařazen do vyššího uzlu.

```
struct tnode *tree(p,w)    /*uložení do p nebo níže*/
struct tnode *p;
```

```
char *w;
{
      struct tnode *talloc();
      char *strsave();
      int cond;
      if (p == NULL)            /*nové slovo*/
      {
          p = talloc();    /*vytvoření nového uzlu*/
          p -> word = strsave(w);
          p -> count = 1;
          p -> left = p -> right = NULL;
      }
      else if ((cond = strcmp(w, p -> word)) == 0)
             p -> count++;    /*opakované slovo*/
           else if (cond < 0)     /*je menší*/
                      p -> left = tree (p -> left, w);
               else                /*je větší*/
                      p -> right = tree (p -> right, w);
      return(p);
}
```

Paměť pro nový uzel je přidělována funkcí talloc, která je obdobou funkce alloc, kterou jsme napsali dříve. Vrací pointr na volné místo pro uzel stromu. (Za chvíli se k tomu vrátíme.) Nové slovo je kopírováno kamsi v paměti funkcí strsave, je inicializován počet výskytu a poduzly jsou vynulovány. Tato část programu je vykonávána pouze tehdy, vytváří-li se nový uzel. Vynechali jsme testování chyby po návratu z funkcí strsave a talloc, což není příliš moudré.

treeprint tiskne strom počínajíc levým podstromem. V každém uzlu tiskne levý podstrom (což jsou všechna slova menší než slovo v uzlu), potom slovo samo, a nakonec pravý podstrom (všechna větší slova). Pokuste se sami si vypsat strom užitím funkce treeprint; je to jedna z nejjasnějších rekurzivních funkcí, na kterou jste kdy narazili.

```
treeprint(p)            /*tiskni p rekurzívně*/
{
if (p != NULL)
  {
  treeprint(p -> left);
  print("%4d % s\n", p -> count, p -> word);
  treeprint(p -> right);
  }
}
```

Praktická poznámka: jestliže strom není vyvážen, protože slova nepřicházejí náhodně, čas výpočtu roste příliš rychle. Nejhorším případem je, přicházejí-li slova již uspořádaná. Program potom vlastně provádí lineární prohledávání. Existuje zobecnění lineárních stromů: stromy typu *2-3* a *AVL*, které jsou výhodnější ale my se o nich zde nebudeme zmiňovat.

Předtím, než opustíme tento příklad, vraťme se ještě k problému přidělování paměti. Ideálně by měl v programu existovat jenom jeden alokační podprogram a měl by přidělovat paměť různým objektům. Ale když má jeden alokační podprogram přidělit paměť pro pointr na char a pointr na tnode struktury, tak vyvstávají dvě otázky. Za prvé, jak je splněna podmínka mnoha počítačů pro zarovnání adresy (např. int musí být umístěno na sudé adrese). Za druhé, jak naložíme s tím, že podprogram vrací pokaždé jiný druh pointru?

Požadavky na zarovnání mohou být snadno splněny za předpokladu určitého plýtvání pamětí. Alokační podprogram bude vždy vracet pointr, který splňuje **všechny** požadavky na srovnání paměti. Např. na PDP-11 mohou být objekty uloženy na sudých adresách. Jedinou cenou za to je jeden zbytečný znak na liché adrese. Obdobně je tomu na jiných počítačích. Proto funkce alloc není přenosná, ale její použití je univerzální. Naše funkce z kapitoly 5 nesplňuje základní požadavky na zarovnávání. V kapitole 8 tuto funkci napíšeme správně.

Otázka způsobu deklarace funkce alloc je klíčová pro všechny jazyky, kde je prováděno kontrolování typu seriózně. V jazyce C je nejlepším způsobem deklarovat, že alloc vrací pointr na char a potom upravit pointr na požadovaný tvar. Jestliže je *p* deklarováno jako

```
char *p;
```

potom

```
(struct tnode*)p
```

jej konvertuje na pointr typu tnode. Proto je talloc napsána takto:

```
struct tnode *talloc()
{
      char *alloc();
      return((struct tnode*)alloc(sizeof(struct tnode)));
}
```

Je to více než požadují současné překladače, ale představuje to nejbezpečnější přístup k budoucím překladačům.

Cvičení 6-4. Napište program, který čte program v jazyku C a tiskne podle abecedy skupiny názvu proměnných, které jsou shodné v prvních sedmi znacích a v ostatních se liší.
Cvičení 6-5. Napište jednoduchý program pro křížové reference. Program má tisknout všechna slova ze vstupu a ke každému slovu uvést čísla řádek, ve kterých se vyskytuje.
Cvičení 6-6. Napište program, který tiskne rozdílná slova ze vstupu setříděná sestupně podle frekvence výskytu. Před každé slovo vytiskněte počet výskytů.

6.6 Prohledávání tabulky

Další vlastnosti struktur budeme ilustrovat souborem programů pro prohledávání tabulek. Tyto programy bývají součástí podprogramů pro rozvíjení maker a pro ukládání parametrů do tabulek. Např. uvažujme příkaz #define v jazyku C. Jestliže je na řádce text

```
#define YES 1
```

tak jméno YES i jeho hodnota 1 jsou uloženy v tabulce. Později, když se objeví jméno YES v příkazu

```
inword = YES
```

tak musí být nahrazeno jedničkou.

Pro operace s názvy a jejich náhradami existují dvě základní funkce. install(s, t) ukládá jméno s a nahrazující řetězec t do tabulky. s a t jsou znakové řetězce. Funkce lookup prohledává tabulku a vrací pointr na místo, kde byl nalezen text, nebo *NULL* nebyl-li text nalezen. V těchto funkcích je použito algoritmu klíče: jména jsou konvertována na malá kladná čísla, která jsou potom použita jako indexy do pole pointrů. Prvek tohoto pole ukazuje na začátek řetězce bloků popisujících jména, která mají tuto hodnotu. Vrací *NULL*, nejsou-li žádná taková jména.

Blok v řetězci je strukturou obsahující pointr na jména, nahrazující text a odkaz na další blok v řetězci. Nulový pointr označuje konec řetězce.

```
struct nlist      /*základní položka*/
{
      char *name;
      char *def;
      struct nlist *next;  /*další vstup v řetězci*/
};
```

Pole pointrů vypadá takto:

```
#define HASHSIZE 100
static struct nlist *hashtab[HASHSIZE];
                        /*tabulka pointrů*/
```

Transformační funkce, které je použito ve funkcích lookup a install sčítá hodnoty znaků v řetězci a vrátí zbytek po dělení délkou hashovací tabulky. (Není to algoritmus nejlepší, ale je jednoduchý).

```
hash(s)            /*transformace řetězce s*/
char *s;
{
      int hashval;
      for(hashval = 0; *s != '\0'; )
            hashval += *s++;
      return(hasval % HASHSIZE);
}
```

Tento proces produkuje začáteční index pole *hashtab*. Ať už byl řetězec nalezen kdekoliv, bude uložen v řetězci bloků začínajících zde. Funkce lookup realizuje prohledávání. Jestliže jméno již existuje, vrací pointr na něj. Jestliže ale neexistuje, vrací *NULL*.

```
struct nlist *lookup(s)      /*prohledávání*/
char *s;
{
struct nlist *np;
for(np = hashtab[hash(s)]; np == NULL; np = np -> next)
```

```
            if(strcmp(s, np -> name) == 0 )
                  return(np);      /*nalezeno*/
      return(NULL);      /*nenalezeno*/
      }
```

Funkce install používá lookup pro zjistění, zda je jméno již uloženo. Jestliže je uloženo, tak nová definice musí nahradit starou. Jinak je uloženo nové jméno. install vrací nulu, když není místo pro nový prvek.

```
      struct nlist *install(name,def)        /*ulož(name,
def)*/
      char *name, *def;                     /*do nashtab*/
      {
            struct nlist *np, *lookup();
            char *strsave(), *alloc();
            int hashval;
      if((np = lookup(name)) == NULL)      /*nenalezeno*/
      {
            np=(struct nlist*) alloc (sizeof(*np));
            if (np == NULL)
                  return (NULL);
            if ((np -> name = strsave (name)) == NULL
                  return (NULL);
            hashval= hash (np -> name);
            np -> next = hashtab[hashval];
            hasthab[hashval] = np;
      }
      else                                  /*existuje*/
            free (np -> def);               /*vymazání*/
      if ((np -> def = strsave(def)) == NULL)
            return(NULL);
      return(np);
      }
```

strsave kopíruje řetězec, který je jeho argumentem na bezpečné místo. Místo je přiděleno funkcí alloc. alloc jsme ukázali v kapitole 5. Protože se volání alloc a free může objevit v libovolném pořadí a protože je potřeba dodržovat pravidla zarovnání adresy, tak tato jednoduchá verze nedostačuje. Viz kapitolu 7 a 8.

Cvičení 6-7. Napište program, který vyjme jméno a nahrazující řetězec tabulky, obhospodařované funkcemi lookup a install.
Cvičení 6-8. Implementuje jednoduchou verzi programu, který zpracovává příkazy #define v programech v jazyce C. Použijte funkce z této kapitoly a také funkce getch a ungetch.

6.7 Pole bitů

Jestliže je otázka paměti na prvním místě, může být nezbytné sloučit různé objekty do jednoho počítačového slova. Hodně se používá souboru tří znakových bitů v aplikacích typu tabulky symbolů překladače.

Představte si fragment překladače, který zachází s tabulkou symbolů. Každý identifikátor s sebou nese jisté informace. Např. je-li klíčovým slovem, zda je externí nebo externí staticky nebo interní atd. Nejkompaktnějším způsobem zápisu těchto informací je jejich zakódování do proměnné typu int nebo char.

Obvyklým způsobem jak to udělat je definice masek, odpovídajících pozici bitu ve slově:

```
#define   KEYWORD 01
#define   EXTERNAL 02
#define   STATIC 04
```

(Čísla musí být mocninami dvou.) Přístup k bitům bude prováděn posouváním, maskováním a doplňkovým operátorem, popsaným v kapitole 2.

Výraz

```
flags |= EXTERNAL | STATIC;
```

Nastavuje bity *EXTERNAL* a *STATIC* ve *flags*, zatímco

```
flags &= (EXTERNAL | STATIC);
```

tyto bity nuluje.

Výraz

```
if ((flags & (EXTERNAL | STATIC)) == 0)
```

je pravdivý, jestliže je některý z obou bitů jednotkový.

Jako alternativu nabízí jazyk C možnost definování a přístup k polím bitů přímo. **Pole bitů** je soustava bitů v jedné proměnné int. Syntaxe definice pole bitů a přístup k němu je založen na struktuře. Např. tabulka symbolů #define může být nahrazena definicí tří polí bitů:

```
struct
{
      unsigned is_keyword : 1;
      unsigned is_extern : 1;
      unsigned is_static : 1;
}  flags;
```

Tím je nadefinována proměnná *flags*, která obsahuje 3 pole bitů. Pole bitů jsou definovány jako unsigned, protože to jsou skutečné veličiny bez znaménka.

Jednotlivá pole bitů jsou: *flags.is_keyword*, *flags.is_extern* atd., stejně jako členy struktur. Pole bitů se chovají jako malé proměnné bez znaménka a mohou vystupovat v aritmetických operacích jako jiné proměnné int. Předchozí příklady mohou být napsány rovněž takto:

```
flags.is_extern = flags.is_static = 1
```

nastavuje bity a

```
flags.is_extern = flags.is_static = 0
```

je nuluje a

```
if (flags.is_extern == 0 && flags.is_static == 0)
```

je testuje. Pole bitů by neměly překročit meze proměnné int: stane-li se to, tak je pole přiřazeno na další adresu. Pole bitů nemusí mít jméno. Nepojmenovaná pole (dvojtečka s udanou délkou) mohou být použita jako výplň.

Pole bitů jsou na některých počítačích přiřazována do proměnné zleva doprava, na některých zprava doleva – záleží na typu počítače.

Je nutne mít na mysli další omezení. Pole bitů nemají znaménko, mohou být přiřazovány do int nebo unsigned. Nejsou to pole jako taková. Nemají adresy a tak pro ně nemůže být používáno unární operace *&*.

6.8 Uniony

Union je proměnná, která může obsahovat v různých chvílích objekty různých typů a velikostí. Uniony umožňují pracovat s různými typy objektů v jedné oblasti paměti, aniž by využívaly nějaké strojově závislé operace.

Pro příklad se vraťme opět k tabulce symbolů překladače. Předpokládejme, že konstanty mohou být int, float nebo znakové pointry. Hodnota konstanty musí být uložena v proměnné odpovídajícího typu. Konstanty by měly zabírat stejné místo a nemělo by záležet na typu. Můžeme použít union, kde různé proměnné mohou sdílet stejné místo. Tak jako u pole bitů je syntaxe definice založena na struktuře:

```
union  u_tag
{
    inf ival;
    float fval;
    char *pval;
} uval;
```

Proměnná *uval* bude dostatečně velká, aby mohla obsahovat největší ze tří typů a nezáleží na hardwaru počítače. Libovolný z těchto typů může být proměnné *uval* přiřazen a potom použit ve výrazu. Typ musí odpovídat naposled uloženému typu. Záleží pouze na programátorovi, aby si pamatoval, co do proměnné *uval* uložit naposledy. Výsledek záleží na počítači pouze tehdy, uloží-li se do unionu něco jiného, než co je později využito.

Členy unionu jsou dosažitelné takto:

```
jmeno_unionu.clen
```

nebo

```
pointr_unionu -> clen
```

tak, jako ve strukturách. Jestliže do proměnné *utype* uložíme typ, který byl přiřazen unionu *uval*, pak můžeme psát

```
if (utype == INT)
      printf ("%d\n", uval.ival);
else if (utype == FLOAT)
      printf ("%f\n", uval.fval);
```

```
else if (utype == STRING)
      printf ("%s\b", uval.pval);
else
      printf ("bad type %d in utype\n", utype);
```

Uniony se mohou objevit ve strukturách a polích a naopak. Zápis pro dosažení členu unionu ve struktuře (a naopak) je identický s vnořenými strukturami. Např. v poli struktury nadefinovaném takto:

```
struct
{
    char *name;
    int flags;
    int utype;
    union
    {
        int ival;
        float fval;
        char *pval;
    }  uval;
} symtab[NSYM];
```

je proměnná *ival* určena takto

```
symtab[i].uval.ival
```

a první znak řetězce *pval* takto

```
*symtab[i].uval.pval
```

Ve skutečnosti je union struktura, ve které mají všechny členy nulový relativní posun adresy, struktura je dostatečně velká, aby mohla „pojmout nejširší“ člen a zarovnáním adresy platí pro všechny členy unionu. Jediná povolená operace s uniony je dosazení jejího členu a určení jeho adresy. Uniony nemohou být parametry funkcí, nemohou stát na levé straně přiřazovacího příkazu a nemohou být vráceny funkcemi. Pointry na uniony mohou být užívány stejným způsobem jako pointry na struktury. V kapitole 8 ukážeme, jak mohou být uniony používány.

6.9 Příkaz typedef

C umožňuje definovat nová datová jména např.

```
typedef int LENGTH;
```

LENGTH je synonymum pro int. „Typ“ LENGTH může být použit v deklaracích stejným způsobem jako int:

```
LENGTH len, maxlen;
LENGTH lenghts[];
```

obdobná deklarace

```
typedef char *STRING;
```

definuje STRING jako synonymum pro char nebo znakový pointr, který potom může být použit v deklaracích STRING *p*, *lineptr* [*LINES*], alloc();

Uvědomte si, že typ, který je deklarován v příkazu typedef se objevuje na pozici jméno proměnné a ne přímo za slovem typedef. Syntakticky je typedef na stejné úrovni jako extern, static atd. Použili jsme rovněž velkých písmen, abychom zdůraznili jméno.

Jako složitější příklad uvedeme definici uzlu stromu s použitím typedef:

```
typedef struct tnode           /*základní uzel*/
{
    char *word;      /*ukazatel na další jméno*/
    int count;                 /*počet výskytů*/
    struct tnode *left          /*levý poduzel*/
    struct tnode *right;       /*pravý poduzel*/
} TREENODE, *TREEPTR;
```

Tato definice vytváří dvě nová slova: TREENODE (struktura) a TREEPTR (pointr na strukturu). Potom funkce talloc může být napsána takto:

```
TREEPTR talloc();
      char *alloc();
      return((TREEPTR) alloc (sizeof (TREENODE)));
```

Musíme zdůraznit, že příkaz typedef **nevytváří** nové datové typy. Přidává pouze další jména existujícím typům. Není ani použito nové sémantiky: proměnné deklarované tímto způsobem mají přesně stejné vlastnosti, jako kdyby byly definovány normálně. Ve skutečnosti je typedef obdobou příkazu #define, ale je interpretován překladačem.

```
typedef int (*PFI)();
```

Vytváří typ PFI pro „pointr na funkci, která vrací int“, kterého může být použito v

```
PFI strcmp, numcmp, swap;
```

v programu sort z kapitoly 5.

Jsou dva způsoby pro používání deklarace typedef. První důvod je, že umožňuje jednoduše řešit problémy přesnosti programu. Je-li některý typ závislý na použitém počítači, tak stačí změnit pouze jeden příkaz typedef. Je také možné používat jmen definovaných v typedef pro různé typy int a pak konkrétně dosadit short, int, long dle potřeby použitého počítače. Druhým důvodem je zajištění lepší dokumentace programu: typu nazvanému TREEPTR je lépe rozumět, než je-li definován pouze jako pointr na strukturu.

Je také možné, že v budoucnu bude překladač nebo program lint užívat informace obsažené v typedef pro kontrolu správnosti programu.

Kapitola 7

Vstup a výstup

Vstup a výstup není částí jazyka C a tak jsme na ně nekladli důraz. Nicméně opravdové programy spolupracují se svým okolím složitějším způsobem, než jak jsme doposud ukázali. V této kapitole popíšeme standardní „knihovnu vstupu a výstupu“ – soubor funkcí, které byly navrženy pro standardní I/O systém jazyka C. Funkce byly navrženy tak, aby umožňovaly pohodlné programování. Jsou dostatečně efektivní, takže uživatel téměř necítí potřebu je „zrychlovat a zefektivňovat“. A nakonec budiž řečeno, že funkce jsou „přenositelné“, což znamená, že existují kompatibilně na počítačích, kde je implementován jazyk C. Mohou být převáděny z jednoho systému do druhého bez podstatných změn. Zde se nebudeme snažit o popis celé knihovny.

Větší důraz klademe na to, abychom ukázali základy psaní programů v jazyku C, než abychom se soustředili na to, jak programy spolupracují se svým okolím.

7.1 Přístup do standardní knihovny

Každý zdrojový soubor, který potřebuje funkce ze standardní knihovny musí obsahovat řádek

```
#include <stdio.h>
```

na začátku souboru. V souboru stdio.h jsou definována jistá makra a proměnné, které jsou standardní I/O knihovnou využívány. Použitím znamének $<$ a $>$ místo uvozovek směrujeme překladač do adresáře, který obsahuje standardní záznamy se záhlavím (na systému UNIX je to /usr/include).

Někdy může být nutné definovat knihovnu při zavádění programu explicitně. Např. na PDP 11, UNIX bude příkaz pro překlad programu vypadat takto:

```
CC zdrojovy soubor,...            -&S
```

kde -&S znamená načítání ze standardní knihovny.

7.2 Standardní vstup a výstup – getchar a putchar

Nejjednodušším mechanismem načítání je čtení znaků ze „standardního vstupu“, obecně uživatelova terminálu funkcí getchar (). Funkce getchar () vrací další znak ze vstupu pokaždé, když je vyvolána. Na mnoha systémech, kde je implementován jazyk C, může být vstup z terminálu nahrazen vstupem ze souboru operátorem $<$. Jestliže program prog používá funkce getchar, tak příkazová řádka

```
prog < infile
```

způsobí, že prog bude číst vstup ze souboru infile místo z terminálu. Toto přepínání vstupu je děláno takovým způsobem, že program prog o něm „neví“. Řetězec '< infile >' není rovněž zařazen do argumentu *argv*. Přepínání vstupu si rovněž můžeme všimnout při situaci, když výstup jednoho programu je vstupem programu dalšího (pipe line). Příkaz

```
program1 | program2
```

spustí dva programy: program1 a program2 a výstup z programu1 bude vstupem programu2.

Funkce getchar vrací *EOF*, když narazí na konec vstupu, ať je vstupem cokoliv (terminál nebo soubor). Ve standardní knihovně je definován *EOF* jako -1 (v souboru stdio.h). Testování má být prováděno s *EOF* a ne s -1, aby bylo nezávislé na konkrétní hodnotě. Při výstupu funkce putchar (c) „zapíše“ znak c do „standardního výstupu“, což je standardně rovněž terminál. Výstup může být přesměrován znakem $>$ do souboru. Jestliže program prog používá putchar, tak příkazem

```
prog > outfile
```

bude prog svůj výstup psát do souboru outfile. V systému UNIX může být rovněž použit znak | pro řetězení programu (pipe line)

```
program1 | program2
```

Ani zde se programy „nestarají“ o změnu vstupu a výstupu. Výstupy z funkce printf a putchar mohou být vzájemně zaměňovány.

Překvapivě velké množství programů čte pouze jeden vstup a zapisuje na jeden výstup. Pro tyto programy jsou funkce getchar, putchar a printf zcela postačující. Máme navíc možnost přesměrování vstupu a výstupu a možnost řetězení programu. Uvažujeme např. program lower, který mění velká písmena ze vstupu na malá

```
#include <stdio.h>
main()   /* konvertování velkých písmen na malá */
{
   int c;
   while ((c=getchar()) != EOF)
      putchar (isupper(c) ? tolower(c) : c);
}
```

„Funkce“ isupper a tolower jsou ve skutečnosti makra definovaná ve stdio.h. Makro issupper testuje, zda jeho argument je velké písmeno. Když je argument velké písmeno, tak vrací nenulovou hodnotu, v opačném případě vrací nulu. Makro tolower konvertuje velká písmena na malá. Nehledě na to, jak jsou tyto funkce implementovány na různých počítačích, tak efekt je vždy a všude stejný a program se nemusí starat o soubor znaků na daném počítači.

Pro konvertování více souborů můžeme použít program cat

```
cat soubor1 soubor2...  | lower > output
```

a nemusíme se učit, jak se má k souborům přistupovat. (Program cat bude popsán dále v této kapitole).

Standardní „funkce“ vstupu a výstupu putchar a getchar mohou být ve skutečnosti makra a tak se nemusíme starat o konvenci volání funkcí. Jak je to uděláno ukážeme v kapitole 8.

7.3 Formátový výstup – printf

Dvě rutiny printf pro výstup a scanf pro vstup (bude popsáno v příštím odstavci) umožňují translaci čísel na znakovou reprezentaci. Umožňují rovněž zápis a čtení formátovaných řádek.

Funkce printf jsme užívali v předchozích kapitolách, aniž bychom znali její přesný popis. Zde uvedeme kompletnější, přesnější definici

```
printf (control, arg1, arg2, ...)
```

Funkce prinf provádí konverze čísel a formátový výstup do standardního souboru. Formát je udáván řídícím řetězcem control. Tento řetězec obsahuje dva typy objektů: obyčejné znaky, které jsou normálně kopírovány na výstup a konverzní specifikace, z nichž každá platí pro patřičný argument funkce printf. Každá konverze je uvedena znakem % a je ukončena znakem pro typ konverze. Mezi znakem % a znakem typu konverze může být:

- Znaménko −, které způsobuje zarovnání argumentů na pozici doleva.
- Číslo udávající minimální délku pole pro zobrazovaný řetězec. Konvertované číslo bude vytištěno do pole minimálně této délky a popř. bude použito pole širšího, bude-li to nezbytné. Jestliže má konvertované číslo méně znaků než specifukuje délka pole, tak výstup bude „vyplněn“ nalevo (nebo napravo bylo-li specifikováno znaménko −). Vyplňovací znaky jsou normálně mezery. Byla-li délka pole specifikována s úvodními nulami, budou vyplňovací znaky nulami.
- Tečka, která odděluje délku pole od dalšího čísla.
- Číselný řetězec (přesnost) který udává maximální počet znaků, které mají být tištěny napravo od desetinné tečky pro čísla float nebo double.
- Modifikátor délky l (písmeno l), které indikuje to, že argumentem je long.

Typ konverze může být následující:

d – argument je konvertován na desítkový zápis

o – argument je konvertován do osmičkové soustavy bez znaménka (bez úvodních nul)

x – argument je konvertován na hexadecimální tvar bez znaménka (bez úvodních nul)

u – argument je konvertován na desítkový zápis bez znaménka

c – argument je dán jako jeden znak

s – argument je řetězec. Řetězec je tištěn, až do znaku \0 nebo do délky, která byla specifikována

e – argument je brán jako float nebo double a konvertován na desítkový zápis ve formě [−]m.nnnnnne[+]xx, kde délkou řetězce je specifikována přesnost (implicitně 6)

f – argument je chápán jako float nebo double a konvertován na desítkový zápis ve tvaru [−]mmm.nnnnnn, kde délka řetězce n je specifikována přesností (implicitně 6), přesnost nijak neovlivňuje tisk významových číslic

g – používá %e nebo %f podle toho, co je kratší, nevýznamné nuly nejsou tištěny

Jestliže znak za znaménkem % není znakem konverze, tak je normálně vytištěn. % může být vytisknuto takto: %%.

Většina formátů již byla použita a vysvětlena v předchozích kapitolách. Jedinou výjimkou je zadání přesnosti a její vliv na tištěný řetězec. V následující tabulce jsou probrány různé varianty specifikace při tisku řetězce „hello, world", který má 12 znaků. Řetězec jsme v příkladech ohraničili znaky:

```
:%10s:        :hello, world:
:%-10s:       :hello, world:
:%20s:        :        hello, world:
:%-20s:       :hello, world        :
:%20.10s:     :          hello, wor:
:%-20.10s:    :hello, wor          :
:%.10s:       :hello, wor:
```

Varování: první argument funce printf udává, kolik argumentů následuje a jakého jsou typu. Jestliže je argumentů méně, nebo jsou nesprávného typu, tak výstup bude nesmyslný.

Cvičení 7.1. Napište program, který bude tisknout údaje ze vstupu rozumným způsobem. Minimálně by měl nepísmenové znaky tisknout v osmičkové nebo šestnáctkové soustavě a rozdělovat dlouhé řádky.

7.4 Formátovy vstup – scanf

Funkce scanf je funkcí analogickou k funkci printf. Provádí obdobné konverze opačným směrem.

```
scanf(control, arg1, arg2, ...)
```

scanf čte znaky standardního vstupu a interpretuje je podle formátových specifikací, uvedených v řetězci *control* a ukládá do argumentu. Řetězec *control* bude dále popsán. Ostatní argumenty, z **nichž každý musí být pointr** určují, na které místo bude ukládán konvertovaný vstup. Řetězec *control* obvykle obsahuje znaky konverzních specifikací, které jsou používány k přímé konverzi. Může obsahovat:

- Mezery, tabelátory nebo znaky pro novou řádku, „oddělovače", které jsou ignorovány
- Obyčejné znaky (vyjma %), o kterých se předpokládá, že budou souhlasit se znaky ze vstupu.
- Konverzní specifikace, sestávající ze znaku %, nepovinného znaku * pro potlačení přiřazení, nepovinného počtu udávajícího maximální délku pole a znaku typu konverze.

Konverzní specifikace je aplikována na další výstupní pole. Normálně je výsledek uložen do proměnné, na kterou ukazuje následující argument. Jestliže se ve specifikaci objeví znak *, je vstup „přeskočen“. Vstupní pole je definováno jako řetězec, který neobsahuje oddělovače. Jeho délka je omezena dalším oddělovačem nebo specifikací délky. scanf tedy nebude při načítání znaků omezována jednotlivými řádky, protože znaky pro novou řádku jsou pro tuto funkci normálními oddělovači.

Znak typu konverze určuje interpretaci vstupního pole. Odpovídající argument musí byt pointr. Jsou povoleny následující konverze:

d – očekává se číslo v desítkovém zápisu. Odpovídající argument by měl být pointr na celé číslo int

o – je očekáváno číslo v osmičkovém tvaru; odpovídající argument musí být pointr na int.

x – je očekáváno číslo v hexadecimálním tvaru. Odpovídající argument musí být pointr na int.

h – je očekáváno číslo typu short int. Odpovídající argument musí být pointr na short int.

c – je očekáván jeden znak, odpovídající argument má být pointr na char. V tomto případě je potlačeno přeskakování oddělovačů. K načtení dalšího znaku, který není oddělovačem, je třeba použít $\%1s$.

s – je očekáván znakový řetězec. Odpovídající argument musí být pointr na pole znaků dostatečně velké, aby se do něho vešel celý vstupní řetězec včetně ukončovacího znaku \0.

f – je očekáváno číslo s plovoucí desetinnou tečkou. Odpovídající argument by měl být pointr na float.

e – vstupní řetězec může obsahovat znaménko, desetinnou tečku a exponent, obsahující E nebo e a číslo integer se znaménkem. E má stejný význam jako f.

Před znaky konverze o, d, x může být uvedeno l (písmeno l), které určuje long int. Obdobně před znaky e a f může být l, které určuje double. Např. volání

```
int i;
float x;
char name[50];
scanf("%d%f%s", &i, &x, name);
```

Se vstupem ve tvaru

```
25   54.32E-1   Thompson
```

přiřadí proměnné i hodnotu 25, $x = 5.432$ a poli *name* řetězec Thompson \0. Tato 3 vstupní pole mohou být od sebe oddělena libovolně mnoha oddělovači.

Volání

```
int i;
float x;
char name[50];
scanf("%2d%f%*d%2s", &i, &x, name);
```

se vstupem ve tvaru

```
56789 0123 45 a72
```

přiřadí proměnné $i = 56$, $x = 789.0$, přeskočí 0123 a umístí řetězec „45“ do *name*. Další vyvolaná vstupní funkce začne načítání od písmene *a*. V těchto dvou příkladech je *name* pole a proto nemusí být označeno znakem &.

V dalším příkladu přepíšeme program pro kalkulátor z kapitoly 4 s použitím funkce scanf:

```
#include<stdio.h>
main()      /*jednoduchý stolní kalkulátor*/
{
  double sum,v;
  sum=0;
  while(scanf("%lf", &v) != EOF)
      printf("\t%.2f\n", sum += v);
}
```

Funkce scanf ukončí činnost, jestliže vyčerpá řídící řetězec, nebo narazí-li na vstup, který neodpovídá zadané specifikaci. Jako hodnotu vrací počet úspěšně načtených vstupních položek. Narazí-li na konec vstupu, vrací *EOF*. To je odlišné od znaku 0, který vrací neodpovídá-li vstup formátové specifikaci. Po dalším vyvolání pokračuje funkce následujícím znakem.

Závěrečné varování: argumenty **musí být pointry**! Obvyklá chyba je:

```
scanf("%d", n); místo scanf("%d", &n);
```

7.5 Formátové konverze u paměti

Funkce scanf a printf mají své obdoby, nazývané sscanf a sprintf, které provádějí obdobné konverze, ale operují s řetězci místo se soubory. Obecný tvar zápisu.

```
sprintf(string, control, arg1, arg2, ...)
sscanf(string, control, arg1, arg2, ...)
```

sprintf konvertuje argumenty stejným způsobem jako printf, ale výsledek píše do řetězce *string* místo do standardního výstupu. Ovšemže *string* musí být dostatečně velký. Jestliže je navíc znakovým polem a *n* je celé číslo, potom

```
sprintf(name, "temp%d", n);
```

vytváří řetězec ve tvaru *tempnnn* v poli *name*. *nnn* je hodnota proměnné *n*.

sscanf provádí opačnou konverzi – prohledává řetězec podle formátové specifikace v řetězci *control* a hodnoty ukládá do *arg1*, *arg2*, což opět musí být pointry.

Volání

```
sscanf(name, "temp%d", &n);
```

nastavuje *n* na hodnotu řetězce za názvem *temp*.

Cvičení 7.2. Přepište kalkulátor z kapitoly s použitím funkce scanf a sscanf.

7.6 Přístup k souborům

Programy, které jsme dosud napsali čtou ze standardního vstupu a zapisují na standardní výstup, které jsou programu „předdefinovány“ operačním systémem.

V dalším kroku napíšeme program, který pracuje se soubory, které k němu nejsou operačním systémem implicitně přiřazeny. Jedním takovým programem je program cat, který řetězí skupinu pojmenovaných souborů na standardní výstup. Program cat je používán pro tisk souborů na terminál a je univerzálním předzpracovacím programem pro ty programy, které nemají přístup k jinému než standardnímu vstupu. Např. příkaz

```
cat x.c y.c
```

Tiskne obsah souboru x.c a y.c do standardního výstupu.

Otázkou zůstává, jak zabezpečit to, aby soubory byly načteny – tzn. jak spojit externí jména souborů s aktuálním příkazem pro načítání dat.

Pravidla jsou jednoduchá. Předtím, než soubor bude čten nebo do něj bude psáno, musí být otevřen pomocí standardního programu fopen. fopen přijímá externí jména (jako x.c, y.c) provádí interakci s operačním systémem (detaily nás nemusí zajímat) a vrací interní jméno, které musí být použito v následujícím čtení nebo psaní.

Toto interní jméno je vlastně pointr (který se nazývá **pointr na soubor**) na strukturu, která obsahuje informace o souboru. Informace obsahuje údaje o umístění vyrovnávací paměti (bufferu), aktuální pozici v této paměti, zda je soubor čten nebo je do něho psáno atd.

Programátor nemusí znát podrobnosti, protože částí standardních definic obsazených ve stdio.h je definice struktury nazvaná FILE.

Jediná potřebná deklarace je

```
FILE *fopen(),*fp;
```

Definice říká, že *fp* je pointr na FILE a fopen vrací pointr na FILE. Uvědomme si, že FILE je název typu zrovna tak jako int a není to vlastně název struktury. (Podrobnosti, jak vše pracuje pod systémem UNIX jsou dány v kapitole 8)

Skutečné volání fopen vypadá takto

```
fp = fopen(name, mode);
```

Prvním argumentem funkce fopen je **název souboru** *name*, zadaný jako znakový řetězec. Druhý argument, *mode*, určuje způsob přístupu k souboru. Je to rovněž znakový řetězec. Volené způsoby přístupu jsou: čtení („r“), zápis („w“) a připisování na konec („a“).

Jestliže otevřeme soubor, který neexistuje, pro zápis nebo připisování na konec, tak je soubor vytvořen. Otevření existujícího souboru pro zápis způsobí jeho smazání. Pokus o čtení neexistujícího souboru způsobí chybu. Chyba při čtení může nastat také z dalších důvodů (např. čtení souboru, ke kterému není povolen přístup). Jestliže při čtení vznikla chyba, je hodnota pointru fopen rovna *NULL* (což je rovněž definováno v stdio.h)

Dále je nutno vědět, jak číst a psát do právě otevřených souborů. Je mnoho možností. Funkce getc a putc jsou nejjednodušší možností. Jako argument je zadán pointr na *file*.

```
c = getc(fp)
```

Vrací znak ze souboru, na nejž ukazuje pointr *fp*. Je-li dosaženo konce souboru, vrací EOF. putc je inverzí getc:

```
putc(c, fp);
```

Zapisuje znak do souboru *fp* a vrací *c*. Zrovna tak jako getchar a putchar, getc a putc mohou být makra.

Jestliže je program spuštěn, tak jsou automaticky otevřeny tři soubory a jsou dostupné pointry na tyto soubory. Těmito soubory jsou: standardní vstup, standardní výstup a standardní výstup pro chybové hlášení. Pointry na tyto soubory se jmenují *stdin*, *stdout* a *stderr*. Normálně jsou spojeny s terminálem, ale *stdin* a *stdout* mohou být přesměrovány na soubor, nebo použity pro řetězení programu (pipe line) viz odst. 7.2. getchar a putchar mohou být definovány pomocí getc a putc a *stdout* takto:

```
#define getchar()      getc(stdin)
#define putchar(c)     putc(c, stdout)
```

Pro formátový vstup a výstup mohou být použity funkce fscanf a fprintf. Tyto funkce jsou identické s funkcemi scanf a printf s tím, že prvním argumentem je pointr na soubor a řetězec *control* je až druhým parametrem.

Nyní můžeme přistoupit k napsání programu cat. Základní princip je použitelný pro mnoho dalších programů: jsou-li zadány argumenty v příkazové řádce, jsou brány popořádku. Jestliže argumenty zadány nejsou, je pracováno se standardním vstupem. Tento program může být použit samostatně, nebo může být částí většího celku

```
#include <stdio.h>
main(argc, argv)      /*cat: spojování souborů*/
int argc;
char *argv[];
{
   FILE *fp, *fopen();
   if(argc == 1)      /*žádné argumenty; kopíruj
                              standardní vstup */
      filecopy(stdin);
   else
      while(--argc > 0) {
         if((fp = fopen(*++argv, "r")) == NULL)
            {
               printf("cat: can't open %s\n", *argv);
               break;
            }
          else {
               filecopy(fp);
               fclose(fp);
              }
      }
}

filecopy(fp) /* zkopírování fp do standardního výstupu */
FILE *fp;
{
      int c;
      while((c=getc(fp)) != EOF)
```

```
            putc(c, stdout);
}
```

Pointry na soubory stdin a stdout jsou předdefinovány v knihovně vstupu a výstupu a mohou být používány všude tam, kde mohou být používány pointry FILE *. Jsou to **konstanty** a ne proměnné, takže se nepokoušejte jim něco přiřazovat.

Funkce fclose je inverzní funkce fopen. Přerušuje spojení mezi externím názvem souboru a pointrem. Pointr může být dále použit pro jiný soubor. Protože každý oprerační systém má omezení počtu současně otevřených souborů, je výhodné používat ty pointry, které už nejsou potřeba – jak jsme to udělali v programu cat. Pro použití funkce fclose je další důvod: uvolňuje vyrovnávací paměť, kterou používá putc (funkce fclose je automaticky vyvolána pro každý otevřený soubor, když program končí normálně svoji činnost)

7.7 Ošetřování chyb – stderr a exit

Ošetření chyb v programu cat není ideální. Jestliže jeden ze souborů nemohl být otevřen, tak chybová hláška je připojena na konec spojených souborů. To není na závadu tehdy, je-li výstup směrován na terminál. Je-li ale výstup vstupem pro další program, tak už tento způsob není přijatelný.

Druhým výstupním souborem je stderr. Výstup, který je psán na stderr se objevuje na terminálu i tehdy, je-li stdout přesměrován jinam.

Přepíšeme nyní program cat s použitím stderr.

```
      #inclunde <stdio.h>
      main(argc, argv)      /*cat:spojování souborů*/
      int argc;
      char *argv[];
      {
            FILE *fp, *fopen();
            if (argc == 1)      /* žádné argumenty. Kopíruj
                                          standardní vstup */
                  filecopy(stdin);
            else
              while(--argc > 0)
              {  if((fp = fopen(*++argv, "r")) == NULL)
                   {
                      fprintf(stderr, "cat: can't open %s\n",
*argv);
                      exit(1);
                    }
                  else
                  { filecopy(fp);
                    fclose(fp);
                   }
            }
            exit(0);
      }
```

Program signalizuje chybu dvěma způsoby. Chybové hlášení jde na *stderr*, což je uživatel terminálu.

Rovněž je zde použita standardní funkce exit, která ukončuje činnost programu, je-li vyvolána.

Argument funkce exit může být testován následujícím programem. Obyčejně se používá 0 pro znamení, že vše proběhlo řádně a nenulová hodnota pro nenormální ukončení.

Funkce exit volá fclose pro každý otevřený výstupní soubor. Obsah vyrovnávací paměti je přitom připsán do souboru a je vyvolána funkce exit, která okamžitě ukončuje činnost programu. Může být rovněž vyvolána přímo.

7.8 Řádkový vstup a výstup

Standardní knihovna obsahuje funkci fgets, která je obdobou funkce getline, kterou jsme používali v této knize.

Volání

```
fgets(line, MAXLINE, fp);
```

čte další řádek (včetně znaku pro novou řádku) ze souboru *fp* do znakového pole *line*. Maximálně je načteno $MAXLINE - 1$ znaků. Řádek je ukončena znakem \0. fgets normálně vrací řádek. Na konci souboru vrací *NULL* (naše funkce getline vrací délku řádku a *NULL* pro konec souboru). Funkce fputs píše řetězec znaků (řetězec nemusí obsahovat znak pro nový řádek) do souboru.

```
fputs(line, fp)
```

Abychom ukázali, že na funkcích fgets a fputs není nic magického, uvedeme přímo jejich kopie ze standardní knihovny

```
#include <stdio.h>
char *fgets(s, n, iop)      /*načtení n znaků z iop*/
char *s;
int n;
register FILE *iop;
{
   register int c;
   register char *cs;
   cs = s;
      while (--n > 0 && (c = getc(iop)) != EOF)
          if((*cs++ = c) == '\n')
               break;
      *cs='\0';
      return((c == EOF && cs == s) ? NULL : s);

}

fputs(s, iop)      /* zapsání řetězce s do souboru iop */
register char *s;
register FILE *iop;
```

```
{
    register int c;
    while (c = *s++)
          putc(c, iop);
}
```

Cvičení 7.3. Napište program, který porovnává dva soubory a tiskne první řádku a pozici, kde se soubory liší.
Cvičení 7.4. Modifikujte program pro nalezení řetězce z kapitoly 5, aby četl vstup ze vstupních souborů, které budou zadány jako jeho argumenty. Jestliže žádný parametr nebyl zadán, tak bude číst ze standardního vstupu. Může být vytištěno jméno souboru, ve kterém je řetězec nalezen?
Cvičení 7.5. Napište program, který tiskne skupinu souborů, každý na novou stránku. Tiskněte také název souborů a číslujte stránky pro každý soubor zvlášť.

7.9 Některé další funkce

Některé funkce ze standardní knihovny jsou mimořádně užitečné. Již jsme se zmínili o funkcích strlen, strcpy, strcat a strcmp. Zde jsou některé další.

7.9.1 Testování znaků a konverzní funkce

Jsou to tato makra

isalpha(c) – nenulová, je-li *c* písmeno, nulová, jestliže není

isupper(c) – nenulová je-li *c* velké písmeno, nulová, jestliže není

islower(c) – nenulová, je-li *c* malé písmeno, nulová, jestliže není

isdigit(c) – nenulová, je-li *c* číslice, nulová, jestliže není

isspace(c) – nenulová, jestliže *c* je mezera, nulová, jestliže není

toupper(c) – konvertuje *c* na velké písmeno

tolower(c) – konvertuje *c* na malé písmeno

ungetc

Ve stantardní knihovně je hodně omezená verze naší funkce ungetch, kterou jsme napsali v kapitole 4.

```
ungetc(c, fp)
```

Vrací znak zpět do souboru *fp*. Je dovoleno vracet pouze jeden znak na řádku. ungetc může být použita s funkcemi vstupu a výstupu a makry jako jsou scanf, getc nebo getchar.

7.9.2 Systémová volání

Funkce system(s) vykoná příkaz, který je specifikován v řetězci s a vrací se zpátky do programu. Obsah řetězce s záleží na operačním systému. Např. v systému UNIX řádka

```
system("date")
```

spustí program date.

7.9.3 Organizace paměti

Funkce calloc je obdobou funkce alloc, kterou jsme používali v předchozích kapitolách

```
calloc(n, sizeof(objekt))
```

Vrací pointr na místo v paměti, kam lze umístit n objektů specifikované délky. Není-li požadovaná paměť dostupná, vrací $NULL$. Alokovaná paměť je vynulována.

Pointr je řádně nastaven na požadovaný objekt, ale měl by být nastaven na správný typ:

```
char *calloc();
int *ip;
ip=(int*)calloc(n, sizeof(int));
```

Funkce cfree(p) uvolňuje místo v paměti, na které ukazuje pointr p. p bylo původně získáno vyvoláním funkce calloc. Uvolňování paměti **nemusí** být děláno popořádku. Je ale chybou uvolňovat paměť, která nebyla přidělena funkci calloc.

V kapitole 8 si ukážeme implementaci funkce calloc.

Kapitola 8

Souvislosti s operačním systémem UNIX

Tato kapitola se zabývá souvislostmi jazyka C a operačního systému UNIX. Protože mnoho uživatelů jazyka C pracuje na systémech typu UNIX, bude tato kapitola užitečná většině čtenářů. Dokonce i těm uživatelům, kteří pracují s jinými systémy, bude užitečné pro pochopení programování v jazyce C prostudování příkladů uvedených v této kapitole.

Kapitola je rozdělena do tří hlavních částí:

– vstup a výstup
– systém souborů
– přidělování paměti.

První dvě části předpokládají minimálně znalost vnějších vlastností systému UNIX. Kapitola 7 se zabývala systémovým interfacem, který je společný pro mnoho operačních systémů. Na jakémkoliv z těchto systémů jsou funkce standardní knihovny napsány pomocí vstupně/výstupních vlastností skutečného systému. V následujících odstavcích vysvětlíme základní systémové vstupní body pro vstupy a výstupy operačního systému UNIX a ukážeme si, jak s jejich pomocí mohou být implementovány některé části standardní knihovny.

8.1 Deskriptory souboru

V opérečním systému UNIX jsou všechny vstupy a výstupy uskutečňovány pomocí čtení nebo zápisu do souboru, protože všechna periferní zařízení, dokonce terminál operátora, jsou chápána jako soubory ze systému souborů. To znamená, že jediný, homogenní interface řídí veškerou komunikaci mezi programem a periferním zařízením.

Před čtením nebo zápisem do souboru je nejdříve nutné o této skutečnosti informovat systém procesem zvaným „otevření souboru". Jestliže chcete do souboru zapisovat je také nutné ho nejprve vytvořit. Systém sám zkontroluje, je-li vše v pořádku (Existuje soubor? Je do něho dovolen přístup?) a jestliže ano, vrátí vašemu programu nízké kladné celé číslo zvané „deskriptor souboru". Toto číslo je pak vždy užíváno při přístupu k souboru namísto jména souboru. (Tato vlastnost je podobná zvyklostem ve FORTRANU např: READ(5,...) nebo WRITE(6,...).) Všechny informace o otevřeném souboru jsou ovládány systémem, uživatelův program používá pouze deskriptor souboru. Protože vstup a výstup terminálu operátora je společný s ovládáním souborů, existuje možnost jak této výhody využít. Kdykoliv interpret příkazové řádky operačního systému (tzv. „shell“) spouští váš program otevře vždy tři soubory, které mají deskriptory souboru 0, 1 a 2 a jsou nazvány názvy: standardní vstup, standardní výstup a standardní výstup chybových hlášení. Všechny tyto soubory jsou normálně přiřazeny k terminálu operátora, takže program, který čte ze souboru určeného deskriptorem 0, a který píše do souborů určených deskriptory 1 a 2 může toto uskutečnit, aniž by bylo nutné tyto soubory otevřít.

Uživatel může změnit vstupně výstupní přiřazení do a ze souboru pomocí znaků < a >:

```
prog < infile > outfile
```

Ve výše uvedeném případě změní shell původní přiřazení deskriptoru souborů 0 a 1 z terminálu na jmenované soubory. Deskriptor souboru 2 většinnou bývá nezměněný, a chybová hlášení vystupují na terminál operátora. Podobné konvence jsou platné v případě, kdy je vstup a výstup užíván při zřetězeném vykonávání programů (pipe). Je nutné poznamenat, že ve všech výše uvedených případech byla změna provedena shellem a nikoliv programem uživatele. Program uživatele neví, odkud přichází vstup, ani kam směřuje výstup, dokud používá soubory 0 pro vstup a soubory 1 a 2 pro výstup.

8.2 Vstup a výstup nejnižší urovně – read a write

Nejnižší úroveň vstupu a výstupu v operačním systému UNIX nevykonává žádné bufferování ani jiné služby, je to ve skutečnosti přímý vstup do operečního systému. Všechny vstupy a výstupy jsou uskutečněny pomocí dvou funkcí: read a write. Pro obě dvě je prvním argumentem deskriptor souboru. Druhý argument je buffer ve vašem programu, do kterého budou data zapisována, či budou z něho čtena. Třetí argument udává počet slabik, které jsou přeneseny. Použití je následující:

```
n_read = read(fd, buf, n);
n_writen = write(fd, buf, n);
```

Každé vyvolání výše uvedených funkcí vrátí počet slabik, které byly skutečně systémem přeneseny. Při čtení může být toto číslo nižší, nežli je zadaná hodnota. Pokud je vrácena hodnota 0, je to znamení konce souboru a hodnota −1 znamená, že došlo k chybě při vykonání funkce. Při zápisu je vrácená hodnota rovna počtu skutečně zapsaných slabik, pokud je různá od hodnoty zadané, je to znamení chyby.

Počet slabik, které se mohou zapsat či přečíst je libovolný. Dvě nejčastěji používané hodnoty jsou 1 slabika (tj. zápis po jednotlivých znacích) a 512 slabik (tato hodnota odpovídá fyzikální délce bloku na mnoha periferních zařízeních). Druhý případ je více efektivní, ale ani první není zcela nevhodný.

Na základě výše uvedených faktů můžeme nyní napsat jednoduchý program, který kopíruje data ze vstupu na výstup. Je ekvivalentní programu, který je uveden v první kapitole. Pod operačním systémem UNIX tento program kopíruje data odkudkoliv kamkoliv, protože může být přeřazen jeho vstup i výstup na libovolné zařízení nebo do souboru.

```
#DEFINE BUFSIZE 512       /*vhodná velikost pro systém
                                    PDP-11 UNIX*/
```

```
main ()                    /*kopíruje vstup na výstup*/
{
      char buf[BUFSIZE];
      int n;
      while ((n = read(0, buf, BUFSIZE)) > 0)
            write(1, buf, n);
}
```

Jestliže délka souboru není násobkem konstanty BUFSIZE bude při některém čtení vráceno číslo menší nežli hodnota BUFSIZE, takový počet slabik bude zapsán funkcí write. Příští volání funkce vrátí hodnotu 0.

Je poučné prostudovat, jak pomocí fukcí read a write napsat podprogram vyšší úrovně, jako jsou třeba podprogramy getchar, putchar atd. Například si uvedeme jednoduchou verzi funkce getchar, která nepoužívá vstup s využitím buferu.

```
#DEFINE CMASK 0377      /*maska, po jejímž použití mají
                                 znaky kladnou hodnotu*/
getchar()    /*vstup jednoho znaku, bez použití buferu*/
{
      char c;
      return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

Proměnná c musí být deklarována jako proměnná typu char, protože funkce read vyžaduje argument typu ukazovátko na proměnnou typu char. Vrácený znak musí být maskován hodnotou 0377, aby bylo zaručeno, že je pozitivní. Jinak by se mohlo stát, že bude díky znaménkovému rozšíření negativní. Konstanta 0377 je vhodná pro systém PDP-11, ale pro jiné systémy může být rozdílná.

Další verze funkce getchar uskutečňuje vstup více znaků najednou, ale výstup je uskutečňován po jednom znaku.

```
#define CMASK 0377  /* aby hodnota char byla > 0*/
#define BUFSIZE 512

getchar()   /* verze s bufferem */
{
    static char buf[BUFSIZE];
    static char *bufp=buf;
    static int n=0;

    if(n == 0) /* je-li prázdný buffer */
     {    n = read(0, buf, BUFSIZE);
          bufp = buf;
     }
     return((--n > 0) ? *bufp++ & CMASK: EOF);
}
```

8.3 open, create, close, unlink

Kromě standardního vstupního, výstupního a chybového souboru musí být každý další explicitně otevřen pro čtení nebo zápis. Můžeme to provést funkcí open nebo creat.

open se podobá fopen, popsané v kapitole 7, vrací však popisovač souboru, který může být deklarován jako int:

```
int fd;
fd = open(name, rwmode);
```

Parametr *name*, stejně jako u fopen, je řetězec obsahující jméno souboru podle systémových konvencí; *rwmode* je 0 pro čtení, 1 pro zápis a 2 pro čtení i zápis. Funkce vrátí hodnotu -1, vyskytla-li se chyba, jinak vrátí správný popisovač souboru.

Pokus o otevření neexistujícího souboru skončí chybou. Pomocí funkce create založíme nový soubor nebo přepíšeme starý

```
fd = create(name, pmode);
```

vrátí popisovač souboru, povedlo-li se založit soubor, -1 když se nepovedlo. Nepovažuje se za chybu, existoval-li soubor s daným jménem; voláním create se zkrátí na nulovou délku.

Při vytváření souboru se nastaví kód ochrany podle *pmode*. Vystému UNIX je *pmode* 9 bitů povolujících čtení, zápis nebo vykonání pro vlastníka souboru, jeho skupinu a všechny ostatní; *pmode* lze čitelně vyjádřit trojmístným oktalovým číslem, např. 0755 specifikuje čtení, zápis a vykonání pro vlastníka souboru, pro skupinu a ostatní pouze čtení a vykonání.

Jako příklad si uvedeme jednoduchou verzi pomocného programu systému UNIX, který má název cp. Jedná se o program, který kopíruje jeden soubor do souboru jiného. Podstatné zjednodušení je v tom, že náš program kopíruje pouze jeden soubor a není možné, aby druhým argumentem byl adresář.

```
      #DEFINE NULL 0
      #DEFINE BUFSIZE 512
      #DEFINE PMODE 0644      /*R/W pro vlastníka, R pro
ostatní*/

      main(argc, argv)        /*cp: kopíruj f1 do f2*/
      int argc;
      char argv[];
      {
          int f1, f2, n;
          char buf[BUFSIZE];
          if(argc != 3)
             error("Usage: cp from to", NULL);
          if((f1 = open(argv[1], 0)) == -1)
             error("cp: can't open %s", argv[1]);
          if((f2 = creat(argv[2], PMODE)) == -1)
             error ("cp: can't create %s", argv[2]);
          while((n = read(f1, buf, BUFSIZE)) > 0)
             if(write(f2, buf, n) != n)
```

```
                error("cp: write error", NULL);
            exit(0);
        }

        error(s1, s2) /*vytiskni chybovou zprávu a ukonči
činnost*/
        char *s1, *s2;
        {
            printf(s1, s2);
            printf("\n");
            exit(1);
        }
```

Existuje maximální počet současně otevřených souborů, typicky 15 až 25. Program, který používá mnoho souborů, musí být schopen vícenásobného použití deskriptoru souborů. Funkce close ukončí spojení mezi deskriptorem souboru a otevřeným souborem. Tím se uvolní deskriptor souboru pro použití s dalším souborem. Ukončení programu funkcí exit nebo návratem z hlavního programu uzavře všechny otevřené soubory.

Funkce unlink *filename* zruší soubor se jménem *filename* v systému souborů.

Cvičení 8-1. Napište znovu program cat z kapitoly 7 s použitím funkcí read, write, open a close na místo jejich ekvivalentů ze standardní knihovny. Porovnejte rychlosti obou verzí.

8.4 Náhodný přístup – seek a lseek

Normální přístup k souborům je sekvenční. Každý nový zápis nebo čtení ze souboru se uskuteční na následující pozici v souboru. Když je to nezbytné, je možné soubory číst nebo do nich zapisovat na libovolné pozici. Systémová funkce lseek umožňuje přístup do libovolného místa souboru, aniž by bylo nutné soubor skutečně číst, či do něho zapisovat.

```
lseek (fd, offset, origin);
```

Funkce lseek nastaví pozici v souboru, který je určený deskriptorem souboru *fd*, na místo určené posunutím *offset*, který je určován od místa v souboru, určeného argumentem *origin*. Následující čtení nebo zápis do souboru se uskuteční na této pozici. Argument *offset* je typu long, argumenty *fd* a *origin* jsou typu int. Argument *origin* může nabýt hodnoty 0, 1 a 2, která určuje, že posunutí je měřeno od počátku, od pravé aktuální pozice v souboru nebo od konce souboru. Např. nastavení na konec souboru (append) se uskuteční následovně :

```
lseek(fd, 0L, 2);
```

Nastavení na začátek souboru (rewind):

```
lseek(fd, 0L, 0);
```

Je nutné poznamenat, že argument 0L je možné psát ve tvaru (long)0.

S pomocí funkce lseek je možné se soubory pracovat jako s rozlehlými poli, na úkor pomalejšího přístupu k jednotlivým položkám. Například následující funkce čte libovolný počet slabik z jakéhokoliv místa v souboru:

```
      get(fd, pos, buf, n)      /*čti n slabik od pozice pos*/
      int fd, n;
      long pos;
      char *buf;
      {
         lseek(fd, pos, 0);     /*nastav ukazatel na pozici
pos*/
         return(read(fd, buf, n));
      }
```

V operačním systému UNIX V7 je hlavní systémovou funkcí, která umožňuje přístup k souborům, funkce seek. Funkce seek je analogická funkci lssek, vyjma té skutečnosti, že argument je typu int. Protože u systému PDP-11 mají čísla typu int velikost 16 bitů, je možná největší hodnota argumentu *offset* 32767. Proto existují další možné hodnoty argumentu (3, 4, 5), které způsobují vynásobení hodnoty *offset* konstantou 512 (tj. počtem slabik v jednom fyzickém bloku dat) a samotný argument *origin* je zpracován, jako kdyby jeho hodnota byla 0, 1 nebo 2. Je tedy nutné při přístupu do rozlehlých souborů uskutečnit dvě vyvolání funkce seek. První určí blok a druhé, jehož argument *origin* má hodnotu 1 určí požadovanou slabiku uvnitř bloku.

Cvičení 8-2. Funkce seek může být napsána pomocí funkce lseek a naopak. Proveďte.

8.5 Příklad – implementace funkcí fopen a getc

Nyní si ukážeme, jak pomocí výše uvedených informací implementovat funkce fopen a getc ze standardní knihovny.

Připomínáme, že soubory jsou ve standardní knihovně určeny ukazovátkem souboru a nikoliv deskriptorem souboru. Ukazovátko na soubor, je ukazovátkem na strukturu, která obsahuje nezbytné informace o souboru, ukazovátko na bufer, takže soubor může být čten po větších částech; čítač počtu znaků, které v buferu zbývají; ukazovátko na znak v buferu, který má být zpracován: několik klíčů určujících módy čtení/zápis atd. a deskriptor souboru.

Datová struktura popisující soubor je definována v souboru stdio.h, který musí být připojen (pomoci #include) ke všem zdrojovým souborům, které používají podprogramy ze standardní knihovny. Je také začleněn funkcemi této knihovny. Následující výtah ze souboru stdio.h určuje ty, které jsou užívány pouze funkcemi standardní knihovny, znakem podtrhnutí („_“) tak, aby nevznikly kolize se jmény v uživatelově programu.

```
#define _BUFSIZE 215
#define _NFILE   20  /*počet současně otevřených souborů*/

typedef struct _iobuf {
        char *_ptr;          /*pozice následujícího znaku*/
        int  _cnt;           /*počet již zpracovaných znaků*
        char  *_base;        /*adresa I/O buferu*/
        int  _flag;          /*způsob přístupu k souboru*/
        int  _fd;            /*deskriptor souboru*/
} FILE;
```

```
extern FILE _iob[_NFILE];

#define stdin  (_iob[0])
#define stdout (_iob[1])
#define sterr  (_iob[2])

#define _READ      01      /*soubor otevřen pro čtení*/
#define _WRITE     02      /*soubor otevřen pro zápis*/
#define _UNBUF     04      /*přístup bez buferu*/
#define _BIGBUF    010     /*přístup s buferem*/
#define _EOF       020     /*příznak konce souboru*/
#define _ERR       040     /*příznak chyby při přístupu*/
#define NULL       0
#define EOF        (-1)
#define getc(P)    (--(P) -> _cnt >= 0\
                     ? *(p) -> _ptr++ & 0377 : _fillbuf(p))
#define getchar()  getc(stdin)
#define putc(x, p) (--(p) -> _cnt >= 0\
                     ? *(p) -> _ptr++ = (x) : _flushbuf((x),
p))
#define putchar(x) putc(x, stdout)
```

Makro getc pouze dekrementuje čítač, nastaví ukazovátko na novou pozici a vrátí znak. (Dlouhá definice #define je prodloužena na další řádku pomocí znaku „\“ jestliže se čítač stane negativní, je vyvolána funkce _fillbuf, která naplní bufer, provede opětnou inicializaci obsahu struktury a vrátí znak. Tyto funkce smějí obsahovat nepřenositelný interface k operačnímu systému, např. funkce getc maskuje znak hodnotou 0377, aby předešla znaménkovému rozšíření, které vzniká na systému PDP-11, a tím zajistí, že všechny znaky budou mít kladné hodnoty. Přestože se nebudeme zaobírat detaily, je zde také definice funkce putc analogická funkci getc, vyvolávající však funkci _flushbuf při zaplněném buferu.

Nyní již můžeme napsat definici funkce fopen. Největší část funkce fopen se zabývá vlastním otevřením souboru a nastavením ukazovátka na správné místo v souboru a nastavením klíčů na požadovaný stav. Funkce fopen nepřiděluje žádný prostor pro bufer, ten je přidělen funkcí _fillbuf při prvém čtení ze souboru.

```
#include <stdio.h>
#define PMODE 0644      /*R/W pro vlastníka, R pro ostatní*/

FILE fopen(name, mode)      /*otevři soubor a vrať
                                  ukazovátko na desk. soub*/
register char *name, *mode;
{   register int fd;
    register FILE *fp;
    if(*mode != 'r' && *mode != 'w' && *mode != 'a')
    {
        fprintf(stderr,"illegal mode %s opening %s \n", mode,
name;
        exit(1);
```

```
    }
    for(fp >= _iob; fp < _iob + _NFILE; fp++)
          if((fp -> _flag & (_READ | _WRITE)) == 0
             break;            /*najdi volnou položku*/
    if(fp) = _iob + _NFILE      /*není volná položka*/
          return(NULL);
    if(*mode == 'w')      /*přistupuj do souboru*/
          fd = creat(name, PMODE);
    else if(*mode == 'a')
         {
          if((fd = open(name, 1)) == -1)
              fd = creat(name, PMODE);
         lseek(fd, 0L, 2);
         }
         else
              fd = open(name, 0);
    if(fd == -1)            /*nemohu otevřít soubor*/
         return(NULL);
    fp -> _fd = fd;
    fp -> _cnt = 0;
    fp -> _base = NULL;
    fp ->_ flag &= ^(_READ | _WRITE);
    fp -> _flag |= (*mode == 'r') ? _READ : _WRITE;
    return(fp);
}
```

Funkce _fillbuf je poněkud komplikovanější. Hlavní problém spočívá ve faktu, že funkce _fillbuf se pokusí umožnit přístup k souboru dokonce i v případě, kdy není dostatek volné paměti pro vstup/výstup s použitím buferu. Jestliže je paměť pro nový buffer přidělená funkcí calloc, je vše v pořádku. Jestliže paměť přidělena není, umožní funkce _fillbuf přístup k souboru bez buferu, tj. znak po znaku s využitím svého vnitřního pole.

```
#include <stdio.h>

_fillbuf(fp)      /*přiděl a naplň vstupní bufer*/
register FILE fp;
{
   static char smallbuf[_NFILE];   /*pro I/0 bez buferu*/
   char *calloc();

   if((fp -> _flag & _READ) == 0 || fp -> _flag & (_EOF |
_ERR)) != 0)
      return(EOF);
   while(fp -> _base == NULL) /*najdi místo pro bufer*/
      if(fp -> _flag & _UNBUF) /*I/0 bez buferu*/
         fp -> _base = &smallbuf[fp -> _fd];
      else if((fp -> _base = calloc(_BUFSIZE, 1)) == NULL)
             fp -> _flag |= _UNBUF; /*nemohu najít místo pro
bufer*/
```

```
            else
             fp -> _flag |= _BIGBUF; /*místo pro bufer bylo
nalezeno*/
   fp -> _ptr = fp -> _base;
   fp -> _cnt = read(fp -> _fa, fp -> _ptr,
                  fp ->_flag & _UNBUF ? 1 : _BUFSIZE);
   if(--fp -> _cnt < 0)
   {
         if(fp -> _cnt == -1)
             fp -> _flag |= _EOF;
          else
             fp -> _flag |= _ERR;
         fp -> _cnt = 0;
         return(EOF);
   }
   return(*fp -> _ptr++ & 0377); /*vrať kladný znak*/
}
```

Při prvním vyvolání funkce getc pro daný soubor bude čítač _cnt roven nule, což způsobí vyvolání funkce _fillbuf. Jestliže funkce _fillbuf zjistí, že soubor není otevřen pro čtení, vrátí bezprostředně hodnotu *EOF*. Jestliže tomu tak není, zkusí nejprve přidělit souboru bufer. Pokud není možné souboru bufer přidělit, označí to v proměnné _flag.

Jakmile je bufer stanoven, je vyvolána funkce read, aby ho naplnila, je nastaven čítač, ukazovátka a je vrácen znak z počátku buferu. Další volání funkce _fillbuf již naleznou bufer souboru přidělený.

Zbýva pouze jediná věc, kterou je nutné vysvětlit. Pole _iob musí být definováno a inicializováno pro soubory stdin, stdout a stderr:

```
      FILE _iob[_NFILE]=(
             (NULL, O, NULL, _READ, 0),         /*stdin*/
             (NULL, 0, NULL, _WRITE, 1),        /*stdout*/
             (NULL, 0, NULL, _WRITE _UNBUF, 2) /*stderr*/
      );
```

Z inicializace proměnné _flag je patrné, že soubor stdin je určen pouze pro čtení, soubor stdout pouze pro zápis a soubor stderr pouze pro zápis bez použití bufferu.

Cvičení 8-3. Přepište funkce fopen a _fillbuf a použijte pole bitů namísto explicitních bitových operací.
Cvičení 8-4. Navrhněte funkce _flushbuf a fclose.
Cvičení 8-5. Součástí standardní knihovny je funkce fseek

```
 fseek(fp, offset, origin)
```

která je identická s funkcí lseek s tím rozdílem, že *fp* je ukazovátko na soubor místo deskriptoru souboru. Napište funkci fseek. Ujistěte se o správné funkci vaší funkce ve spolupráci s ostatními funkcemi ze standardní knihovny.

8.6 Příklad – výpis adresáře

Rozdílnou úlohou při práci se soubory je získávání informací o souboru, nikoliv práce s jeho obsahem. Příkaz operačního systému UNIX ls (list directory = výpis adresáře) vytiskne jména souborů v adresáři a pokud je žádoucí tak i další informace, jako je velikost, způsoby přístupu atd.

Protože v operačním systému UNIX je adresář chápán jako soubor, je práce s ním stejná jako s jinými soubory, systém nejprve soubor s adresářem přečte a vyjme z něho všechny nezbytné informace o souborech, které v něm nalezne. Formát informace v adresáři je určen operačním systémem, nikoliv uživatelovým programem, proto musí program ls znát, jaká je reprezentace informací v adresáři.

Ukážeme si některé výše uvedené zásady na programu nazvaném fsize. Program fsize je specializovaná forma programu ls, která vytiskne velikosti všech souborů, které jsou uvedeny v seznamu argumentů příkazové řádky. Jestliže jeden z těchto argumentů je adresářem, uplatní se na něm rekurzivně opět program fsize. Jestliže nejsou žádné argumenty bude zpracován právě používaný adresář.

Nyní stručný přehled o systému souborů. Adresář je soubor, který obsahuje seznam jmen souborů a informace o tom, kde je soubor uložen. Informace o umístění souboru je ve skutečnosti indexem v tabulce zvané „inod table“. Položka adresáře se skládá pouze ze dvou částí. Ze jména souboru a čísla určujícího položku v tabulce „inod table“. Tato položka obsahuje veškeré informace o souboru a jeho umístění. Přesná specifikace je začleněna v souboru sys/dir.h, který obsahuje:

```
#define DIRSIZ 14 /*maximální délka jména souboru*/
struct direct     /*struktura položky adresáře*/
{
   ino_t d_ino;            /*hodnota inode*/
   char  d_name[DIRSIZ];   /*jméno souboru*/
} ;
```

Typ ino_t je typ definovaný pomocí konstrukce typedef a popisuje index do tabulky „inod table“. Na systému PDP-11 UNIX je typu unsigned, ale na jiných systémech může být rozdílného typu. Kompletní přehled všech systémových typů je obsažen v souboru sys/types.h.

Funkce stat má vstupním argumentem jméno souboru a vrací veškerou informaci, která je obsažena v tabulce „inod table“, nebo hodnotu −1, pokud došlo k chybě.

```
struct stat stbuf;
char *name;
stat(name, &stbuf);
```

Funkce stat zaplní strukturu stbuf informací obsaženou v tabulce „inod table“ pro daný soubor. Struktura popisující vrácenou informaci je obsažena v souboru sys/stst.h a je následovná:

```
struct stat   /*struktura vrácená funkcí stat*/
{
   dev_t   st_dev;   /*typ zařízení*/
   ino_t   st_ino;   /*hodnota inode*/
   short   st_mode   /*způsob přístupu*/
```

```
    short   st_nlink; /*počet směrníků souborů*/
    short   st_uid;   /*vlastníkův identifikátor*/
    short   st_gid;   /*skupinový identifikátor*/
    dev_t   st_rdev;  /*příznak zvláštních souborů*/
    off_t   st_size   /*délka souboru ve znacích*/
    time_t  st_atime; /*datum posledního přístupu*/
    time_t  st_mtime; /*datum poslední modifikace*/
    time_t  st_ctime; /*datum vytvoření souboru*/
};
```

Mnoho z výše uvedené struktury je objasněno v komentářích obsažených v definici. Položka st_mode obsahuje soubor klíčů, které popisují soubor. Definice klíčů je součástí souboru sys/stat.h.

```
#define  S_IFMT   0160000  /*typ souboru*/
#define  S_IFDIR  0040000  /*adresář*/
#define  S_IFCHR  0020000  /*zvláštní znak*/
#define  S_IFBLK  0060000  /*zvláštní blok*/
#define  S_IFREG  0100000  /*regulární*/
#define  S_ISUID  04000    /*nastav vlastníkův
identifikátor*/
#define  S_ISGID  02000    /*nastav skupinový
identifikátor*/
#define  S_ISVTX  01000    /*uschovej text po použití*/
#define  S_IREAD  0400     /*čtení dovoleno*/
#define  S_IWRITE 0200     /*zápis dovolen*/
#define  S_IEXEC  0100     /*vykonání dovoleno*/
```

Teprve nyní můžeme začít psát program fsize. Jestliže informace obdržená funkcí stat určuje, že soubor není ardesářem, může být jeho velikost přímo obdržena a vytisknuta. Jestliže se jedná o adresář, zpracujeme rekurzivně jeho položky. Nejvyšší úroveň programu se zabývá zpracováním argumentů příkazové řádky. Každý argument této řádky poskytne ke zpracování funkci fsize.

```
#include <stdio.h>
#include <sys/types.h>   /*definice typu*/
#include <sys/dir.h>     /*struktura položky adresáře*/
#include <sys/stat.h>    /*struktura vrácená funkcí stat*/
#define  BUFSIZE 256

main(argc, argv)    /*fsize: tiskni velikost souboru*/
char *argv[];
{
   char buf[BUFSIZE];
   if(argc == 1)      /*default: aktuální adresář*/
   {
         strcpy(buf, ".");
         fsize(buf);
   }
   else
```

```
            while (--argc > 0)
            {
                  strcpy(buf, *++argv);
                  fsize(buf);
            }
      }
```

Funkce fsize tiskne velikosti souborů. Jestliže se jedná o adresář, vyvolá funkce fsize funkci directory, aby mohla zpracovat soubory obsažené v adresáři. Všimněte si použití názvů klíčů S_IFMT a S_IDFIR ze souboru stat.h.

```
fsize(name)      /*tiskni velikost daného souboru*/
char *name;
{
    struct stat stbuf;
    if (stat (name, &stbuf) == -1)
    {
        fprintf (stderr,"fsize: can't find %s n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        directory(name);
    printf ("%81d %s \n", stbuf.st_size, name);
}
```

Funkce directory je poněkud komplikovaná. Jejím hlavním účelem je vytvoření úplného jména souboru, kterým se právě zabývá.

```
directory(name)      /*velikosti všech souborů adresáře*/
char *name;
{
    struct direct dirbuf;
    char *nbp, *nep;
    int i, fd;
    nbp = name + strlen(name);
    *nbp++ = '/';      /*přidej znak "/" ke jménu adresáře*/
    if(nbp + DIRSIZ + 2 >= name + BUFSIZE) /*jméno je
                                              příliš dlouhé*/
          return;
    if((fd = open(name, 0)) == -1)
          return;
    while(read(fd, (char*) & dirbuf, sizeof(dirbuf)) > 0)
    {
          if(dirbuf.d_ino == 0) /*položka není použita*/
                continue;
          if(strcmp(dirbuf.d_name, ".") == 0
                 || strcmp(dirbuf.d_name, ".." == 0)
                continue;  /*přeskoč sama sebe a své rodiče*/
          for (i = 0, nep = nbp; i < DIRSIZ; i++)
                *nep++ = dirbuf.d_name[i];
```

```
            *nep++ = '\0';
            fsize(name);
        }
        close(fd);
        *--nbp = '\0';    /*obnov jméno*/
}
```

Jestliže položka adresáře není použita, protože byl soubor zrušen, je index na tabulku „inod table" roven nule. Potom je tato položka při zpracování přeskočena. Každý adresář obsahuje také položku pro sebe sama, která je označena „." a pro své „otce", která je označena „..". Tyto položky musí být také přeskočeny, protože by došlo k zacyklení programu.

Přestože je program fsize poněkud specializovaný, lze si na něm ukázat několik důležitých ideí. Za prvé, mnoho programů, přestože nejsou programy systémovými, používá informace obhospodařované operačním systémem. Za druhé, pro takové programy je rozhodující, že reprezentace takových informací je uložena pouze ve standardních souborech, jako jsou např. stat.h a dir.h, a že programy, které tyto soubory začleňují, nemusí skutečné deklarace v sobě obsahovat.

8.7 Příklad – přidělování paměti

V kapitole 5 byla uvedena jednoduchá verze funkce alloc. Verze, která bude nyní uvedena má neomezené možnosti. Volání funkcí alloc a free mohou být prostřídána v libovolném pořadí. Funkce alloc používá volání operačního systému pro přidělení paměti. Tyto funkce také ilustrují možnost psát programy závislé na technickém vybavení systému relativně nezávisle od tohoto vybavení. Dále ukazují užití struktur, unionů a definic typu typedef v praxi.

Namísto přidělování paměti z pole pevně dané velikosti určené při překladu využívá funkce alloc žádosti k operačnímu systému o přidělení paměti. Protože i další činnosti programu mohou vyžadovat přidělení paměti, je paměť přidělována nekontinuálně. Proto musí být volná paměť (tj. informace o ní) udržována v řetězu volných bloků paměti. Každý blok obsahuje svoji velikost, ukazovátka na další blok a vlastní volnou paměť. Bloky jsou zřetězeny v pořadí stoupající velikosti adresy operační paměti a blok poslední (s nejvyšší adresou) ukazuje zpět na první blok, takže řetěz vytváří ve skutečnosti kruh.

Při žádosti o přidělení paměti je prohledáván seznam volných bloků paměti, dokud není nalezen dostatečně veliký blok paměti. Jestliže má blok požadovanou velikost, je ze seznamu volných bloků vyřazen a přidělen uživateli. Jestliže je blok příliš veliký, je rozdělen a přiměřená část je přidělena uživateli, zatímco část zbývající je začleněna zpět do seznamu volných bloků. Jestliže není nalezen žádný blok dostatečné velikosti, je žádán operační systém o přidělení dalšího bloku paměti, který je začleněn do seznamu volných bloků.

Uvolňování bloků paměti způsobuje prohledávání seznamu volných bloků proto, aby bylo nalezeno vhodné místo pro začlenění právě uvolněného bloku paměti. Jestliže právě uvolňovaný blok paměti je přiléhající k bloku jinému, je k němu připojen, aby nedocházelo k přílišnému rozdělení paměti. Nalezení přiléhajících bloků je snadné, protože seznam volných bloků paměti je organizován dle pořadí ukládání.

Jeden problém, o kterém byla zmínka v kapitole 5, je zajištění toho, aby blok paměti, přidělený funkcí alloc byl vhodný pro objekty, které do něho budou uloženy. Ačkoliv existují různé druhy výpočetních systémů, existuje pro každý z nich nejvíce omezený objekt. Nejvíce omezený objekt může být ukládán pouze na některé adrese. Na tyto adresy pak mohou být ukládány také všechny objekty méně omezené. Například na systémech IBM 360/370, Honeywell 6000 a některých dalších mohou být libovolné objekty uloženy na místa vhodná pro uložení objektu typu double. Na systému PDP-11 je tímto objektem objekt typu int.

Volný blok obsahuje ukazovátko na následující blok v řetězci, záznam o velikosti bloku a vlastní volnou paměť. Kontrolní informace umístěná na počátku bloku se nazývá hlavičkou („header"). Pro zjednodušení přidělování mají všechny bloky velikost rovnu násobku velikosti hlavičky, která musí být správně umístěna. Toho je dosaženo unionem, který obsahuje požadovanou strukturu hlavičky a příkladem nejvíce omezeného typu.

```
typedef int ALIGN;  /*vhodné přiřazení pro PDP-11*/
union header        /*volná hlavička bloku*/
 {    struct{
        union header *ptr; /*následující volný blok*/
        unsigned size;  /*velikost tohoto volného bloku*/
      } s;
      ALIGN x;      /*přiřaď správně blok*/
 };
typedef union header HEADER;
```

Funkce alloc zaokrouhlí požadovanou velikost přidělené paměti v násobcích velikosti hlavičky. Skutečná velikost bloku přidělené paměti je větší o jednu jednotku velikosti hlavičky, protože blok musí obsahovat i vlastní hlavičku. Tato hodnota je obsažena v proměnné *size* hlavičky. Ukazovátko vrácené funkci alloc, však ukazuje na počátek volné paměti, nikoliv na hlavičku bloku volné paměti.

```
static HEADER base;      /*počáteční prázdný seznam*/
static HEADER *allocp = NULL; /*naposledy přidělený blok*/

char *alloc(nbytes)  /*základní funkce přidělování paměti*/
unsigned nbytes;
{
      HEADER *morecore();
      register HEADER *p, *q;
      register int nunits;
      nunits = 1 + (nbytes + sizeof(HEADER) - 1) /
sizeof(HEADER);
      if(( q = allocp) == NULL) /*seznam volných bloků
neexistuje*/
      {
            base.s.ptr = allocp = q = &base;
            base.s.size = 0;
      }
      for(p = q -> s.ptr; ; q = p -> s.ptr)
      {
         if(p -> s.size >= nunits      /*je větší*/
         {
```

```
                if(p -> s.size == nunits) /*je roven*/
                      q -> s.ptr = p -> s.ptr;
                else      /*přiděl konec bloku*/
                {
                      p -> s.size -= nunits;
                      p += p -> s.size;
                      p -> s.size = nunits;
                }
                allocp = q;
                return ((char*)(p+1));
              }
          if(p == allocp) /*zamez opakovanému prohledávání*/
                if ((p = morecore(nunits)) == NULL)
                   return (NULL); /*není žádný volný blok*/
          }
}
```

Proměnná *base* je použita pouze při odstartování procesu přidělování paměti. Jestliže je ukazovátko *allocp* hodnoty *NULL*, provádí se první volání funkce alloc, při kterém je vytvořen degenerovaný seznam volné paměti, který obsahuje jeden blok nulové velikosti a odkazuje sám na sebe. Při dalších voláních funkce alloc, již bude blok volné paměti hledán. Vyhledání volného bloku paměti započne z místa, kde byl přidělen volný blok paměti naposledy, což napomáhá k udržení homogennosti seznamu volné paměti. Jestliže je nalezen příliš veliký blok paměti, je uživateli přidělena paměť od konce bloku. Tímto způsobem je hlavička původního bloku zachována. Pouze je změněna hodnota size udávající velikost volné paměti bloku. Ve všech případech je však uživateli vráceno ukazovátko na skutečně volnou paměť, která začíná jednu jednotku velikosti hlavičky bloku za počátkem bloku (tj. za hlavičkou). Poznamenejme, že proměnná t je konvertována na proměnnou typu ukazovátko, před jejím vrácením funkcí alloc.

Funkce morecore obdrží volnou paměť od operačního systému. Podrobnosti o tom jak, se samozřejmě liší na různých systémech. V systému UNIX vrátí systémové volání sbrk(n) ukazovátko na n dalších slabik volné paměti. Ukazovátko vrácené voláním uspokojuje všechny požadavky na správné přidělení paměti. Protože přidělování paměti prostřednictvím operačního systému je poměrně náročná operace, není žádoucí, aby každé volání funkce alloc používalo služby operačního systému. Proto funkce morecore zaokrouhlí počet zadaných jednotek volné paměti na větší hodnotu. Takový blok potom může být znovu přidělován, jak je zapotřebí. Pravidlo zaokrouhlení je parametrem, který může být nastaven tak, jak je žádoucí.

```
#define NALLOC 128  /*počet najednou přidělených jednotek*/
static HEADER *morecore(nu)  /*žádej systém o volnou paměť*/

unsigned nu;
{
      char *sbrk();
      register char *cp;
      register HEADER *up;
      register int rnu;
      rnu = NALLOC * ((nu + NALLOC - 1) / NALLOC);
```

```
	cp = sbrk(rnu * sizeof(HEADER));
	if((int) sp == -1)	/*není žádná volná paměť*/
		return(NULL);
	up = (HEADER*)cp;
	up -> s.size = rnu;
	free((char*)(up + 1));
	return(allocp);
}
```

Funkce sbrk vrátí hodnotu -1, jestliže už ani systém nemůže přidělit žádnou volnou paměť, ačkoliv by byla vhodnější hodnota *NULL*. Hodnota -1 musí být konvertována na hodnotu typu int, aby mohla být porovnána. Naproti tomu použití konstrukce 'cast' (vid. 7.2) umožňuje, aby funkce byla poměrně imunní proti různým reprezentacím ukazovátek na rozdílných systémech.

Funkce free jednoduše prohledává seznam volných bloků paměti, počínaje od bloku určeného ukazovátkem *allocp*. Hledá místo pro zařazení volného bloku. Takové místo existuje buď mezi dvěma bloky nebo za posledním blokem v seznamu. Každopádně však budou volné bloky, které spolu sousedí, spojeny v blok jediný. Jediná obtíž programu je udržet ukazovátka tak, aby ukazovala na správná místa a velikosti volných bloků paměti byly správné.

```
free(ap)	/*vlož blok ap do seznamu volných bloků*/
char *ap;
{
	register HEADER *p, *q;
	p = (HEADER*) ap - 1; /*ukazovátko na hlavičku bloku*/
	for(q = allocp; (p > q && p < q -> s.ptr); q = q ->
s.ptr)
	  if(q >= q -> s.ptr && (p > q || p < q -> s.ptr))
		break;	/*nepřiléhá-li*/
	if(p + p -> s.size == q -> s.ptr) /*připoj k násled.
bloku*/
	{
		p -> s.size += q -> s.ptr -> s.size;
		p -> s.ptr = q -> s.ptr -> s.ptr;
	}
	else
		p -> s.ptr = q -> s.ptr;
	if(q + q -> s.size == p) /*připoj k předch. bloku*/
	{
		q -> s.size += p -> s.size;
		q -> s.ptr = p -> s.ptr;
	}
	else
		q -> s.ptr = p;
	allocp = q;
}
```

Ačkoliv je přidělování paměti v podstatě závislé na technickém vybavení systému, program výše uvedený ukazuje, jak tyto závislosti mohou být řízeny a svěřeny nevelké části programu. Užití konstrukcí typedef a union je vhodné pro řízení přidělení správného úseku paměti (za předpokladu, že funkce sbrk vrátí správné ukazovátko). Použití konstrukce 'cast' zabezpečí explicitní uskutečnění konverze ukazovátka a to dokonce i v případě, kdy je špatně navržený systémový interface. Přesto že se jedná o podrobnosti uplatněné při navrhování funkcí zabezpečujících přidělování volné paměti, je jejich všeobecné uplatnění daleko širší.

Cvičení 8-6. Funkce ze standardní knihovny calloc(n, *size*) vrátí ukazovátko na volný blok paměti o velikosti n**size*, která je inicializována nulami. Napište funkci calloc, použitím funkce alloc jako vzoru nebo jako volané funkce.

Cvičení 8-7. Funkce alloc přijme žádost o přidělení volné paměti dané velikosti aniž kontroluje reálnost takové žádané velikosti paměti. Funkce free dále předpokládá, že blok, který je zpracováván, obsahuje korektní hodnotu v proměnné *size*. Pozměňte výše uvedené funkce tak, aby byly schopny více rozpoznávat takové chyby.

Cvičení 8-8. Napište funkci bfree(p, n), která uvolní libovolný blok po n znacích a zařadí jej do seznamu volné paměti. Použitím funkce bfree může uživatel přiřadit statické nebo externí pole do seznamu bloku volné paměti v libovolném okamžiku.