Kapitola 3

Řízení běhu programu

Příkazy pro řízení běhu programu určují pořadí výpočtů. V dřívějších příkladech jsme se již s nejběžnějšími řídícími konstrukcemi setkali; v této kapitole probereme i zbývající a podíváme se detailněji na ty, o nichž jsme již hovořili.

3.1 Příkazy a bloky

Z výrazů typu `x = 0` nebo `i++` nebo `printf(...)` se stanou *příkazy*, pokud za nimi bude následovat středník, například

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

V jazyce C středník slouží jako ukončující znak příkazu, nikoli jejich oddělovač, jak je tomu v jazycích, jako je Pascal.

Složené závorky { a } se používají pro seskupení deklarací a příkazů do složených příkazů neboli *bloků*, čímž se stanou syntakticky ekvivalentní jedinému příkazu. Zjevným příkladem jsou složené závorky obklopující příkazy funkcí; jiným příkladem jsou složené závorky okolo příkazů za `if`, `while`, `else` nebo `for`. (Proměnné lze deklarovat uvnitř *libovolného* bloku; o tom budeme hovořit ve čtvrté kapitole.) Za pravou složenou závorkou ukončující blok se žádný středník nepíše.

3.2 If-else

Příkaz `if-else` se používá, chceme-li vyjádřit rozhodování. Jeho formální syntaxe je

```
if (výraz)
    příkaz1
else
    příkaz2
```

část `else` je nepovinná. Nejprve je vyhodnocen *výraz*; je-li pravdivý (to jest, má-li nenulovou hodnotu), je vykonán $příkaz_1$. Je-li nepravdivý (nulový) a je-li uvedena část `else`, je vykonán $příkaz_2$.

Protože `if` jednoduše testuje číselnou hodnotu výrazu, je možné použít jisté programátorské zkratky. Příkladem, který se sám nabízí, je

```
if (výraz)
```

místo

```
if (výraz != 0)
```

Někdy je tento zápis jasný a přirozený; jindy nemusí být jeho význam zřejmý.

Protože je část `else` příkazu `if-else` nepovinná, může dojít k nejednoznačné situaci, vynecháme-li ji v posloupnosti do sebe vnořených příkazů `if`. Tuto nejednoznačnost řeší kompilátor připojením `else` k nejbližšímu předchozímu `if` bez `else`. Například v

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

náleží `else` vnitřnímu `if`, což jsme zvýraznili odsazením. Chceme-li něco jiného, musíme použít složené závorky:

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

Tato nejednoznačnost je obzvláště destruktivní v situacích, jako je tato:

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

Odsazení jasně vyjadřuje záměr, ale ke kompilátoru se tato informace nedostane a ten sdruží `else` s vnitřním `if`. Tento typ chyby se těžko hledá; u vnořených `if` se vyplatí uvádět složené závorky.

Mimochodem, všimněte si, že po příkazu `z = a` v

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

následuje středník, a to proto, že gramaticky za `if` následuje *příkaz* a výraz typu `"z = a;"` je vždy ukončen středníkem.

3.3 Else-if

Konstrukce

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

se objevuje tak často, že stojí za to ji probrat odděleně. Tato řada příkazů `if` představuje nejobecnější způsob, jak zapsat vícecestné rozhodování. *Výrazy* jsou postupně vyhodnocovány; je-li jeden *výraz* pravdivý, pak je vykonán odpovídající *příkaz* a tím je celý řetěz ukončen. Jako vždy může kód celého *příkazu* tvořit jediný příkaz nebo skupina příkazů ve složených závorkách.

Poslední část `else` řeší případ „nic z předchozího“, tedy když není splněna žádná z podmínek. Někdy pro tento případ není nutná žádná explicitní akce; v takovém případě můžeme koncové

```
else
   příkaz
```

vynechat, nebo je můžeme použít pro kontrolu chyb, tedy pro odchytávání „nemožného“ stavu.

Pro ilustraci trojcestného rozhodování si ukážeme funkci implementující binární vyhledávání, která zjistí, zda se daná hodnota `x` nachází v uspořádaném poli `v`. Prvky pole `v` musí být uspořádány ve vzestupném pořadí. Funkce vrací pozici (číslo mezi `0` a `n-1`), jestliže se `x` nachází ve `v`, a `-1`, jestliže se ve `v` nenachází.

Binární vyhledávání nejprve porovná vstupní hodnotu `x` s prostředním prvkem pole `v`. Je-li `x` menší než prostřední hodnota, soustředí se hledání na dolní polovinu tabulky, v opačném případě na horní polovinu tabulky. V obou případech je dalším krokem porovnání `x` se středním prvkem zvolené poloviny. Tento proces dělení rozsahu na poloviny pokračuje, dokud není hodnota nalezena nebo dokud je v čem hledat.

```
/* binvyhl: najdi x ve v[0] <= v[1] <= v[2] ... <= v[n-1] */
int binvyhe(int x, int v[], int n)
{
   int dolni, horni, prostredni;

   dolni = 0;
   horni = n-1;
   while (dolni <= horni) {
      mid = (dolni + horni) / 2;
      if (x < v[prostredni])
         horni = prostredni - 1;
      else if (x > v[prostredni])
         dolni = prostredni + 1;
      else /* nalezena shoda */
         return prostredni;
   }
   return -1; /* shoda nenalezena */
}
```

Fundamentálním rozhodnutím je, zda je `x` menší, větší nebo rovno prostřednímu prvku `v[prostredni]`; to je práce pro `if-else`.

Cvičení 3-1. Naše binární vyhledávání provádí uvnitř cyklu dva testy, i když by stačil jeden (za cenu většího počtu testů mimo cyklus). Napište verzi s jediným testem v cyklu a změřte rozdíl při běhu programu.

3.4 Switch

Příkaz `switch` slouží k vícecestnému rozhodování testujícímu, zda výraz odpovídá jedné z několika *konstantních* celočíselných hodnot, a podle toho se větví.

```
switch (výraz) {
   case konstantní-výraz: příkazy
   case konstantní-výraz: příkazy
   default: příkazy
}
```

Každá alternativa je označena jednou nebo několika celočíselnými konstantami nebo konstantními výrazy. Výpočet začne alternativou, která odpovídá hodnotě výrazu. Výrazy označující jednotlivé alternativy se musí lišit. Alternativa `default` je vykonána, jestliže hodnota výrazu neodpovídá žádné jiné alternativě. Alternativa `default` je nepovinná; není-li uvedena a neodpovídá-li žádná alternativa, nic se nestane. Alternativy (včetně `default`) mohou být zapsány v libovolném pořadí.

V první kapitole jsme napsali program, který počítal výskyty všech číslic, bílých znaků a všech ostatních znaků pomocí posloupnosti `if … else if … else`. Zde je stejný program s příkazem `switch`:

```
#include <stdio.h>

main() /* spočítá číslice, bílé znaky a ostatní */
{
   int z, i, pbilych, postatnich, pcisel[10];

   pbilych = postatnich = 0;
   for (i = 0; i < 10; i++)
      pcisel[i] = 0;
   while ((z = getchar()) != EOF) {
      switch (z) {
      case '0': case '1': case '2': case '3':
      case '4': case '5': case '6': case '7':
      case '8': case '9':
         pcisel[z-'0'] ++;
         break;
      case ' ':
      case '\n':
      case '\t':
         pbilych++;
         break;
      default:
         postatnich++;
         break;
      }
   }
   printf("čísla = ");
   for (i = 0; i < 10; i++)
      printf(" %d", pcisel[i]);
   printf(" , neviditelné znaky = %d, ostatní = %d\n",
              pbilych, postatnich);
   return 0;
}
```

Příkaz `break` způsobí okamžité ukončení příkazu `switch`. Protože alternativy slouží pouze jako návěští, po vykonání příkazů v jedné akternativě *pokračuje* výpočet dalšími alternativami, pokud nezadáme příkaz pro ukončení. `break` a `return` jsou nejběžnější způsoby opuštění příkazu `switch`. Příkaz `break` lze také použít pro vynucené opuštění cyklů `while`, `for` a `do`, jak ukážeme později v této kapitole.

Pokračování dalšími alternativami je kontroverzní výhoda. Na jednu stranu umožňuje připojit několik alternativ k jedné akci, jako u čísel v předchozím příkladu. Na druhou stranu je nutné ukončit každou z alternativ příkazem `break`, abychom zabránili vykonávání příkazů dalších alternativ. Pokračování dalšími alternativami není robustní a je náchylné k chybám při modifikacích programu. S výjimkou vícenásobných návěští pro jediný výpočet by se pokračování dalšími alternativami mělo využívat zřídka a jejich použití by mělo být dobře popsáno v komentářích.

Kvůli dobrému programátorskému stylu umístěte `break` i za poslední alternativu (zde `default`), i když to logika programu nevyžaduje. Někdy v budoucnu bude možná na konec přidána další alternativa a tento kousek defenzivního programování vám ušetří spoustu nervů.

Cvičení 3.2. Napište funkci `escape(r,s)`, která při kopírování řetězce `s` do `r` převede znaky jako znak nového řádku nebo tabulátor na viditelné řídicí posloupnosti typu `\n` a `\t`. Použijte příkaz `switch`. Napište také funkci pro převod v opačném směru, tj. z řídicích posloupností na skutečné znaky.

3.5 Cykly – while a for

S cykly `while` a `for` jsme se již setkali. V konstrukci

```
while (výraz)
    příkaz
```

se vyhodnotí *výraz*. Je-li nenulový, provede se *příkaz* a *výraz* se znovu vyhodnotí. Tento cyklus pokračuje, dokud je *výraz* nenulový. Pak program pokračuje příkazy, které se nacházejí za *příkazem*.

Příkaz `for`

```
for(výraz1; výraz2; výraz3)
    příkaz
```

je ekvivalentní konstrukci

```
výraz1;
while (výraz2) {
    příkaz
    výraz3;
}
```

s výjimkou chování příkazu `continue`, který je popsán v oddílu 3.7.

Gramaticky se tři komponenty cyklu `for` řadí mezi výrazy. Nejčastěji představují $výraz_1$ a $výraz_3$ přiřazení nebo volání funkce a $výraz_2$ je relačním výrazem. Kteroukoli z těchto tří částí je možné vynechat, i když středníky musí zůstat. Vynecháme-li $výraz_1$ nebo $výraz_3$, budou prostě chybět ve výše uvedeném rozvoji. Není-li ale přítomen test $výraz_2$, je tato část brána jako stále pravdivá. Proto představuje

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

nekonečný cyklus, ze kterého je nutno vyskočit pomocí příkazů `break` či `return`.

Volba mezi `while` a `for` závisí zcela na osobních preferencích. Například použití `while` se nabízí v příkazu

```
while ((z = getchar()) == ' ' || z == '\n' || z == '\t')
   ;  /* přeskočí bílé znaky */
```

protože tento cyklus neobsahuje žádnou inicializaci ani reinicializaci.

Příkaz `for` se hodí v případech, kdy je potřeba provést jednoduchou inicializaci a reinicializaci, protože v něm se řídicí příkazy nacházejí blízko sebe a na vrcholu cyklu jsou dobře viditelné. To je evidentní u příkazu

```
for(i = 0; i < n; i++)
   ...
```

což je idiom (ustálený obrat) jazyka C pro zpracování prvních n prvků pole. Jde o jasnou analogii cyklu `DO` jazyka Fortran nebo cyklu `for` jazyka Pascal. Avšak nejde o dokonalou analogii, protože index a meze cyklu `for` jazyka C můžeme uvnitř cyklu měnit a indexová proměnná `i` si uchovává svoji hodnotu i po ukončení cyklu (ať skončí z jakéhokoli důvodu). Cyklus `for` se nemusí držet aritmetických postupů, protože komponentami cyklu mohou být libovolné výrazy. Na druhou stranu, cpát navzájem nesouvisející výpočty do míst pro řídící příkazy není dobrou programátorskou praxí.

Jako větší příklad uvádíme další verzi funkce `atoi` pro převod řetězců na jejich číselné ekvivalenty. Tato verze je o něco obecnější než verze z druhé kapitoly; dokáže se vypořádat s úvodními bílými znaky a volitelným znaménkem + nebo -. (V kapitole 4 narazíte na funkci `atof`, která provádí stejný převod pro čísla s pohyblivou řádovou čárkou.)

Ve struktuře programu se odráží formát vstupních dat:

přeskoč bílé znaky, pokud existují
získej znaménko, pokud existuje
získej celočíselnou část a převeď ji

Každý krok vykoná svoji část a předá data v konzistentním stavu k dalšímu zpracování. Celý proces skončí s prvním znakem, který nemůže být součástí čísla.

```
#include <ctype.h>

/* atoi: převede r na celé číslo; 2. verze */
int atoi(char r[])
{
   int i, n, znamenko;

   for (i = 0; isspace(r[i]); i++) /* přeskočí bílé znaky */
      ;
   znamenko = (r[i] == '-') ? -1 : 1;
   if (r[i] == '+' || r[i] == '-') /* přeskočí znaménko */
      i++;
   for (n = 0; isdigit(r[i]); i++)
      n = 10 * n + (r[i] - '0');
   return znamenko * n;
}
```

Standardní knihovna poskytuje propracovanější funkci `strtol` pro převod řetězců na velká celá čísla; viz oddíl 5 přílohy B.

Výhody centralizace řízení cyklu se naplno projeví, máme-li několik do sebe vnořených cyklů. Následující funkce implementuje Shellův algoritmus pro třídění pole celých čísel. Základní myšlenka tohoto třídicího algoritmu, který vymyslel D. L. Shell v roce 1959, spočívá v tom, že v raných fázích jsou porovnávány vzdálené prvky, nikoli sousední, jako je tomu u jednodušších třídicích algoritmů. Tím se daří rychle eliminovat velký chaos, takže pozdější fáze už mají méně práce. Interval mezi porovnávanými prvky se postupně snižuje k jedné, čímž se tento algoritmus prakticky stává metodou založenou na výměně sousedních prvků.

```
/* shellsort: setřídí v[0]...v[n-1] v rostoucím pořadí */
void shellsort(int v[], int n)
{
   int mezera, i, j, pom;

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

Máme zde tři vnořené cykly. První určuje mezeru mezi porovnávanými prvky, kterou zmenšuje z `n/2` při každém průchodu cyklem o polovinu, dokud neklesne na nulu. Prostřední cyklus prochází jednotlivými prvky. Vnitřní cyklus porovnává dvojice prvků, jejichž indexy se liší o `mezera`, a přehazuje ty, které jsou ve špatném pořadí. `mezera` se nakonec zmenší na `1` a prvky se seřadí ve správném pořadí.

Všimněte si, že díky obecnosti cyklu `for` má vnější cyklus stejnou formu jako ostatní, i když nejde o průchod aritmetickou posloupností.

Posledním operátorem jazyka C je čárka „,“, která nachází uplatnění nejčastěji v cyklu `for`. Dvojice výrazů oddělených čárkou je vyhodnocována zleva doprava a typ a hodnota výsledku jsou dány typem a hodnotou pravého operandu. Proto je v příkazu `for` možné umístit do různých částí více výrazů, například pro paralelní zpracování dvou indexů. To ilustruje funkce `obrat(r)` obracející na místě řetězec `r`.

```
#include <string.h>

/* obrat: obrací na místě řetězec r */
void obrat(char r[]);
{
   int z, i, j;

   for (i = 0, j = strlen(r) - 1; i < j; i++, j--) {
      z = r[i];
      r[i] = r[j];
      r[j] = z;
   }
}
```