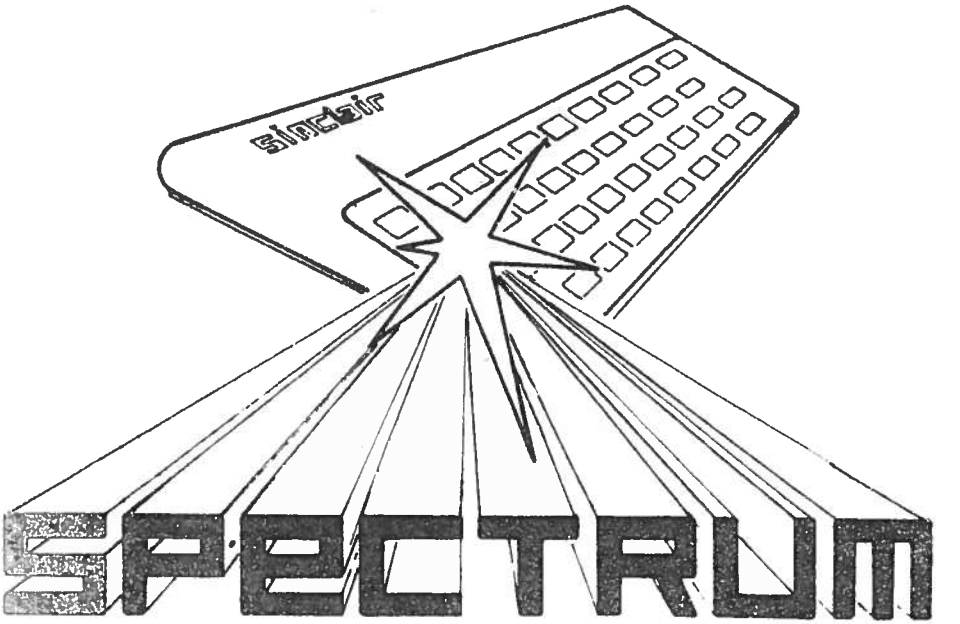
ZO SVAZARM KAROLINKA

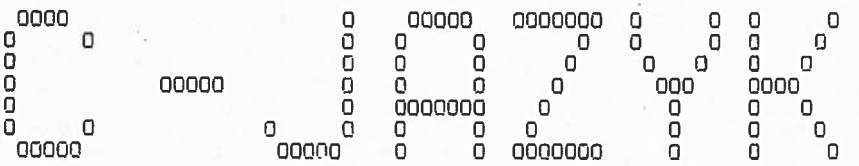
Sborník č. 22

C - JAZYK



OBSAH

str.



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.

Tato kniha je koncipovaná tak, aby pomohla naučit se čtenáři programovat v jazyku C. Je určena především uživatelům TNS s operačním systémem CP/M. 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 pouhym učením zákonitosti. Všechny uvedené příklady tvoří úplné, skutečné programy (nikoliv pouze části),které byly odladěny na počítači TNS. 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, zvlaště 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.

KAPITOLA 0: ÚVOD

C je univerzální programovací jazyk. Jazyk C není vázán k určitému operačnímu systému nebo určitému typu počítače, přestože je nazýván "systémovym programovacím jazykem" a je použitelný pro psaní operačních systémů. Je stejně dobře použitelný i pro tvoření programů numerického charakteru, programů pro zpracování textu a hromadných dat.

Jazyk C je jazyk relativně nízké úrovně. Tento jeho rys nesnižuje však jeho význam; je tím řečeno, že C jazyk pracuje se stejnou třídou objektu 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 mechanismy 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 může proto 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 již s existujícími překladači.

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. 13 000 řádek v jazyku C a jenom 800 řádků assembleru na nejnižší úrovni. Navíc veškerý aplikační software systému UNIX je psán v jazyku C.

Pro ty programátory, kteří pracují s jinými jazyky, může být užitečné pro srovnání se dozvědět o historických, technických a filozofických aspektech jazyka C. Většina nejdůležitějších myšlenek jazyka C pochází z dosti starého, ale stále živého jazyka BCPL. Vliv BCPL na C se uskutečnil nepřímo jazykem B, který byl napsán v r. 1970 pro první systém UNIX.

Přestože jazyk C má s BCPL mnoho společných znaků, není v žádném případě jeho kopií. Jazyky BCPL a B jsou jazyky bez "typů". 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 a celá čísla různých délek. 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 byt volány rekurzívně a lokální proměnné jsou převážně "automatické", tj. jsou vytvořeny znovu pri 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, nebo 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 verzé, ale neprovádí automatické konverze dat s velkou přehlížívostí jazyka PL/1. Dosud vytvořené překladače nekontrolují při výpočtu meze polí, typy argumentů atd.

Pro situace, kdy je nezbytné kontrolovat typy dat, se používá speciální 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, 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 programování. 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, nemyslíme si, že tato kniha bude příručkou datových struktur a algoritmů. Vždy, když jsme si mohli vybrat, tak 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 pointru a aritmetických adres. Kapitola 6 obsahuje popis struktur a unionů.

Kapitola 7 popisuje standardní knihovnu vstupů a výstupů, která zajišťuje styk s operačním systémem.

Kapitola 8 popisuje interface mezi jazykem C a operačním systémem CP/M se zaměřením na vstupy a výstupy, systémy souborů a přesnost. Přestože je tato kapitola zaměřena na CP/M, tak programátor, který tento systém nepoužívá, zde může nalézt užitečný materiál.

Protože se jazyk C stále vyvíjí a existuje již na mnoha systémech, některá část této knihy nemusí korespondovat se současným stavem vývoje jazyka C na určitém systému.

Následující řádky jsou určeny těm programátorům, kteří si budou poznatky získané v následujících kapitolách ověřovat na počítači TNS s operačním systémem CP/M V 2.25.

Vzhledem k tomu, že překladače, knihovny a další moduly nutné pro práci v C jazyku zabírají na disketě poměrně značný prostor, není možné pracovat v C jazyku pouze s jednou disketou.

Nejvýhodnějším řešením se zdá být umístění všech "neměnných" modulů na jedné disketě (kompilační), která může být později doplněna o moduly pro práci s pseudo-floppy.

Na druhé disketě (pracovní) pak udržujeme pouze nezbytné moduly a zdrojový text laděného programu. Zbytek volné kapacity této diskety slouží pro ukládání meziproduktů překladačů.

Výhodou tohoto řešení je především možnost udržovat si několik pracovních disket s různými laděnými programy, přičemž kompilační disketa zůstává pouze jedna.

Později je možné založit také disketu různých ladících a služebních programů, kterou využijeme především pro ladění programů na pracovních disketách.

Zde je návrh kompilační a pracovní diskety:

O b s a h k o m p i l a č n í d i s k e t y:

Soubor	Typ	F u n k c e
PIP	COM	program pro kopírování souborů
STAT	COM	informace o souborech a zařízeních
CC	COM	kompilátor SOFT - C (překlad do U-kódu)
C2	COM	optimizér SOFT - C (překlad do assembleru)
M80	COM	překladač do assembleru (I8080 i Z80)
L80	COM	spojovací a zaváděcí program
LIB	COM	program pro práci s knihovnami
C2PRE	REL	inicializační modul pro binární tvar programu
C2POST	REL	uzavírací modul pro binární tvar programu
LIBC	REL	knihovna standardních funkcí
TNSLIB	REL	knihovna funkcí pro TNS
IZOT	REL	modul pro práci s magn. páskou
STDIO	H	definiční modul pro práci se soubory
CBRACK	H	definiční modul vybraných znaků a operátorů
KEYSP	H	definiční modul kláves apod.
IZOT	H	záhlaví pro programy pracující s magn. páskou

O b s a h p r a c o v n í d i s k e t y:

Soubor	Typ	F u n k c e
PIP	COM	program pro kopírování souborů
ED	COM	textový editor
SUBMIT	COM	program pro vykonávání příkazových souborů
C	SUB	pomocný soubor SUBMITu pro kompilaci
C2	RTM	pomocný soubor pro optimizér
???	C	zdrojový text v C jazyku (laděný program)

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 jen je 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 programů a další podrobnosti.

Tento přístup má ovšem své nevýhody. Zvláštnosti je to, že kompletní popis určité vlastnosti jazyka není na jednom místě. Vzhledem k tomu, že nelze 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, které jsou modifikací programů uvedených.

1.1. Začínáme

Jediny způsob, jak se naučit nový programovací jazyk, je psát programy v tomto jazyku. Zde je náš první program:

Vytiskni slova

a h o j , s v e t e

To je základní úkol. K tomu, abychom ho splnili, 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 "ahoj, svete" vypadá v jazyku C takto:

```
main ()

{
        printf ("ahoj, svete\n");
}
```

Následuje popis vytvoření, přeložení, sestavení a spuštění programu v jazyku C na počítači TNS.

Zdrojový text programu v C jazyku vytvoříme pomocí textového editoru. { se na TNS zadává současným stisknutím kláves [SPEC] [(] a } stisknutím kláves [SPEC] [)] na malých písmenech. Nechť se nazývá např. AHOJ.C. Nejprve je třeba vytvořit tzv. U-kód; to provedeme příkazem:

```
b:cc ahoj.c
```

Z mechaniky B se zavede kompilátor SOFT - C, který zpracuje zdrojový modul a vytvoří soubor AHOJ.COD. Z tohoto souboru nyní vytvoříme assemblerovský modul pomocí příkazu:

```
b:c2 ahoj.cod
```

Z mechaniky B se zavede optimizér SOFT - C, který vytvoří soubor AHOJ.ASM. Nyní již nepotřebujeme soubor AHOJ.COD, a proto ho můžeme vymazat - uvolní se tím místo na disketě:

```
era ahoj.cod
```

V dalším kroku přeložíme soubor AHOJ.ASM makroassemblerem a získáme tzv. přemístitelný tvar programu:

```
b:m80 =ahoj.asm
```

Z mechaniky B se zavede makroassembler, který vytvoří soubor AHOJ.REL. Nyní již nepotřebujeme soubor AHOJ.ASM, a proto ho vymažeme:

```
era ahoj.asm
```

Posledním krokem je spojení našeho přemístitelného modulu s dalšími potřebnými moduly C jazyka. To provedeme příkazem:

```
b:l80 b:c2pre,ahoj,b:tnslib/s,b:libc/s,b:c2post,ahoj/n/e
```

Výsledkem je již hotový program AHOJ.COM. Modul AHOJ.REL můžeme vymazat:

```
era ahoj.rel
```

Pochopitelně, pokud byly v našem programu nějaké chyby, musíme se vrátit na začátek - opravit zdrojový soubor a kompilovat znovu. Že je program bez formálních chyb, to se dozvíme podle hlášení jednotlivých průchodů:

```
0 compilation errors          (kompilátor)
0 code generation errors      (optimizér)
No  Fatal error(s)            (makroassembler)
```

Program s názvem ahoj.com můžeme spustit zadáním jeho jména:

```
ahoj
```

Pokud byl předchozí postup v pořádku, vytiskne se text:

```
ahoj, svete
```

Pro usnadnění kompilace je výhodné používat program SUBMIT, který umožňuje vykonávat příkazove soubory. Celý postup kompilace je napsán (editorem) v souboru C.SUB:

```
era *.bak
b:cc $1.c
b:c2 $1.cod +prgl
era $1.cod
b:m80 =$1.asm
era $1.asm
b:l80 b:c2pre,$1,b:tnslib/s,b:libc/s,b:c2post,$1/n/e/y
era $1.rel
KONEC PREKLADU
```

Symbol "$1" zde nahrazuje skutečně jméno souboru, proto může být tento pomocný soubor využíván pro kompilaci různých programů.
Rozdíly oproti našemu postupu jsou následující:

- první řádek smaže všechny staré soubory (kvůli prostoru);
- ve třetím řádku parametr "+prgl" způsobí vypisování názvu jednotlivých optimalizovaných funkcí zdrojového souboru;
- v sedmém řádku parametr "/y" způsobí navíc vytvoření souboru typu .SYM, který obsahuje symbolické konstanty pro ladění programem SLAP nebo ZSID;
- poslední řádek oznámí textem i zvukem ukončení překladu.

Vlastní překlad nyní provedeme tímto jednoduchým příkazem (předpokládáme náš program AHOJ.C):

```
submit c ahoj
```

Nejprve se zavede program SUBMIT, kterému se odevzdává jméno příkazového souboru (C) a jméno, které má dosazovat za symbol "$1". Také lze přejmenovat program SUBMIT.COM na S.COM a tím se příkaz pro kompilaci ještě zjednoduší:

```
s c ahoj
```

Pozn.: Pozor na příkaz: s c ahoj.c !!!
Pokud zadáme omylem i typ souboru, dosadí SUBMIT ve 4. řádku příkazového souboru

```
era ahoj.c.cod
```

Tento příkaz nám velmi jednoduše vymaže zdrojový soubor ! Je třeba prací SUBMITu včas zastavit.

C v i č e n í 1 - 1:

Spusťte tento program a potom zkoušejte vynechávat některé jeho části a pozorujte chybová hlášení.

Nyní něco o programu samém. Program v jazyce C, ať je jeho velikost jakákoliv, se 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 m a i n .

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 používají 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 { } združují příkazy, které vytvářejí tělo funkce. Jsou analogické příkazům DO-ENO 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.

Programový řádek

```
printf ("ahoj, svete\n");
```

je voláním funkce, která se jmenuje printf, jediným argumentem je řetězec znaků "ahoj, svete\n".

Printf je knihovní funkce, která zobrazuje výstup na displej (jestliže není specifikováno jiné medium). V tomto případě zobrazí řetězec znaků, který je jejím argumentem.

Souvislá řada libovolného množství znaků vložená do uvozovek "....." je nazývaná znakový řetězec nebo řetězcová konstanta. Od této chvíle budou znakové řetězce používány jako argumenty funkcí jako je např. funkce p r i n t f.

Dvojice znaků \n v řetězci je v jazyku C symbolem pro znak nového řádku, který, když je nalezen, posune kurzor na levý okraj nového řádku. Uvedení dvojice znaků \n je jediný způsob přechodu na nový řádek.

Když budete zkoušet něco podobného jako

```
printf ("ahoj, svete
");
```

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

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

```
main ()

{
        printf ("ahoj,");
        printf (" svete");
        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 mechanismus pro reprezentaci těžko zobrazitelných nebo neviditelných znaků. Např. \t je symbol pro tabelátor, \b pro zpětný posun kurzoru, \" pro uvozovky a \\ pro obrácené lomítko samo.

C v i č e n í 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 tabulku druhých mocnin pro čísla 0 až 10:

```
0       0
1       1
2       4
3       9
..      ...
9       81
10      100
```

Program vypadá takto:

```
/* tisk tabulky druhých mocnin pro základ = 0, 1, ..., 10 */

int dolní_mez, horní_mez, krok;

main ()

{
        int zaklad, mocnina;

        dolni_mez = 0;          /* dolní mez tabulky     */
        horni_mez = 10;         /* horní mez tabulky     */
        krok = 1;               /* hodnota kroku         */
        zaklad = dolni_mez;

        while (zaklad <= horni_mez)
        {
                mocnina = zaklad * zaklad;
                printf ("%3d\t%4d\n",zaklad,mocnina);
                zaklad = zaklad + krok;
        }
}
```

Znak _ se na TNS zadává současným stisknutím kláves [SPEC] ["mezera"] na velkých písmenech.

První řádek programu

```
/* tisk tabulky druhých mocnin pro základ = 0, 1, ..., 10 */
```

je 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 dolni_mez, horni_mez, krok;
        int zaklad, mocnina;
```

Typ i n t znamená, že všechny uvedené proměnné budou celočíselné . Na počítači TNS je int 16-ti bitové číslo se znaménkem a leží v rozsahu -32768 až +32767.

Typ c h a r znamená, že se jedná o znak, ovšem často se char používá jako 8-bitové číslo bez znaménka v rozsahu 0 - 255.

V kapitole 2 jsou uvedeny ostatní typy. Základní typy jsou rovněž pole, struktury, uniony, ukazatele na ně, a funkce, které s nimi pracují. Se všemi těmito typy se postupně v textu shledáme.

Skutečný výpočet v programu pro výpočet tabulky druhých mocnin začínáme přiřazením

```
dolni_mez = 0;
horni_mez = 10;
krok = 1;
zaklad = dolni_mez;
```

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 w h i l e :

```
while (zaklad <= horni_mez)
{
        ...
}
```

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

Tělo příkazu w h i l e 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, jsou odsazeny 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 nezáleží příliš na pozici příkazu v textu, je vhodné zdůrazňovat logickou strukturu a užívat mezery, abychom zdůraznili členění programu. Doporučujeme psát pouze jeden příkaz na řádek a nechávat mezery okolo operátorů. 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 činnosti našeho programu je vykonána v těle smyčky while. Druhá mocnina zakladu je vypočítána a přiřazena proměnné mocnina příkazem

```
mocnina = zaklad * zaklad;
```

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 ("%3d\t%4d\n",zaklad,mocnina);
```

specifikace %3d znamená, že číslo typu int má být zobrazeno s délkou tří znaků; %4d určuje, že další číslo bude v délce čtyři znaky. Tato specifikace odpovídá formátové specifikaci I3 a I4 v jazyku FORTRAN. \t znamená, že tisk dalšího čísla bude odsazen o tabelátor a \n znamená, že další tisk bude proveden na nový řádek.

Printf rozpoznává %d pro dekadické 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 dává program nesmyslné výsledky.

Mimochodem, funkce p r i n t f 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 s c a n f v kap. 7, odstavec 7.4. Scanf je funkce obdobná jako printf, až na to, že čte vstup místo psaní výstupu.

C v i č e n í 1 - 3:

Upravte program pro výpočet druhých mocnin tak, aby vytiskl záhlaví tabulky.

C v i č e n í 1 - 4:

Napište program, který bude tisknout tabulku druhých mocnin v obráceném pořadí (od 10 do 0).

1.3. Příkaz for

Jak je možno očekávat, je mnoho způsobů, jak napsat program; uvedeme jinou variantu našeho programu pro výpočet druhých mocnin:

```
main ()         /* tabulka výpočtu druhých mocnin */

{
        int zaklad;

        for (zaklad = 0; zaklad <= 10; zaklad = zaklad + 1)
                printf("%3d\t%5d\n", zaklad, zaklad * zaklad);
}
```

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

Poslední změna je příkladem obecného pravidla jazyka C; v kterémkoliv místě, kde se může vyskytnout proměnná, je možné použít výraz stejného typu. Protože argumenty funkce printf mají být typu int, aby je bylo možno zobrazit konverzí %d, tak na jejich pozici se může použít i proměnná typu char.

Příkaz f o r je podmíněný příkaz, který je zobecněním příkazu w h i l e. Jestliže jej porovnáme s příkazem while, tak jeho činnost by nám měla být jasná. Sestává ze tří částí, které jsou od sebe odděleny středníky.

První část, inicializace

```
zaklad = 0;
```

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

```
zaklad <= 10;
```

Podmínka je vyhodnocena a 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:

```
zaklad = zaklad + 1
```

a znovu je vyhodnocena podmínka v druhé časti 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 v tomto případě kompaktnější, protože příkazy pro řízení cyklu jsou zde pohromadě.

C v i č e n í 1 - 5:

Modifikujte program pro výpočet druhých mocnin tak, aby ve třetím sloupci tiskl poloviční hodnotu druhých mocnin. Operátor dělení (/) znamená celočíselné dělení, takže případné desetinné části čísel budou automaticky odřezány, proto je nemusíte uvažovat.

1.4. Symbolické konstanty

Předtím, než opustíme náš program pro výpočet druhých mocnin, učiňme ještě závěrečnou poznámku. Je špatné "pohřbívat magická čísla" jako jsou čísla 10 a 1 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

d e f i n e

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 DOLNI_MEZ       0       /* dolní mez tabulky     */
#define HORNI_MEZ       10      /* horní mez tabulky     */
#define KROK            1       /* hodnota kroku         */

main ()                         /* tabulka výpočtu druhých mocnin */

{
        char zaklad;

        for (zaklad = DOLNI_MEZ; zaklad <= HORNI_MEZ;
                                        zaklad = zaklad + KROK)
                printf ("%3d\t%5d\n",zaklad,zaklad * zaklad);
}
```

Položky DOLNI_MEZ, HORNI_MEZ, KROK jsou konstanty, proto se neobjevují v deklaracích. Symbolická jména je vhodné psát velkými písmeny, aby je bylo možno jednoduše odlišit od názvu proměnných, která jsou psána 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

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

Vstup a výstup znaků

Standardní knihovna obsahuje funkce pro čtení a vypsání znaků. Funkce g e t c h a r 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 getchar je na TNS implementována tak, že navíc zobrazí načtený znak. Pokud chceme tuto závadu odstranit, připojíme ke každému zdrojovému textu novou funkci getchar, ktera bude mít tvar :

```
getchar ()

{
        register char c ;

        while (!(c = bdos(6,0xFF)))
                return c;
}
```

Význam jednotlivých příkazů nechme zatím stranou. Nyní už máme i na TNS zaručeno, že popisované příklady budou dělat to, co mají.

Funkce p u t c h a r je doplňkem funkce getchar, takže

```
putchar (c);
```

vytiskne obsah proměnné c na nějaké medium - obyčejně na obrazovku. Volání funkce putchar a printf můžeme kombinovat; výstup se objeví v tomto 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.

Kopírování souborů

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 schema 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, že může existovat více konvencí pro indikaci konce souboru. My jsme se této nepříjemnosti vyhnuli tím, že používáme symbolické jméno EOF pro hodnotu "konec souboru", ať už je jakákoliv. Na TNS je EOF rovno 26 (1A hex), a proto musí být na začátku programu definice

```
#define EOF     26
```

Pozor! Pri použiti knihovní funkce getchar na TNS musí být na začátku definice

```
#define EOF     -1
```

protože funkce getchar přiřadí symbolu EOF (z klávesnice ho zadáme [SPEC] [Z]) hodnotu -1.

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

Současně musíme deklarovat proměnnou c typu int a ne c h a r, aby mohla obsahovat hodnotu, jakou 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, která by mohla být i -1, což je číslo int.

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

```
c = getchar ();
```

použít 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 ve 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 to, že proměnné c bude přiřazena logická hodnota 0 nebo 1 podle toho, zda funkce getchar narazila na konec souboru nebo ne (více o této problematice bude uvedeno v kapitole 2).

V operačním systému CP/M není vstup z klávesnice soubor, proto si textovým editorem vytvoříme soubor UCIMSECJ.C, který nám bude nahrazovat vstup z klávesnice. Může obsahovat různé věty, dlouhé řádky, komentáře apod. Nyní uvedeme způsob, který umožní uživateli TNS čtení ze souboru.

Předpokládejme, že na disketě už máme soubor UCIMSECJ.C a že chceme číst z tohoto souboru. Nejprve musíme tento soubor zpřístupnit (tj. otevřít). To provedeme funkcí fopen. Samotné čtení provádí funkce fgetc, která vrací hodnotu -1, jestliže soubor nebyl otevřen nebo bylo dosaženo konce souboru, jinak funguje stejně jako funkce getchar. Po skončení čtení je nutné soubor uzavřít funkcí fclose. Detailnější popis funkcí fopen, fgetc a fclose necháme zatím stranou. Ještě uvedeme program na kopírování souboru UCIMSECJ.C na výstup, na kterém budeme demonstrovat použití funkcí fopen, fgetc a fclose.

```
#define ERROR (-1)        /* chyba */
#define BUF    128        /* velikost bufferu */

main ()

{
        unsigned pop_souboru;
        int c;

        pop_souboru = fopen("UCIMSECJ.C","r",BUF);
        while ((c = fgetc(pop_souboru)) != ERROR )
              putchar(c);
        fclose(pop_souboru);
}
```

Počítání znaků

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

```
main ()                    /* počítání znaků na vstupu */

{
        unsigned pocet_znaku;

        pocet_znaku = 0;
        while (getchar () != EOF)
              ++ pocet_znaku;
        printf ("%d\n",pocet_znaku);
}
```

Příkazem

```
++ pocet_znaku
```

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

Program pro počítání znaků sčítá znaky v proměnné pocet_znaku, která je typu u n s i g n e d (celočíselná proměnná bez znaménka s hodnotou 0 až 65535). Výstupní konverze %d umožňuje i tisk proměnných deklarovaných jako unsigned.

Také použijeme příkaz for místo příkazu while, abychom ukázali jiný způsob tvorby cyklu.

```
main ()                    /* počítání znaků na vstupu */

{
        unsigned pocet_znaku;

        for (pocet_znaku = 0;getchar () != EOF; ++ pocet_znaku)
                ;
        printf ("%d\n",pocet_znaku);
}
```

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 raději píšeme na samostatnou řádku, abychom si ho snadněji všimli.

Předtím, než program pro počítání znaků opustíme, všimněme 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). Program počítá správně, i když vstup nenabízí "žádné znaky".

Počítání řádků

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

```
main ()                 /* počítání řádků */

{
        int c, pocet_radku;

        pocet_radku = 0;
        while ((c = getchar ()) != EOF)
                if (c == '\n')
                        ++ pocet_radku;
        printf("%d\n",pocet_radku);
}
```

Na TNS při zadávání dat z klávesnice místo \n používáme klávesu [VEZMI], místo \t [SPEC] [I], místo \b [SPEC] [H].

Tělo cyklu while nyní obsahuje příkaz if, který řídí přírůstek ++ pocet_radku. 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 nový řádek. Uvědomte si, že '\n' je jeden znak a ve výrazu je roven číslu typu int a na druhé straně, "\n" je řetězec znaků, který obsahuje jeden znak. Tato problematika je podrobne popsána v kap. 2.

C v i č e n í 1 - 6:

Napište program, který počítá mezery, tabelátory a znaky pro nový řádek.

C v i č e n í 1 - 7:

Napište program, který kopíruje text ze vstupu na výstup a nahrazuje jednu nebo více mezer vždy jen jednou mezerou.

C v i č e n í 1 - 8:

Napište program, který nahrazuje každý tabelátor dvojicí znaků: > a -, a každý zpětný znak ('\b') obdobnou dvojicí: < a -. To nám zviditelní znaky pro tabelátor a backspace. Program vyzkoušejte na nějakém cvičném souboru.

Počítání slov

Čtvrtý ze serie užitečných programů počítá řádky, slova a znaky. Slovo je zde definováno jako skupina znaků, která neobsahuje mezeru, tabelátor nebo znak pro nový řádek.

Zde je program:

```
#define ANO     1
#define NE      0

main ()     /* zjištění počtu řádků, slov, znaků ze vstupu */

{
        int c, poc_radku, poc_slov, poc_znaku, v_slove;

        v_slove = NE;
        poc_radku = poc_slov = poc_znaku = 0;
        while ((c = getchar ()) != EOF)
        {
                ++ poc_znaku;
                if (c == '\n')
                        ++ poc_radku;
                if (c == ' ' || c == '\n' || c == '\t')
                        v_slove = NE;
                else    if (v_slove == NE)
                        {
                                v_slove = ANO;
                                ++ poc_slov;
                        }
        }
        printf ("%d\t%d\t%d\n",poc_radku,poc_slov,poc_znaku);
}
```

Znak | zadáváme na TNS současným stisknutím kláves [SPEC] [\] na malých písmenech.

Pokaždé, když program narazí na první znak slova, započítá slovo. Proměnná v_slove indikuje, zda je program právě uprostřed slova nebo ne; na začátku "není uprostřed slova", což je vyjádřeno hodnotou NE. Dáváme přednost symbolickým konstantám ANO a NE před hodnotami 1 a 0, protože je program srozumitelnější.

Ovšem, v tak malém programu, jako je náš, to přináší spíše potíže, ale potom ve větších programech se nám 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.

Řádek

```
poc_radku = poc_slov = poc_znaku = 0;
```

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

```
poc_radku = ( poc_slov = ( poc_znaku = 0 ));
```

Operátor || znamená nebo (logický součet) a tak řádek

```
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 odleva 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 nový řádek. V krátkém programu to není důležité, ale na významu to nabývá v komplikovanějších situacích, jak brzy uvidíme.

Náš program také ukazuje činnost příkazu e l s e, jenž určuje alternativní činnost, která má být konána, když podmínka příkazu if není splněna. Obecný tvar je

```
if (podminka)
        prikaz_1
else
        prikaz_2
```

Za této situace bude vykonán právě jeden ze dvou příkazů spojených s příkazem if. Jestliže je podmínka pravdivá, bude vykonán prikaz_1; jestliže ne, bude vykonán prikaz_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.

C v i č e n í 1 - 9:

Jak budete testovat program pro počítání slov? Jaká jsou omezení?

C v i č e n í 1 - 10:

Napište program, který tiskne slova ze vstupu vždy na novou řádku.

C v i č e n í 1 - 11:

Upravte definici slova v programu pro počítání slov. Např. slovo je posloupnost písmen a číslic, která začíná písmenem. Napište program, kde použijete novou definici slova. Srovnejte výsledky programu pro různé definice slova.

1.6. Pole

Napišme program, který bude zjišťovat počet výskytů každého z čísel, oddělovačů (tj. mezer, tabelátoru a znaku nového řádku) 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, oddel, ostat;
        int poccis [10];

        oddel = ostat = 0;
        for (i = 0; i < 10; ++ i)
                poccis [i] = 0;
        while ((c = getchar ()) != EOF)
                if (c >= '0' && c <= '9')
                        ++ poccis [c - '0'];
                else    if(c == ' ' || c == '\n' || c == '\t')
                                ++ oddel;
                        else
                                ++ ostat;
        printf ("pocet cislic = ");
        for (i = 0; i < 10; ++ i)
                printf ("%d, ",poccis [i]);
        printf ( "oddelovace = %d, ostatni = %d",oddel,ostat);
}
```

Popis

```
int poccis [10];
```

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

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 poccis.

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

```
if (c >= '0' && c <= '9')
        ++ poccis [c - '0'];
else    if (c == ' ' || c == '\n' || c == '\t')
                ++ oddel;
        else
                ++ ostat;
```

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 příkazů 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 příkazů).

Příkaz s w i t c h, který bude popsán v kapitole 3, umožňuje další způsob mnohonásobného rozhodování. Je vhodný pro zjišťování, z kterého souboru je dané číslo nebo znakový výraz.

C v i č e n í 1 - 12:

Napište program, který bude tisknout histogram délek načtených slov. Nejsnadnější je nakreslit histogram horizontálně, vertikální uspořádání je složitější. Nakonec program vypíše tabulku, např.:

Delka slova	Pocet slov
1	2
2	0
3	8
...	...
9	1
10	0

Je možné omezit délku slova např. na 10 znaků a všechna delší slova počítat dohromady.

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 jen jednou volaná. 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 jazyk C neobsahuje operátor mocnění (jako je ** ve FORTRANU nebo PL/1), vysvětlíme si způsob vytvoření funkce a napíšeme funkci p o w e r (m,n), která umocňuje celé číslo m na celé číslo n. Např. hodnota power(2,5) je 32. Tato funkce nenahrazuje zcela operátor **, protože pracuje pouze s malými celými čísly, ale je vhodné začínat jednoduchou ukázkou.

Následuje funkce power a hlavní program main, který ji ověří, 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 i ve více souborech. Ovšem když je text programu ve více 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

Při každém vyvolání funkce power jsou jí předávány dva argumenty a návratovou hodnotou funkce je celé číslo a to 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 řádku

```
int x, n;
```

který následuje za řádkem se jménem funkce. Deklarace argumentů je umisteěna mezi seznam argumentů a první složenou závorku; každá deklarace je zakončena středníkem.

Jména, která se pro argumenty používaji 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 (bez závorek !) předává pouze řízení volající jednotce a nepředává hodnotu. Ostatně závorky nejsou důležité a nemusí se v příkazu return uvádět.

C v i č e n í 1 - 13:

Napište program, který převádí velká písmena ze vstupu na malá písmena a používá k tomu existující funkci tolower. Funkce tolower vrací c, když c není písmeno, a hodnotu malého písmene c, jestliže c je písmeno. Vyvolá se příkazem

```
tolower(c);
```

1.8, Argumenty - volání hodnotou

Jedna z vlastností jazyka C může být některým programátorům, kteří používají FORTRAN a 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 proměnnými.

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 volaná.

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

Je-li jako argument uvedeno jméno pole, potom 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 nejdelší. Hrubý diagram je velmi jednoduchý:

```
while   (existuje další řádek)
        if (je delší než předchozí nejdelší)
                uchovej ho a jeho délku
vytiskni nejdelší řádek
```

Tento diagram objasňuje činnost programu a umožňuje nám rozdělit ho na jednotlivé části. Jedna část načítá řádek, druhá část ho 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 řádek; getline je zobecnění funkce getchar. Aby byla funkce getline užitečná i v dalších programech, napišme ji tak univerzálně, jak jen to půjde. Přinejmen-

ším musí funkce getline signalizovat konec souboru; dalším zobecněním bude, aby její hodnota byla rovna délce načteného řádku, nebo nule, byl-li načten konec souboru. Každý řádek má minimálně jeden znak a tak nula není platná hodnota pro délku řádku ani tehdy, obsahuje-li řádek jen znak pro nový řádek \n (takový řádek má délku 1).

Když narazíme na řádek, který je delší než předcházející nejdelší řádek, tak ho musíme někde uchovat. To bude vykonávat druhá funkce, c o p y, která bude nejdelší řádek kopírovat na bezpečné místo. A na závěr potřebujeme hlavní program, který bude činnost funkcí getline a copy řídit. Zde je výsledný program:

```
#define MAXLINE 1000    /* max. velikost vstupujícího řádku */

main ()                 /* nalezení nejdelšího řádku */

{
        int delka;              /* délka běžného řádku      */
        int max;                /* max. dosažená velikost   */
        char line [MAXLINE];    /* běžný řádek              */
        char save [MAXLINE];    /* nejdelší uchovaný řádek */

        max = 0;
        while ((delka = getline (line,MAXLINE)) > 0)
                if (delka > max)
                {
                        max = delka;
                        copy (line,save);
                }
        if (max>0)              /* byl načten min. 1 řádek */
                printf("%s",save);
}


getline (s,lim)         /* načti řádek 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 hodoty. 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 specifikovaná 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 námi požadované hodnoty, jiné, jako např. copy, jsou užívány pouze pro to, co dělají a hodnotu nevracejí.

getline vklada znak \0 (NULL)

(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

```
"ahoj\n"
```

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

```
a h o j \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 řádek, který je delší než zvolené maximum ? Funkce getline pracuje správně. Přestane číst znaky, je-li pole plné, i když nebyl načten znak nového řádku. Kontrolou délky řádku a testováním posledního znaku může main určit, zda řádek nebyl příliš dlouhý 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.

C v i č e n í 1 - 14:

Upravte hlavní program main v programu pro tisk nejdelšího řádku tak, aby správně vytiskl délku libovolně dlouhého vstupního řádku a co nejvíce jeho obsahu.

C v i č e n í 1 - 15:

Napište program, který bude tisknout všechny řádky delší než 80 znaků.

C v i č e n í 1 - 16:

Napište program, který vyjme úvodní mezery a tabelátory z každého řádku na vstupu a vymaže prázdné řádky.

C v i č e n í 1 - 17:

Napište funkci reverse, která převrací znakový řetězec s (znaky v řetězci přeskládá naopak). Použijte ji v programu, který převrací vstupní řádky. Funkcí reverse předávejte jako jeden z argumentů skutečnou délku řádků, o které předpokládejte, že není větší než pevně zvolené maximum.

1.10. Externí proměnné

Proměnné v programu main (line, save, atd.) jsou soukromé, místní jednotky main, protože jsou v jednotce main 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 aktivovaná 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 proto 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 nejdelšího řádku s proměnnými line, save a max definovanými jako externí proměnné. To vyžaduje změnit volání funkce, změnit deklarace a upravit všechny funkce. Zde je uveden upravený program:

```
#define MAXLINE 1000     /* max. délka vstupní věty       */

int     max;             /* předposl. max. délka řádku    */

main ()          /* nalezení nejdelšího řádku; zvláštní verze */

{
        int delka;
        extern int max;
        extern char save [];

        max = 0;
        while ((delka = getline ()) > 0)
                if (delka > max)
                {
                        max = delka;
                        copy ();
                }
        if (max > 0)             /* byla načtena aspoň 1 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;
}

char    line [MAXLINE]; /* vstupní řádek                */
char    save [MAXLINE]; /* nejdelší řádek uchován zde   */
```

Při kompilaci tohoto programu se objeví následující chybové hlášení :

```
char line [1000];
     Array or pointer being lengthened
char save [1000];
     Array or pointer being lengthened
```

Tato hlášení v tomto případě značí, že bude zvětšeno přidělené místo v paměti proměnným line a save. Pokud nebude v programu jiná chyba, bude tento program fungovat.

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 je tedy deklarace extern int max v jednotce main nadbytečná. 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 soub1 a používána v souboru soub2, potom je deklarace extern v soub2 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 přidělena přitom 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 plný nebezpečí, protože vede k programům, kde je vzájemná komunikace nejasná. Za prvé proměnné mohou být neočekávané 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.

C v i č e n í 1 - 18:

Podmínka v příkazu for ve funkci getline je poněkud neohrabaná. Přepište program tak, aby byla jasnější a přitom zachovejte 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. Jistě bude vhodné, když si je procvičíte před dalším studiem. Následující příklady by vám měly dát náměty pro složitější programy, než jsou popsány v této kapitole.

C v i č e n í 1 - 19:

Napište program detab, který nahrazuje tabelátory ze vstupů odpovídajícím počtem mezer. Předpokládejte, že tabelační pozice jsou 1, n+1, 2n+1,... .

C v i č e n í 1 - 20:

Napište program entab, který nahrazuje řetězec mezer minimálním počtem tabelátorů a mezer. Užívejte stejné pozice tabelátorů jako ve funkci detab.

C v i č e n í 1 - 21:

Napište program, který rozděluje dlouhé řádky ze vstupů po posledním nemezerovém znaku, který se objeví před n-tým znakem vstupu, kde n je parametr. Pokud bude některé slovo delší než je n, tak program ukončí svoji činnost a ohlásí chybu.

C v i č e n í 1 - 22:

Napište program, který vypíše program v jazyce C bez všech komentářů. Neopomeňte správně používat znakové řetězce a znakové konstanty.

C v i č e n í 1 - 23:

Napište program, který kontroluje počet základních symbolů programu, to je neuzavřené kulaté, hranaté a složené závorky, nezapomeňte na apostrofy, uvozovky a komentáře. Tento program je složitý, jestliže jej plně zobecníte. Závorky v apostrofech, uvozovkách a komentářích ignorujte.

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 "_" (podčárka) 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 šest znaků jmen je významných, ačkoliv jich může být použito více. Podrobnosti jsou v dodatku A.

Klíčová slova, jako jsou if, else, int, double, 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 zaměnit navzájem.

2.2. Typy dat a jejich délka

V jazyku C je jen několik základních typů dat:

char	délka 1 byte, může obsahovat jeden znak (kód 00 - FF hex)
int	celočíselná proměnná se znaménkem délka 2 byte (hodnota -32768 až 32767)
unsigned	celočíselna proměnná bez znaménka délka 2 byte (hodnota 0 až 65535)

Dále existují speciální typy dat (používají se zřídka, na TNS v podstatě nemají význam):

short	celočíselná hodnota délka 1 byte (hodnota 0 až 255)
long	celočíselná hodnota délka 4 byte
float	číslo s pohyblivou desetinnou čárkou a jednoduchou přesností délka 4 byte
double	číslo s pohyblivou desetinnou čárkou a dvojnásobnou přesností délka 8 byte

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 se v takových případech neuvádí. Přesnost těchto typů proměnných záleží na konkrétním typu počítače; int obyčejně představuje "přirozenou" délku pro daný typ počítače. 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 jsme již používali, takže upozorněme na nezvyklé tvary, s kterými se můžete setkat např. ve Fortranu

```
123.456e-7
```

nebo

```
0.12E3
```

Tento tvar konstant se nazývá "vědecká" notace a současná verze kompilátoru takové konstanty nedovoluje používat. Také je třeba se vyvarovat používání obyčejných celočíselných konstant, které jsou delší než int (char); budou redukovány na typ int (char) a to je zpravidla chyba.

Existuje také možnost zápisu oktalových a hexadecimálních konstant: je-li na první pozici konstanty znak \ (zpětné lomítko), tak je konstanta brána jako oktalová; jsou-li na začátku znaky 0x nebo 0X, tak se jedná o hexadecimální konstantu.

Například dekadické číslo 31 může být psáno oktalově jako \037 nebo \37 a hexadecimálně jako 0x1f nebo 0X1F.

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, učiníme tak program nezávislým na konkrétní reprezentaci. Znaková konstanta se může zúčastnit číselných operací jako ostatní čísla, přestože se používa 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      tabelátoru
\0      prázdny znak (NULL) - jeho hodnota je nulová
\\      obrácené lomítko
\'      apostrof
```

atd,, která vypadá jako dva znaky, ale ve skutečnosti to je jen jeden znak. Navíc může být libovolný vzor bitů v byte 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. '\0' píšeme často 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 (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

```
sekundy = 60 * 60 * hodiny;
```

Znaková konstanta je sekvence několika znaků jako např.

```
"Ja jsem retezec"
""                          /* prázdný řetězec */
```

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

```
"Ja jsem retezec \" a toto je uvozovka uvnitr"
```

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. Deklarace specifikuje typ a za ním následuje seznam jedné nebo více proměnných tohoto typu, jako např.:

```
int dolni_mez, horni_mez, krok;
char c, radek [1000];
```

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

```
int dolni_mez;
int horni_mez;
int krok;
char c;
char radek [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.

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

Operátory + a - mají stejnou prioritu, která je nižší než priorita operátorů *, / a %. Unární mínus 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 operátory logického součinu a logického součtu:

```
&&   ||
```

Výrazy s těmito operátory jsou vyhodnocovány zleva doprava a vyhodnocování skončí tehdy, je-li zřejmé, že výraz je pravdivý nebo nepravdivý. Tuto vlastnost je třeba si 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 m u s í být vykonán dříve než načteme další znak. A ne jenom to. Když totiž není tato podmínka splněna, n e s m í m e 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 operátory 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 (pravdivé operátory) na 0, a nulové (nepravdivé operátory) na 1. Rozšířeným použitím operátoru ! je výraz jako

```
if (! je_slovo)
```

spíše než

```
if (je_slovo == 0)
```

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

C v i č e n í 2 - 1:

Napište příkaz cyklu ekvivalentní cyklu for v předchozí ukázce bez užití &&.

2.7. Konverze typu

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. 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 konvertová-

na na int. To umožňuje dostatečnou flexibilitu v určitých druzích znakové transformace. To je užito ve funkci a t o i , 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 t o l o w e r, která převádí velká písmena na malá (platí pouze pro ASCII !). Jestliže znak není velké písmeno, vrací jej funkce tolower nezměněn.

```
tolower (c)       /* prevod c na male pismeno, jen pro ASCII */

        int c;

{
        if (c >= 'A' && c <= 'Z')
                return c + 'a' - 'A';
        else
                return c;
}
```

Tato fukce může pracovat pouze se znakovým souborem ASCII, protože velká a malá písmena mají od sebe konstantní vzdálenost a číselné hodnoty obou abeced jsou ve vzestupném pořadí a mezi 'A' a 'Z' není nic jiného než písmena.

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 vždy k l a d n ý (vyšší byte bude vynulován). To však nemusí platit u jiných počítačů.

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, a EOF záporné. Výsledkem je to, že test c == EOF nemůže být nikdy pravdivý. Abychom se tomuto problému vyhnuli, musíme 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 přirazena, n e s m í 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:

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

nastavuje proměnnou je_cislo na 1, je-li c číslicí, a na 0, není-li číslicí. (V podmínkách příkazu 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 (short) jsou konvertovány na int,

- jestliže je nějaký operand unsigned, ostatní jsou konvertovány na unsigned a výsledek je unsigned.

- jinak operandy musí být int a výsledek je int.

Konverze se rovněž uplatňují při přiřazování; hodnota pravé strany je konvertována na typ proměnné na levé straně, což je typem výsledku. Znak je konvertován na celé číslo. 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 nezměněna.

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

Na závěr budiž řečeno, že konverze lze provádět rovněž explicitně způsobem zvaným v l a s t n o s t . V konstrukci

```
(nazev typu) vyraz
```

je v y r a z konvertován na vyjmenovaný typ podle výše uvedených pravidel. Přesný význam v l a s t n o s t i je v tom, že hodnota výrazu je přiřazena proměnné specifikovaného typu a ta je potom použita místo celého výrazu. Např. bude-li argument funkce s q r t proměnná typu double, pak, 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 funkcí sqrt (uvědomte si, že vlastnost vytváří h o d n o t u n spravného 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.

C v i č e n í 2 - 2:

Napište funkci htoi(s), která hexadecimální řetězec dlouhý maximálně 4 znaky převede na jeho desítkový ekvivalent.

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 p ř e d použitím jeho hodnoty a n ++ zvětšuje n p o t o m, co jeho hodnota byla použita. To znamená, že v kontextu, kde má být použita hodnota proměnné, konstrukce ++ n a n ++ jsou odlišné. Jestliže n je 5 potom po

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

je x také 5, zatímco po

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

je x 6.

Operátory ++ a -- mohou byt aplikovány pouze na proměnné. V případech, že nepotřebujeme ve výrazech používat hodnotu proměnné, jako v

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

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

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

```
squeeze (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 pismenem c je kop...ován 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.

C v i č e n í 2 - 3:

Napište modifikaci funkce squeeze (s1,s2), která zruší všechny znaky v s1, které jsou shodné se znaky v řetězci s2.

C v i č e n í 2 - 4:

Napište funkci any (s1,s2), která vrátí pozici prvního výskytu řetězce s2 v řetězci s1 anebo -1, jestliže s1 neobsahuje s2.

2.9. Bitové logické operátory

V jazyku C jsou k dispozici operátory pro manipulaci s bity. Tyto operátory mohou být aplikovány pouze na proměnné int nebo char.

&	bitové AND
!	bitové OR
^	bitové exkluzivní OR
<<	bitový posun doleva
>>	bitový posun doprava
~	bitový jedničkový doplněk

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

```
c = n & 0x7f;
```

vynuluje všechny bity proměnné n kromě nejnižších 7 bitů.

Bitové OR (!) je používáno pro nastavení bitů

```
x = x ! MASK;
```

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

Bitové exkluzivní OR (^) (^ se na TNS zadává klávesou ["obdélník"] v levém horním rohu klávesnice pod [SPEC]) představuje ve skutečnosti "negaci ekvivalence".

S opatrností rozlišujte bitové operátory & a | od logických operátorů && a ||, které provádějí vyhodnocování pravdivosti odleva doprava. Např. jestliže

```
x = 1   a   y = 2,
```

potom x & y je nula, zatímco x && y je jedna (proc ?).

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 proměnné doprava vyplní prázdná místa vlevo nulami.

Výsledkem unární operace operátorem ~ (~ se na TNS zadáva [SPEC] ["obdélník"]) je doplněk celého čísla; to znamená, že převádí bity 0 na 1 a naopak. Tyto operátory se obvykle používají ve výrazech jako

```
x & ~ 0x3f
```

který nuluje posledních šest bitů. Uvědomte si, že operace x & ~ 0x3f je nezávisla na délce slova a proto je lepší ji používat místo operace x & 0xffc0, která předpokládá, že x je šestnáctibitové. Výraz ~ 0x3f nestojí při výpočtu žádný čas navíc, protože je jako konstantní vyčíslován již při překladu.

Abychom si ilustrovali použití některých bitových operátorů, napíšeme funkci g e t b i t s (x, p, n), která vrací pole n-bitů proměnné x, které začínají 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 bitu od pozice p */

        unsigned x, p, n;

{
        return ((x >> (p + 1 - n)) & ~ (~0 << n));
}
```

Vyraz 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. ~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 rovno 1.

C v i č e n í 2 - 5:

Modifikujte getbits pro číslování bitů odleva doprava.

C v i č e n í 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.

C v i č e n í 2 - 7:

Napište funkci rightrot (n,b), která rotuje celé číslo n o b bitů doprava.

C v i č e n í 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 zhusteným způsobem

```
i += 2;
```

použitím p ř i ř a z o v a c í h o o p e r á t o r u 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 fukci 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 & 1)
                        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 (+=, -=, atd.) se rovněž může objevit ve výrazech, přestože je to méně obvyklé. Typ operátoru přiřazení je shodný s typem na levé straně.

C v i č e n í 2 - 9:

U systému s dvojkovým doplňkem výraz x & (x - 1) nuluje bit s hodnotou 1 proměnné x, který je nejvíce vpravo (proč ?). Použijte této skutečnosti a napište rychlejší verzi funkce bitcount.

2.11. Podmíněné výrazy

Příkaz

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

počítá maximum z čísel a a b.

P o d m í n ě n ý v ý r a z psaný s operátorem "?:", umožňuje jiným způsobem napsat podobnou konstrukci. Ve výrazu

```
e1 ? e2 : e3
```

je výraz e1 spočítán nejdříve. Jestliže je nenulový (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 max z a a b můžeme napsat

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

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

```
(c > 0) ? n : c;
```

je typem int. 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.následující cyklus tiskne N prvků pole, 10 na řádku oddělené jednou mezerou a každý řádek (i poslední) je ukončen přesně jedním znakem pro nový řádek.

```
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.

C v i č e n í 2 - 10:.

Přepište funkci tolower, která konvertuje velká písmena na malá s užitím podmíněného výrazu místo if - else.

2.12. Priorita a pořadí vyhodnocování

Tabulka uvedená níže shrnuje pravidla priorit a asociativnosti všech operátorů včetně těch, se kterými jsme se ještě neseznámili. Operátory na jednom řádku mají stejnou prioritu; řádky jsou seřazeny sestupně, tak např. *, / a % mají stejnou prioritu vyšší než + a -.

Tabulka priorit operátorů a směru vyhodnocování

Operátor	Asociativnost
() [] -> .	zleva doprava
! - (unarni) ++ -- ~ (typ) * & sizeof	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 -> 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 &, ^ 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 (*, +, &, ^, |) 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 operátorů vyčísleny. Např. v příkazu

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

může být f vyčíslena dříve než g nebo také naopak; proto jestliže 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

```
printf ("%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 volěním funkce power nebo ne. Správný zápis vypadá takto:

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

Volání funkcí s 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 index je 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 programu vykonány. Už jsme se setkali s nejobvyklejšími příkazy pro větvení programu v dřívějších příkazech, zde uvedeme kompletní sestavu těchto příkazů a upřesníme předchozí.

3.1. Příkazy a bloky

V ý r a z jako je x = 0 nebo i ++ nebo printf (...) se stává p ř í k a z e m, je-li následován středníkem, jako v

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

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

Složené závorky { a } se používají pro sdružování deklarací a příkazů ve s l o ž e n ý p ř í k a z neboli b l o k, 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 l i b o v o l n é h o 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)
        p ř í k a z - 1
else
        p ř í k a z - 2
```

kde část else není povinná. V ý r a z je vyhodnocen, jestliže je "pravdivý" (tj. má nenulovou hodnotu), p ř í k a z - 1 je vykonán, jestliže je nepravdivý (tj. má nulovou hodnotu) a jestliže je uvedeno else, je vykonán p ř í k a z - 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 příkazů if vzniká nejednoznačnost. Else je logicky spojeno s nejbližším předchozím výrazem if, který nemá část else. Např.

```
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 sdružení příkazů if-else v konkrétních případech 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("chyba - n je nula\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 (v y r a z)
        p r i k a z
else    if(v y r a z)
                p r i k a z
else    if(v y r a z)
                p r i k a z
else
        p r i k a z
```

se objevují tak často, že stojí zato se podrobněji o nich zmínit. Sekvence příkazů if je nejobecnějším způsobem mnohonásobného rozhodování. V ý r a z y jsou vyhodnocovány v tomto pořadí: jestliže je v ý r a z pravdivý, tak příkaz s ním spojený je vykonán, a tím je celá sekvence ukončena.

P ř í k a z znamená buď jeden příkaz, nebo skupinu příkazů v závorkách. Příkaz spojený s posledním else je vykonán, jestliže žádná z předchozích podmínkek nebyla splněna.
V některých případech může být část

```
else
        p ř í k a z
```

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, anebo -1, jestliže není.

```
binary (x,v,n)          /* nalezení x ve v[0]...v[n-1] */

        int x, v [], n;

{
        int dolni_mez, horni_mez, stred;

        dolni_mez = 0;
        horni_mez = n - 1;
        while (dolni_mez <= horni_mez)
        {
                stred = (dolni_mez + horni_mez) / 2;
                if (x < v [stred])
                        horni_mez = stred - 1;
                else    if(x > v [stred])
                                dolni_mez = stred + 1;
                        else            /*hodnota je nalezena*/
                                return (stred);
        }
        return (-1);
}
```

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

3.4. Přepínač

Přepínač je speciální, mnohonásobný rozhodovací příkaz, který testuje, zda se výraz v závorkách rovná některé z k o n s t a n t n í c h 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, oddel, ostat, poccis [10];

        oddel = ostat = 0;
        for(i = 0; i < 10; i ++)
                poccis [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':
                                poccis [c - '0'] ++;
                                break;
                case ' ':
                case '\n':
                case '\t':
                                oddel ++;
                                break;
                default:
                                ostat ++;
                                break;
                }
        printf ("pocet cislic = ");
        for (i = 0; i < i0; i ++)
                printf ("%d, ",poccis [i]);
        printf("oddelovace = %d, ostatni = %d\n",oddel,ostat);
}
```

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 žádný case neodpovídá hodnotě výrazu. Položka default nemusí být uváděna; 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 b r e a k způsobuje okamžité ukončení příkazu switch. case slouží pouze jako návěští, tzn. jakmile jsou vykonány příkazy odpovídající návěští case, program dále p r o c h á z í 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 řídce.

Je dobrým zvykem ukončovat poslední 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.

C v i č e n í 3 - 1:

Napište funkci expand (s,t), která při kopírování řetězce s na t konvertuje znaky jako je znak pro nový řádek a tabelátor do viditelného tvaru \n a \t. Použijte switch.

3.5. Cykly w h i l e a f o r

Již jsme se setkali s příkazy cyklu while a for. Ve

```
while (v ý r a z)
        p ř í k a z
```

je výraz vyhodnocen. Pokud je nenulový, je p ř í k a z vykonán a v ý r a z je znovu vyhodnocen. Cyklus pokračuje do té doby, než je výraz nulový. Potom program pokračuje za příkazem while.

Příkaz

```
for (v ý r a z 1;  v ý r a z 2;  v ý r a z 3)
        p ř í k a z
```

je ekvivalentní

```
v ý r a z 1;
while ( v ý r a z 2 )
{
        p ř í k a z
        v ý r a z 3
}
```

Z gramatického hlediska jsou všechny tři položky příkazu for výrazy. Obyčejně výraz1 a výraz3 jsou přiřazovací příkazy nebo volání funkce a výraz2 je relační výraz. Kterákoliv položka může být z příkazu for vypuštěna, ale středníky musí být uvedeny. Jestliže vypustíme výraz1 a výraz3, proměnná i se přestane měnit. Jestliže vypustíme relační výraz2, 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. v

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

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

Příkaz for ma 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/I. Není to ale úplná analogie, protože limity cyklu for mohou být uvnitř měněny a proměnná i si uchová svoji hodnotu, i když je cyklus for z jakýchkoliv důvodů ukončen. Protože složky cyklu for jsou libovolné výrazy, není cyklus for omezen jen na aritmetické posloupnosti. Nicméně ale není dobrým stylem "vnutit" do příkazu for výrazy, které spolu nemají nic společného.

Jako další příklad uvedeme jinou verzi funkce atoi pro konvertování řetězce na číselný ekvivalent. Tato verze je univerzálnější, počítá totiž s úvodním oddělovačem a znaménkem + a -. 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, znamenko;

        for(i=0;s[i] == ' ' || s[i] == '\n'|| s[i] =='\t';i++)
                ;           /* přeskočení oddělovačů */
        znamenko = 1;
        if (s [i] == '+' || s [i] == '-')       /* znak znam.*/
                znamenko = (s [i ++] == '+') ? 1 : -1;
        for (n = 0; s [i] >= '0' && s [i] <= '9'; i ++)
                n = 10 * n + s [i] - '0';
        return (znamenko * n);
}
```

Výhoda soustředění podmínek cyklu na jedno místo tkví v tom, že je mnohem jasnější, než několik vnořených cyklů.

Následující funkce je třídění typu Shell 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 hodné 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] vzestupné */

        int v [], n;

{
        int krok, i, j, pom_prvek;

        for (krok = n / 2; krok > 0; krok /= 2)
                for (i = krok; i < n; i ++)
                        for (j = i - krok; j >= 0 && v [j] >
                                        v [j + krok]; j -= krok)
                        {
                                pom_prvek = v [j];
                                v [j] = v [j + krok];
                                v [j + krok] = pom_prvek;
                        }
}
```

V příkladu 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 krok, vnitřní cyklus zaměňuje ty prvky, které jsou nesprávně uspořádány. Protože proměnná krok je nakonec zmenšena na jedna, 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 oddělené čárkou je vykonávána zleva doprava, 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; i < 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 n e z a r u č u j í vyhodnocování zleva doprava.

C v i č e n í 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.

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 jazyku C, příkazy do - while, mají rozhodování umístěno na konci p o vykonání příkazu těla cyklu; tělo cyklu je tedy vykonáno minimálně jednou.

Syntaxe vypadá takto

```
do
        p ř í k a z
while (v ý r a z);
```

Příkaz je vykonán a potom je vyhodnocen výraz. Pokud 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, znamenko;
        if ((znamenko = 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 (znamenko < 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 bez ohledu 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.

C v i č e n í 3 - 3:

Naše verze programu itoa při reprezentaci čísel ve dvojkovém doplňku neumí zacházet s největším záporným číslem, tj. n = -(2 ** (délka slova - 1)). Proč? Upravte ji tak, aby tisklo i tuto hodnotu správně a nezávisle na typu počítače.

C v i č e n í 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.

C v i č e n í 3 - 5:

Napište verzi funkce itoa, která má tři argumenty místo dvou. Třetím argum.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 a do a stejně tak z přepínače switch. Příkaz break působí na nejvnitřnější vložený cyklus (nebo switch).

Následující program odstraňuje koncové mezery a tabelátory z každého řádku ze 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'
                                        && line [n] != '\n')
                                break;
                line [n + 1] = '\0';
                printf ("%s\n",line);
        }
}
```

Funkce getline vrací délku řádku. Vnitřní cyklus while začíná na posledním znaku řádku (uvědomte si, že -- n zmenšuje n před použitím jeho hodnoty), prohledává ho odzadu a hledá první znak, který není mezera, tabelátor nebo znak pro nový řádek. Cyklus je přerušen tehdy, je-li takový znak nalezen nebo když n se stane záporným (tzn. celý řádek byl prohledán). Měli byste si ověřit, že program funguje správně, i když řádek 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, neboť testovací podmínka není již tak jasná. Podmínkám, které obsahují směs &&, ||, ! nebo závorek je lepší se vyhnout.

3.8. Continue

Příkaz continue je obdobou příkazu break, ale užívá se ho méně; způsobuje skok na začátek d a l š í i t e r a c e 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; 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.

C v i č e n í 3 - 6:

Napište program, který kopíruje vstup na výstup s tou výjimkou, že kopíruje pouze jeden řádek ze skupiny sousedních totožných řádků.

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 něho. 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, v které je skok goto s tímto návěštím.

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

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

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

```
pozice = 0;,
for(i = 0; i < N && ! pozice; i ++)
        for(j = 0; j < M && ! pozice; j ++)
                pozice = v[i] [j] < 0;
if (pozice)
        /* 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 člení velké výpočetní úlohy na menší a umožňují lidem stavět na tom, co ostatní již udělali - nemusí začínat od začátku. Vhodné funkce mohou často skrýt detaily částí programu, o kterých není třeba nic vědět, a tak vyjasnit celek a usnadnit případné změny v programu.

Jazyk C byl sestaven tak, aby umožnil vytvářet vlastní funkce. Programy v jazyku C sestávají spíše z více malých funkcí 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.

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

4.1. Základy

Pro začátek napíšeme program, který tiskne každý řádek ze vstupu, který má jistou vlastnost nebo obsahuje určitý řetězec znaků. Např. hledejme řádky, které obsahují řetězec "se" v souboru řádků

```
Tak dlouho se chodi
se dzbanem
pro vodu,
az se ucho utrhne.
```

Výsledkem bude:

```
Tak dlouho se chodi
se dzbanem
az se ucho utrhne.
```

Základní strukturu můžeme popsat ve třech částech

```
while (existuje další řádek)
        if (řádek obsahuje daný řetězec)
                vytiskni jej
```

Přestože by jistě bylo možné napsat tento program jako celek, je lépe 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.

"existuje další řádek" je funkce getline, která je popsána v kap.1 a "vytiskni jej" je funkce printf, kterou již pro nás někdo napsal. To znamená, že musíme pouze napsat funkci, která rozhoduje, zda řádek obsahuje daný řetězec znaků. Tento problém můžeme vyřešit tím, že ukradneme ideu z jazyka PL/1: funkce index (s,t) vrací pozici v řetězci s, na které začíná řetězec t, nebo -1, jestliže t není obsaženo v s. Pro počáteční pozici v řetězci s použijeme spíše 0 než 1, protože pole v jazyku C začínají pozicí nula. Budeme-li později 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 tento případ bude řetězec, který hledáme, znakovou konstantou v argumentu funkce index, což není úplně obecný způsob. 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,"se") >= 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 v jazyku C má strukturu

```
        název  (seznam argumentů, jsou-li nějaké)

                deklarace argumentů, jsou-li nějaké

        {
                deklarace a příkazy, jsou-li nějaké
        }
```

Jak je vidět, různé části funkce 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, pokud funkce vrací něco jiného než celočíselnou hodnotu; o tom budeme hovořit v příštím odstavci.

Program je jen souborem definic jednotlivých funkcí. Komunikace mezi funkcemi se uskutečňuje (v tomto případě) pomocí argumentů a hodnot jimi vracenými; 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 se žádná hodnota nevrací. Řízení se volající jednotce vrací rovněž 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.

Nyní uvedeme způsob, jak překládat a sestavovat v jazyku C program, který je obsažen v několika zdrojových souborech. Předpokládejme, že tři funkce jsou ve třech souborech nazvaných main.c, getline.c, index.c. Nejjednodušší postup je každou funkci překládat zvlášť až do přemístitelného tvaru .rel a potom všechny funkce pomocí linkeru L80 spojit do jednoho programu. Jestliže se např. v main.c vyskytne chyba, tak může být později přeložen sám a znovu spojen spolu s dříve preloženými soubory.

C v i č e n í 4 - 1:

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

4.2. Funkce, které nevracejí 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, 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.

Hodnota v

```
return (výraz)
```

je vždy konvertována na typ funkce předtím, než je příkaz return vykonán.

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 vyvolaná.

Jestliže se jako argument funkce objeví identifikátor pole, je předána adresa začátku tohoto pole. Prvky pole nejsou kopírovány - funkce může měnit hodnoty prvků pole. Pole jsou tedy předávána adresou. V kapitole 5 budeme hovořit o použití pointrů, které umožňují funkcím měnit také jednotlivé proměnné.

Ještě je třeba se zmínit o tom, že 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 by počítala maximum libovolného množství argumentů, jako to je u funkcí MAX ve FORTRANU nebo PL/1. Později budeme hovořit o tzv. l i s t funkci.

Bezpečně můžeme nakládat s proměnným množstvím parametrů, pokud volaná funkce nepoužívá ty argumenty, které jí nebyly ve skutečnosti dodány, a pokud 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, jestliže 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 (užívá se nuly), která ukončuje seznam argumentů.

4.4. Externí proměnné

Program v jazyku C je složen z množiny externích objektů, kterými jsou buď proměnné nebo funkce. Přídavné jméno "externí (vnější)" je užito v kontrastu se slovem "interní (vnitřní)", které popisuje argumenty a automatické proměnne uvnitř funkcí. Externí proměnné jsou definovány mimo funkce a jsou potencionálně 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 lze definovat externí proměnné, které jsou dosažitelné pouze z jednoho zdrojového souboru, ale nikoliv obecně. (Pozor! Názvy externích proměnnych nesmí být shodné s názvy registrů - A, B, C, D, E, F, H, I, L, R, AF, BC, DE, FY, HL, IX, IY, PC, SP.)

Protože externí proměnné jsou všeobecně dosažitelné, je možné je používat pro komunikaci mezi funkcemi namísto seznamu argumentů. Libovolné funkci lze zpřístupnit externí proměnnou odkazem na její identifikátor, pokud tento identifikátor byl nějak deklarován.

Jestliže musí být funkcemi sdílen velký počet proměnných, pak 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.

Dalším důvodem pro používání externích proměnných je jejich pole působnosti a životnost. Automatické proměnné jsou vnitřní, interní, dané funkcí; začnou existovat při vyvolání funkce a zmizí, jakmile funkce činnost ukončí. 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 programový kalkulátor, který používá 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 poměrně 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 nejsou složité, ale jakmile k nim přidáme chybová hlášení, jsou již dostatečně dlouhé 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 struktura programu bude 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, čímž jsme se zatím nezabývali, 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 zásobnik a jemu přidružene proměnné budou externí a dosažitelné jen funkcím push - pop.

Naprogramovat to, co jsme v hlavních rysech načrtli, je poměrně snadné. Jednotka main bude velký příkaz switch, který pracuje s operátory a operandy. Jedná se o typičtější příklad pro použití příkazu switch, než jaký byl uveden v kap. 3.

```
#define MAXOP   5               /* maximální délka čísla */
#define CISLO   '0'             /* symbol pro číslo */
#define DLOUHY  '9'             /* symbol pro příliš dlouhý řetě-
                                   zec */
#define CLEAR   '#'             /* symbol pro vymazání celého
                                   zásobníku */

main ()                         /* RPN kalkulátor */
{
        int typ,op2;
        char s[MAXOP+1];

        clear ();
        init ();

        while ((typ = getop(s,MAXOP)) != EOF)
                switch (typ=tolower(typ))
                {
                        case CISLO: push (atoi(s));
                                    break;
                        case '+':   push (pop() + pop());
                                    break;
                        case '*':   push (pop() * pop());
                                    break;
                        case '-':   op2 = pop();
                                    push (pop() - op2);
                                    break;
                        case '/':   op2 = pop();
                                    if (op2 != 0)
                                            push (pop() / op2);
                                    else
                                      printf("dělení nulou\n");
                                    break;
                        case '=':   printf("\t%d\n",
                                                push(pop()));
                                    break;
                        case CLEAR: clear ();
                                    break;
                        case DLOUHY:printf("dlouhý řetězec\n");
                                    break;
                        default:  printf("neznámý příkaz %c\n",
                                                typ);
                                    break;
                }
}
```

```
#define MAXVAL 100      /* maximální velikost zásobníku */
int zp;                 /* ukazatel zásobníku */
int val[MAXVAL];        /* zásobník čísel */

push (i)                /* ulož i do zásobníku */
        int i;
{
        if (zp < MAXVAL)
                return (val[zp++] = i);
        else
        {
                printf ("chyba : plný zásobník\n");
                clear ();
                return (0);
        }
}

pop ()                  /* vyjmi vršek zásobníku */
{
        if (zp > 0)
                return (val [--zp]);
        else
        {
                printf ("chyba : prázdný zásobník\n");
                return (0);
        }
}

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

Příkaz CLEAR vymaže zásobník. Užívá k tomu funkci clear, kterou rovněž používá funkce push v případě chyby. Obdobně funkce init nuluje další externí proměnnou, která souvisí s funkcí getop, k níž se za chvíli vrátíme.

Jak bylo uvedeno v kap. 1, proměnná je externí, je-li definovaná 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 skrytá. 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.

C v i č e n í 4 - 2.

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 obsažen 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?

P o l e p ů s o b n o s t i identifikátoru proměnné je ta část programu, ve kterém je identifikátor definován. Pro automatické proměnné deklarované na začátku funkce je polem 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 na konci tohoto souboru. Jestliže např. val, zp, push, pop a clear jsou definovány v jednom souboru v tomto pořadí, tj.

```
int zp;
int val[MAXVAL];
push (i) (...)
pop ()  ( ...)
clear ()  (...)
```

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

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

Je důležité rozlišovat mezi d e k l a r a c í externí proměnné a její d e f i n i c í. Deklarace popisuje vlastnosti proměnné (tj. typ, rozměr, atd.), zatímco definice této proměnné vyhrazuje současně místo v paměti. Jestliže se řádky

```
int zp;
int val[MAXVAL];
```

uvedou mimo funkce, potom d e f i n u j í externí proměnné zp a val, vyhrazují jim místa v paměti a zároveň slouží jako deklarace až do konce zdrojového souboru.

Na druhé straně řádky

```
extern int zp;
extern int val [];
```

d e k l a r u j í pro zbytek zdrojového textu proměnnou zp jako int a val jako pole typu int (jehož velikost je někde jinde definovaná), ale nevytvářejí proměnné ani pro ně nevyhrazují místa v paměti.

Ve všech souborech, které tvoří daný program, může být pouze 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 zp a val 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 zp;                     /* ukazatel zásobníku */
        int val[MAXVAL];            /* zásobník */
v souboru 2
        extern int zp;
        extern int val [];
        push (i) (...)
        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. Její základní úkol je jasný: přeskoč mezery, tabelátory a symboly pro nový řádek. Jestliže načtený znak není číslo, vrať jej. Jinak načti řetězec číslic a vrať CISLO, 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 - jestliže nedošlo k přetečení, vrací CISLO a řetězec číslic. Pokud číslo bylo příliš dlouhé, getop ignoruje zbytek vstupu, takže uživatel může přepsat řádek od místa chyby; jako signál, že číslo bylo dlouhé, vrací DLOUHY.

```
getop (s, lim)          /* načten operátor nebo operand */
        char s[];
        int lim;
{
        int i, c;
        while ((c=getch())==' '||c=='\t'||c=='\n')
        if ( c < '0' || c > '9' )
                return (c);
        s [0] = c;
        for (i=1;(c=getch())>='0' && c<='9';i++)
                if (i < lim+1)
                        s [i] = c;
```

```
        if ( i< lim+1)                  /* číslo je OK */
        {
                ungetch (c);
                s [i] = '\0';
                return (CISLO);
        }
        else                    /* je příliš dlouhé, přeskoč
                                   zbytek řetězce */
        {
                while (c!= ' '&&c!= '\t'&&c!= '\n'&&c!= EOF)
                        c = getch();
                s [lim-1] = '\0';
                return (DLOUHY);
        }
}
```

Co dělají funkce getch a ungetch? Často se vyskytne situace, že program načítající vstup nedokáže rozhodnout, zda již načetl dost, dokud nenačte 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 zřejmě vyřešen, kdyby bylo možno tento znak "vrátit zpátky" do vstupu - jako kdyby nebyl načten. Tento problém snadno řeší dvojice doplňujících se funkcí.

Jejich spolupráce je jednoduchá - funkce ungetch vrátí znak do společně sdíleného bufferu - znakového pole; funkce getch čte z tohoto pole tehdy, když není prázdné, v opačném případě zavolá getchar. Musí rovněž existovat index, který určuje pozici znaku v bufferu.

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

```
#define BUF     100
char buf[BUF];          /* definice bufferu */
int bufp;               /* volná pozice v bufferu */

getch ()                /* načtení vráceného znaku */
{
        return((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch (c)             /* vrať znak zpět do vstupu */
        int c;
{
        if ( bufp > BUF)
                printf("ungetch: příliš mnoho znaků\n");
        else
                buf[bufp++] = c;
}

init ()                 /* vymaž buffer */
{
        bufp = 0;
}
```

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

C v i č e n í 4 - 3.

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?

C v i č e n í 4 - 4.

Předpokládejme, že nikdy nebude potřeba vracet více než jeden znak. Modifikujte odpovídajícím způsobem funkce getch a ungetch.

C v i č e n í 4 - 5.

Naše funkce getch a ungetch nefungují správně, je-li načteno EOF. Rozhodněte, co by se mělo stát, je-li EOF vrácenc zpět, zdůvodněte a implementujte svůj názor.

4.6. Statické proměnné

S t a t i c k é 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. Slouží tedy k uschování jmen jako buf a bufp ve funkcích getch a ungetch. Tato jména musí být externí, aby je bylo možno oběma funkcemi sdílet, ale neměla by být přístupná uživateli funkcí getch a ungetch, aby nevznikl konflikt. Jestliže jsou obě funkce a obě tyto proměnné překládány současně v jednom souboru jako

```
static char buf[BUF];          /* buffer pro ungetch */
static int bufp;               /* 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 stejnými jmény v jiných souborech téhož programu.

Statická paměť, ať již interní nebo externí, je specializovaná slovem s t a t i c , které se uvede před normální deklarací. 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.

E x t e r n í s t a t i c k é 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 znaků; proměnné buf a bufp by měly být statické, aby nebyly dosažitelné zvenku. Funkce push, pop a clear vytvářejí stejným způsobem "modul" pro operace se zásobníkem, a tak proměnné val a zp jsou definovány jako externí statické.

4.7. Proměnné typu register

Poslední typ proměnné je nazýván r e g i s t e r . Deklarace register oznamuje překladači, že proměnná bude hodně využívána. Pokud je to možné, 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.
```

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

```
f (c,n)
        register char c;
        int 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. Omezení se mění s typem počítače; např. na TNS je brána v úvahu pouze první deklarace typu register v každém bloku.

4.8. Blokové struktury

Jazyk C není jazykem blokových struktur v tom smyslu jako ALGOL nebo PL/1; funkce nemohou být definovány uvnitř jiných funkcí. Na druhé straně ale proměnné mohou být 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 ()
{
        unsigned x;
        ...
}
```

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

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

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

4.9. Inicializace

O inicializaci jsme se již několikrát zmiňovali, ale vždy jen okrajově. Poté, co jsme si uvedli nové typy proměnných, můžeme shrnout některá pravidla.

Verze kompilátoru, který máme k dispozici, nepřipouští zkrácenou inicializaci (při deklaraci) tvaru

```
int i = 1;
```

Inicializaci automatických proměnných a proměnných typu register si ukážeme na příkladu z kap. 3. Pro jejich inicializaci nemusí být použity jenom konstanty, ale i libovolné výrazy, v nichž vystupují dříve definované proměnné nebo dokonce i funkce:

```
binary (x, v, n)
        int x, v[], n;
{
        int l, h, m;

        l = 0;
        h = n - 1;
        ...
        {
                int i,j;
                i = 1;
                j = v[0];
                ...
        }
        ...
}
```

Pro inicializaci externích nebo statických proměnných můžeme použít "inicializační" funkce jako např. clear nebo init v příkladu s kalkulátorem.

4.10. Rekurze

Funkce v jazyku C mohou byt používány rekurzívně, to znamená, že funkce může volat s a m a s e b e. 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 : ↑L

```
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 nejprve volá 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');
}
^L
```

Když funkce volá sebe sama, tak pokaždé má "čerstvou" sadu automatických proměnných, které jsou 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 i 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. Zejména je výhodná pro rekurzivně definované struktury dat jako jsou stromy, jak se ukáže na příkladu v kap. 6.

C v i č e n í 4 - 6.

Použijte ideu z printd a napište novou funkci itoa, tj. převeďte integer na řetězec znaků použitím rekurze.

C v i č e n í 4 - 7.

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.

Vkládání souborů

Každý řádek, který má následující tvar

```
#include "filename"
```

je nahrazen obsahem souboru filename (uvozovky jsou povinné). V souboru se na začátku často objevují 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.

Makroinstrukce

Definice ve tvaru

```
#define ANO     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 řádku. Dlouhé definice mohou pokračovat na dalším řádku, mají-li jako poslední znak \. "Obor působnosti" jména definovaného #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 ANO definované jméno, tak v příkazu printf("ANO") nebude nahrazeno. Protože implementace příkazu #define je makroinstrukce a není součástí překladače, neexistuje pro tento příkaz 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
```

KAPITOLA 5: POINTRY A POLE

Pointer (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, s 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 objektů "nepřímo" - právě pomocí pointrů. Předpokládejme, že x je proměnná typu int a px je pointer, vytvořený zatím nespecifikovaným způsobem. Unární operátor & udává adresu objektu, a tudíž příkaz

```
px = &x;
```

přiřazuje adresu proměnné x do proměnné px; říkáme, že px "ukazuje" na x. Operátor & může být aplikován pouze na proměnné a na 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 pointer 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 pointrů je ale novinkou.

Deklarace

```
int *px;
```

říká, že kombinace *px je typu int, což znamená, jestliže se objeví px v souvislosti s *px, je *px ekvivalentní proměnné typu int. Ve skutečnosti syntaxe deklarace tohoto typu proměnné simuluje syntaxi výrazu, ve kterém se daná proměnná vyskytuje. To je uzitečné ve všech komplikovaných deklaracích. Např. deklarace

```
int atoi (), *dp;
```

určuje, že ve výrazu mají funkce atoi () a *dp hodnotu typu int.

Měli byste si uvědomit, že v deklaraci je pointer 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 pointer 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 jedničku 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 pointry jsou proměnné, může být s nimi nakládáno jako s normálními proměnnými. Jestliže py je pointer 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", nemůže funkce 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 nadefinovaná takto:

```
swap (x,y)                          /* chybně ! */

        int x, y;

{
        int pom_prom;

        pom_prom = x;
        x = y;
        y = pom_prom;
}
```

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

Naštěstí existuje způsob, jak může dosáhnout žádaného efektu. Volající program bude předávat p o i n t r y proměnných:

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

Protože operátor & udává adresu proměnné, je &a pointer 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 pom_prom;

        pom_prom = *px;
        *px = *py;
        *py = pom_prom;
}
```

Obvykle používáme pointry jako argumenty funkcí tehdy, jestliže vyžadujeme, 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. Funkce getint vrací hodnotu, která byla načtena, nebo EOF, když narazila na konec vstupu. Tyto hodnoty musí být vráceny v různých proměnných, protože ať už zvolíme pro EOF 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 funkční 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 pointer. Tento způsob odděluje určování konce souboru od předání číselných hodnot. Následující cyklus vyplňuje pole celými čísly, načtené funkcí getint:

```
int n, v, array [POCET];

for (n = 0; n < POCET && 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, dostaneme nesmyslný výsledek, protože getint počítá s tím, že argument je pointer.

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

```
getint (pn)             /* načti číslo ze vstupu */

        int *pn;

{
        int c, znamenko;

        while((c = getch ()) == ' ' || c == '\n' || c =='\t')
                ;       /* ignoruj oddělovače */
        znamenko = 1;
        if(c == '+' || c == '-')        /* znaménko */
        {
                znamenko = (c == '+') ? 1 : (-1);
                c = getch ();
        }
        for (*pn = 0 ; c >= '0' && c <= '9' ; c = getch () )
                *pn = 10 * *pn + c - '0' ;
        *pn *= znamenko ;
        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.

5.3. Pointry a pole

V jazyku C je úzký vztah mezi pointry a poli. Dokonce tak úzký, že s pointry 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 težší 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 pointer 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 + i);
```

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

Tyto poznatky platí bez ohledu na to, jakého typu jsou proměnné pole a. Definice "přičti i k pointru" a obecně celá aritmetika pointrů je založena na tom, že přírůstek je vynásoben velikostí objektu, na který pointer ukazuje. Proto ve výrazu pa + i je i vynásobeno rozměrem objektu, na který ukazuje pointer pa.

Souhlas mezi indexováním a aritmetikou pointru je zřejmý. Ve skutečnosti je odkaz na pole překladačem převeden na pointer na začátek pole. Z toho plyne, že název pole je vlastně pointer. 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 překvapivější, aspoň 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 ta skutečnost, že když je pa pointer, může být použit s indexem: pa [i] je totožné s *(pa + i). Krátce řečeno, indexový výraz z libovolného pole může být napsán jako pointer a offset (posunutí) a naopak; dokonce ve stejném příkazu.

Je jenom jeden rozdíl mezi jménem pole a pointrem, který si musíme uvědomit. Pointer je proměnná, a proto pa = a a pa ++ jsou dovolené operace. Název pole je naproti tomu k o n s t a n t a a ne proměnná. Proto jsou výrazy typu a = pa nebo a ++ nebo p = &a neplatné.

Když je funkcí předáváno jméno pole, tak je předána adresa počátku pole. Uvnitř volané funkce je argument již ale normální proměnnou a tak jméno pole je opravdu pointer, 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 != '\0'; s ++)
                n ++;
        return n;
}
```

Zvětšování proměnné s je dovolené, protože s je pointer a s ++ nijak neovlivňuje řetezec ve funkci, která strlen volá, ale jenom zvětšuje privátni kopii adresy tohoto pole.

Formální parametr ve funkci může být deklarován 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 pointer a podle toho s ním také nakládá.

Funkcí je také možno předat pouze část pole předáním pointru začátku tohoto podřetězce. Například, je-li a pole, potom

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

nebo

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

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 p je pointer, potom operace p ++ zvětšuje p o jeden prvek a p potom ukazuje na další prvek a p += i zvětšuje p o i, takže ukazuje o i prvků dále. 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 tři podprogramy:

1. Funkce allinic () nastaví ukazatel na začátek paměti vyhrazené pro alokace. Tuto funkci je třeba zavolat jen jednou, a to před prvním použitím funkce alloc.

2. Funkce alloc (n) vrací pointer p na n po sobě jdoucích znakových pozic, které mohou být použity pro uložení znaku.

3. Funkce free (p) uvolňuje alokovanou paměť.

Funkce jsou opravdu "základní", protože funkce free musí být vyvolána, až je-li vyvolána funkce alloc. To znamená, že paměť obhospodařovaná těmito funkcemi je zásobník neboli LIFO (last in, first out - poslední dovnitř, první ven). 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 indexy, jiné funkce nemusí o tomto poli nic vědět. Toto pole může být tedy deklarováno jako extern static, což znamená, že je místní ve zdrojovém souboru obsahující funkce alloc a free a mimo ně je "neviditelné".

Další informaci, kterou potřebujeme znát, je ta, jak často je allocbuf využíváno. Budeme používat pointer 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, jestliže je p uvnitř allocbuf.

```
#define NULL 0          /* pointer pro chybová hlášení  */
#define MAX_ALLOC_PAM 1000  /* maximální dosažitelná paměť  */

static char allocbuf[MAX_ALLOC_PAM]; /* paměť pro alloc */
static char *allocp;              /* další volná pozice  */

allinic ()
{
        allocp = allocbuf;
}

char *alloc (n)         /* vrať ukazatel na n znaku      */

        int n;
{
        if (allocp + n <= allocbuf + MAX_ALLOC_PAM)
                                        /* jde to ? */
        {
                allocp += n;
                return (allocp - n);    /* staré p */
        }
        else    return NULL;            /* je málo místa */
}

free (p)                        /* uvolní paměť */

        char *p;

{
        if (p >= allocbuf && p < allocbuf + MAX_ALLOC_PAM)
                allocp = p;
}
```

Obecně může být pointer 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;
```

definuje allocp jako znakový pointer. Je třeba ho nastavit na začátek allocbuf (pomocí funkce allinic), což je vlastně první volné místo v paměti, když program začíná činnost. Přiřazení v allinic by ale také stejně dobře mohlo být napsáno ve tvaru

```
allocp = &allocbuf [0];
```

protože jméno pole je adresa jeho nultého prvku.

Podmínka

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

testuje, zda je ještě dostatek paměti pro n znaků. Jestliže je, tak allocp bude maximálně o jednu za koncem pole allocbuf. Jestliže je podmínka splněna, vrací alloc normální pointer (všimněte si vlastní definice funkce). Jestliže nemůže být požadavek na paměť splněn, musí alloc nějak tuto skutečnost signalizovat. V jazyku C je zaručeno, že žádný pointer nebude obsahovat nulu jako hodnotu, a proto nula může být 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 + MAX_ALLOC_PAM)
```

a

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

ukazuji 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 pointer může být porovnán s NULL. Ale všechny výhody jsou pryč, porovnáváte-li pointry, z nichž každý ukazuje na něco jiného.

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

```
p + n
```

znamená n-tý prvek za místem, kam ukazuje pointer p. Počítač vynásobí n odpovídajícím rozměrem objektu, na který pointr ukazuje. Např. na počítači TNS je pro char násobný faktor 1, pro int a unsigned 2, pro long 4, pro double 8 a pro strukturu délka struktury.

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;

        p = s;
        while (*p != '\0')
                p ++;
        return p - s;
}
```

Nejprve 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 ++;
```

Protoze p ukazuje na znak, p ++ posouvá p na další znak a p - s udává počet znaků, o který je p posunuto - tj. délku řetězce.

Aritmetika pointrů je konzistentní. Jestliže pracujeme s int, p ++ se posune na další int. Tak můžeme napsat další funkci alloc, která bude pracovat s int místo s char. Toho docílíme tím, že všude ve funkcích alloc a free nahradíme deklaraci char deklarací int. Operace s pointry budou prováděny opět správně.

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

5.5 Znakové pointry a funkce

Z n a k o v á k o n s t a n t a, psaná jako

```
"Ja jsem retezec"
```

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 nejčastěji se řetězce vyskytují jako parametry funkcí:

```
        printf ("Ahoj, svete\n");
```

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

Znaková pole nemusí ale být pouze argumenty funkcí. Jestliže řetězec je deklarován takto

```
char *retezec;
```

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

```
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 strcpy nebude napsaná tak, jak jsme uvedli. Další možnost 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 kvůli tomu, 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 znaků, ve kterých se řetězce s a t liší.

```
strcmp (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 p ř e d vybíráním znaku, na který ukazuje.

C v i č e n í 5 - 1:

Napište pointrovou verzi funkce strcat, kterou jsme uvedli v kapitole 2. strcat (s,t) kopíruje řetězec t na konec řetězce s.

C v i č e n í 5 - 2:

Přepište vhodné programy z dřívějších kapitol s použitím pointrů. Např. itoa, reverse, index aj.

5.6. Pointry nejsou celá čísla

V minulých programech v jazyku C jste si mohli všimnout kavalírského přístupu k pointrům. Obecně platí, že na mnoha počítačích je pointrům přiřazeno celé číslo a naopak. To ale vedlo k přílišné svobodě. V podprogramech, které vrací pointry, které jsou předávány dále, jsou pak vynechány žádoucí deklarace pointrů. 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;
}
```

V praxi ale bývá tendence vynechávat deklarace:

```
strsave (s)                  /* ulož někam řetězec */

        char *s;

{
        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 pointer 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, tedy i použitých knihovních funkcí. Program LINT nás bude při takových konstrukcích varovat.

5.7. Vícerozměrná pole

C umožňuje používání vícerozměrných polí, přestože jsou v praxi mnohem méně používána 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:

den_v_roce bude konvertovat měsíc a den na den v roce a
mesic_den bude konvertovat den v roce na měsíc a den.

Protože tato funkce vrací dvě hodnoty, tak měsíc a den budou pointry

```
mesic_den (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 přestupny nebo ne, je jednodušší použít dvourozměrného pole. Funkce budou vypadat takto:

```
den_v_roce (rok,mesic,den)

        int rok, mesic, den;

{
        int tab_dnu [2] [13];
        int i, pom;

        tab_dnu [0] [0] = 0;    tab_dnu [1] [0] = 0;
        tab_dnu [0] [1] = 31;   tab_dnu [1] [1] = 31;
        tab_dnu [0] [2] = 28;   tab_dnu [1] [2] = 29;
        tab_dnu [0] [3] = 31;   tab_dnu [1] [3] = 31;
        tab_dnu [0] [4] = 30;   tab_dnu [1] [4] = 30;
        tab_dnu [0] [5] = 31;   tab_dnu [1] [5] = 31;
        tab_dnu [0] [6] = 30;   tab_dnu [1] [6] = 30;
        tab_dnu [0] [7] = 31;   tab_dnu [1] [7] = 31;
        tab_dnu [0] [8] = 31;   tab_dnu [1] [8] = 31;
        tab_dnu [0] [9] = 30;   tab_dnu [1] [9] = 30;
        tab_dnu [0] [10] = 31;  tab_dnu [1] [10] = 31;
        tab_dnu [0] [11] = 30;  tab_dnu [1] [11] = 30;
        tab_dnu [0] [12] = 31;  tab_dnu [1] [12] = 31;

        pom = rok % 4 == 0 && rok % 100 != 0 || rok % 400 == 0;
        for (i = 1; i < mesic; i ++)
                den += tab_dnu[pom][i];
        return (den);
}

mesic_den (rok, prubden, pmesic, pden)

        int rok, prubden, *pmesic, *pden;
```

```
{
        int tab_dnu [2] [13];
        int i, pom;

        tab_dnu [0] [0] = 0;     tab_dnu [1] [0] = 0;
        tab_dnu [0] [1] = 31;    tab_dnu [1] [1] = 31;
        tab_dnu [0] [2] = 28;    tab_dnu [1] [2] = 29;
        tab_dnu [0] [3] = 31;    tab_dnu [1] [3] = 31;
        tab_dnu [0] [4] = 30;    tab_dnu [1] [4] = 30;
        tab_dnu [0] [5] = 31;    tab_dnu [1] [5] = 31;
        tab_dnu [0] [6] = 30;    tab_dnu [1] [6] = 30;
        tab_dnu [0] [7] = 31;    tab_dnu [1] [7] = 31;
        tab_dnu [0] [8] = 31;    tab_dnu [1] [8] = 31;
        tab_dnu [0] [9] = 30;    tab_dnu [1] [9] = 30;
        tab_dnu [0] [10] = 31;   tab_dnu [1] [10] = 31;
        tab_dnu [0] [11] = 30;   tab_dnu [1] [11] = 30;
        tab_dnu [0] [12] = 31;   tab_dnu [1] [12] = 31;

        pom = rok % 4 == 0 && rok % 100 != 0 || rok % 400 == 0;
        for (i = 1; prubden > tab_dnu[pom][i]; i ++)
                prubden -= tab_dnu[pom][i];
        *pmesic = i;
        *pden = prubden;
}
```

Pole tab_dnu je prvním vícerozměrným polem, se kterým jsme se setkali. V jazyku C je dvojrozměrné pole jednorozměrným polem, jehož prvky jsou zase jednorozměrná pole. Proto je psáno

```
        tab_dnu [i] [j]
```

namísto

```
        tab_dnu [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 třeba inicializovat (v našem případě na začátku funkcí). První prvek pole tab_dnu 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 zesložitění indexu pole.

Jestliže je funkcí předáváno dvojrozměrné pole, tak v deklaraci argumentů m u s í být uveden počet sloupců pole. Deklarace počtu řádek není rozhodující, protože je předáván pointer. V tomto případě je to pointer na objekty, které jsou 13-ti dimenzionálními poli. Je-li tedy předáváno pole tab_dnu funkci, tak deklarace ve funkci f musí vypadat takto

```
f (tab_dnu)
        int tab_dnu [2] [13];
{
        ...
}
```

Deklarace argumentů může stejně dobře vypadat takto:

```
int tab_dnu [] [13];
```

protoze počet řádků není důležitý, nebo takto

```
int (*tab_dnu) [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 nez *. Bez závorek bude

```
int *tab_dnu [13];
```

deklarováno pole 13 pointrů. To uvidíme v dalším odstavci.

5.8 Pole pointrů. Pointry na pointry

Protože pointry jsou proměnné, můžeme předpokládat, že můžeme využít pole pointrů. Ilustrujeme to na programu, který bude třídit soubor řádků podle abecedy.

V kapitole 3 jsme uvedli funkci Shell sort, 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 dostatečné 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 každý řádek může být reprezentován pointrem na jeho první znak. Pointry mohou být uloženy do pole. Dva řádky mohou být potom porovnávány funkcí strcmp. Jestliže chceme prohodit dva řádky, potom stačí prohodit pouze pointry na ně. To značně zjednodušuje celou operaci.

Třídící postup sestává ze tří částí:

```
načtení všech řádků ze vstupu
třídění
tisk seřazených řádků
```

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 každého řádku a sestavit pole pointrů na tyto řádky. Protože vstupní funkce může nakládat pouze s konečným počtem řádků, mohla by vrátit nesmyslnou hodnotu, pokud by řádků bylo příliš mnoho. Výstupní funkce pouze řádky tiskne podle pořadí pole pointrů.

```
#define NULL    0
#define MAX_RADKU       100     /* maximální počet řádků */

main ()                         /* třídění vstupních řádků */

{
        char *ptr_rad [MAX_RADKU];      /* pointry na řádky */
        int poc_rad;            /* počet načtených řádků */

        if ((poc_rad = readlines (ptr_rad,MAX_RADKU)) >= 0)
        {
                sort (ptr_rad,poc_rad);
                writelines (ptr_rad,poc_rad);
        }
        else
                printf ("prilis vstupnich radku\n");
}

#define MAX_DEL_RAD     1000        /* maximální délka řádku */

readlines (ptr_rad,max_rad)             /* čti vstupní řádky */

        char *ptr_rad [];
        int max_rad;

{
        int len, poc_rad;
        char *p, *alloc (), radek [MAX_DEL_RAD];

        poc_rad = 0;
        while ((len = getline (radek,MAX_DEL_RAD)) > 0)
                if (poc_rad >= max_rad)
                        return (-1);
                else    if((p = alloc (len+1)) == NULL)
                                return (-1);
                        else                    /* nový řádek */
                        {
                                strcpy (p,radek);
                                ptr_rad [poc_rad ++] = p;
                        }
        return (poc_rad);
}
```

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

```
writelines (ptr_rad,poc_rad)            /* výpis řádků */

        char *ptr_rad [];
        int poc_rad;

{
        int i;

        for (i = 0; i < poc_rad; i ++)
                printf ("%s\n",ptr_rad [i]);
}
```

Hlavní novinkou je deklarace ptr_rad:

```
char *ptr_rad [MAX_RADKU];
```

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

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

```
writelines (ptr_rad,poc_rad)                    /* výpis řádků */

        char *ptr_rad [];
        int poc_rad;

{
        while (-- poc_rad >= 0)
                printf ("%s\n",*ptr_rad ++);
}
```

Proměnná ptr_rad na začátku ukazuje na první řádek. Každý inkrement ji posouvá na další řádek a přitom se proměnná poc_rad zmenšuje.

Když jsme se postarali o vstup a výstup, můžeme přejít ke třídění. Program Shell sort 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] vzestupně */

        char *v [];
        int n;

{
        int krok, i, j;
        char *pom_ptr;

        for (krok = n / 2; krok > 0; krok /= 2)
                for (i = krok; i < n; i ++)
                        for (j = i - krok; j >= 0; j -= krok)
                        {
                                if (strcmp(v[j],v[j+krok]) <=0)
                                        break;
                                pom_ptr = v [j];
                                v [j] = v [j + krok];
                                v [j + krok] = pom_ptr;
                        }
}
```

Celý program by mohl být ovšem rychlejší. Mohl by totiž kopírovat řádky ze vstupu přímo do pole ptr_rad a ne do pole radek a potom dále. Je ale lepší udělat první verzi jasně a o "účinnost" se starat později. Tato úprava by stejně 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 p ř e d prvním vykonáním těla cyklu. To nám zaručuje, že program bude fungovat správně, i když na vstupu nejsou žádné řádky. Projděte si program pro třídění a zjišťujte, co se stane, nebyl-li načten žádný vstup.

C v i č e n í 5 - 3:

Přepište readlines tak, aby řádky ukládal v jednotce main a ne v sobě. O kolik bude program rychlejší?

5.9 Inicializace pole pointrů

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

Syntaxe je obdobná jako dříve:

```
char *mesic_jmeno(n)     /*vrať jméno n-tého měsíce*/

        int n;

{
        static char *jmeno[13] ;
        int i;

        i = 0;
        jmeno [i++] = "Spatne cislo mesice";
        jmeno [i++] = "Leden";
        jmeno [i++] = "Unor";
        jmeno [i++] = "Brezen";
        jmeno [i++] = "Duben";
        jmeno [i++] = "Kveten";
        jmeno [i++] = "Cerven";
        jmeno [i++] = "Cervenec";
        jmeno [i++] = "Srpen";
        jmeno [i++] = "Zari";
        jmeno [i++] = "Rijen";
        jmeno [i++] = "Listopad";
        jmeno [i] = "Prosinec";

        return((n < 1 || n > 12) ? jmeno[0] : jmeno[n]);
}
```

Deklarace proměnné jméno, která je polem znakových pointrů,je obdobná deklaraci ptr_rad 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 pointer na něj je uložen v jmeno[i].

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 použití a a b je 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 používáno 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 pointer ukazuje na pole int o rozměru 10, tak bude alokováno celkem 100 buněk plus 10 buněk pro pointry. Pole pointrů potřebuje tedy více paměti a může tak požadovat explicitní inicializaci. Má ale také dvě výhody: adresování prvků 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 pole o 2 prvcích, jiný na pole s 20 prvky a další na prázdné pole.

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

C v i č e n í 5 - 4:

Přepište funkce den_v_roce a mesic_den s použitím pointrů.

5.11 Argumenty ve tvaru příkazového řádku

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 příkazového řádku, kterou byl program vyvolán. Druhý parametr (argv) je pointer 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 ahoj, svete
```

bude mit vystup

```
AHOJ, SVETE
```

Výstup v tomto tvaru je způsoben tím, že všechna malá písmena v příkazovem řádku jsou operačním systémem CP/M konvertována na velká. Tuto vlastnost musíme mít v následujících příkladech na zřeteli.

argc je vždy přinejmenším 1. argv[0] je pointer na prázdný řetězec. V předchozím příkladě je argc 3 a argv[0] je "", argv[1] je "AHOJ," a argv[2] je "SVETE". Prvním skutečným argumentem je argv[1] a posledním argv[argc-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 pointer 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');
}
```

Jelikož je argv pointer 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 příkladu 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. Zde je program find :

```
#define MAX 1000          /* maximální délka řádku */

main (argc, argv)         /*nalezni první výskyt řetězce s */

        int argc;
        char *argv [];

{
        char radek [MAX];

        if ( argc != 2 )
                printf("Uziti programu: find vzor\n");
        else
                while ( getline (radek, MAX) > 0)
                        if ( index (radek, argv[1]) >= 0 )
                                printf("%s", radek);
}
```

Je obecnou konvencí v programech v jazyku C těm argumentům v příkazovém řádku, které začínají znaménkem + nebo - , říkat parametry. Tyto zvláštní argumenty neboli parametry určují způsob běhu programu. Program by měl být v takovémto případě napsán tak, aby nezáleželo na pořadí parametrů, dokonce i tak, aby mohl být kterýkoliv parametr vynechán, nebo aby mohly být parametry slučovány.

Nejlépe bude, když tuto problematiku demonstrujeme na jednoduchém příkladu. Vyjdeme ze základního modelu programu find. Předpokládejme, že chceme používat dva zvláštní argumenty - parametry. První z nich bude určovat, které řádky se mají tisknout, a druhý, zda se budou řádky tisknout s číslováním. Zvolíme si následující konvenci pro hodnotu prvního parametru :

```
v ( vyber )     - vyber všechny řádky, které obsahují
                  daný vzor, a vytiskni je
-v              - vyber všechny řádky, které neobsahují
                  daný vzor, a vytiskni je
```

Pro druhý parametr zvolíme následující konvenci :

```
c ( cislovani ) - tiskni řádky bez číslování
-c              - tiskni řádky s číslováním
```

Potom

```
find -v -n mus
```

se vstupem ve tvaru

```
Kdo chce dobrou
musku miti,
musi kola-
loku piti.
```

vytiskne

```
1: Kdo chce dobrou
4: loku piti.
```

za předpokladu, že jsme zpětně konvertovali řetězec mus na malá písmena. Jestliže bude některý z parametrů vynechán, program implicitně předpokládá, že byla zadána kladná hodnota tedy v nebo c.

Zde je program:

```
#define MAX     1000    /* maximální délka řádku */

main (argc,argv)                        /* najdi řetězec */

        int argc;
        char *argv [];

{
        char radek [MAX], *s;
        unsigned cis_rad;
        int vyber, cislovani;

        cis_rad = vyber = cislovani = 0;
        while (-- argc > 0 && (* ++ argv) [0] == '-')
                for (s = argv [0] + 1; *s != '\0'; s ++)
                        switch (tolower(*s))
                        {
                        case 'v';
                                vyber = 1;
                                break;
                        case 'c';
                                cislovani = 1;
                                break;
                        default;
                           printf("find: spatny parametr%c\n",
                                                        *s);
                                argc = 0;
                                break;
                        }
        if(argc != 1)
                printf("Uziti programu: find -v -c vzor\n");
        else
                while(getline(radek, MAX) > 0)
                {
                        cis_rad++;
                        if((index(radek, *argv) >= 0) != vyber)
                        {
                                if(cislovani)
                                        printf("%d: ",cis_rad);
                                printf("%s", radek);
                        }
                }
}
```

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 pointer 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 možnou formou zápisu je **++argv.

C v i č e n í 5 - 5:

Napište program add, který vyčísluje v obrácené polské notaci výraz, zadaný jako argument. Např.

```
add 2 3 4 + * =
```

vypočte 2 * (3 + 4).

C v i č e n í 5 - 6:

Modifikujte program entab a detab (z kap.1) tak, aby tabelátory byly zadávány jako argumenty. Použijte normální tabelátor, nebyl-li zadán žádný argument.

C v i č e n í 5 - 7:

Rozšiřte entab a detab tak, aby umožňoval zadání ve tvaru

```
entab m + n
```

což znamená, že tabelační pozice jsou m, n+m, 2*n+m,...

C v i č e n í 5 - 8:

Napište program tail, který tiskne posledních n řádků 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 jazyku C není proměnná, ale je možno definovat p o i n t e r f u n k c e, se kterým může být manipulováno (může být předáván jako argument funkcím, ukládán do pole atd.).

Budeme to ilustrovat tím, že upravíme třídící program tak, že když bude uveden parametr -n, budou řádky setříděny číselně a ne podle abecedy.

Třídění obvykle sestává ze tří částí:

1. p o r o v n á v á n í, které určuje pořadí porovnávaného páru,

2. v ý m ě n y, která mění pořadí páru,

3. t ř í d í c í h o a l g o r i t m u, 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 MAX_RAD 100      /* maximální počet řádků */

main (argc,argv)         /* setřídění vstupních řádků */

        int argc;
        char *argv [];

{
        char *ptr_rad [MAX_RAD];/* pointry na text. řádky  */
        int poc_rad;            /* počet načtených řádek   */
        int strcmp(), numcmp(); /* porovnávací funkce      */
        int swap ();            /* funkce výměny           */
        int numeric;            /* 1 pro numerickou výměnu */

        numeric = 0;
        if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n')
                numeric = 1;
        if ((poc_rad = readlines (ptr_rad,MAX_RAD)) >= 0)
        {
                if (numeric)
                        sort (ptr_rad,poc_rad,numcmp,swap);
                else
                        sort (ptr_rad,poc_rad,strcmp,swap);
                writelines (ptr_rad,poc_rad);
        }
        else
                printf ("prilis vstupnich radku ");
}
```

Názvy strcmp, numcmp a swap jsou adresy funkcí. Protože víme, že jde o funkce, není operátor & 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 krok, i, j;
```

```
        for (krok = n / 2; krok > 0; krok /= 2)
                for (i = krok; i < n; i ++)
                        for (j = i - krok; j >= 0; j -= krok)
                        {
                                if ((*comp)(v[j],v[j+krok])<=0)
                                        break;
                                (*exch)(&v[j],&v[j+krok]);
                        }
}
```

Musíme dát pozor na deklarace. Deklarace

```
int (*comp) ();
```

říká, že comp je pointer funkce, která vrací int. První pár závorek je nezbytný, protože bez nich deklarace

```
int *comp ();
```

znamená, že comp je funkce, která vrací pointer, což, jak vidíme, je něco zcela odlišného. Použití funkce comp v řádku

```
if ((*comp) (v[j],v[j + krok])) <= 0)
```

je v souhlase s deklarací: comp je pointer na funkci, *comp je funkce a

```
(*comp) (v[j],v[j + krok])
```

je volání funkce. Závorky jsou nezbytné. Ještě uvedeme funkci numcmp:

```
numcmp (s1,s2)          /* Porovnávání s1 a s2 číselné */

        char *s1, *s2;

{
        int atoi (), v1, v2;

        v1 = atoi (s1);
        v2 = atoi (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 *pom_ptr;

        pom_ptr = *px;
        *px = *py;
        *py = pom_ptr;
}
```

C v i č e n í 5 - 9:

Upravte sort tak, aby bylo možno zadat parametr -r, který požaduje třídění sestupné; -r musí rovněž pracovat s -n.

C v i č e n í 5 - 10:

Přidejte parametr -f, který dává na roveň malá a velká písmena.

C v i č e n í 5 - 11:

Přidejte parametr -d ("slovníkové srovnávání"). Potom budou porovnávány pouze písmena, číslice a mezery, ostatní znaky jsou ignorovány. Zkontrolujte, zda bude pracovat spolu s -f.

Organizace paměti

Funkce calloc je obdobou funkce alloc, kterou jsme používali v předchozích kapitolách. Příkaz

```
calloc (n,sizeof(objekt));
```

vrací pointer na místo v paměti, kam lze umístit n objektů specifikované délky. Není-li požadovaná paměť dostupná, vrací NULL. Alokovaná paměť je vynulována.

Pointer je řádně nastaven na požadovaný objekt, ale měl by být nastaven na správný typ:

```
char *calloc();
int *ip, n;

ip = (int*) calloc (n,sizeof(n));
```

Funkce free (p) uvolňuje místo v paměti, na který ukazuje pointer p. Pointer p byl původně získán vyvoláním funkce calloc. Uvolňování paměti n e m u s í být děláno popořádku. Je ale chybou uvolňovat paměť, která nebyla přidělena voláním funkce calloc nebo alloc.

KAPITOLA 6. STRUKTURY

S t r u k t u r a 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 znovu 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 datum
{
        int den;
        int mesic;
        int rok;
        int den_roku;
        char jmeno_mesice [4];
};
```

Klíčové slovo struct uvádí deklaraci struktury, která je seznamem deklarací, vložených do závorek. Nepovinným parametrem je jméno struktury uvedené za slovem struct. V našem příkladu je to jméno datum.

Jméno strukturu označuje a může být použito jako zkratka pro podobnou deklaraci. Prvky nebo proměnné uvedené ve struktuře jsou nazývány jejími č l e n y .

Pravá závorka ukončuje seznam členů a za ní může následovat seznam obyčejných proměnných. Tj.

```
struct {...} x, y, z;
```

je z hlediska syntaxe analogické

```
int x, y, z;
```

v tom smyslu, že x, y a z jsou deklarovány jako proměnné téhož typu.

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 datum, potom

```
struct datum d;
```

definuje proměnnou d, která je strukturou typu datum. Člen struktury je možno dosáhnout takto:

```
jmeno_struktury. clen
```

Operátor "." spojuje jméno struktury s jejím členem. Např. nastavení proměnné pom ze struktury datum může být provedeno takto:

```
pom = d.rok % 4 == 0 && d.rok % 100 != 0
        || d.rok % 400 == 0;
```

Kontrola názvu měsíce:

```
if (strcmp (d.jmeno_mesice, "Aug") == 0) ...
```

nebo konvertování prvního znaku měsíce na malé písmeno:

```
d.jmeno_mesice [0] = tolower (d.jmeno_mesice [0]);
```

Struktury mohou byt vkládány do sebe. Data zaměstnance mohou vypadat takto:

```
struct zamestnanec
{
        char jmeno [DL_JMENA];
        char adresa [DL_ADRESY];
        char c_o_p [10];
        unsigned os_cislo;
        int mzda;
        struct datum dat_nar;
        struct datum dat_nastupu;
};
```

Struktura zamestnanec obsahuje dva datumy. Jestliže budeme deklarovat strukturu emp jako:

```
struct zamestnanec emp;
```

potom

```
emp.dat_nar.mesic
```

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 jejich 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.

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 funkci den_v_roce z kapitoly 5:

```
d.den_roku = den_v_roce (d.rok, d.mesic, d.den);
```

Druhý způsob je předání pointru. Jestliže jsme deklarovali dat_nastupu jako

```
struct datum dat_nastupu;
```

a přepsali funkci den_v_roce, můžeme napsat:

```
dat_nastupu.den_roku = den_v_roce (&dat_nastupu);
```

Funkce musí být modifikována, protože jejím argumentem je nyní pointer a ne seznam proměnných.

```
den_v_roce (pd)             /* výpočet dne roku z měsíce a dne */

        struct datum *pd;
{
        int i, dat, pom;

        dat = pd -> den;
        pom = pd -> rok % 4 == 0 && pd -> rok % 100 != 0 ||
                                    pd -> rok % 400 == 0;
        for (i=1; i < pd -> mesic; i++)
                dat += tab_dnu [pom] [i];
        pd -> den = dat;
}
```

Deklarace

```
struct datum *pd
```

určuje, že pd je pointer na strukturu typu datum. Notace

```
pd -> rok
```

je novinkou.

Jestliže p je pointer na strukturu, potom

```
p -> člen struktury
```

ukazuje na určitý člen. Operátor je sestaven ze znaménka minus a znaku > .

Protože pd je pointer na strukturu, tak člen rok je možno dosáhnout takto:

```
(*pd).rok
```

Pointry na struktury jsou často používány a tak operátor -> je výhodnou zkratkou. Ve výrazu (*pd).rok jsou závorky nezbytné, protože operátor "." má vyšší prioritu než *, operátory -> a . pracují zleva doprava.

```
p -> q -> člen struktury
```

je emp.dat_nar.mesic

```
( p -> q ) -> člen struktury
```

je (emp.dat_nar).mesic

```
mesic_den (pd)          /* výpočet měsíce a dne ze dne roku */

        struct datum *pd;
{
        int i, pom;

        pom = pd -> rok % 4 == 0 && pd -> rok % 100 != 0
                                 || pd -> rok % 400 == 0;
        pd -> den = pd -> den_roku;
        for (i = 0; pd -> den > tab_dnu [pom] [i]; i++)
                pd -> den -= tab_dnu [pom] [i];
        pd -> mesic = i;
}
```

Operátory -> 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 dosažením 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 dosažení objektu (právě tak jako *s++). (*p -> y)++ zvětšuje to, na co ukazuje y. *p++ -> y zvětšuje p po dosažení 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, která budou obsahovat jména a pole čísel int, kde bude ukládán počet. Jednou z možností je použití dvou paralelních polí - klic_slovo a pocet_slov:

```
char *klic_slovo [P_KLICU];
int pocet_slov [P_KLICU];
```

Právě fakt, že jsou pole paralelní, indikuje možnost použití jiné organizace. Každý prvek je vlastně pár:

```
char *klic_slovo;
int pocet_slov;
```

Deklarace

```
struct klic
{
        char *klic_slovo;
        int pocet_slov;
}
tab_klicu [P_KLICU];
```

udává, že každý prvek pole tab_klicu je strukturou, a alokuje paměť. To může být napsáno takto:

```
struct klic
{
        char *klic_slovo;
        int pocet_slov;
};
struct klic tab_klicu [P_KLICU];
```

Program pro zjišťování počtu klíčových slov začíná deklarací struktury tab_klicu a její inicializací například pomocí funkce inic. Hlavní program čte vstup opakovaným voláním funkce getword, která načítá ze vstupu vždy jedno slovo. Pro každé slovo je prohledána tabulka tab_klicu pomocí funkce pro binární hledání, kterou jsme uvedli v kapitole 3. (Seznam klíčových slov musí být uspořádán vzestupně.)

```
#define         MAX_DELKA       20
#define         EOF             (-1)
#define         PISMENO         'a'
#define         CISLICE         '0'
#define         P_KLICU         10

struct klic
{
        char *klic_slovo;
        int pocet_slov;
} tab_klicu [P_KLICU];


main()                                  /* počítání klíčových slov */

{
        int n, t;
        char slovo [MAX_DELKA];

        inic ();
        init ();

        while ((t = getword (slovo,MAX_DELKA)) != EOF)
              if((n = binary (slovo,tab_klicu,P_KLICU) >= 0)
                                tab_klicu [n].pocet_slov++;
        for (n = 0; n < P_KLICU; n++)
                if (tab_klicu[n].pocet_slov > 0)
                        printf ("%4d %s\n",
                                tab_klicu[n].pocet_slov,
                                tab_klicu[n].klic_slovo);
}


binary (slovo,tab,n)                    /* najdi slovo v tab[n] */

        char *slovo;
        struct klic tab [];
        int n;

{
        int dol_mez, hor_mez, stred, pod;

        dol_mez = 0;
        hor_mez = n - 1;
        while (dol_mez <= hor_mez)
        {
                stred = (dol_mez + hor_mez) / 2;
                if((pom=strcmp(slovo,tab[stred].klic_slovo))<0)
                        hor_mez = stred - 1;
                else if (pom > 0)
                        dol_mez = stred + 1;
                else
                        return (stred);
        }
        return (-1);
}
```

Následující verze funkce inic nevyčerpává všechna klíčová slova jazyka C.

```
inic ()                                 /* inicializace struktury */
{
        int j;

        for (j = 0;j <= P_KLICU-1; j++)
        {
                tab_klicu[j].pocet_slov = 0;
                tab_klicu[j].klic_slovo = alloc(10);
        }
        tab_klicu[0].klic_slovo = "break";
        tab_klicu[1].klic_slovo = "continue";
        tab_klicu[2].klic_slovo = "else";
        tab_klicu[3].klic_slovo = "for";
        tab_klicu[4].klic_slovo = "if";
        tab_klicu[5].klic_slovo = "int";
        tab_klicu[6].klic_slovo = "return";
        tab_klicu[7].klic_slovo = "struct";
        tab_klicu[8].klic_slovo = "unsigned";
        tab_klicu[9].klic_slovo = "while";
}
```

Nyní k funkci getword. Funkce getword vrací PISMENO pokaždé, když nalezne slovo. Toto slovo kopíruje do prvního argumentu. Slovo je řetězec znaků a čísel začínající písmenem, nebo jeden znak. Funkce getword vrací EOF pro konec souboru nebo bylo-li načteno dlouhé slovo.

```
getword (s,lim)                                 /* načti slovo */
        char s[];
        int lim;
{
        int c,i;

        while ((type (c = getch ())) != PISMENO && c != EOF)
                ;
        if (c == EOF)
                return (c);
        s[0] = c;
        for (i=1; (type (c = getch ())) == PISMENO ||
                                        type (c) == CISLICE; i++)
                if(i < lim+1)
                        s[i] = c;
        if (i < lim+1)
        {
                ungetch (c);
                s[i] = '\0';
                return (PISMENO);
        }
        else
                return (EOF);
}
```

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 (PISMENO);
        else if(c >= '0' && c <= '9')
                return (CISLICE);
        else
                return (c);
}
```

Symbolické konstanty PISMENO a CISLICE mohou mít libovolnou hodnotu, která není v rozporu s nealfanumerickými znaky a EOF. Obvykle se volí

```
#define PISMENO  'a'
#define CISLICE  '0'
```

C v i č e n í 6 - 1.

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.

Globální deklarace pole tab_klicu může zůstat nezměněna. Musíme změnit main a binary.

```
main()              /* počítej klíčová slova, verze s pointry */

{
        int t;
        char slovo [MAX_DELKA];
        struct klic *binary(), *p;

        while ((t = getword (slovo,MAX_DELKA)) != EOF)
              if((p = binary(slovo,tab_klicu,P_KLICU)) != NULL)
                        p -> pocet_slov++;
        for (p = tab_klicu; p < tab_klicu + P_KLICU; p++)
                if (p -> pocet_slov > 0)
                        printf("%4d %s\n",
                                p -> pocet_slov,
                                p -> klic_slovo);
}

struct klic *binary (slovo, tab, n)
                                    /* najdi slovo v tab [n] */
```

```
        char *slovo;
        struct klic tab[];
        int n;

{
        int pom;
        struct klic *dol_mez;
        struct klic *hor_mez;
        struct klic *stred;

        dol_mez = &tab [0];
        hor_mez = &tab [n-1];
        while (dol_mez <= hor_mez)
        {
                stred = dol_mez + (hor_mez - dol_mez) / 2;
                if ((pom = strcmp (slovo,stred
                        -> klic_slovo)) < 0)
                        hor_mez = stred - 1;
                else if (pom > 0)
                        dol_mez = stred + 1;
                          else
                                return (stred);
        }
        return (NULL);
}
```

Za zmínku zde stojí více věcí. Za prvé, deklarace funkce binary musí indikovat, že funkce vrací pointer na strukturu typu klic. To musí být deklarováno jak v jednotce main, tak ve funkci binary. Jestliže funkce binary slovo najde, tak vrací pointer na něj. Když slovo nenajde, vrací NULL. Za druhé, přístup k prvkům pole tab_klicu je přes pointry. Proto musíme změnit funkci binary. Prostřední prvek nemůže už být jednoduše zjišťován výrazem

```
stred = (dol_mez + hor_mez) / 2
```

protože součet pointrů produkuje nesmyslný výsledek (dokonce i když je vydělen 2). To musí být změněno na:

```
stred = dol_mez + (hor_mez - dol_mez) / 2
```

což nastavuje střed na prvek na polovině cesty mezi hor_mez a dol_mez.

Všimněte si rovněž inicializace dol_mez a hor_mez. Je možné totiž inicializovat pointer na adresu dříve definovaného objektu.

V main jsme napsali:

```
for (p = tab_klicu; p < tab_klicu + P_KLICU; p++)
```

p je pointer 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 rozměrů jejích členů.

Nakonec se podívejme na formát programu. Jestliže funkce vrací komplikovaný typ jako

```
struct klic *binary (slovo, tab, n)
```

tak můžeme v deklaraci jméno funkce přehlédnout. Proto někdy používáme zápis ve tvaru

```
struct klic *
binary (slovo, 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 v š e c h 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 načítaná slova neustále třídit. To znamená ukládat právě načtené slovo tam, kam patří. To ale nemůžeme dělat v lineárním poli, protože by to rovněž trvalo dlouho. Místo toho použijeme datovou strukturu nazývanou
b i n á r n í s t r o m.

Tento strom obsahuje vždy jeden uzel pro každé slovo. Každý uzel obsahuje:

```
pointer na 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 obsaženo 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 tisky je rekurzivní.

Vraťme se zpět k popisu uzlu. Je to struktura se čtyřmi položkami:

```
struct uzel_stromu                    /* základní uzel */
{
        char *slovo;                 /* pointer na text */
        int pocet;                    /* počet výskytů */
        struct uzel_stromu *levy;     /* levý poduzel */
        struct uzel_stromu *pravy;    /* pravý poduzel */
}*ptr;
```

"Rekurzivní" definice uzlu může vypadat zmateně, ale je úplně správná. Je zakázáno, aby struktura obsahovala sebe samu jako položku, ale

```
struct uzel_stromu *levy;
```

deklaruje levy jako p o i n t e r na uzel. Tedy struktura uzel_stromu neobsahuje sebe samu.

Text programu je velice krátký a využívá funkce, které jsme napsali dříve. Je to getword pro načtená slova ze vstupu, alloc, která zajišťuje místo v paměti pro jednotlivá slova.

Hlavní program jednoduše čte slova a ukládá je do stromu.

```
#define PISMENO         'a'
#define CISLICE         '0'
#define MAX_DELKA       20

main()                                /* počítání frekvence slov */

{
        struct uzel_stromu *koren, *strom ();
        char sl [MAX_DELKA];
        int t;

        koren = NULL;
        while ((t = getword (sl, MAX_DELKA)) != EOF)
                koren = strom (koren,sl);
        treeprint (koren);
}
```

Funkce strom je jednoduchá. Slovo je uloženo na vrcholek stromu (kořen). V každé fázi je porovnáno se slovem uloženým v uzlu. Potom se pokračuje do levého nebo pravého poduzlu rekurzivním voláním funkce strom. Slovo je buď ve stromu nalezeno (a je přičtena jednička k množství výskytu), nebo je výsledkem nulový pointer, který indikuje, že uzel musí být ve stromu vytvořen. Jestliže je vytvořen nový uzel, tak strom vrací pointer na něj a tento pointer je zařazen do vyššího uzlu.

```
struct uzel_stromu *
strom (p,w)                              /* uložení do p nebo níže */

        struct uzel_stromu *p;
        char *w;

{
        struct uzel_stromu *talloc ();
        char *strsave();
        int pom;

        if (p == NULL)                                  /* nové slovo */
        {
                p = talloc();                            /* nový uzel */
                p -> slovo = strsave (w);
                p -> pocet = 1;
                p -> levy = NULL;
                p -> pravy = NULL;
        }
        else if ((pom = strcmp (w,p -> slovo)) == 0)
                p -> pocet ++ ;                 /* opakované slovo */
                else if (pom < 0)                          /* je menší */
                     p -> levy = strom (p -> levy,w);
                        else                               /* je větší */
                            p -> pravy = strom (p -> pravy, w);
        return (p);
}
```

Paměť pro nový uzel je přidělována funkcí talloc, která je obdobou funkce alloc, kterou jsme napsali dříve. Vrací pointer na volné místo pro uzel stromu. (Za chvíli se k tomu vrátíme.) Nové slovo je kopírováno kamsi do 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.

Funkce treeprint tiskne strom levým podstromem počínajíc. 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 rekurzivně */

        struct uzel_stromu *p;
{
        if (p != NULL)
        {
                treeprint (p -> levy);
                printf("\n %4d \t %s\n",p -> pocet,p -> slovo);
                treeprint (p -> pravy);
        }
}
```

Praktická poznámka: jestliže strom není vyvážen, protože slova nepřicházejí náhodně, čas výpočtu roste příliš rychle. Nejhorším případem je, přicházejí-li slova již uspořádaná. Program vlastně potom provádí lineární prohledávání. Existuje zobecnění lineárních stromů: stromy typu 2-3 a AVL, které jsou výhodnější, ale my se o nich zde nebudeme zmiňovat.

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.

Poznámka:
V jazyku C existuje unární operátor s i z e o f, 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 proměnna, pole, struktura, jméno typu int nebo jméno odvozené od struct.

Otázka způsobu deklarace funkce alloc je klíčová pro všechny jazyky, kde je prováděno kontrolování typu seriózně. Nejlepším způsobem v jazyku C je deklarovat, že alloc vrací pointer na char, a potom upravit pointer na požadovany tvar. Jestliže je p deklarováno jako

```
char *p;
```

potom

```
(struct uzel_stromu *) p
```

jej konvertuje na pointer typu uzel_stromu. Proto je funkce talloc napsána takto:

```
struct uzel_stromu *talloc ()

{
        char *alloc ();
        return ((struct uzel_stromu *) alloc (sizeof(*ptr)));
}
```

Je to více než požadují současné překladače, ale představuje to nejbezpečnější přístup k budoucím překladačům.

C v i č e n í 6 - 2.

Napište program, který čte program v jazyku C a tiskne podle abecedy skupiny slov, které jsou shodné v prvních pěti znacích.

C v i č e n í 6 - 3.

Napište jednoduchý program pro křížové reference. Program má tisknout všechna slova ze vstupu a ke každému slovu uvést čísla řádek, ve kterých se vyskytuje.

C v i č e n í 6 - 4.

Napište program, který tiskne rozdílná slova ze vstupu setříděná sestupně podle frekvence výskytu. Před každé slovo vytiskněte počet výskytů.

6.6. Prohledávání tabulky

Další vlastností struktur budeme ilustrovat souborem programů pro prohledávání tabulek. Tyto programy bývají součástí podprogramu pro rozvíjení maker a pro ukládání parametrů do tabulek. Např. uvažujeme příkaz #define v jazyku C. Jestliže je na řádce text:

```
#define ANO 1
```

tak jméno ANO i jeho hodnota 1 jsou uloženy v tabulce. Později, když se objeví jméno ANO v příkazu

```
v_slove = ANO;
```

tak musí být nahrazeno jedničkou.

Pro operace s názvy a jejich náhradami existují dvě základní funkce. Funkce install(s,t) ukládá jméno s a nahrazující řetězec t do tabulky. Parametry s a t jsou znakové řetězce. Funkce lookup prohledává tabulku a vrací pointer na místo, kde byl nalezen text, nebo NULL, nebyl-li text nalezen. V těchto funkcích je použito klíčovacího algoritmu: 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 pointer na jména, nahrazující text, a odkaz na další blok v řetězci. Nulový pointer označuje konec řetězce.

```
struct polozky                    /* základní položka */
{
      char *jmeno;
      char *def;
      struct polozky *next; /* další vstup v řetězci */
}*ptr;
```

Pole pointrů vypadá takto:

```
#define HASHVEL 100

static struct polozky *hashtab [HASHVEL];
                                  /* tabulka pointerů */
```

Transformační funkce, které je použito ve funkcích lookup a install, sčítá hodnoty znaků v řetězci a vrátí zbytek po dělení rozměrem tabulky. (Není to algoritmus nejlepší, ale je jednoduchý).

```
nash (s)                                /* transformace řetězce s */

        char *s;

{
        int hashval;

        for (hashval = 0; *s != '\0';)
                hashval += *s++;
        return (hasval % HASHVEL);
}
```

Tento proces produkuje počáteční 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 pointer na něj. Jestliže ale neexistuje, vrací NULL.

```
struct polozky *
lookup (s)                                      /* prohledávání */

        char *s;

{
        struct polozky *np;

        for (np = hashtab[hash(s)];np == NULL;np=np -> next)
                if (strcmp (s,np -> jmeno) == 0 )
                        return (np);                    /* nalezeno */
        return (NULL);                                  /* nenalezeno */
}
```

Funkce install používá lookup pro zjištění, zda je jméno již uloženo. Jestliže je uloženo, tak nová definice musí nahradit starou. Jinak je uloženo nové jmeno. Install vrací nulu, když není místo pro nový prvek.

```
struct polozky *
install (jm,df)                         /* ulož (jm,df) do hashtab */

        char *jm, *df;

{
        struct polozky *np, *talloc ();
        char *strsave();
        int hashval;

        if ((np = lookup (jm)) == NULL)         /* nenalezeno */
        {
                np = talloc ();
                if (np == NULL)
                        return (NULL);
                if ((np -> jmeno = strsave (jm)) == NULL
                        return (NULL);
                hashval = hash (np -> jmeno);
                np -> next = hashtab [hashval];
                hashtab [hashval] = np;
```

```
        }
        else                                        /* existuje */
                free (np -> def);                   /* vymazání */
        if ((np -> def = strsave (df)) == NULL )
                return (NULL);
        return (np);
}
```

Strsave kopíruje řetězec, který je jeho argumentem, na bezpečné místo. Místo je přiděleno funkci alloc - ukázali jsme ji v kap. 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 kapitola 7 a 8.

C v i č e n í 6 - 5.

Napište program,který vyjme jméno a nahrazující text (definicí) z tabulky, obhospodařované funkcemi install a look-up.

6.7. Pole bitů

Jestliže je na prvním místě otázka paměti, je často nezbytné sloučit různé objekty do jednoho počítačového slova. Hodně se používá souboru příznakových 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, externí statický, interní, atd. Nejkompaktnějším způsobem zápisu těchto informací je jejich zakódování do proměnné typu int nebo char.

Obvyklý způsob je definovat "masky", odpovídající pozici bitů ve slově (definování a přímý přístup k polím bitů nelze na TNS použít):

```
#define KLIC_SL 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);
```

nuluje tyto bity.

Výraz

```
if ((flags & (EXTERNAL | STATIC)) == 0)
```

je pravdivý, jestliže je některý z obou bitů jednotkový.

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, unsigned 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.

```
union   u_tag
{
        int ival;
        unsigned sval;
        char *pval;
} uval;
```

Proměnná uval bude dostatečně velká, aby mohla obsahovat největší ze tří typů; nezáleží na hardware počítače. Libovolný z těchto typů může být proměnné uval přiřazen a potom použit ve výrazu. Typ musí odpovídat naposledy uloženému typu. Záleží pouze na programátorovi, aby si pamatoval, co do proměnné uval uložil naposledy. Výsledek záleží na počítači pouze tehdy, uloží-li se do unionu něco jiného, než co je později využito.

Členy unionu jsou dosažitelné jako ve strukturách:

```
jmeno_unionu.clen
```

nebo

```
pointer_unionu -> clen
```

Jestliže do proměnné utyp uložíme typ, který byl přiřazen unionu uval, pak můžeme psát

```
if (utyp == INT)
        printf ("%d\n", uval.ival);
else if (utyp == UNSIGNED)
        printf ("%d\n", uval.sval);
else if (utyp == RETEZ)
        printf ("%s\n",uval.pval);
else
        printf ("spatny typ %d\n", utyp);
```

Uniony se mohou objevit ve strukturách a polích a naopak. Zápis pro dosažení členů unionu ve struktuře a naopak je identický s vnořenými strukturami. Např. v poli struktur nadefinovaném takto:

```
struct
{
        char *jmeno;
        int flags;
        int utyp;
        union
        {
                int ival;
                unsigned sval;
                char *pval;
        } uval;
} symtab [P_SYM];
```

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 všechny členy mají relativní posun adresy nulový; struktura je dostatečně velká, aby mohla "pojmout nejširší člen", a zarovnání adresy platí pro všechny členy unionu. Jediná povolená operace s unionem je dosažení jeho členu a určení jeho adresy. Uniony nemohou byt 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 lze uniony používat.

6.9. Příkaz typedef

Jazyk C umožňuje definovat nová datová jména např.

```
typedef int DELKA;
```

DELKA je synonymum pro int. Verze 1.23 kompilátoru SOFT - C nemá však příkaz typedef implementován.

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 popiseme standardní "knihovnu vstupů a výstupů" - soubor funkcí, které byly navrženy pro standardní I/O systém jazyka C. Funkce byly navrženy tak, aby umožňovaly pohodlné programování. Funkce 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 programu v jazyku C, než abychom se soustředili na to, jak programy spolupracují se svým okolím.

7.1. Přístup do standardní knihovny

Pokud v programu používáte funkce ze standardní knihovny (případně z vlastní knihovny), musíte jméno knihovny uvést v příkazovém řádku pro spojovací program L80.

V jazyku C musí každý zdrojový soubor, který má nějakým způsobem přistupovat k souborům na disku, obsahovat řádek

```
#include <stdio.h>
```

na začátku souboru. V souboru stdio.h jsou definice, které jsou standardní I/O knihovnou využívány.

7.2. Standardní vstup a výstup - getchar a putchar

Nejjednodušším mechanismem načítání je čtení znaků ze "standardního vstupu", obecně klávesnice uživatelova terminálu, funkcí getchar (). Funkce getchar () vrací další znak ze vstupu pokaždé, když je vyvolána.

Při výstupu funkce putchar (c) "zapíše" znak c do "standardního výstupu", což je standardně rovněž terminal.

Překvapivě velké množství programů čte pouze z jednoho vstupu a zapisuje na jeden výstup. Pro tyto programy jsou funkce getchar, putchar a printf zcela postačující.

Uvažujme např. program t o l o w e r , který mění velká písmena ze vstupu na malá:

```
main ()         /* konvertování velkých písmen na malá */

{
        int c;

        while ((c = getchar ()) != EOF)
        putchar (isupper (c) ? tolower (c) : c);
}
```

Funkce isupper a tolower jsou funkce z knihovny libc. Funkce isupper testuje, zda její argument je velké písmeno. Pokud je argument velké písmeno, vrací nenulovou hodnotu, v opačném případě vrací nulu. Funkce tolower konvertuje velká písmena na malá. Nehledě na to, jak jsou tyto funkce implementovány na různých počítačích, je efekt vždy a všude stejný a program se nemusí starat o soubor znaků na daném počítači.

7.3. Formátovaný výstup - printf

Dvě rutiny - printf pro výstup a scanf pro vstup (bude popsáno v příštím odstavci) - umožňují převod čísel na znakovou reprezentaci. Umožňují rovněž zápis a čtení formátovaných řádků.

Funkci printf jsme užívali v předchozích kapitolách, aniž bychom znali její přesný popis. Zde uvedeme kompletnější a přesnější definici

```
printf (ridici_retezec,arg1,arg2,...)
```

Funkce printf provádí konverze čísel a formátový výstup do standardního výstupu. Formát je udáván řídícím řetězcem. 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 "-" způsobuje zarovnání argumentů na pozici doleva.

Číslo udává 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 specifikuje 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 nuly.

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

Jestliže znak za znakem % není znakem konverze, tak je normálně vytištěn. Znak % může být vytisknut takto: %%.

Většina formátu 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 "ahoj, svete", který má 11 znaků. Řetězec jsme v příkladech ohraničili znaky:

```
:%10s:          :ahoj, svete:
:%-10s:         :ahoj, svete:
:%20s:          :          ahoj, svete:
:%-20s:         :ahoj, svete          :
:%20.10s:       :          ahoj, svet:
:%-20.10s:      :ahoj, svet           :
:%.10s:         :ahoj, svet:
```

V a r o v á n í : první argument funkce printf udává, kolik argumentů následuje a jakého jsou typu. Jestliže je argumentů méně, nebo jsou nesprávného typu, bude výstup nesmyslný.

C v i č e n í 7 - 1:

Napište program, který bude tisknout údaje ze vstupu rozumným způsobem. Minimálně by měl nepísmenové a nemezerové znaky tisknout v osmičkové nebo šestnáctkové soustavě a rozdělovat dlouhé řádky.

7.4. Formátový vstup - scanf

Funkce scanf je analogická funkci printf. Provádí obdobné konverze opačným směrem.

```
scanf (ridici_retezec,arg1,arg2,...)
```

Funkce scanf čte znaky standardního vstupu, interpretuje je podle formátových specifikací uvedených v řídícím řetezci a ukládá do argumentů. Řídící řetězec bude popsán dále. Ostatní argumenty, z n i c h ž k a ž d ý m u s í b ý t p o i n t e r určují, na které místo bude ukládán konvertovaný vstup. Řídící řetězec obvykle obsahuje znaky konverzních specifikací, které jsou používány k příme konverzi. Řídící řetězec může obsahovat:

mezery, tabelátory nebo znaky pro nový řádek, tedy "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 nový řádek jsou pro tuto funkci normálními oddělovači.

Znak typu konverze určuje interpretaci vstupního pole. Odpovídající argument musí být pointer. Následující konverze jsou povoleny:

d je očekáváno číslo v desítkovém zápisu. Odpovídající argument by měl být pointer na číslo int

o je očekáváno číslo v osmičkovém tvaru. Odpovídající argument musí být pointer na int.

x je očekáváno číslo v hexadecimálním tvaru. Odpovídající argument musí být pointer na int.

c je očekáván jeden znak, odpovídající argument má být pointer 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 pointer na pole znaků dostatečně velké, aby se do něho vešel celý vstupní řetězec včetně ukončovacího znaku \0.

[vstupní hodnota je interpretována jako řetězec znaků. Odpovídající argument by měl být pointer na pole znaků dostatečně velké i pro uložení znaků \0. Znak [(levá hranatá závorka) je následován řadou znaků ukončených znakem] (pravá hranatá závorka). Jestliže prvním znakem této řady znaků je znak "stříška" (jde o znak totožný se znakem používaným pro exkluzívní OR viz 2. kap.), bude vstupní pole ukončeno prvním znakem, který je různý od znaků uvedených mezi levou a pravou hranatou závorkou. Jestliže je prvním znakem za [znak "stříška", bude vstupní pole zakončeno prvním znakem, který je totožný s některým znakem uvedeným mezi [a]. Znak "stříška", který následuje bezprostředně za levou hranatou závorkou, se neporovnává.

Např. volání

```
int i;
char x;
char jmeno [50];

scanf ("%d%c%s",&i,&x,jmeno);
```

Se vstupem ve tvaru

```
25      a       Novak
```

přiřadí proměnné i hodnotu 25, x='a' a poli jmeno řetězec Novak\0. Tato tři vstupní pole mohou být od sebe oddělena libovolně mnoha oddělovači.

V o l á n í

```
int i;
int x;
char jmeno [50];

scanf ("%2d%d%*d%2s",&i,&x,jmeno);
```

se vstupem ve tvaru

```
56789 0123 45 a72
```

přiřadí proměnné i=56, x=789, přeskočí 0123 a umístí řetězec "45" do jmeno. Další vyvolaná vstupní funkce začne načítání od písmene a. V těchto dvou příkladech je j m é n o pole a proto nemusí být označeno znakem &.

V dalším příkladu přepíšeme program pro kalkulátor z kapitoly 4 s použitím funkce scanf:

```
main()                  /* jednoduchý stolní kalkulátor */
                        /* pouze pro sčítání */
{
        int sum,v;

        sum=0;
        while (scanf("%d",&v) != EOF)
                printf("\t%d\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.

Z á v ě r e č n é v a r o v á n í: 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,ridici_retezec,arg1,arg2,...)
sscanf (string,ridici_retezec,arg1,arg2,...)
```

Funkce 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 (jmeno,"ahoj%d",n);
```

vytváří řetězec ve tvaru "ahojnnn" v poli jmeno, nnn je hodnota proměnné n.

Funkce sscanf provádí opačnou konverzi - prohledává řetězec podle formátové specifikace v řídícím řetězci a hodnoty ukládá do arg1, arg2, což opět musí být pointry.

Volání

```
sscanf (jmeno,"ahoj%d",&n);
```

nastavuje n na hodnotu, která je uvedena v poli jmeno za nápisem ahoj.

7.6. Přístup k souborům

Nyní napíšeme program, který pracuje se soubory, které k němu nejsou operačním systémem implicitně přiřazeny. Jedním takovým programem je program cat, který řetězí skupinu vyjmenovaných souborů na standardní výstup. Program cat je používán pro tisk souborů na terminal 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 souborů 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 knihovní funkce fopen. Funkce 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 pointer (který se nazývá p o i n t e r n a s o u b o r) 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 nazvané FILE.

Jediná potřebná deklarace je

```
FILE *fopen (), *fp;
```

Definice říká, že fp je pointer na FILE a fopen vrací pointer na FILE. Uvědomme si, že FILE je název typu zrovna tak jako je int a není to vlastně název struktury.

(Podrobnosti, jak vše pracuje pod systémem CP/M jsou dány v kapitole 8.)

Skutečné volání fopen vypadá takto

```
fp = fopen (jmeno,mode,bufsiz);
```

Prvním argumentem funkce fopen je n á z e v s o u b o r u (jmeno), 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"). Třetí argument (bufsiz) udává požadovanou velikost vyrovnávací paměti (většinou se uvádí číslo 128 - délka sektoru na disku).

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,který vrací funkce 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 fgetc a fputc jsou nejjednodušší možnosti. Jako argument je zadán pointer na soubor.

Příkaz

```
c = fgetc (fp);
```

vrací znak ze souboru, na který ukazuje pointer fp. Je-li dosaženo konce souboru, vrací ERROR.

Funkce fputc je inverzí fgetc:

```
fputc (c,fp);
```

Zapisuje znak c do souboru fp a vrací SUCCESS (tj. 0) v případě úspěšného zápisu, v opačném případě ERROR.

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 pointer na soubor a řídící_řetězec je až druhým parametrem.

Nyní můžeme přistoupit k napsání programu cat. Základní princip je použitelný pro mnoho dalších programů: jsou-li zadány argumenty v příkazovém řádku, jsou brány popořádku. Tento program může být použit samostatně, nebo může být částí většího celku

```
#include <stdio.h>
#define BUF     128
#define EOF      26

main (argc, argv)       /*cat:spojování souborů*/

        int argc;
        char *argv [];
{
        FILE *fp, *fopen();

        if(argc == 1)                   /* žádné argumenty  */
                printf ("nebyl zadan soubor");
        else
                while(--argc > 0)
                {
                        if((fp=fopen(*++argv,"r",BUF)) == NULL)
                        {
                                printf("nejde otevrit %s\n",
                                                        *argv);
                                break;
                        }
                        else
                        {
                                filecopy(fp);
                                fclose (fp);
                        }
                }
}

filecopy (fp) /* zkopírování fp do standardního vystupu */

        FILE *fp;
{
        int c;
        while ((c=fgetc(fp)) != EOF && c != ERROR)
                putchar(c);
}
```

Funkce fclose je inverzní funkce fopen. Přerušuje spojení mezi externím názvem souboru a pointrem. Pointer může být dále použit pro jiný soubor. Protože každý operační systém má omezení počtu současně otevřených souborů, je výhodné používat ty pointry, které už nejsou potřeba - jako jsme to udělali v programu cat. Pro použití funkce fclose je další důvod: uvolňuje vyrovnávací paměť, kterou používá fputc (funkce fclose je automaticky vyvolána pro každý otevřený soubor, když program končí normálně svoji cinnost).

7.7. Ošetřování chyb - exit

Přepíšeme nyní program cat s použitím standardní funkce exit, která ukončuje činnost programu, je-li vyvolána.

```
#include <stdio.h>
#define BUF     128

main (argc, argv)       /*cat:spojování souborů*/

    int argc;
    char *argv [];

{
    FILE *fp, *fopen();

    if (argc == 1)  /* žádné argumenty  */
        printf ("nebyl zadan soubor");
    else
        while(--argc > 0)
        {
            if((fp=fopen(*++argv,"r",BUF)) == NULL)
            {
                printf("nejde otevrit %s\n",
                                        *argv);
                exit(1);
            }
            else
            {
                filecopy(fp);
                fclose(fp);
            }
        }
        exit(0);
}
```

Chybové hlášení jde na terminál.

Funkce exit volá fclose pro každý otevřený výstupní soubor. Obsah vyrovnávací paměti je přitom připsán do souboru a je vyvolána funkce - exit. Funkce - exit okamžitě ukončuje činnost programu. Může být rovněž přímo vyvolána.

7.8. Řádkový vstup a výstup

Standardní knihovna obsahuje funkci fgets, která je obdobou funkce getline, kterou jsme používali v této knize.

P ř í k a z

```
fgets(line,MAXLINE,fp);
```

čte další řádek (včetně znaku pro nový řádek) ze souboru fp do znakového pole line. Maximálně je načteno MAXLINE - 1 znaků. Řádek je ukončen znakem \0. Fgets v případě úspěšného čtení vrací řádek. V opačném případě, tedy též na konci souboru, vrací NULL (naše funkce getline vrací délku řádku a nulu pro konec souboru).

Funkce fputs píše řetězec znaků (řetězec nemusí obsahovat znak pro nový řádek, ale musí být zakončen znakem '\0') do souboru příkazem

```
fputs(line,fp);
```

Abychom ukázali, že na funkcích fgets a fputs není nic magického, tak zde uvedeme jejich zdrojový tvar.

```
#include <stdio.h>
#define EOF      26

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 && c != ERROR )
        {
                if((*cs++=c) == '\n')
                {
                        *cs='\0';
                        return s;
                }
        }
        *cs = '\0';
        return ( ((c == EOF || c == ERROR ) && *s == '\0' ) ?
                                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);
}
```

Standardní knihovna obsahuje též funkce gets a puts pro řádkový vstup a výstup, které jsou analogicke funkcím fgets a fputs.

Příkaz

```
gets (ret);
```

čte řádek ze standardního vstupu do řetězce ret, který musí být dostatečně velký, neboť hrozí přepsání části paměti. Funkce gets nahrazuje znak '\n' nebo "\r\n" znakem '\0'.

Příkaz

```
puts (ret);
```

píše obsah řetězce ret až po znak '\0' na standardní vystup. Řetězec ret v tomto případě může obsahovat take řídící znaky jako '\n', '\t' apod.

C v i č e n í 7 - 2:

Napište program, který porovnává dva soubory a tiskne první řádek a pozici, kde se soubory liší.

C v i č e n í 7 - 3:

Modifikujte program pro nalezení řetězce z kapitoly 5, aby četl vstup ze vstupních souborů, které budou zadány jako jeho argumenty. Jestliže nebyl zadán žádný argument, bude číst ze standardního vstupu. Může být vytištěno jméno souboru, ve kterém je řetězec nalezen?

C v i č e n í 7 - 4:

Napište program, který tiskne skupinu souborů, každý na novou stránku. Tiskněte také nazev souboru 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.

Testování znaků a konverzní funkce

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

Ve standardní knihovně je omezena verze naší funkce ungetch, kterou jsme napsali v kapitole 4. Příkaz

```
ungetc(c,fp);
```

vrací znak zpět do souboru fp. Ungetc může být použita pouze jednou mezi použitím funkcí vstupu jako jsou fscanf, fgetc nebo fgets.

KAPITOLA 8: SYSTÉM. SOUVISLOSTI S OP. SYSTÉMEM CP/M

Tato kapitola se zabývá souvislostmi jazyka C a operačního systému CP/M. Protože mnoho uživatelů jazyka C pracuje na systémech typu CP/M, bude tato kapitola užitečná většině čtenářů. Dokonce i těm uživatelům, kteří pracují s jinými systémy, bude užitečné pro pochopení programování v jazyku C prostudování příkladů uvedených v této kapitole.

Kapitola je rozdělena do tří hlavních částí:

- vstup a výstup
- systém souborů
- přidělování paměti

Prvé dvě části předpokládají minimálně znalost vnějších vlastností systému CP/M.

Kapitola 7 se zabývala systémovym 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 CP/M a ukážeme si, jak s jejich pomocí mohou být implementovány některé části standardní knihovny.

8.1. Deskriptory souborů

V operačním systému CP/M jsou všechny vstupy a výstupy uskutečňovány pomocí čtení nebo zápisu do souboru, protože téměř všechna periferní zařízení jsou chápána jako soubory ze systému souborů.

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 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.

8.2. Vstup a výstup nejnižší úrovně - read a write

Nejnižší úroveň vstupu do souboru a výstupu ze souboru v operačním systému CP/M nevykonává žádné bufferování ani jiné služby. Nejnižší úroveň vstupu do souboru má funkce write, nejnižší úroveň výstupu ze souboru má funkce read. 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:

```
načteno = read (fd,buf,n);

zapsáno = write (fd,buf,n);
```

Každé vyvolání výše uvedených funkcí vrátí počet slabik, které byly skutečně systémem přeneseny. Při čtení může být toto číslo nižší, než-li 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 vrácena 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 128 slabik (tato hodnota odpovídá fyzikální délce bloku na mnoha periferních zařízeních).Druhý případ je efektivnější, ale ani první není zcela nevhodný.

8.3. Funkce open, creat, close, unlink

Pomocí funkce creat lze založit nový soubor. Příkazem

```
fd = creat (jmeno,mod);
```

založíme na disketě soubor jmeno. Tento soubor bude otevřen pro zápis i čtení v tom případě, že hodnota proměnné mod bude 0x80 (hexadecimální zápis), v opačném případě bude otevřen pouze pro čtení. Pokud bude soubor úspěšně založen, vrátí funkce creat deskriptor souboru. Argument jmeno je pointer na char, argument mod je typu unsigned.

Funkce open otevírá již existující soubor a v případě úspěšného otevření vrací deskriptor souboru. Příkazem

```
open (jmeno,mod);
```

otevřeme soubor jmeno pro čtení, jestliže je argument mod roven 0, pro zápis, jestliže mod = 1, a pro čtení i zápis pro mod = 2. Stejně jako funkce creat ani funkce open nevykonává žádné bufferování.

Použití funkcí creat a open budeme ilustrovat na jednoduchém programu, který kopíruje jeden soubor do druhého.

```
#define ERROR   (-1)
#define VEL_BUF 128     /* vhodná velikost bufferu pro CP/M */
#define NULL    0

main (argc,argv)        /* kopíruje 1.soubor do 2.souboru */

        int argc;
        char *argv [];

{
        char buf [VEL_BUF];
        int n;
        unsigned fd1, fd2;

        if ( argc != 3 )
                error ("\nSpatne uziti programu \n",NULL);
        if ((fd1 = open (* ++ argv,0)) == ERROR )
                error ("\n nejde otevrit soubor %s\n",* argv);
        if ((fd2 = creat (* ++ argv,INIT_DMA)) == ERROR )
                error ("\n nejde zalozit soubor %s\n",* argv);
        while ((n = read (fd1,buf,VEL_BUF)) > 0)
                if ( write (fd2,buf,n) != n )
                        error ("\nchyba pri zapisu\n",NULL);
        exit (0);
}

error (s1,s2)           /* tiskne chybovou zprávu */

        char *s1, *s2;

{
        printf (s1,s2);
        exit (1);
}
```

Existuje maximální počet současně otevřených souborů, většinou 15 až 25. Program, který používá mnoho souborů, musí být schopen vícenásobného použití deskriptoru souboru. Funkce close ukončí spojení mezi deskriptorem souboru a otevřeným souborem. Tím se uvolní deskriptor souboru pro použití s dalším souborem. Ukončení programu funkcí exit uzavře všechny otevřené soubory.

Funkce unlink (jmeno) zruší soubor se jménem jmeno v systému souborů.

8.4. Přímý přístup - SEEK

Normální přístup k souborům je sekvenční. Každý nový zápis nebo čtení ze souboru se uskuteční na následující pozici v souboru. Pokud to situace vyžaduje, je možné soubory číst nebo do nich zapisovat na libovolné pozici. Systémová funkce seek umožňuje přístup do libovolného místa souboru, aniž by bylo nutné soubor skutečně číst, či do něho zapisovat.

```
seek (fd,offset,origin);
```

Funkce seek nastaví pozici v souboru, který je určený deskriptorem souboru fd, na místo určené posunutím offset, který je určován od místa v souboru, které je určené argumentem origin. Následující čtení nebo zápis do souboru se uskuteční na této pozici. Argumenty offset a origin jsou typu int. Argument origin může nabýt hodnoty 0, 1 a 2, která určuje, že posunutí je měřeno od počátku, od právě aktuální pozice v souboru nebo od konce souboru. Např. nastavení na konec souboru (append) se uskuteční následovně:

```
seek (fd,0,2);
```

Nastavení na začátek souboru (rewind):

```
seek (fd,0,0);
```

S pomocí funkce seek 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  pos, n;
        char *buf;
        unsigned fd;

{
        seek (fd,pos,0);    /* nastav ukazatel na pozici pos */
        return read (fd,buf,n);
}
```

Protože u systému TNS mají čísla typu int velikost 16 bitů, je možná největší hodnota argumentu offset 32767. Proto existují další možné hodnoty argumentu origin (3, 4, 5), které způsobují vynásobení hodnoty offset konstantou 512, 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. Prvé určí blok a druhé, jehož argument origin má hodnotu 1, určí požadovanou slabiku uvnitř bloku.

8.5. Výpis adresáře

Rozdílnou úlohou při práci se soubory je získávaní informací o souboru, nikoliv práce s jeho obsahem. Příkaz operačního systému CP/M dir (directory = adresář) vytiskne jména souborů v adresáři.

V operačním systému CP/M je adresář disku vždy umístěn v počátečních alokačních blocích. Systém ovládání souborů považuje alokační bloky adresáře za obsazené. Položka adresáře je tvořena popisem jednoho souboru na daném disku. Položka adresáře zabírá 32 bytů. Tedy do jednoho sektoru adresáře se vejdou 4 položky. Položka obsahuje příznak obsazení položky, jméno a verzi souboru, počet sektorů souboru a čísla alokačních bloků, které byly souboru přiřazené. Pro delší soubory je v adresáři rezervována další položka, která popisuje toto další rozšíření (extension) souboru.

Jestliže soubor zrušíme, tak se v adresáři do příznaku obsazenosti položky daného souboru zapíše 0E5H. Místo v adresáři, které tato položka zabírá, může být využito pro popis jiného souboru.

Programátor, který pracuje s diskovým souborem, se na tento soubor odvolává pomocí popisovače souboru. Popisovač souboru je vlastně struktura se jménem filedesc, jejíž definice je obsazena v souboru STDIO.H . Položky struktury filedesc jsou v podstatě stejně organizovány jako položka v adresáři k danému souboru.

Je celkem jasné, že existují i jiné možnosti, jak získat některé informace o souborech.

8.6. Příklad - Přidělování paměti

V kapitole 5 byla uvedena jednoduchá verze funkce alloc. Verze, která bude nyní uvedena, má poněkud větší 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 na tomto vybavení. Dále ukazují užití struktur a unionů.

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 zřeřezeny v pořadí stoupající velikosti adresy operační paměti. Poslední blok (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 velký blok paměti. Jestliže má blok požadovanou velikost, je ze seznamu volných bloků vyřazen a přidělen uživateli. Jestliže je blok příliš velký, je rozdělen a přiměřená část je přidělena uživateli, zbývající část 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 bloku paměti způsobuje prohledávání seznamu volných bloků proto, aby bylo nalezeno vhodné místo pro začlenění právě uvolněného bloku paměti. Jestliže právě uvolňovaný blok paměti přiléhá k jinému bloku, 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 5. kapitole, je zajištění toho, aby blok paměti, přidělený funkci 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 objektů typu double. Na systému TNS 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 hlavička (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 nejvíce omezený typ.

```
union header                        /* volná hlavička bloku */
      {       struct
              {
                      union header *ptr;
                              /* následující volný blok */
                      unsigned vel;
                        /* velikost tohoto volného bloku */
               } s;
               int x;                     /* přiřaď správně blok */
      } *hdr;
```

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 vel hlavičky. Ukazovátko vrácene funkcí alloc, však ukazuje na počátek volné paměti, nikoliv na hlavičku bloku volné paměti.

```
#define HEADER          union header
#define NULL            0

static  HEADER  base;           /* počáteční prázdný seznam */
static  HEADER  *allocp;        /* naposledy přidělený blok */

char *alloc (nbytes)    /*základní funkce přidělování paměti*/

        unsigned nbytes;

{
        register int nunits;
        register HEADER *p, *q;
        HEADER *morecore ();

        nunits = 1 + (nbytes + sizeof(hdr) - 1) / sizeof(hdr);
        if (( p = allocp) == NULL)      /* seznam volných bloků
                                                neexistuje */
        {
                base.s.ptr = allocp = q = &base;
                base.s.vel = 0;
        }
        for (p = q -> s.ptr; ; q = p -> s.ptr)
        {
                if (p -> s.vel >= nunits)       /* je větší */
                {
                        if (p -> s.vel == nunits)
                                                /* je roven */
                                q -> s.ptr = p -> s.ptr;
                        else            /* přiděl konec bloku */
                        {
                                p -> s.vel -= nunits;
                                p += p -> s.vel;
                                p -> s.vel = 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 má ukazovátko allocp hodnotu NULL, provádí se prvé 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, to 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 vel udávající velikost volné paměti bloku. Ve všech případech je však uživateli vráceno ukazovátko na skutečně volnou paměť, která počíná jednu jednotku velikosti hlavičky bloku za počátkem bloku (tj. za hlavičkou). Poznamenejme, že proměnná p je konvertována na proměnnou typu ukazovátko, před jejím vrácením funkcí alloc.

Před prvním použitím funkce alloc je nutné použít funkci inial().

```
inial ()

{
        allocp = NULL;
}
```

Funkce morecore obdrží volnou paměť od operačního systému. Podrobnosti o tom, jak ji obdrží, se samozřejmě liší na různých systémech. V systému CP/M 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 náročnější operace, není žádoucí, aby každé volání funkce alloc používalo služby operačního systému. Proto funkce morecore zaokrouhlí počet žádaných jednotek volné paměti na větší hodnotu. Takový blok potom může být znovu podle potřeby přidělován. Pravidlo zaokrouhlení je parametr, který může být nastaven tak, jak je žádoucí.

```
#define NALLOC 128     /* počet jednotek najednou přidělených */

static HEADER *morecore (nu)  /* žádej systém o volnou paměť */

        unsigned nu;

{
        register char *cp;
        register HEADER *up;
        register int rnu;
        char *sbrk ();

        rnu = NALLOC * ((nu + NALLOC - 1) / NALLOC);
        cp = sbrk(rnu * sizeof(hdr));
        if ((int) cp == -1)          /* není žádná volná paměť */
                return (NULL);
        up = (HEADER*) cp;
        up -> s.vel = 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 byt porovnána.

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 volné bloky, které spolu sousedí, budou spojeny v blok jediný. Jediná obtíž programu je udržet ukazovátka tak, aby ukazovala na správná místa a aby 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.vel == q -> s.ptr) /* připoj k následu.
                                                bloku */
        {
                p -> s.vel += q -> s.ptr -> s.vel;
                p -> s.ptr = q -> s.ptr -> s.ptr;
        }
        else
                p -> s.ptr = q -> s.ptr;
        if (q + q -> s.vel == p)   /* připoj k předch. bloku */
        {
                q -> s.vel += p -> s.vel;
                q -> s.ptr = p -> s.ptr;
        }
        else
                q -> s.ptr = p;
        allocp = q;
}
```

Ačkoliv je přidělování paměti v podstatě závislé na technickém vybavení systému, výše uvedený program ukazuje, jak tyto závislosti mohou být řízeny a svěřeny nevelké části programu. Užití konstrukce union je vhodné pro řízení přidělení správného úseku paměti (za předpokladu, že funkce sbrk vrátí správně ukazovátko).

Vyřešená cvičení zadávaná v textu učebnice

CVIČENÍ: 1 - 3

```
/* tisk tabulky druhých mocnin pro základ = 0, 1, ..., 10 */

int dolni_mez, horni_mez, krok;

main ()

{
        int zaklad, mocnina;

        dolni_mez = 0;          /* dolní mez tabulky     */
        horni_mez = 10;         /* horní mez tabulky     */
        krok = 1;               /* hodnota kroku         */
        zaklad = dolni_mez;

        printf ("Tabulka druhych mocnin");
        printf ("\nZaklad\tDruha mocnina\n");

        while (zaklad <= horni_mez)
        {
                mocnina = zaklad * zaklad;
                printf ("%3d\t%4d\n",zaklad,mocnina);
                zaklad = zaklad + krok;
        }
}
```

CVIČENÍ: 1 - 4

```
/* tisk tabulky druhých mocnin pro základ = 0, 1, ..., 10 */

int dolni_mez, horni_mez, krok;

main ()

{
        int zaklad, mocnina;

        dolni_mez = 0;          /* dolní mez tabulky     */
        horni_mez = 10;         /* horní mez tabulky     */
        krok = 1;               /* hodnota kroku         */
        zaklad = horni_mez;

        printf ("Tabulka druhych mocnin");
        printf ("\nZaklad\tDruha mocnina\n");

        while (zaklad >= dolni_mez)
        {
                mocnina = zaklad * zaklad;
                printf ("%3d\t%4d\n",zaklad,mocnina);
                zaklad = zaklad - krok;
        }
}
```

CVIČENÍ: 1 - 5

```
main ()         /* tabulka výpočtu druhých mocnin */

{
        int zaklad,i;

        for (zaklad = 0; zaklad <= 10; zaklad = zaklad + 1)
                printf("%3d\t%5d\t%4d\n", zaklad, i=zaklad *
                        zaklad, i/2);
}
```

CVIČENÍ: 1 - 6

```
#define EOF     -1      /* indikace konce souboru */

main ()         /* zjistí počet mezerových znaků v textu */

{
        int c;
        unsigned poc_mezer,poc_tab,poc_radku;

        poc_mezer=0;
        poc_tab=0;
        poc_radku=0;

        while ((c=getchar())!=EOF)
        {
                if (c==' ')
                        ++poc_mezer;
                if (c=='\t')
                        ++poc_tab;
                if (c=='\n')
                        ++poc_radku;
        }

        printf ("Pocet mezer, tabelatoru a radku :\n");
        printf ("%d\t%d\t%d",poc_mezer,poc_tab,poc_radku);
}
```

CVIČENÍ: 1 - 7

```
#define ANO      1
#define NE       0
#define EOF     -1

main ()         /* tiskni místo skupiny mezer pouze jednu */

{
        int c;
        unsigned byla_mezera;

        byla_mezera=NE;         /* indikace předchozího znaku */
```

```
        while ((c=getchar())!=EOF)
        {
                if (byla_mezera==NE)
                {
                        putchar (c);
                        if (c==' ')
                                byla_mezera=ANO;
                }
                else
                        if (c!=' ')
                        {
                                putchar (c);
                                byla_mezera=NE;
                        }
        }
}
```

CVIČENÍ: 1 - 8

```
#define EOF     -1

main ()         /* zobraz znaky pro tabelaci a zpětný znak */

{
        int c;

        while ((c=getchar())!=EOF)
        {
                if (c!='\t'&&c!='\b')
                        putchar (c);
                else
                {
                        if (c=='\t')
                                printf (">-");
                        else
                                printf ("<-");
                }
        }
}
```

CVIČENÍ: 1 - 10

```
#define ANO      1
#define NE       0
#define EOF     -1

main ()         /* piš kazdé slovo na nový řádek */

{
        int c;
        unsigned byl_oddelovac;
```

```
        byl_oddelovac=NE;
        while ((c=getchar())!=EOF)
        {
                if (c!=' '&&c!='\n'&&c!='\t')
                {
                        if (byl_oddelovac==ANO)
                                byl_oddelovac=NE;
                        putchar (c);
                }
                else            /* oddělovače */
                        if (byl_oddelovac==NE)
                        {
                                putchar ('\n');
                                byl_oddelovac=ANO;
                        }
        }
}
```

CVIČENÍ: 1 - 11

```
#define ANO      1
#define NE       0
#define EOF     -1

main ()         /* zjisti počet slov */

{
        int c;
        int pocet_slov,v_slove;

        pocet_slov=0;
        v_slove=NE;

        while ((c=getchar())!=EOF)
        {
                if (v_slove==NE&&((c>='a'&&c<='z')||
                    (c>='A'&&c<='Z')))
                {
                        ++pocet_slov;
                        v_slove=ANO;
                }
                else
                        if (!(c>='0'&&c<='9')&&!((c>='a'&&
                            c<='z')||(c>='A'&&c<='Z')))
                                v_slove=NE;
        }

        printf ("Pocet slov = %d",pocet_slov);
}
```

CVIČENÍ: 1 - 12

```
#define EOF     (-1)
#define MEZ     10      /* maximální délka slova */
```

```
main ()         /* zjisti počet slov různých délek */

{
        int c,i;
        unsigned pocet_znaku,slovo[MEZ];
        unsigned slovo[MEZ];    /* pole slov délek i až MEZ */

        for (i=0;i<MEZ;i++)
                slovo[i]=0;
        pocet_znaku=0;

        while ((c=getchar())!=EOF)
        {
                if (c==' '||c=='\n'||c=='\t')
                {
                        if (pocet_znaku>0)
                        {
                                if (pocet_znaku>MEZ)
                                        pocet_znaku=MEZ;
                                ++slovo[pocet_znaku-1];
                                pocet_znaku=0;
                        }
                }
                else
                        ++pocet_znaku;
        }

        printf ("\nDelka slova\tPocet slov");
        for (i=1;i<=MEZ;i++)
                printf ("\n%d\t\t%d",i,slovo[i-1]);
}


        CVIČENÍ:        1 - 13
        ----------------------

#define EOF     (-1)

main ()         /* převeď velká písmenna na malá */

{
        int c;

        while ((c=getchar())!=EOF)
                putchar (tolower(c));
}


        CVIČENÍ:        1 - 14
        ----------------------

#define EOF     (-1)
#define MAX     20

main ()         /* zjisti skutečnou délku řádku, vytiskni
                   nejvýše MAX znaků */
```

```
{
        int delka;
        char radek[MAX+1];

        while ((delka=getline(radek,MAX))>0)
                printf ("%d\n%s\n",delka,radek);

}

getline (s,lim)

        char s[];
        int lim;
{
        int c,i;

        for (i=0;(c=getchar())!=EOF;i++)
        {
                if (i<lim)
                        s[i]=c;
                if (c=='\n')
                {
                        if (i==0)   /* prázdný řádek vynechá */
                                --i;
                        else
                        {
                                s[minim(i,lim)]='\0';
                                return (i);
                        }
                }
        }
        if (i>0)
                s[minim(i,lim)]='\0';
        return (i);
}

minim (cisi,cis2)                   /* vrací menší ze dvou čísel */

        int cisi,cis2;
{
        if (cisi<=cis2)
                return (cisi);
        else
                return (cis2);
}
```

CVIČENÍ: 1 - 15

```
#define EOF     (-1)
#define MIN     40
#define MAX     65

main ()         /* tiskni řádky delší než MIN - nejvýše
                   MAX znaků */
```

```
{
        int delka;
        char radek[MAX+1];

        while ((delka=getline(radek,MAX))>0)
                if (delka>MIN)
                        printf ("%s\n",radek);
}

getline (s,lim)

        char s[];
        int lim;

{
        int c,i;

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

CVIČENÍ: 1 - 16

```
#define ANO      1
#define NE       0
#define EOF     (-1)

main ()         /* vymaž mezery a tabelátory na počátku řádku
                   a zruš prázdné řádky */
{
        char c;
        unsigned v_radku;       /* indikace zda minulý znak
                                   byl uvnitř řádku */
        v_radku=NE;

        while ((c=getchar())!=EOF)
        {
                if (v_radku==NE)
                {
                        if (c!=' '&&c!='\t'&&c!='\n')
                                /* nový řádek */
                        {
                                v_radku=ANO;
                                putchar (c);
                        }
                }
                else
                {
                        if (c=='\n')
                                /* konec řádku */
                                v_radku=NE;
                        putchar (c);
                }
        }
}
```

CVIČENÍ: 1 - 17

```
#define EOF     -1
#define MAX     65

main ()         /* převrať vstupní řádky */

{
        int delka;
        char radek[MAX+1];

        while ((delka=getline(radek,MAX+1))>=0)
        {
                reverse(radek,delka);
                printf ("%s\n",radek);
        }
}
```

CVIČENÍ: 1 - 22

```
#define ANO      1
#define NE       0
#define EOF     (-1)

main ()         /* vynechej všechny komentáře */
{
        int c;
        unsigned v_koment;
        unsigned kom_znak;      /* znaky komentáře : / a * */

        v_koment=NE;
        kom_znak=NE;    /* zda minulý znak  byl /,resp.* */
        while ((c=getchar())!=EOF)
        {
                if (v_koment==NE)       /* vně komentáře */
                {
                        if (c!='/'&&c!='*')
                        {
                                if (kom_znak==ANO)
                                {
                                        putchar ('/');
                                        kom_znak=NE;
                                }
                                putchar (c);
                        }
                        else            /* znak komentáře */
                        {
                                if (c=='/')
                                        if (kom_znak==ANO)
                                                putchar ('/');
                                        else
                                                kom_znak=ANO;
                                else
                                {
```

```
                                        if (kom_znak==ANO)
                                        { /* počátek koment. */
                                                v_koment=ANO;
                                                kom_znak=NE;
                                        }
                                        else
                                                putchar ('*');
                                }
                        }
                }

                else            /* uvnitř komentáře */
                {
                        if (c!='/'&&c!='*')
                                if (kom_znak==ANO)
                                        kom_znak=NE;
                                else;
                        else
                        {
                                if (c=='*')
                                        if (kom_znak==NE)
                                                kom_znak=ANO:
                                        else;
                                else
                                        if (kom_znak==ANO)
                                        {   /* konec koment. */
                                                v_koment=NE;
                                                kom_znak=NE;
                                        }
                        }
                }
        }
}
```

CVIČENÍ: i - 23

```
#define ANO     1
#define NE      0
#define EOF     (-1)
#define APOS    39      /* dekadický kód apostrofu */

main ()         /* zkontroluj počet závorek, uvozovek,... */

{
        int c;
        int kul,hran,sloz,apo,uvoz,koment;
        unsigned v_apo,v_uvoz,v_koment;
        unsigned kom_znak;      /* zda před '*' byl '/',resp.
                                   před '/' byl '*' */

        kul= hran= sloz= apo= uvoz= koment=0;
        v_apo= v_uvoz= v_koment= kom_znak=NE;

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

```
if (v_apo==NE&&v_uvoz==NE&&v_koment==NE)
        /* vne apostrofu, uvoz. a komentaru */
{
        if (c!='*'&&kom_znak==ANO)
                kom_znak=NE;
        if (c=='('||c=='['||c=='{'||c==')'||
            c==']'||c=='}'||c==APOS||c=='"'||
            c=='/'||c=='*')
        {
                if (c=='(')
                        ++kul;
                if (c=='[')
                        ++hran;
                if (c=='{')
                        ++sloz;
                if (c==')')
                        --kul;
                if (c==']')
                        --hran;
                if (c=='}')
                        --sloz;
                if (c==APOS)
                        /* počátek apostrofu */
                {
                        ++apo;
                        v_apo=ANO;
                }

                if (c=='"')
                        /* počátek uvozovek */
                {
                        ++uvoz;
                        v_uvoz=ANO;
                }
                if (c=='/')
                        kom_znak=ANO;
                if (c=='*'&&kom_znak==ANO)
                        /* počátek komentáře */
                {
                        ++koment;
                        kom_znak=NE;
                        v_koment=ANO;
                }
        }
}
else
   /* uvnitř komentáře, mezi apostrofy a mezi
   uvozovkami se až do uzavření ostatní
   znaky ignorují */
{
        if (c!='/'&&kom_znak==ANO)
                kom_znak=NE;
        if (c==APOS||c=='"'||c=='/'||c=='*')
        {
                if (c==APOS&&v_apo==ANO)
```

```
                        {       /* konec apostrofu */
                                --apo;
                                v_apo=NE;
                        }
                        if (c=='"'&&v_uvoz==ANO)
                        {       /* konec uvozovek */
                                --uvoz;
                                v_uvoz=NE;
                        }
                        if (v_koment==ANO)
                        {
                                if (c=='*')
                                        kom_znak=ANO;
                                if (c=='/'&&
                                    kom_znak==ANO)
                                /* konec komentáře */
                                {
                                        --koment;
                                        v_koment=NE;
                                        kom_znak=NE;
                                }
                        }
                }
        }

        if (kul||hran||sloz||apo||uvoz||koment)
        {
           printf ("kulate %d  hranate %d  ",kul,hran);
           printf ("slozene %d  apostrofy %d  ",sloz,apo);
           printf ("uvozovky %d  komentare %d",uvoz,koment);

/* 0 značí, že počet daných symbolů je vpořádku; u závorek
   záporné číslo značí více uzavíracích závorek, kladné číslo
   počet neuzavřených */

        }
        else
           printf ("V ramci tohoto programu.vsechno O.K.");
}
```

CVIČENÍ: 2 - 1

```
#define MAXLINE 1000
#define EOF     (-1)

main ()

{
        printf ("\n\t%d\n",getline ());
}                       /* tiskne počet znaků řádku */

getline ()              /* modifikace funkce */
```

```
{
        int c,i;
        char line [MAXLINE];

        i = 0;
        while ((c = getchar ()) != EOF)
        {
                if (i < MAXLINE - 1)
                {
                        if (c != '\n')
                                line [i] = c;
                                ++i;
                }
        }
        if (c == '\n')
        {
                line [i] = c;
                ++i;
        }
        line [i] = '\0';
        return (i);
}
```

CVIČENÍ: 2 - 2

```
#define MAXLINE 5
#define EOF     (-1)

main ()

{
        char ret [MAXLINE];

        getline (ret,MAXLINE);
        printf ("\n\t%6d",htoi (ret));
}

htoi (s)                /* akceptuje jen velká písmena */

        char s[];

{
        int i,n;

        n = 0;
        for (i = 0;s[i] != '\0';i++)
        {
                if (s[i] >= '0'&&s[i] <= '9')
                        n = 16 * n + s[i] - '0';
                if (s[i] >= 'A'&&s[i] <= 'F')
                        n = 16 * n + s[i] + 10 - 'A';
        }
        return (n);
}

getline (s,lim)
```

```
        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);
}
```

CVIČENÍ: 2 - 3

```
#define EOF     (-1)
#define MAX     1000

main ()

{
        char ret1 [MAX],ret2 [MAX];

        getline (ret1,MAX);
        printf ("\n");
        getline (ret2,MAX);
        squeeze (ret1,ret2);
}

squeeze (s1,s2)

        char s1[], s2[];
{
        int i,j,k;
        char c;

        k = 0;
        while (s2[k] != '\0')
        {
                c = s2[k++];
                for (i = j = 0;s1[i] != '\0';i++)
                        if (s1[i] != c)
                                s1[j++] = s1[i];
                s1[j] = '\0';
        }
        printf ("\n%s",s1);
}

getline (s,lim)

        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);
}
```

CVIČENÍ: 2 - 4

```
#define EOF     (-1)
#define MAX     1000

main ()

{
        char ret1[MAX], ret2[MAX];

        getline (ret1,MAX);
        printf ("\n");
        getline (ret2,MAX);
        printf ("\n%3d",any (ret1,ret2));
}

any (s1,s2)

        char s1[], s2[];

{
        int i,j;

        i = j = 0;
        while (s1[i] != '\0')
        {
                if (s1[i] != s2[j])
                        i++;
                else
                {
                        while (s1[i] == s2[j])
                        {
                                i++;
                                j++;
                        }
                        if (s2[j] == '\0')
                                return (i-j);
                        else
                                j = 0;
                }
```

```
        }
        return EOF;
}

getline (s,lim)

        char s[];
        int lim;

{
        int c;

        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);
}
```

CVIČENÍ: 2 - 5

```
#define MAX     100
#define EOF     (-1)

main ()

{
        char ret1[MAX],ret2[MAX];

        getline (ret1,MAX);
        printf ("\n");
        getline (ret2,MAX);
        printf ("\n%6d",any (ret1,ret2));
}

any (s1,s2)

        char s1[], s2[];

{
        int i,j;

        i = j = 0;
        while (s1[i] != '\0')
        {
                if (s1[i] != s2[j])
                        i++;
                else
                {
                        while (s1[i] == s2[j])
```

```
                        {
                                i++;
                                j++;
                        }
                        if (s2[j] == '\0')
                                return (i - j + i);
                        else
                                j = 0;
                }
        }
        return EOF;
}

getline (s,lim)

        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);
}
```

CVIČENÍ: 2 - 6

```
main ()

{
        printf ("\n\n\t%d",wordlength ());
}

wordlength ()

{
        int slovo;
        int n;

        slovo = 0;
        n=1;
        while (~slovo >> n)
                n++;
        return n;
}
```

CVIČENÍ: 2 - 7

```
#define XMAX    100
#define NMAX    3
#define EOF     (-1)

main ()

{
        char xret[XMAX],nret[NMAX];
        unsigned cislo,pocet;

        getline (xret,XMAX);
        cislo = atoi (xret);
        printf ("\n");
        getline (nret,NMAX);
        pocet = atoi (nret);
        printf ("\n");
        printf ("\n%6d",rightrot (cislo,pocet));
}

rightrot (x,n)

        unsigned x,n;

{
        return (( x >> n) | ( x << (16 - n)));
}

getline (s,lim)

        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);
}
```

CVIČENÍ: 2 - 8

```
#define MAX     100
#define EOF     (-1)
#define PMAX    3
#define NMAX    3
```

```
main ()

{
        char ret[MAX],pret[PMAX],nret[NMAX];
        unsigned cislo,pozice,pocet;

        getline (ret,MAX);
        cislo = atoi (ret);
        printf ("\n");
        getline (pret,PMAX);
        pozice = atoi (pret);
        printf ("\n");
        getline (nret,NMAX);
        pocet = atoi (nret);
        printf ("\n%6d",invert (cislo,pozice,pocet));
}

invert (x,p,n)

        unsigned x,p,n;

{
        return (((~x >> (p+1-n) & ~(~0 << n)) << (p+1-n)) |
                (((~0 << p) | (~0 >> (15-p+n))) & x ));
}

getline (s,lim)

        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);
}
```

CVIČENÍ: 2 - 9

```
#define MAX     100
#define EOF     (-1)

main ()

{
        char ret[MAX];
        unsigned cislo;
```

```
        getline (ret,MAX);
        cislo = atoi (ret);
        printf ("\n\t%d",bitcount (cislo));
}

bitcount (n)            /* vrací počet jednotkových bitů */

        unsigned n;

{
        unsigned b;

        b = 0;
        while (n)
        {
                n &= (n - 1);
                b++;
        }
        return (b);
}
getline (s,lim)

        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);
}
```

CVIČENÍ: 2 - 10

```
#define EOF     (-1)
#define MAX     1000

main ()

{
        int i;
        char ret0[MAX], ret1[MAX];

        getline (ret0,MAX);
        i = 0;
        while (ret0[i] != '\0')
        {
                ret1[i] = tolower ( ret0[i] );
                i++;
```

```
        }
        reti[i] = '\0';
        printf ("\n%s",reti);
}

getline (s,lim)

        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);
}

tolower (c)

        char c;

{
        return (c >= 'A' && c <= 'Z') ? c + 'a' - 'A' : c;
}
```

CVIČENÍ: 3 - 1

```
/* Program konvertuje některé neviditelné znaky ze vstupního
   řetězce jako např. tabelátor, nový řádek do viditelného
   tvaru \t,\n  apod. */

#define MAX     80      /* maximální délka řetězce */
#define EOF     (-1)    /* konec souboru */

main()

{
        char radek[MAX];
        char novy_radek[MAX*2];

        while ( getline(radek,MAX) > 0 )
        {
                expand (radek,novy_radek);
                printf ("\n%s",novy_radek);
        }
}

getline (s,lim)         /* načti řádek do s, vrať délku */
```

```
        char s [];
        int lim;

{
        int c,i;

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

expand (s,t)          /* zviditelňuje někt. neviditel. znaky */

        char s[], t[];

{
        int c,i,j;

        i=j=0;
        while ((c=s[i++])!='\0')
                switch (c)
                {
                case '\n':t[j++]='\\';  /* nový řádek */
                          t[j++]='n';break;
                case '\t':t[j++]='\\';  /* tabulátor */
                          t[j++]='t';break;
                case '\b':t[j++]='\\';  /* o znak zpět */
                          t[j++]='b';break;
                case '\r':t[j++]='\\';  /* návrat vozu */
                          t[j++]='r';break;
                default:  t[j++]=c;
                          break;
                }
        t[j]='\0';
}
```

CVIČENÍ: 3 - 2

```
/* program rozepisuje zkratky typu a-z, A-Z, 0-9 v kompletní
   seznam */

#define EOF     (-1)     /* konec souboru */
#define MAX       3      /* délka zkratky */
#define DEL_ZKR   27     /* délka výčtu zkratky */

main()
```

```
{
        char zkratka[MAX+1];
        char seznam[DEL_ZKR+1];
        int n;

        while ((n=getline(zkratka,MAX+1)) > 0 )
        {
                n=expand(zkratka,seznam);
                if (n)
                     printf("\n%s\n",seznam);
                else
                     printf("\nNebyla to povolena zkratka\n");
        }
}

expand (s1,s2)          /* rozepisuje zkratky */

        char s1[], s2[];

{
        int c, d, i;

        if (s1[1]!='-')                 /* není zkratka */
                return(0);
        c=s1[0];d=s1[2];
        if (c > d )                     /* špatná zkratka */
                return(0);
        if (!((c>='a'&&c<='z'&&d<='z')||
              (c>='A'&&c<='Z'&&d<='Z')||      /* nepřípustná */
              (c>='0'&&c<='9'&&d<='9')))        /* zkratka */
                return(0);
        for (i=0;c<=d;i++)
                s2[i]=c++;
        s2[i]='\0';
        return i;
}

getline (s,lim)         /* načti řádek do s, vrať délku */

        char s [];
        int lim;

{
        int c,i;

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

CVIČENÍ: 3 - 3

```
/* program převádí dekadické číslo na znakový řetězec;
   jiné znaky než cifry a minus ze vstupu ignoruje */

#define N       5       /* maximální počet cifer */
#define EOF     (-1)    /* konec souboru */

main()

{
        int x,i;
        char ret[N+2];

        while ( getline(ret,N+2) > 0 )
        {
                x=atoi(ret);
                itoa(x,ret);
                printf("\n%s\n",ret);
        }
}

itoa (n,s)              /* konverze n na znaky v s */

        char s [];
        int n;

{
        int i, znamenko, j;

        znamenko = ( n < 0 ) ? (-1) : 1 ;
                                /* zjištění znaménka */
        i = 0;
        do          /* generování číslic v opačném pořadí */
        {
                j = n / 10 ;
                j = n - j * 10 ;
                s [i ++] = j * znamenko + '0' ;
                                        /* další číslice  */
        }
        while ((n /= 10 ) * znamenko > 0 );/*smazání číslice */
        if (znamenko < 0)
                s [i ++] = '-';
        s [i] = '\0';
        reverse (s);
}

reverse (s)             /* invertován řetězec s */

        char s [];

{
        int c, i, j;

        for (i = 0, j = strlen (s) - 1; i < j; i ++, j --)
        {
```

```
                c = s [i];
                s [i] = s [j];
                s [j] = c;
        }
}

getline (s,lim)         /* načti řádek číslic a vřať délku */

        char s [];
        int lim;

{
        int c,i;

        for (i=0;i<lim-1;++i)
                if ((c=getchar())!=EOF)
                {
                        s[i]=c;
                        if (c=='\n')
                                lim=i;
                        if (!(c>='0'&&c<='9'||c=='-'))
                                --i;
                }
                else
                        lim=--i;
        s[i]='\0';
        return (i);
}
```

CVIČENÍ: 3 - 4

```
/* program konvertuje nezáporné celé číslo na jeho binární
   a hexadecimální reprezentaci */

#define DEL_SL  16      /* velikost int v bitech */
#define DEL_HX   4      /* délka hexadec. řetězce */
#define N       5       /* maximální počet cifer */
#define EOF     (-1)    /* konec souboru */

main()

{
        unsigned x;
        int i;
        char ret[N+1];
        char bin[DEL_SL+1];
        char hex[DEL_HX+1];

        while ( getline(ret,N+1) > 0 )
        {
                x=atoi(ret);
                printf("\ndekadicky\t%d\n",x);
                itob(x,bin);
                printf("binarne\t%s\n",bin);
                itoh(x,hex);
```

```
                printf("hexadecimalne\t%s\n",hex);
        }
}

itob(n,s)       /* převod nezáp. čísla na binární repr. */

        unsigned n;
        char s[];

{
        unsigned i;

        i = 0;
        do              /* generování číslic v opačném pořadí */
        {
                s [i ++] = n % 2 + '0' ;
                                        /* další číslice  */
        }
        while ((n /= 2 ) > 0 );
        s [i] = '\0';
        reverse (s);
}

itoh (n,s)              /* převod nezápor. čísla na hex. repr.*/

        unsigned n;
        char s[];

{
        unsigned i ,j;

        i=0;
        do
        {
                j = n % 16 ;
                if ( j>=0&&j<=9 )
                        s[i++] = j + '0' ;
                else
                        switch (j)
                        {
                        case 10:s[i++]='A';break;
                        case 11:s[i++]='B';break;
                        case 12:s[i++]='C';break;
                        case 13:s[i++]='D';break;
                        case 14:s[i++]='E';break;
                        case 15:s[i++]='F';break;
                        }
        }
        while ((n/=16) > 0 );
        s[i]='\0';
        reverse (s);
}

reverse (s)             /* invertován řetězec s */

        char s [];
{
        int c, i, j;
```

```
        for (i = 0, j = strlen (s) - 1; i < j; i ++, j --)
        {
                c = s [i];
                s [i] = s [j];
                s [j] = c;
        }
}

getline (s,lim)          /* načti řádek číslic a vrať délku */

        char s [];
        int lim;
{
        int c,i;

        for (i=0;i<lim-1;++i)
                if ((c=getchar())!=EOF)
                {
                        s[i]=c;
                        if (c=='\n')
                                lim=i;
                        if (!(c>='0'&&c<='9'))
                                --i;
                }
                else
                        lim=--i;
        s[i]='\0';
        return (i);
}


        CVIČENÍ:        3 - 5
        ---------------------

/* program převádí celé číslo na jeho řetězovou reprezentaci a
  řetězec doplní zleva mezerami na jeho stanovenou min.délku */

#define N       5       /* maximální počet cifer */
#define EOF     (-1)    /* konec souboru */
#define M       9       /* minimální délka řetězce bez znam.*/

main ()

{
        int x,i;
        char ret[N+2], ret1[M+2];

        while ( getline(ret,N+2) > 0 )
        {
                x=atoi(ret);
                if ( M>=N )
                {
                        itoa(x,ret1,M+2);
                        printf("\n%s\n",ret1);
                }
                else
                {
```

```
                        itoa(x,ret,M+2);
                        printf("\n%s\n",ret);
                }
        }
}

itoa (n,s,m)                    /* konverze n na znaky v s */

        char s [];
        int n, m;
{
        int i, znamenko, j;

        znamenko = ( n < 0 ) ? (-1) : 1 ;
        i = 0;
        do              /* generování číslic v opačném pořadí */
        {
                j = n / 10 ;
                j = n - j * 10 ;
                s [i ++] = j * znamenko + '0' ;
        }                                       /* další číslice  */
        while ((n /= 10 ) * znamenko > 0 );/*smazání číslice */
        if (znamenko < 0)
                s [i ++] = '-';
        for (n=i;n<m-1;n++)
                s [n] = ' ';
        s [n] = '\0';
        reverse (s);
}

reverse (s)                     /* invertován řetězec s */

        char s [];

{
        int c, i, j;

        for (i = 0, j = strlen (s) - 1; i < j; i ++, j --)
        {
                c = s [i];
                s [i] = s [j];
                s [j] = c;
        }
}

getline (s,lim)                 /* načti řádek číslic a vrať délku */

        char s [];
        int lim;

{
        int c,i;

        for (i=0;i<lim-1;++i)
                if ((c=getchar())!=EOF)
                {
```

```
                        s[i]=c;
                        if (c=='\n')
                                lim=i;
                        if (!(c>='0'&&c<='9'||c=='-'))
                                --i;
                }
                else
                        lim=--i;
        s[i]='\0';
        return (i);
}
```

CVIČENÍ: 3 - 6

```
/* program kopíruje vstup na výstup; kopíruje pouze jeden
   řádek ze skupiny sousedních totožných řádků */

#define MAX     80      /* maximální délka řádku */
#define EOF     (-1)    /* konec souboru */

main ()

{
        int n1, n2;
        char radek1[MAX+1];
        char radek2[MAX+1];

        if ((n1=getline(radek1,MAX+1)) > 0 )
        {

           printf("\n%s\n",radek1);
           while ((n2=getline(radek2,MAX+1)) > 0 )
                if (n1!=n2)
                {
                        printf("\n%s\n",radek2);
                        n1=n2;
                        copy(radek2,radek1);
                }
                else
                        if (porovnej(radek1,radek2,n1) < 0 )
                                ;
                        else
                        {
                                printf("\n%s\n",radek2);
                                n1=n2;
                                copy(radek2,radek1);
                        }
        }
}

porovnej (s,t,n)        /* porovnává 2 řetězce délky n */

        char s[], t[];
        unsigned n;
```

```
{
        --n;
        while (n>=0&&s[n]==t[n])
                --n;
        return(n);
}

getline (s,lim)          /* načti řádek do s, vrať délku */

        char s [];
        int lim;

{
        int c,i;

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

CVIČENÍ: 4 - 1

```
rindex (s,t)    /* vrať index posledního výskytu řetězce t
                   v řetězci s (oba musí být ukončeny '\0') */

        char s[],t[];
{
        int i,j,k;
        int pozice;

        pozice=-1;

        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')
                {
                        pozice=i;
                        i=--j;
                }
        }
        return (pozice);
}
```

CVIČENÍ: 4 - 2

```
  . . .
  . . .
#define ANO      1
#define NE       0
#define MINUS    '~'            /* symbol pro unární mínus */
#define ERASE    '$'            /* symbol pro vymazání vrcholu
                                   zásobníku */
#define NULA     '@'            /* symbol pro vymazání hodnot
                                   zvolených proměnných */
#define PROM     26             /*-počet proměnných */

main ()                         /* RPN kalkulátor */
{
        int typ,op1,op2,i,c;
        char s[MAXOP+1];
        int prom[PROM];         /* pole proměnných */
        int ctr[PROM];          /* indikace, zda proměnná je už
                                   "naplněná" */

        for (i=0;i<=PROM-1;i++)         ctr[i]=NE;
        clear (); init ();
        while ((typ = getop(s,MAXOP)) != EOF)
                switch (typ=tolower(typ))
                {
                        . . .
                case MINUS: push (0 - pop()); break;
                case '%':   op2 = pop(); op1 = pop();
                            if (op2 != 0)
                                push (op1-op1/op2*op2);
                            else
                              printf("deleni nulou\n");
                            break;
                case ERASE: pop (); break;
                case NULA:
                    while ((c=tolower(getchar()))!=EOF)
                    {       if (c>='a'&&c<='z')
                                    ctr[c-'a']=NE;
                            else    putchar ('\b');
                    }   break;
                default:
                     if (typ>='a'&&typ<='z')
                     {    typ=typ-'a';
```

```
                        if (ctr[typ]==NE)
                        {       ctr[typ]=ANO;
                                prom[typ]=pop(); }
                        else  push (prom[typ]);
                    }
                    else printf("neznamy prikaz %c\n"
                                typ);   break;
                }
}
```

CVIČENÍ: 4 - 3

```
ungets (s)
        char s[];
{
        int i;

        i=0;
        while (s[i]!='\0')
                ungetch(s[i++]);
}
```

CVIČENÍ: 4 - 4

```
char buf;
int bufp;

getch ()
{
        if (bufp==ANO)
        {
                bufp=NE;
                return (buf);
        }
        else
                return (getchar());
}

ungetch (c)
        int c;
{
        buf=c;
        bufp=ANO;
}

init ()
{
        bufp=NE;
}
```

CVIČENÍ: 4 - 5

```
. . .
#define BUF         100
int buf[BUF];    !!
int bufp;
. . .
```

CVIČENÍ: 4 - 6

```
itoa (n,s)                  /* převeď integer  n  na řetězec
                               s (předpoklad : s[0]='\0') */
        int n;
        char s[];
{
        int i,znamenko;

        if ((znamenko=sign(n))<0)
                n=-n;
        if ((i=n/10)>0)
                itoa (i*znamenko,s);
        i=strlen (s);
        if (i==0&&znamenko<0)
                s[i++]='-';
        s[i++]=n%10+'0';
        s[i]='\0';
}

sign (n)                    /* vrať znaménko čísla n */

        int n;
{

        if (n>0)
                return (1);
        if (n==0)
                return (0);
        if (n<0)
                return (-1);
}
```

CVIČENÍ: 4 - 7

```
reverse (s)                 /* převrať daný řetěz s (s musí být
                               ukončen nul-znakem)  */
        char s[];
{
        int delka;
        char c;

        delka=strlen (s);
        c=s[--delka];
```

```
        if (delka>i)
        {
                s[delka]='\0';
                reverse (s);
        }
        while (delka>=i)
                s[delka]=s[--delka];
        s[0]=c;
}
```

CVIČENÍ: 5 - 1

```
/*    pointrová verze funkce strcat     */

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

CVIČENÍ: 5 - 2

```
/*  pointrové verze funkcí itoa, reverse, index  */

itoa (n,s)                  /* konverze n na znaky v s */
        char *s;
        int n;
{
        int znamenko;
        char *x;

        x = s;
        if ((znamenko = n) < 0) /*zjištění znaménka*/
                n = -n;         /*n bude kladné*/
        do              /* generování číslic v opačném pořadí */
        {
                *s ++ = n % 10 + '0'; /* další číslice  */
        }
        while ((n /= 10) > 0);          /* smazání číslice */
        if (znamenko < 0)
                *s  ++ = '-';
        *s = '\0';
        reverse (x);
}

reverse (s)                 /* invertován řetězec s */
```

```
        char *s;

{
        int c;
        int i, j;

        for ( i = 0 , j = strlen (s) - 1; i < j ; i ++ , j --)
        {
                c = *(s + i);
                *(s + i) = *(s + j);
                *(s + j) = c;
        }
}

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);
}
```

CVIČENÍ: 5 - 3

```
#define NULL    0
#define MAX_RADKU       100     /* maximální počet řádků */
#define MAX_DEL_RAD     1000       /* maximální délka řádku */
#define EOF             (-1)

main ()                         /* třídění vstupních řádků */
{
        char *ptr_rad [MAX_RADKU];      /* pointry na řádky */
        int poc_rad;            /* počet načtených řádků */
        char *ptr, *readlines();
        int i;

        poc_rad = 0;
        while ( poc_rad < MAX_RADKU )
        {
                ptr = readlines (&i);
                if ( i <= 0)
                        break;
                ptr_rad [poc_rad ++] = ptr;
        }
        if ( i == 0 )
```

```
        {
                sort (ptr_rad,poc_rad);
                writelines (ptr_rad,poc_rad);
        }
        else
                printf ("malo pameti pro vstupni radky\n");
}

readlines (i)           /* čti vstupní řádky */

        int *i;
{
        int len;
        char *p, *alloc(), radek [MAX_DEL_RAD];

        if ((len = getline (radek,MAX_DEL_RAD)) > 0)
        {
                if((p = alloc (len+1)) == NULL)
                {
                        *i = -1;
                        return NULL;
                }
                else                    /* nový řádek */
                        strcpy (p,radek);
                *i = 1;
                return p;
        }
        else
                return *i = NULL;
}

writelines (ptr_rad,poc_rad)            /* vypiš řádku */

        char *ptr_rad [];
        int poc_rad;
{
        int i;

        for (i = 0; i < poc_rad; i ++)
                printf ("%s\n",ptr_rad [i]);
}

sort (v,n)      /* roztřiď řetězec v[0] ... v[n-1] vzestupně */

        char *v [];
        int n;

{
        int krok, i, j;
        char *pom_ptr;

        for (krok = n / 2; krok > 0; krok /= 2)
                for (i = krok; i < n; i ++)
                        for (j = i - krok; j >= 0; j -= krok)
                        {
                                if (strcmp(v[j],v[j+krok]) <=0)
                                        break;
```

```
                        pom_ptr = v [j];
                        v [j] = v [j + krok];
                        v [j + krok] = pom_ptr;
                }
}

getline (s,lim)         /* načti řádek do s, vrať délku */

        char s [];
        int lim;
{
        int c,i;

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

CVIČENÍ: 5 - 4

```
/* pointrové verze funkcí den_v_roce a mesic_den    */

den_v_roce (rok,mesic,den)

        int rok, mesic, den;

{
        int (*tab_dnu) [13];
        int i, pom;
        char *alloc();

        tab_dnu = alloc(70);

        (*tab_dnu) [0] = 0;     (*(tab_dnu+i)) [0] = 0;
        (*tab_dnu) [1] = 31;    (*(tab_dnu+i)) [1] = 31;
        (*tab_dnu) [2] = 28;    (*(tab_dnu+i)) [2] = 29;
        (*tab_dnu) [3] = 31;    (*(tab_dnu+i)) [3] = 31;
        (*tab_dnu) [4] = 30;    (*(tab_dnu+i)) [4] = 30;
        (*tab_dnu) [5] = 31;    (*(tab_dnu+i)) [5] = 31;
        (*tab_dnu) [6] = 30;    (*(tab_dnu+i)) [6] = 30;
        (*tab_dnu) [7] = 31;    (*(tab_dnu+i)) [7] = 31;
        (*tab_dnu) [8] = 31;    (*(tab_dnu+i)) [8] = 31;
        (*tab_dnu) [9] = 30;    (*(tab_dnu+i)) [9] = 30;
        (*tab_dnu) [10] = 31;   (*(tab_dnu+i)) [10] = 31;
        (*tab_dnu) [11] = 30;   (*(tab_dnu+i)) [11] = 30;
        (*tab_dnu) [12] = 31;   (*(tab_dnu+i)) [12] = 31;
```

```
        pom = rok % 4 == 0 && rok % 100 != 0 || rok % 400 == 0;
        for (i = 1; i < mesic; i ++)
                den += (*(tab_dnu+pom))[i];
        free (tab_dnu);
        return (den);
}

mesic_den (rok, prubden, pmesic, pden)

        int rok, prubden, *pmesic, *pden;

{
        int (*tab_dnu) [13];
        int i, pom;
        char *alloc();

        tab_dnu = alloc(70);

        (*tab_dnu) [0] = 0;     (*(tab_dnu+1)) [0] = 0;
        (*tab_dnu) [1] = 31;    (*(tab_dnu+1)) [1] = 31;
        (*tab_dnu) [2] = 28;    (*(tab_dnu+1)) [2] = 29;
        (*tab_dnu) [3] = 31;    (*(tab_dnu+1)) [3] = 31;
        (*tab_dnu) [4] = 30;    (*(tab_dnu+1)) [4] = 30;
        (*tab_dnu) [5] = 31;    (*(tab_dnu+1)) [5] = 31;
        (*tab_dnu) [6] = 30;    (*(tab_dnu+1)) [6] = 30;
        (*tab_dnu) [7] = 31;    (*(tab_dnu+1)) [7] = 31;
        (*tab_dnu) [8] = 31;    (*(tab_dnu+1)) [8] = 31;
        (*tab_dnu) [9] = 30;    (*(tab_dnu+1)) [9] = 30;
        (*tab_dnu) [10] = 31;   (*(tab_dnu+1)) [10] = 31;
        (*tab_dnu) [11] = 30;   (*(tab_dnu+1)) [11] = 30;
        (*tab_dnu) [12] = 31;   (*(tab_dnu+1)) [12] = 31;

        pom = rok % 4 == 0 && rok % 100 != 0 || rok % 400 == 0;
        for (i = 1; prubden > (*(tab_dnu+pom))[i]; i ++)
                prubden -= (*(tab_dnu+pom))[i];
        *pmesic = i;
        *pden = prubden;
        free (tab_dnu);
}
```

CVIČENÍ: 5 - 5

```
#define CISLO   '0'             /* symbol pro číslo */
#define MINUS   '~'             /* symbol pro unární minus */
#define ERASE   '$'             /* symbol pro vymazání vrcholu
                                   zásobníku */
main (argc, argv)               /* RPN kalkulátor */
        int argc;
        char *argv [];
{
        int typ,op1,op2,ctr,a;

        ctr=0;  clear ();
        while ( -- argc > 0 )
        {
```

```
                if ( ** ++ argv < '0' || ** argv > '9' )
                        typ = tolower (** argv);
                else    typ = CISLO;
                switch (typ)
                {
                        case CISLO: push (atoi(*argv)); break;
                        case MINUS: push (0 - pop());   break;
                        case '+':   push (pop() + pop());break;
                        case '*':   push (pop() * pop());break;
                        case '-':   op2 = pop();
                                    push (pop() - op2); break;
                        case '/':   op2 = pop();
                                    if (op2) push (pop()/op2);
                                    else
                                      printf("deleni nulou\n");
                                    break;
                        case '%':   op2 = pop(); op1 = pop();
                                    if (op2 != 0)
                                        push (op1-op1/op2*op2);
                                    else
                                      printf("deleni nulou\n");
                                    break;
                        case '=':   printf("\t%d\n",
                                                push(pop()));
                                    break;
                        case ERASE: pop ();
                                    break;
                        default:  printf("neznamy prikaz %c\n",
                                                typ);
                }
        }
}

#define MAXVAL 100      /* maximální velikost zásobníku */

int zp;                 /* ukazatel zásobniku */
int val[MAXVAL];        /* zásobník čísel */

push (i)                /* ulož i do zásobníku */

        int i;

{
        if (zp < MAXVAL)
                return (val[zp++] = i);
        else
        {
                printf ("chyba : plny zasobnik\n");
                clear ();
                return (0);
        }
}

pop ()                  /* vyjmi vršek zásobníku */

{
        if (zp > 0)
```

```
                return (val [--zp]);
        else
        {
                printf ("chyba : prazdny zasobnik\n");
                clear ();
                return (0);
        }
}

clear  ()                /* vymaz zasobnik */

{
        zp = 0;
}
```

CVIČENÍ: 5 - 6.A

```
/* tabelátory nahrazuje patřičným počtem mezer */

#define ERROR (-1)
#define TAB      8       /* tabelační pozice N*TAB+1, N=0,1,..*/
#define BUF    128
#define CR      13
#define LF      10
#define EOF     26

main (argc,argv)

        int argc;
        char *argv [];
{
        unsigned n;

        if (argc < 3)
        {
                if (argc == 2)
                        n = atoi(*++argv);
                else
                        n = TAB;
                detab (n);
        }
        else
                printf("\nSpatny pocet parametru");
}

detab (n)
        unsigned n;
{
        int c,i;
        unsigned sb;

        i=1;
        sb=fopen("cvicny.txt","r",BUF);
        while ((c=fgetc(sb))!=EOF&&c!=ERROR)
        {
```

```
                if ((i%n) == 1)
                        i=1;
                if (c!='\t')
                {
                        putchar (c);
                        ++i;
                        if (c == CR || c == LF)
                                i=1;
                }
                else    while (i ++ <= n)
                                putchar (' ');
        }
        fclose(sb);
}
```

CVIČENÍ: 5 - 6.8

```
/* program nahradí mezery co nejmenším počtem mezer
   a tabelátorů */

#define ERROR (-1)
#define TAB      8      /* tabelační pozice N*TAB+1, N=0,1,..*/
#define BUF    128
#define EOF     26

main (argc,argv)

        int argc;
        char *argv [];
{
        if (argc < 3)
                entab (argc == 2 ? atoi(*++argv) : TAB);
        else    printf("\nSpatny pocet parametru");
}


entab (n)

        unsigned n;
{
        int c,i,pocet_mezer;
        unsigned sb;

        i=1;    pocet_mezer=0;
        sb=fopen("cvicny.txt","r",BUF);
        while ((c=fgetc(sb))!=EOF&&c!=ERROR)
        {
                if ((i%n)==1)   /* tabelacni pozice */
                {
                        if (pocet_mezer>0)
                        {
                                putchar ('\t');
                                pocet_mezer=0;
                        }
                        i=1;
```

```
                }
                ++i;
                if (c!=' ')
                {
                        if (pocet_mezer>0)
                                while (pocet_mezer -- > 0)
                                        putchar (' ');
                        putchar (c);
                }
                else    ++pocet_mezer;
        }
        fclose(sb);
}
```

CVIČENÍ: 5 - 7.A

```
/* tabelátory nahrazuje patřičným počtem mezer */
/* možné způsoby vyvolání : detab
                            detab m
                            detab + n
                            detab m + n          */
/* tabelační pozice jsou n*i+m, i=0,1,..        */

#define ERROR (-1)
#define TAB      8      /* tabelační pozice N*TAB+1, N=0,1,..*/
#define BUF    128
#define CR      13
#define LF      10
#define EOF     26

main (argc,argv)

        int argc;
        char *argv [];

{
        unsigned m, n;

        switch( argc )
        {
        case 1: m=1; n=TAB; argc=0;
                detab(m,n); break;
        case 2: m=atoi(*++argv); n=TAB; argc=0;
                detab(m,n); break;
        case 3: if (*++argv!='+')
                        break;
                m=1; n=atoi(*++argv); argc=0;
                detab(m,n); break;
        case 4: if (*(argv+2)!='+')
                        break;
                m=atoi(*++argv);
                n=atoi(*(argv+2));
                argc=0;
                detab(m,n);
        }
```

```
        if (argc)
                printf("\nParametry jsou chybne");
}

detab (m,n)

        unsigned m, n;

{
        int c,i,j;
        unsigned sb;

        i=j=1;
        sb=fopen("cvicny.txt","r",BUF);

        while ((c=fgetc(sb))!=EOF&&c!=ERROR)
        {
                if ((i == m && j == m && m > 1)||(i==n&&j>=m))
                        i=1;
                if (c!='\t')
                {
                        putchar (c);
                        ++j;
                        ++i;
                        if (c==CR || c==LF)
                        {
                                j=1;
                                i=1;
                        }
                }
                else
                        while ((i<m&&j<m&&m>1)||(i<n&&j>=m))
                        {
                                putchar (' ');
                                ++j;
                                ++i;
                        }
        }
        fclose(sb);
}
```

CVIČENÍ: 5 - 7.8

```
/* program nahradí mezery co nejmenším počtem mezer
   a tabelátorů                                        */
/* možné způsoby vyvolání : entab
                            entab m
                            entab + n
                            entab m + n               */
/* tabelační pozice jsou n*i+m, i=0,1,..               */

#define ERROR (-1)
#define TAB      8      /* tabelační pozice N*TAB+1, N=0,1,..*/
#define BUF    128
#define CR      13
#define LF      10
#define EOF     26
```

```
main (argc,argv)

        int argc;
        char *argv [];

{
        unsigned m, n;

        switch( argc )
        {
        case 1: m=1; n=TAB; argc=0;
                entab(m,n); break;
        case 2: m=atoi(*++argv); n=TAB; argc=0;
                entab(m,n); break;
        case 3: if (**++argv!='+')
                        break;
                m=1; n=atoi(*++argv); argc=0;
                entab(m,n); break;
        case 4: if (**(argv+2)!='+')
                        break;
                m=atoi(*++argv);
                n=atoi(*(argv+2));
                argc=0;
                entab(m,n);
        }
        if (argc)
                printf("\nParametry jsou chybne");
}

entab (m,n)

        unsigned m,n;

{
        int c,pocet_mezer;
        unsigned sb, i, j;

        i=j=1;
        pocet_mezer=0;
        sb=fopen("cvicny.txt","r",BUF);

        while ((c=fgetc(sb))!=EOF&&c!=ERROR)
        {
                if ((i==m&&j==m&&m>1)||(i==n&&j>=m))
                                /* tabelacni pozice */
                {
                        if (pocet_mezer>0)
                        {
                                putchar ('\t');
                                pocet_mezer=0;
                        }
                        i=1;
                }
                ++i;
                ++j;
                if (c!=' ')
                {
```

```
                        if (pocet_mezer>0)
                                while (pocet_mezer>0)
                                {
                                        putchar (' ');
                                        --pocet_mezer;
                                }
                        putchar (c);
                        if (c==CR||c==LF)
                        {
                                i=j=1;
                                pocet_mezer=0;
                        }
                }
                else
                        ++pocet_mezer;
        }
        fclose(sb);
}
```

CVIČENÍ: 5 - 8

```
/* program tiskne posledních n řádků ze vstupu */
/* maximálně vytiskne MAX_RAD */

#define NULL    0
#define POCET  10
#define MAX_RAD 100

main (argc,argv)
        int argc;
        char *argv [];
{
        int poc_rad;            /* počet načtených řádků */
        char *ptr, *readlines();
        char *ptr_rad [MAX_RAD]; /* pointry na řádky */
        int i, n, j;

        j = 0;
        if ( argc == 2 )
        {
                if ( ** ++argv == '-' )
                {
                        n = atoi( argv [0] + 1 );
                        j = 1;
                }
        }
        if ( argc == 1 )
        {

                n = POCET;
                j = 1;
        }
        if ( j )
        {
```

```
                poc_rad = 0;
                while ( 1 )
                {
                        ptr = readlines (&i);
                        if ( i <= 0)    break;
                        if ( poc_rad < n )
                                ptr_rad [poc_rad ++] = ptr;
                        else
                        {
                                cykl (ptr_rad,n,ptr);
                                poc_rad ++;
                        }
                }
                if ( i == 0 )   writelines (ptr_rad,n);
                else    printf ("malo pameti pro vstup\n");
        }
        else    printf ("\nchybne parametry ");
}

cykl (ptr_rad,n,ptr)

        char *ptr_rad [], *ptr;
        int n;

{
        int i;

        for ( i = 0 ;i < n - 1; )
                ptr_rad [i] = ptr_rad [ ++i];
        ptr_rad [i] = ptr;
}

#define MAX_DEL_RAD     1000        /* maximální délka řádku */

readlines (i)           /* čti vstupní řádky */

        int *i;

{
        int len;
        char *p, *alloc(), radek [MAX_DEL_RAD];

        if ((len = getline (radek,MAX_DEL_RAD)) > 0)
        {
                if((p = alloc (len+1)) == NULL)
                {
                        *i = -1;
                        return NULL;
                }
                else                    /* nový řádek */
                        strcpy (p,radek);
                *i = 1;
                return p;
        }
        else
                return *i = NULL;
}
```

```
writelines (ptr_rad,poc_rad)            /* vypiš řádku */

        char *ptr_rad [];
        int poc_rad;

{
        int i;

        for (i = 0; i < poc_rad; i ++)
                printf ("%s\n",ptr_rad [i]);
}

#define EOF     -1

getline (s,lim)         /* načti řádek do s, vrať délku */

        char s [];
        int lim;

{
        int c,i;

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

CVIČENÍ: 5 - 9

```
#define MAX_RAD 100     /* maximální počet řádků */

main (argc,argv)        /* setřídění vstupních řádků */

        int argc;
        char *argv [];

{
        char *ptr_rad [MAX_RAD];/* pointry na text. řádky  */
        int poc_rad;            /* počet načtených řádek   */
        int strcmp(), numcmp(); /* porovnávací funkce      */
        int swap ();            /* funkce výměny           */
        int numeric;            /* 1 pro numerickou výměnu */
        int sestup;             /* 1 pro sestupné uspořád. */
        char *s;

        numeric = sestup = 0;
        while (-- argc > 0 && (* ++ argv) [0] == '-')
```

```
                for (s = argv [0] + 1; *s != '\0'; s ++)
                        switch (tolower(*s))
                        {
                        case 'n':
                                numeric = 1;
                                break;
                        case 'r':
                                sestup = 1;
                                break;
                        default:
                           printf("sort: spatny parametr%c\n",
                                                tolower(*s));
                                argc = -1;
                                break;
                        }
        if ( argc >= 0 )
        {
                if ((poc_rad=readlines(ptr_rad,MAX_RAD)) >= 0)
                {
                        if (numeric)
                                sort (ptr_rad,poc_rad,numcmp,
                                                        swap);
                        else
                                sort (ptr_rad,poc_rad,strcmp,
                                                        swap);
                        writelines (ptr_rad,poc_rad,sestup);
                }
                else
                        printf ("prilis vstupnich radku ");
        }
}

sort (v,n,comp,exch)                    /* setřídění v[0] ... v[n-1] */

        char *v [];
        int n;
        int (*comp) (), (*exch) ();

{
        int krok, i, j;

        for (krok = n / 2; krok > 0; krok /= 2)
                for (i = krok; i < n; i ++)
                        for (j = i - krok; j >= 0; j -= krok)
                        {
                                if ((*comp)(v[j],v[j+krok])<=0)
                                        break;
                                (*exch)(&v[j],&v[j+krok]);
                        }
}

numcmp (s1,s2)          /* Porovnávání s1 a s2 číselné */

        char *s1, *s2;

{
        int atoi (), v1, v2;
```

```
        v1 = atoi (s1);
        v2 = atoi (s2);
        if ( v1 < v2 )
                return (-1);
        else
                if ( v1 > v2 )
                        return (1);
                else
                        return (0);
}

swap (px,py)                    /* výměna *px a *py */

        char *px [], *py [];

{
        char *pom_ptr;

        pom_ptr = *px;
        *px = *py;
        *py = pom_ptr;
}

#define NULL    0
#define MAX_DEL_RAD     1000        /* maximální délka řádku */

readlines (ptr_rad,max_rad)             /* čti vstupní řádky */

        char *ptr_rad [];
        int max_rad;

{
        int len, poc_rad;
        char *p, *alloc (), radek [MAX_DEL_RAD];

        poc_rad = 0;
        while ((len = getline (radek,MAX_DEL_RAD)) > 0)
                if (poc_rad >= max_rad)
                        return (-1);
                else    if((p = alloc (len+1)) == NULL)
                                return (-1);
                        else                    /* nový řádek */
                        {
                                strcpy (p,radek);
                                ptr_rad [poc_rad ++] = p;
                        }
        return (poc_rad);
}

writelines (ptr_rad,poc_rad,j)          /* vypiš řádku */

        char *ptr_rad [];
        int poc_rad, j;

{
        int i;

        if ( j )
```

```
                for (i = poc_rad - 1; i >= 0; i --)
                        printf ("%s\n",ptr_rad [i]);
        else
                for (i = 0; i < poc_rad; i ++)
                        printf ("%s\n",ptr_rad [i]);

}

#define EOF     (-1)

getline (s,lim)         /* načti řádek do s, vrať délku */

        char s [];
        int lim;

{
        int c,i;

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


        CVIČENÍ:        5 - 10
        ----------------------

#define MAX_RAD 100     /* maximální počet řádků */

main (argc,argv)        /* setřídění vstupních řádků */

        int argc;
        char *argv [];

{
        char *ptr_rad [MAX_RAD];/* pointry na text. řádky  */
        int poc_rad;            /* počet načtených řádek   */
        int strcmp(), numcmp(),
            strcmi();           /* porovnávací funkce      */
        int swap ();            /* funkce výměny           */
        int numeric;            /* 1 pro numerickou výměnu */
        int sestup;             /* 1 pro sestupné uspořád. */
        int vel;                /* 1 dává na roveň velká
                                        a malá písmena     *
        char *s;

        numeric = sestup = vel = 0;
        while (-- argc > 0 && (* ++ argv) [0] == '-')
                for (s = argv [0] + 1; *s != '\0'; s ++)
                        switch (tolower(*s))
```

```
                        {
                        case 'n':       numeric = 1;
                                        break;
                        case 'r':       sestup = 1;
                                        break;
                        case 'f':       vel = 1;
                                        break;
                        default:
                           printf("sort: spatny parametr%c\n",
                                                tolower(*s));
                                argc = -1;
                        }

        if ( argc >= 0 )
        {
                if ((poc_rad=readlines(ptr_rad,MAX_RAD)) >= 0)
                {
                        if (numeric)
                                sort (ptr_rad,poc_rad,numcmp,
                                                        swap);
                        else
                                sort (ptr_rad,poc_rad,vel ?
                                        strcmi : strcmp,swap);
                        writelines (ptr_rad,poc_rad,sestup);
                }
                else    printf ("prilis vstupnich radku ");
        }
}

sort (v,n,comp,exch)                    /* setřídění v[0] ... v[n-1] */

        char *v [];
        int n;
        int (*comp) (), (*exch) ();

{
        int krok, i, j;

        for (krok = n / 2; krok > 0; krok /= 2)
                for (i = krok; i < n; i ++)
                        for (j = i - krok; j >= 0; j -= krok)
                        {
                                if ((*comp)(v[j],v[j+krok])<=0)
                                        break;
                                (*exch)(&v[j],&v[j+krok]);
                        }
}

numcmp (s1,s2)                  /* Porovnávání s1 a s2 číselně */

        char *s1, *s2;

{
        int atoi (), v1, v2;

        v1 = atoi (s1);
        v2 = atoi (s2);
```

```
        if ( v1 < v2 )
                return (-1);
        else
                if ( v1 > v2 )
                        return (1);
                else
                        return (0);
}

strcmi (s,t)                    /* porovnání s a t lexikálně ;velká
                                   a malá písmena jsou si rovna     */

        char *s, *t;

{

        for (; tolower(*s) == tolower(*t); s ++, t ++)
                if ( *s == '\0' )
                        return (0);
        return ( tolower(*s) - tolower(*t));
}

swap (px,py)                    /* výměna *px a *py */

        char *px [], *py [];

{
        char *pom_ptr;

        pom_ptr = *px;
        *px = *py;
        *py = pom_ptr;
}

#define NULL    0
#define MAX_DEL_RAD     1000        /* maximální délka řádku */

readlines (ptr_rad,max_rad)             /* čti vstupní řádky */

        char *ptr_rad [];
        int max_rad;

{
        int len, poc_rad;
        char *p, *alloc (), radek [MAX_DEL_RAD];

        poc_rad = 0;
        while ((len = getline (radek,MAX_DEL_RAD)) > 0)
                if (poc_rad >= max_rad)
                        return (-1);
                else    if((p = alloc (len+1)) == NULL)
                                return (-1);
                        else                    /* nový řádek */
                        {
                                strcpy (p,radek);
                                ptr_rad [poc_rad ++] = p;
                        }
        return (poc_rad);
```

```
}

writelines (ptr_rad,poc_rad,j)          /* vypiš řádku */

        char *ptr_rad [];
        int poc_rad, j;

{
        int i;

        if ( j )
                for (i = poc_rad - 1; i >= 0; i --)
                        printf ("%s\n",ptr_rad [i]);
        else
                for (i = 0; i < poc_rad; i ++)
                        printf ("%s\n",ptr_rad [i]);

}

#define EOF     (-1)

getline (s,lim)         /* načti řádek do s, vrať délku */

        char s [];
        int lim;

{
        int c,i;

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


        CVIČENÍ:        5 - 11
        ----------------------

#define MAX_RAD 100     /* maximální počet řádků */

main (argc,argv)        /* setřídění vstupních řádků */

        int argc;
        char *argv [];
{
        char *ptr_rad [MAX_RAD];/* pointry na text. řádky  */
        int poc_rad;            /* počet načtených řádek   */
        int strcmp(), numcmp(),
            strcmi();           /* porovnávací funkce      */
```

```
        int swap ();                /* funkce výměny              */
        int numeric;                /* 1 pro numerickou výměnu    */
        int sestup;                 /* 1 pro sestupné uspořád.    */
        int vel;                    /* 1 dává na roveň velká
                                           a malá písmena          */
        int slovnik;                /* 1 pro slovníkové srov.     */
        char *s;

        numeric = sestup = vel =slovnik = 0;
        while (-- argc > 0 && (* ++ argv) [0] == '-')
                for (s = argv [0] + 1; *s != '\0'; s ++)
                        switch (tolower(*s))
                        {
                        case 'n': numeric = 1; break;
                        case 'r': sestup = 1; break;
                        case 'f': vel = 1; break;
                        case 'd': slovnik = 1; break;
                        default:
                           printf("sort: spatny parametr%c\n",
                                                tolower(*s));
                                argc = -1; break;
                        }
        if ( argc >= 0 )

        {
                if ((poc_rad=readlines(ptr_rad,MAX_RAD,
                                                slovnik)) >= 0)
                {
                        if (numeric)
                                sort (ptr_rad,poc_rad,numcmp,
                                                        swap);
                        else    if ( vel )
                                        sort (ptr_rad,poc_rad,
                                                strcmi,swap);
                                else    sort (ptr_rad,poc_rad,
                                                strcmp,swap);
                        writelines (ptr_rad,poc_rad,sestup);
                }
                else    printf ("prilis vstupnich radku ");
        }
}

sort (v,n,comp,exch)                /* setřídění v[0] ... v[n-1] */

        char *v [];
        int n;
        int (*comp) (), (*exch) ();

{
        int krok, i, j;

        for (krok = n / 2; krok > 0; krok /= 2)
                for (i = krok; i < n; i ++)
                        for (j = i - krok; j >= 0; j -= krok)
                        {
                                if ((*comp)(v[j],v[j+krok])<=0)
                                        break;
                                (*exch)(&v[j],&v[j+krok]);
                        }
```

```
}

numcmp (s1,s2)          /* Porovnávání s1 a s2 číselné */

        char *s1, *s2;

{
        int atoi (), v1, v2;

        v1 = atoi (s1);
        v2 = atoi (s2);
        if ( v1 < v2 )
                return (-1);
        else
                if ( v1 > v2 )
                        return (1);
                else
                        return (0);
}

strcmi (s,t)            /* porovnání s a t lexikálně ;velká
                           a malá písmena jsou si rovna     */

        char *s, *t;
{

        for (; tolower(*s) == tolower(*t); s ++, t ++)
                if ( *s == '\0' )
                        return (0);
        return ( tolower(*s) - tolower(*t));
}

swap (px,py)            /* výměna *px a *py */

        char *px [], *py [];
{
        char *pom_ptr;

        pom_ptr = *px;
        *px = *py;
        *py = pom_ptr;
}

#define NULL    0
#define MAX_DEL_RAD     1000        /* maximální délka řádku */

readlines (ptr_rad,max_rad,i)           /* čti vstupní řádky */

        char *ptr_rad [];
        int max_rad, i;
{
        int len, poc_rad;
        char *p, *alloc (), radek [MAX_DEL_RAD];

        poc_rad = 0;
        while ((len = getline (radek,MAX_DEL_RAD,i)) > 0)
                if (poc_rad >= max_rad)
                        return (-1);
```

```
                else    if((p = alloc (len+1)) == NULL)
                                return (-1);
                        else                    /* nový řádek */
                        {
                                strcpy (p,radek);
                                ptr_rad [poc_rad ++] = p;
                        }
        return (poc_rad);
}

writelines (ptr_rad,poc_rad,j)          /* vypiš řádku */

        char *ptr_rad [];
        int poc_rad, j;
{
        int i;

        if ( j )
                for (i = poc_rad - 1; i >= 0; i --)
                        printf ("%s\n",ptr_rad [i]);
        else
                for (i = 0; i < poc_rad; i ++)
                        printf ("%s\n",ptr_rad [i]);
}

#define EOF     (-1)

getline (s,lim,j)               /* načti řádek do s, vrať délku */

        char s [];
        int lim, j;

{
        int c,i;

        for (i=0;i<lim-1;++i)
                if ((c=getchar())!=EOF)
                {
                        if ( j )
                                if ((c>='a'&&c<='z')
                                    ||(c>='A'&&c<='Z')
                                    ||(c>='0'&&c<='9')
                                    ||c==' '||c=='\n')
                                        s[i]=c;
                                else
                                        -- i;
                        else
                                s[i]=c;
                        if (c=='\n')
                                lim=--i;
                }
                else
                        lim=--i;
        s[i]='\0';
        return (i);
}
```

CVIČENÍ: 6 - 1

```
#define BUF             128
#define EOF             '0'
#define MAX_DELKA       20
#define TRUE            1
#define NULL            0
#define P_KLICU         10

struct klic
{
        char *klic_slovo;
        int pocet_slov;
} tab_klicu[P_KLICU];

char buffslovo [MAX_DELKA];
char buff [BUF];
char *ukaz;

main()                                  /* počítej klíčová slova, */
                                /* ignoruj řetězce v uvozovkách */
{
        int n;
        char *dej_slovo ();
        char *t;

        inic ();
        while (TRUE)
        {
                if(!(t = dej_slovo ()))
                        break;
                plus_slovo (t);
                if((n = binary (buffslovo,tab_klicu,P_KLICU))
                                                        >= NULL)
                        tab_klicu [n].pocet_slov++;
        }
        for (n = 0; n < P_KLICU; n++)
                printf ("\n%d %s",tab_klicu[n].pocet_slov,
                                        tab_klicu[n].klic_slovo);
}

binary (slovo,tab,n)                    /* najdi slovo v tab[n] */

        char *slovo;
        struct klic tab [];
        int n;

{
        int dol_mez, hor_mez, stred, pom;

        dol_mez = 0;
        hor_mez = n - 1;
        while (dol_mez <= hor_mez)
        {
                stred = (dol_mez + hor_mez) / 2;
                if((pom=strcmp(slovo,tab[stred].klic_slovo))<0)
                        hor_mez = stred - 1;
```

```
                else if (pom > 0)
                        dol_mez = stred + 1;
                else
                        return (stred);
        }
        return (-1);
}

inic ()                         /* inicializace tabulky klíčů */

{
        int j;

        for (j=0;j<=P_KLICU-1;j++)
        {
                tab_klicu[j].pocet_slov = 0;
                tab_klicu[j].klic_slovo = alloc(10);
        }
        tab_klicu[0].klic_slovo = "break";
        tab_klicu[1].klic_slovo = "continue";
        tab_klicu[2].klic_slovo = "else";
        tab_klicu[3].klic_slovo = "for";
        tab_klicu[4].klic_slovo = "if";
        tab_klicu[5].klic_slovo = "int";
        tab_klicu[6].klic_slovo = "return";
        tab_klicu[7].klic_slovo = "struct";
        tab_klicu[8].klic_slovo = "unsigned";
        tab_klicu[9].klic_slovo = "while";
}

dej_slovo ()                        /* vyhledej slovo v buff */

{
        register char *s;

        if (ukaz == buff)
                if (nacti_radek () == NULL)
                        return NULL;
        s = ukaz;
        while (*s && !isalpha (*s))
                ++s;
        if (!*s)
        {
                ukaz = buff;
                return dej_slovo ();
        }
        if (!(ukaz = dalsi_slovo (s)))
                ukaz = buff;
        return s;
}

plus_slovo (adr)            /* načti slovo z adr do buffslova */

        char *adr;
{
        register char *s;
        int i;
```

```
        s = buffslovo;
        for (i=0;i < MAX_DELKA; i++)
                if (isalnum (*(adr+i)))
                        *s++ = *(adr+i);
                else
                        break;
        *s = NULL;
}

dalsi_slovo (slovo)             /* vyhledej následující slovo */

        char *slovo;
{
        register char *s;

        s = slovo;
        while (*s && isalnum (*s))
                ++s;
        while (*s && !isalpha (*s))
                ++s;
        return *s ? s : NULL;
}

nacti_radek ()                  /* načti řádek ze vstupu do buff */

{
        int i;
        char c;

        for (i=0;i < BUF-1 && (c=getchar ()) != EOF &&
                                                c != '\n';i++)
        {
                if (c == '"')
                        while ((c = getchar ()) != '"')
                                ;
                else
                        buff [i] = c;
        }
        if ( c == '\n')
        {
                buff [i] = c;
                buff [++i] = NULL;
        }
        if (c == EOF)
                return NULL;
}
```

CVIČENÍ: 6 - 2

```
#define         MAX_DELKA       20
#define         ERROR   (-1)
#define         BUF     128
#define         NULL    0
#define         TRUE    1
```

```
struct uzel_stromu
{
        char *slovo;
        struct uzel_stromu *levy;
        struct uzel_stromu *pravy;
}*ptr;

int buff [BUF];
char buffslovo [MAX_DELKA];
unsigned fd;
char *ukaz;
char *tab[100];
int k;

main ()

{
        struct uzel_stromu *koren, *strom ();
        char *c, *w;

        koren = NULL;
        ukaz = buff;
        if ((fd = fopen ("TEXT.C","r",BUF)) != ERROR)
        {
                while (TRUE)
                {
                        if (!(c = dej_slovo()))
                                break;
                        plus_slovo (c);
                        koren = strom (koren,buffslovo);
                }
                k = 0;  napln (koren);
                tab [k] = NULL; k = 0;
                while (tab [k] != NULL)
                {
                        if (!(strncmp (tab [k],tab [k+1], 5))
                                                &&tab[k])
                        {       w = strsave (tab [k]);
                                while (!(strncmp (w,tab[k],5)))
                                {       printf ("\n%s",tab[k]);
                                        ++k;    }
                                printf ("\n");  }
                        ++k;    }
                fclose (fd);    }
        else    printf ("\n\tsoubor neni mozno otevrit");
}

plus_slovo (adr)            /* načti slovo z adr do buffslova */

        char *adr;
{
        register char *s;
        int i;

        s = buffslovo;
        for (i=0; i < MAX_DELKA; i++)
```

```
                if (isalnum (*(adr+i)))
                        *s++ = *(adr+i);
                else
                        break;
        *s = NULL;
}

dalsi_slovo (slovo)                 /* najdi následující slovo */

        char *slovo;
{
        register char *s;

        s = slovo;
        while (*s && isalnum (*s))
                ++s;
        while (*s && !isalpha (*s))
                ++s;
        return *s ? s : NULL;
}

dej_slovo ()                /* najdi slovo a vrať pointer na něj */

{
        register char *s;

        if (ukaz == buff)
                if (fgets (buff,BUF,fd) == NULL)
                        return NULL;
        s = ukaz;
        while (*s && !isalpha (*s))
                ++s;
        if (!*s)
        {
                ukaz = buff;
                return dej_slovo ();
        }
        if (!(ukaz = dalsi_slovo (s)))
                ukaz = buff;
        return s;
}

struct uzel_stromu *
strom (p,w)

        struct uzel_stromu *p;
        char *w;
{
        struct uzel_stromu *talloc ();
        int pom;

        if (p == NULL)
        {
                p = talloc ();
                p -> slovo = strsave (w);
                p -> levy = NULL;  p -> pravy = NULL;
        }
```

```
        else if ((pom = strcmp (w,p -> slovo)) == 0)
                        ;
                else if (pom < 0)
                        p -> levy=strom(p->levy,w);
                     else       p->pravy=strom(p->pravy,w);
        return p;
}

struct uzel_stromu *
talloc ()

{
        return ((struct uzel_stromu *) alloc(sizeof (*ptr)));
}

strsave (s)

        char *s;
{
        char *p;

        if((p=alloc(strlen(s)+1))!=NULL)
                strcpy(p,s);
        return p;
}

napln (p)           /* načti adresy slov do tab podle abecedy */

        struct uzel_stromu *p;
{
        if (p!=NULL)
        {
                napln (p -> levy);
                tab [k] = p -> slovo;   k++;
                napln (p -> pravy);
        }
}


        CVIČENÍ:        6 - 3
        --------------------

#define         EOF     '0'
#define         MAX_DELKA       20
#define         ERROR   (-1)
#define         BUF     128
#define         NULL    0
#define         TRUE    1

struct uzel_stromu
{
        char *slovo;
        int radky[10];
        struct uzel_stromu *levy;
        struct uzel_stromu *pravy;
}*ptr;
```

```
char buff [BUF];
char buffslovo [MAX_DELKA];
char *ukaz;
int radek;

main ()

{
        struct uzel_stromu *koren, *strom ();
        char *c;

        radek = 0;  koren = NULL;
        ukaz = buff;
        while (TRUE)
        {
                if (!(c = dej_slovo ()))
                        break;
                plus_slovo (c);
                koren = strom (koren,buffslovo);
        }
                tisk (koren);
}

plus_slovo (adr)            /* načti slovo z adr do buffslova */

        char *adr;
{
        register char *s;
        int i;

        s = buffslovo;
        for (i=0; i < MAX_DELKA; i++)
                if (isalnum (*(adr+i)))
                        *s++ = *(adr+i);
                else    break;
        *s = NULL;
}

dalsi_slovo (slovo)             /* vrať pointer na další slovo */

        char *slovo;
{
        register char *s;

        s = slovo;
        while (*s && isalnum (*s))
                ++s;
        while (*s && !isalpha (*s))
                ++s;
        return *s ? s : NULL;
}

dej_slovo ()                                /* najdi slovo v buff */

{
        register char *s;
```

```
        if (ukaz == buff)
                if (nacti_radek ()==NULL)
                        return NULL;
        s = ukaz;
        while (*s && !isalpha (*s))
                ++s;
        if (!*s)
        {
                ukaz = buff;    return dej_slovo ();
        }
        if (!(ukaz = dalsi_slovo (s)))
                ukaz = buff;
        return s;
}

nacti_radek ()                          /* načti řádek ze vstupu */

{
        int i;
        char c;

        for (i=0;i < BUF-1 && (c=getchar()) != EOF &&
                                                c != '\n';i++)
                buff[i] = c;
        if (c == '\n')
        {
                buff [i] = c;   buff[++i] = NULL;
        }
        radek++;
        if (c==EOF)
                return NULL;
}

struct uzel_stromu *
strom (p,w)

        struct uzel_stromu *p;
        char *w;
{
        struct uzel_stromu *talloc ();
        int i;
        int pom;

        if (p == NULL)
        {
                p = talloc ();
                p -> slovo = strsave (w);
                p -> levy = NULL;
                p -> pravy = NULL;
                p -> radky[0] = radek;
                p -> radky[1] = NULL;
        }
        else if ((pom = strcmp (w,p -> slovo)) == 0)
             {
                i = 0;
                while (p -> radky[i])
                        i++;
```

```
                p -> radky[i] = radek;
                i++;
                p -> radky[i] = NULL;
            }
                else if (pom < 0)
                        p -> levy=strom(p->levy,w);
                else
                        p->pravy=strom(p->pravy,w);
        return p;
}

struct uzel_stromu *
talloc ()
{
        return ((struct uzel_stromu *) alloc(sizeof (*ptr)));
}

strsave (s)
        char *s;
{
        char *p;

        if((p=alloc(strlen(s)+1))!=NULL)
                strcpy(p,s);
        return p;
}

tisk (p)                /* tiskni slova a řádky jejich výskytu */

        struct uzel_stromu *p;

{
        int i;

        if (p!=NULL)
        {
                tisk (p -> levy);
                printf ("\n%s\t\t",p -> slovo);
                for (i = 0;p -> radky [i] != 0; i++)
                        printf (" %d,",p -> radky[i]);
                tisk (p -> pravy);
        }
}
```

CVIČENÍ: 6 - 4

```
#define         EOF     '0'
#define         MAX_DELKA       20
#define         ERROR   (-1)
#define         BUF     128
#define         NULL    0
#define         TRUE    1

struct uzel_stromu
{
```

```
        char *slovo;
        int pocet;
        struct uzel_stromu *levy;
        struct uzel_stromu *pravy;
}*ptr;

char buff [BUF];
char buffslovo [MAX_DELKA];
char *ukaz;
int pom;

main ()

{
        struct uzel_stromu *koren, *strom ();
        char *c;

        koren = NULL;
        ukaz = buff;
        while (TRUE)
        {
                if (!(c = dej_slovo ()))
                        break;
                plus_slovo (c);
                koren = strom (koren,buffslovo);
        }

        pom = koren -> pocet;
        max (koren);
        while (pom)
        {
                tisk (koren);
                --pom;
        }
}

plus_slovo (adr)            /* načti slovo z adr do buffslova */

        char *adr;
{
        register char *s;
        int i;

        s = buffslovo;
        for (i=0; i < MAX_DELKA; i++)
                if (isalnum (*(adr+i)))
                        *s++ = *(adr+i);
                else
                        break;
        *s = NULL;
}

dalsi_slovo (slovo)             /* vrať pointer na další slovo */

        char *slovo;
{
        register char *s;
```

```
        s = slovo;
        while (*s && isalnum (*s))
                ++s;
        while (*s && !isalpha (*s))
                ++s;
        return *s ? s : NULL;
}

dej_slovo ()                            /* najdi slovo v buff */

{
        register char *s;

        if (ukaz == buff)
                if (nacti_radek ()==NULL)
                        return NULL;
        s = ukaz;
        while (*s && !isalpha (*s))
                ++s;
        if (!*s)
        {
                ukaz = buff;
                return dej_slovo ();
        }
        if (!(ukaz = dalsi_slovo (s)))
                ukaz = buff;
        return s;
}

nacti_radek ()                          /* načti řádek ze vstupu */

{
        int i;
        char c;

        for (i=0;i < BUF-1 && (c=getchar()) != EOF &&
                                                c != '\n';i++)

                buff[i] = c;
        if (c == '\n')
        {
                buff [i] = c;
                buff[++i] = NULL;
        }
        if (c==EOF)
                return NULL;
}

struct uzel_stromu *
strom (p,w)

        struct uzel_stromu *p;
        char *w;
{
        struct uzel_stromu *talloc ();
        int i;
        int pom;
```

```
        if (!p)
        {
                p = talloc ();
                p -> slovo = strsave (w);
                p -> levy = NULL;
                p -> pravy = NULL;
                p -> pocet = 1;
        }
        else if ((pom = strcmp (w,p -> slovo)) == 0)
                p -> pocet++;
                else if (pom < 0)
                        p -> levy=strom(p->levy,w);
                        else
                                p->pravy=strom(p->pravy,w);
        return p;
}

struct uzel_stromu *
talloc ()

{
        return ((struct uzel_stromu *) alloc(sizeof (*ptr)));
}

strsave (s)

        char *s;
{

        char *p;

        if((p=alloc(strlen(s)+1))!=NULL)
                strcpy(p,s);
        return p;
}

tisk (p)                /* tiskni slova podle frekvence výskytu */

        struct uzel_stromu *p;
{
        int i;

        if (p)
        {
                tisk (p -> levy);
                if (p -> pocet == pom)
                        printf ("\n%2d\t%s", p -> pocet,
                                                p -> slovo);
                tisk (p -> pravy);
        }
}

max (p)                         /* vyhledej max frekvenci výskytu */

        struct uzel_stromu *p;
{
```

```
        if (p)
        {
                max ( p -> levy);
                if (p -> pocet > pom)
                        pom = p -> pocet;
                max ( p -> pravy);
        }
}
```

CVIČENÍ: 6 - 5

```
#define BUF     101
#define EOF     (-1)
#define MAX_DELKA       20
#define PISMENO 'a'
#define CISLICE '0'
#define NULL    0
#define HASHVEL 100

int buf[BUF];
int bufp;

struct polozky
{
        char *jmeno;
        char *def;
        struct polozky *next;
}*ptr;

static struct polozky *hashtab [HASHVEL];

main ()

{
        char slovo[MAX_DELKA];
        int t;
        struct polozky *np;

        init ();
        napln ();
        puts ("\n");
        if ((t = getword (slovo,MAX_DELKA)) == PISMENO)
        {
                if (strcmp (slovo,(np = hashtab [hash (slovo)])
                                        -> jmeno)==0)
                        printf ("\n%s\t%s",np -> jmeno,
                                                np -> def);
                else
                        printf ("\nslovo neni v tabulce ");
        }
}

hash (s)                                /* transformace řetězce s */

        char *s;
```

```
        int hashval;

        for (hashval = 0; *s != '\0';)
                hashval += *s++;
        return (hashval % HASHVEL);
}
struct polozky *                        /* prohledávání tabulky */
lookup (s)

        char *s;
{
        struct polozky *np;

        for (np = hashtab [hash (s)];np == NULL;
                                                np = np -> next)
        {
                if (strcmp (s,np -> jmeno) == 0)
                        return (np);
        }
        return (NULL);
}

struct polozky *                        /* ulož (jm,d) do hashtab */
install (jm, d)

        char *jm, *d;.
{
        struct polozky *np, *talloc ();
        char *strsave ();
        int hashval;

        if ((np = lookup (jm)) == NULL)
        {
                np = talloc ();
                if (np == NULL)
                        return (NULL);
                if ((np -> jmeno = strsave (jm)) == NULL)
                        return (NULL);
                hashval = hash (np -> jmeno);
                np -> next = hashtab [hashval];
                hashtab [hashval] = np;
        }
        else free (np -> def);
        if ((np -> def = strsave (d)) == NULL)
                return (NULL);
        return (np);
}

napln ()                /* vkládání jmen a definic do struktur */

{
        struct polozky *np;
        int t1,t2;
        char s1[MAX_DELKA], s2[MAX_DELKA];

        while ((t1 = (getword (s1,MAX_DELKA)))!=EOF)
```

```
        {
                if ((t2 = getword (sl2,MAX_DELKA)) != EOF)
                        {
                        if ((np=install (sl1, sl2))!=NULL)
                                printf ("\n%s\t\t%s",np->jmeno,
                                                        np->def);
                        }
                else break;
        }
}

getword (s,lim)                                 /* načti slovo */

        char s[];
        int lim;
{
        int c,i;

        while ((type (c = getch ())) != PISMENO && c != EOF)
                ;
        if (c == EOF)
                return (c);
        s[0] = c;
        for (i=1;(type (c=getch ())) == PISMENO
                                || type (c) == CISLICE; i++)
                if(i < lim+1)
                        s[i] = c;
        if (i < lim+1)
        {
                ungetch (c);
                s[i] = '\0';
                return (PISMENO);
        }
        else
                return (EOF);
}

getch ()
{
        return ((bufp > 0 ? buf[--bufp] : getchar ()));
}

ungetch (c)

        int c;
{
        if (bufp > BUF)
                printf ("ungetch:prilis mnoho znaku");
        else
                buf [bufp++] = c;
}

init ()

{
        bufp = 0;
}
```

```
type (c)

        int c;
{
        if(c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
                return (PISMENO);
        else if (c >= '0' && c <= '9')
                return (CISLICE);
        else
                return (c);
}

strsave (s)

        char *s;
{
        char *p;

        if ((p = alloc (strlen (s) + 1)) != NULL)
                strcpy (p,s);
        return (p);
}

struct polozky *
talloc ()

{
        return ((struct polozky *) alloc (sizeof (*ptr)));
}


        CVIČENÍ:        7 - 1
        --------------------

#define ANO     1
#define NE      0
#define EOF     (-1)
#define N       30      /* nejvýše N-1 znaků v řádku */

main ()         /* rozděl řádek za posledním nemezerovým
                   znakem před N-tým znakem; nepísmenové a
                   nemezerové znaky tiskne v osmičkovém kódu */
{
        int c,pocet,pocet1,pozice;
        char radek[N];
        unsigned byla_mezera;

        pocet=0;                /* počet zobrazovaných znaků */
        pocet1=0;               /* počet načtených znaků */
        pozice=N+1;             /* pozice poslední skupiny
                                   mezerových znaků */
        byla_mezera=NE;         /* indikace zda předchozí
                                   znak byl mezerový */

        for ( c = 0; c < N; c ++ )
                radek [c] = '\0';
        while ((c=getchar())!=EOF&&pocet<N)
```

```
        {
                radek[poceti]=c;
                pocet +=zobr(c)%2;
                if (c==' '||c=='\t'||c=='\n')
                                /* mezerové znaky */
                {
                        if (pocet>0)
                        {
                                if (byla_mezera==NE)
                                {
                                        pozice=poceti;
                                        byla_mezera=ANO;
                                }
                        }
                        else     /* nový řádek nezačíná meze-
                                    rovým znakem */
                        { --pocet; --poceti; }
                }

                else    if (byla_mezera==ANO)
                                byla_mezera=NE;
                ++pocet; ++poceti;
                if (pocet>=N&&pozice<N)
                {
                        radek[pozice]='\0';
                        tisk (poceti,radek);
                        poceti=presun(radek,pozice+1,&pocet);
                        byla_mezera=NE; pozice=N+1;
                }

                        /* přidáním následujícího  e l s e
                           získáme další variantu programu */
                else
                        if (c=='\n'&&pocet>0)
                        {
                                radek[poceti-1]='\0';
                                tisk (poceti,radek);
                                pocet=poceti=0;
                                byla_mezera=NE; pozice=N+1;
                        }
        }
        if (pocet>=N)
                printf ("Prilis dlouhe slovo v textu !!");
        else
                                /* tisk poslední části textu */
        {
                --poceti;
                while((radek[poceti]==' '||radek[poceti]=='\t'
                          ||radek[poceti]=='\n')&&poceti>=0)
                        --poceti;
                radek[poceti+1]='\0';
                tisk (poceti,radek);
        }
}

presun(s,ind,pp)        /* přesouvá znaky řetězce s od indexu
                           ind na počátek řetězce */
```

```
        char s[];
        int ind, *pp;
{
        int i;

        i=*pp=0;
        while (*pp+ind<=N-1 && s[i+ind]!='\0')
        {
                if (i==0)
                        /* hledá se první nemezerový znak */
                        while ((s[ind]==' '||s[ind]=='\t'||
                                s[ind]=='\n')&&ind<=N-1)
                                ++ind;
                if (ind==N)
                        /* ve zbytku řetězce se vyskytují
                           pouze mezerové znaky */
                        return (0);
                s[i]=s[i+ind]; ++i; ++ *pp;
                *pp +=zobr(s[i])*2;
        }
        for ( ind = i; ind < N ;  ind ++ )
                s [ind] = '\0';
        return (i);
}

tisk (i,s)

        int i;
        char s[];
{
        int j;

        for ( j=0; j < i && s[j] !='\0'; j++)
                if ( zobr(s[j]) )
                        printf ("%o",s[j]);
                else
                        printf ("%c",s[j]);
        printf ("\n");
}

zobr (c)

        char c;
{

        if ( (c>='a'&&c<='z')||(c>='A'&&c<='Z')||c=='\f'||
             (c>='0'&&c<='9')||c==' '||c=='\t'||c=='\n')
                return (0);
        return (1);
}
```

CVIČENÍ: 7 - 2

```
#include <b:stdio.h>
#define BUF     128

main (argc,argv)        /* porovnává dva soubory; tiskne první
                                řádek, kde se liší              */
        int argc;
        char *argv [];
{
        char *radek1, *radek2;
        char *fgets (), *alloc ();
        FILE *fd1, *fd2, *fopen ();
        int i;

        if ( argc != 3 )
                printf ("spatny pocet parametru");
        else
        {
           if ( ! strcmp ( * ++ argv,* ++ argv ) )
                        argc = 0;
           else

           {
                if ((fd1 = fopen ( * -- argv,"r",BUF)) == NULL
                   ||(fd2 = fopen ( * ++argv,"r",BUF)) == NULL)
                        printf("\njeden soubor nejde otevrit");
                else
                {       radek1 = alloc (BUF + 1);
                        radek2 = alloc (BUF + 1);
                        while ( 1 )
                        {       for ( i = 0; i <= BUF; i ++)
                                      radek1[i]=radek2[i]='\0';
                                if ( fgets (radek1,BUF,fd1)
                                                == NULL )
                                {
                                   if ( fgets (radek2,BUF,fd2)
                                                == NULL )
                                        argc = 0;
                                   else argc = 1;
                                   break;
                                }
                                else
                                    if ( fgets (radek2,BUF,fd2)
                                                == NULL )
                                     {  argc = 2; break; }
                                if ( strcmp (radek1,radek2) )
                                {       i = poz(radek1,radek2);
                                        argc = 4; break; }
                        }
                        fclose (fd1); fclose (fd2);
                }
           }

           switch ( argc )
           {
           case 0: printf ("soubory se rovnaji");  break;
```

```
            case 1: printf("soubor %s jiz skoncil",* -- argv);
                    printf ("\nsoubor %s je delsi",* ++ argv);
                    break;
            case 2: printf ("soubor %s je delsi\n",* -- argv);
                    printf("soubor %s jiz skoncil",* ++ argv);
                    break;
            case 4: printf ("%s\n\t%s",* -- argv,radek1);
                    printf ("\n%s\n\t%s\n",* ++ argv,radek2);
                    printf ("soubory se lisi na %d .pozici",i);
            }
        }
}

poz (s1,s2)

        char *s1, *s2;

{
        int i;

        for ( i = 0; ; i ++ )
                if ( *s1  != *s2 )
                        return(i + 1);
                else
                        if ( *s1 == '\0' )
                                return (0);
                        else
                        {
                                s1 ++;  s2 ++;
                        }
}
```

CVIČENÍ: 7 - 3

```
#include <b:stdio.h>
#define MAX     1000
#define EOF       26
#define BUF      128

main (argc,argv)   /* najdi všechny řádky obsahující řetězec */

        int argc;
        char *argv [];

{
        char radek [MAX], *vzor, *fgets ();
        FILE *fd, *fopen ();

        if ( argc < 2 )
                printf ("nebyl zadan retezec");
        if ( argc == 2 )
                while (getline (radek,MAX + 1) > 0)
                        if (index (radek,* ++ argv) >= 0)
                                printf ("%s",radek);
        vzor = * (argv + argc - 1);
```

```
        while ( argc -- > 2 )
        {
                if ((fd = fopen ( * ++ argv,"r",BUF)) == NULL)
                {
                        printf ("\nnejde otevrit %s",*argv);
                        break;
                }
                else
                {
                        printf ("\n%s",*argv);
                        while (fgets (radek,MAX,fd) != NULL)
                                if (index (radek,vzor) >= 0)
                                        printf("\n\t%s",radek);
                }
                fclose (fd);
        }
}

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

        i = 0;
        while (-- lim > 0 && (c=getchar())!=ERROR && 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);
}
```

CVIČENÍ: 7 - 4

```
#include <b:stdio.h>
#define MAX       65            /* počet znaků na řádek */
#define EOF       26
#define BUF      128
#define RAD_STR   55            /* počet řádků na stránku */

main (argc,argv)        /* tiskni soubory s číslem stránky */

        int argc;
        char *argv [];

{
        char radek [MAX];
        char *fgets ();
        FILE *fd, *fopen ();
        int i, j;

        if ( argc < 2 )
                printf ("nebyl zadan zadny soubor");
        while ( argc -- > 1 )
        {
                if ((fd = fopen ( * ++ argv,"r",BUF)) == NULL)
                {
                        printf ("\nnejde otevrit %s",*argv);
                        break;
                }
                else
                {
                        i = 1;  j = 0;
                        printf("\f%s\t\t\t\t\t\tstrana %d",
                                                *argv,i);
                        while (fgets (radek,MAX,fd) != NULL)
                        {
                          j ++;
                          printf("\n%s",radek);
                          if ( j == RAD_STR )
                          {
                            i ++;   j = 0;
                            printf("\f%s\t\t\t\t\t\tstrana %d",
                                                *argv,i);
                          }
                        }
                }
                fclose (fd);
        }
}
```