4 Řídicí struktury v programování

Kreslíková, 22. 9. 2023

4.1 Úvod

V této kapitole se budeme zabývat řídícími strukturami programu. Jde o ty části programovacího jazyka, které řídí, jakým směrem se bude ubírat výpočet. Jsou zde demonstrovány opět pomocí jazyka C, proto zde najdeme řídící struktury, které jsou typické pro procedurální, neboli imperativní paradigma.

V první řadě jsou to podprogramy, které mají v jazyce C podobu funkcí. Umožňují nám rozdělovat složité problémy na menší části a šetřit námahu díky opakovanému používání stejného kódu. Protože problémy řešené pomocí podprogramů můžeme parametrizovat, vede programátora jejich používání k obecnějšímu přemýšlení nad problémy a robustnějšímu návrhu programů.

Dále se zde budeme zabývat příkazy zabudovanými přímo do programovacího jazyka. Tyto příkazy nám umožní vyhodnocovat podmínky a provádět opakované činnosti pomocí cyklů.

Tato kapitola slouží jako přehled těchto řídících struktur. Obsahuje popis jejich syntaxe i příklady použití. Jejich použitím při řešení skutečných algoritmických úloh se budeme zabývat v dalších kapitolách.

4.2 Funkce

Program se na nejvyšší úrovni skládá z funkcí a globálních proměnných. Jedna z funkcí se jmenuje vždy ***main***[1] a musí být v programu vždy uvedena - je to funkce, která je volána jako první po spuštění programu. Tělo každé funkce obsahuje obvykle řadu příkazů a definicí lokálních proměnných. Vytváříme-li program, zapisujeme vlastně posloupnost příkazů. Tato posloupnost bude za běhu programu vykonávána. Naším cílem je provádět právě ty příkazy, které odpovídají zvolenému záměru. Výběr z příkazů je určen stavem dosavadního běhu programu, vstupními údaji a řídicími strukturami, které jsme použili. Příkazy specifikují „tok výpočtu“, neboli „běh programu“. Provádějí se postupně v pořadí, v jakém jsou zapsány

Přemýšlíme-li o tom, jak naprogramovat nějaký problém, je nutné vymyslet algoritmus a zapsat jej pomocí vhodné posloupnosti příkazů (a volání funkcí), která povede k požadovanému řešení problému.

Pomůcka zvláště pro dřívější uživatele Pascalu

V jazyce C se všechny příkazy ukončují středníkem (kromě složeného příkazu, viz dále). Středník je nedílnou součástí příkazu, přesněji řečeno - slouží k jeho ukončení.

Funkce jsou základními stavebními kameny programu v jazyce C. Většina reálných programů obsahuje mnoho funkcí. Funkce umožňují rozdělit složité problémy na množinu jednodušších podproblémů. Jde o základní vyjadřovací prostředek strukturovaného programování.

4.2.1 Obecný formát programu s více funkcemi v jazyce C

```
/*  zde vložit hlavičkové soubory */
/*  zde umístit případné prototypy funkcí */
/*  globální  definice */
návratový-typ funkce1(seznam-parametrů)
{
  /*   tělo funkce1  */
}
návratový-typ funkce2(seznam-parametrů)
{
  /*   tělo funkce2  */
}
.
.
 /*  globální  definice */
návratový-typ funkceN(seznam-parametrů)
{
  /*   tělo funkceN …… */
}

int main(seznam-parametrů)
{
  /*   tělo funkce main   */
}
```

[1] V jiných jazycích než C může být startovní bod programu označen jiným způsobem. Například v Pascalu je za startovní bod považován první příkaz begin, který není součástí žádného podprogramu. V některých jednodušších skriptovacích jazycích program začíná prvním příkazem, který interpret v souboru nalezne.

Položka *návratový-typ* představuje datový typ, jehož hodnotu funkce vrací. Nevrací-li funkce žádnou hodnotu, měl by být její návratový typ **void**. Nepoužívá-li funkce parametry, měl by její seznam parametrů obsahovat klíčové slovo **void**.

Klíčové slovo void
Klíčové slovo **void** představuje prázdný datový typ. Používá se pro označení funkcí, které nevracejí hodnotu, pro označení prázdného seznamu parametrů funkcí a pro vytváření obecného ukazatele. Nelze vytvářet proměnné tohoto typu. Nejde na něj aplikovat ani operátor **sizeof** (viz dále).

Pomocí parametrů (*seznam-parametrů*) můžeme funkci předat podklady pro výpočet, vstupní informace, hodnoty apod. Počet parametrů není omezen – funkce nemusí mít žádný parametr nebo jich může mít desítky.
Pozor: Vysoký počet parametrů svědčí o špatné dekompozici problému.

4.2.2 Deklarace funkce (prototyp funkce)

návratový-typ jménoFunkce (*seznam-parametrů*) ;

Prototyp funkce deklaruje funkci před jejím použitím a před tím než je definována. Prototyp se skládá ze jména funkce, jejího návratového typu a seznamu parametrů. Je ukončen středníkem. Překladač potřebuje znát tyto informace, aby mohl správně zpracovat vyvolání funkce.

Prototyp funkce nalezne využití v hlavičkových souborech a u funkcí, které je potřeba používat dříve, než je známa jejich implementace (blíže viz přednášku o rekurzi). U funkcí, které mají zůstat součástí jediného modulu a není potřeba je sdílet s jinými moduly, není potřeba prototypy dávat do hlavičkových souborů (*.h).

Poznámka:
Funkce **main** prototyp nepotřebuje, protože jde o funkci, která má zvláštní postavení. Každý spustitelný program musí obsahovat jednu funkci **main** a její návratový typ musí být **int**. Všechny její prototypy jsou zabudovány do jazyka. Nemělo by smysl poskytovat funkci **main** ve veřejném rozhraní modulu.

4.2.3 Definice funkce (implementace funkce)

návratový-typ jménoFunkce (*seznam-parametrů*)
{
  seznam-příkazů
}

Příklad: Definice funkce.

```
void mojeFunkce(void)
{
  printf("Toto je test.\n");
}
```

Její prototyp je:

```
void mojeFunkce(void);
```

4.2.4 Volání funkcí

Když je volána funkce, předá se zpracování této funkci. Po dosažení konce jejího těla se zpracování vrací bezprostředně za místo volání funkce. Jinak řečeno když funkce skončí, pokračuje zpracování v místě programu, které následuje bezprostředně za voláním funkce. Každá funkce může volat jakoukoli jinou funkci ve stejném programu, která je viditelná v jejím kontextu. Funkce **main**() se obvykle z žádné jiné funkce nevolá, žádná technická omezení tomu však nebrání.

Použití funkcí pro vracení hodnot

V jazyce C může funkce vracet hodnotu. Vrácenou hodnotu lze použít kdekoliv na pozici výrazu odpovídajícího typu.
Funkce můžeme chápat jako zobecnění pojmu operátor.

Příklad: Použití funkce, která vrací hodnotu.

```
double sqrt (double x);
```

Je nutné zajistit, aby typ hodnoty vracené funkcí odpovídal typu proměnné, které se hodnota přiřazuje. Je také důležité, aby typy argumentů funkce odpovídaly typům, které funkce požaduje.

```
#include <stdio.h>
#include <math.h> // potřebné pro použití sqrt()
int main(void)
{
  double x;
  double vysledek;
  printf("Zadejte racionalni cislo : \n");
  scanf("%lf", &y);
  vysledek = sqrt(y);
  printf("Odmocnina z cisla je: %f\n", vysledek);

  return 0;
}
```

S výsledkem funkce můžeme pracovat jako s jakýmkoli jiným výrazem. Zde je například funkce použita jako parametr funkce *printf():*

```
printf("Odmocnina z cisla je: %f\n",sqrt(y));
```

Příkaz return

Při psaní vlastních funkcí je možné vracet hodnotu volající funkci pomocí příkazu **return**.
Příkaz **return** má obecný formát:

`return` *expression*$_{opt}$ **`;`**

kde *expression* je vrácená hodnota. Pokud má funkce návratový typ **void**, za klíčovým slovem **return** následuje středník. Příkaz **return** se může vyskytnout kdekoli v těle funkce. **Return** ukončuje provádění funkce.
Příkaz **return** bude zmíněn v rámci této přednášky ještě v souvislosti s příkazy skoku (viz kap. 4.3.7.3).

Příklad: Definice funkce pro výpočet druhé mocniny zadaného čísla.

```
#include <stdio.h>

int get_sqr(void); // prototyp

int main(void)
{
  int sqrm = get_sqr( );
  printf("Square: %d", sqrm);

  return 0;
}

int get_sqr(void)  // definice
{
  int num;

  printf("Zadejte cele cislo: \n");
  scanf("%d", &num);
  return num*num; // druhá mocnina čísla
}
```

Použití argumentů funkcí

Některé funkce v jazyce C při zavolání nemusí mít žádný argument, jiné jich mohou mít několik. Aby mohla funkce převzít argumenty, musí být deklarovány speciální proměnné pro příjem těchto argumentů. Nazývají se **formální parametry** funkce. Parametry se deklarují v závorkách následujících za jménem funkce. Při volání funkce dochází k substituci, kdy jsou parametry funkce nahrazeny odpovídajícím počtem, pořadím a typem argumentů.

Příklad: Definice funkce, která sečte dvě čísla a vytiskne jejich součet.

```
#include <stdio.h>

void tisk_soucet(int x, int y);

int main(void)
{
  int num1, num2;

  printf("Zadejte dve cela cisla: ");
  scanf("%d%d", &num1, &num2);
  tisk_soucet(num1,num2);
  return 0;
}

void tisk_soucet(int x, int y)
{
  printf("Soucet cisel je: %d \n", x + y);
}
```

Pokud funkci definujeme dříve než je použita, nemusí být předtím uveden prototyp funkce. Následující funkce ani nevrací žádnou hodnotu.

Příklad: Definice funkce, která nevrací hodnotu.

```
#include <stdio.h>

void tisk_soucet(int x, int y)
{
  printf("Soucet cisel je: %d \n", x + y);
}

int main(void)
{
  int num1, num2;

  printf("Zadejte dve cela cisla: ");
  scanf("%d%d", &num1, &num2);
  tisk_soucet(num1,num2);
  return 0;
}
```

Funkce s parametry, která vrací hodnotu

Příklad: Definice funkce, která sečte dvě čísla.

```
#include <stdio.h>

int soucet(int x, int y)
{
  return x + y;
}

int main(void)
{
  int num1, num2;

  printf("Zadejte dve cela cisla: ");
  scanf("%d%d", &num1, &num2);
  printf("Soucet cisel je: %d \n", soucet(num1,num2));
  return 0;
}
```

4.3 Příkazy

Syntaxe příkazu [C99]:

statement:

- *labeled-statement*
- *compound-statement*
- *expression-statement*
- *selection-statement*
- *iteration-statement*
- *jump-statement*

4.3.1 Složený příkaz (blok)

Všude v C, kde se může vyskytovat příkaz, se může vyskytovat i složený příkaz. Složený příkaz je tvořen seznamem položek bloku. Konstrukce, která složený příkaz vymezuje, začíná levou a končí pravou složenou závorkou { }. Za ukončovací závorkou se nepíše středník.

Položka bloku může obsahovat příkazy a lokální deklarace a definice. Jejich platnost je omezena na blok a případné další vnořené bloky. Vnořený blok také není ukončen středníkem, protože představuje složený příkaz a jeho konec je jasně určen.
Deklarace jednotlivých lokálních proměnných se mohou v bloku vyskytovat kdekoli (stará norma umožňovala deklarace (definice) pouze na začátku bloku).
Není na škodu si uvědomit, že tělo každé funkce je složeným příkazem. Proto jsme mohli v těle funkce main() definovat a používat lokální proměnné.
Zkontrolovat! Syntaktický diagram funkce?

Syntakticky můžeme složený příkaz popsat následovně:
compound-statement:
{ *block-item-list*$_{opt}$ **}**
block-item-list:
block-item
block-item-list block-item
block-item:
declaration
statement

Blok tedy může obsahovat žádnou, jednu či více deklarací. Deklarace jednotlivých lokálních proměnných se mohou v bloku vyskytovat kdekoli (stará norma umožňovala deklarace pouze na začátku bloku). Dále blok může obsahovat libovolně dlouhou posloupnost příkazů.

Příklad: Převod stop na metry (volba 1) a metry na stopy (volba 2). Jde o ilustrativní příklad, ve kterém z prostorových důvodů nejsou ošetřeny všechny vstupy (v programu označeno !!!). To znamená, že program může skončit neošetřenou chybou, když například v místě, kde očekává od uživatele zadání čísla typu **float**, zadáme něco jiného než číslo (např. „abcd“). Ošetření těchto situací probereme později.

```
/*****************************************/
/* Převod stop na metry a metry na stopy   */
/*****************************************/
#include <stdio.h>
const float KOEF_PREVODU=3.28;
void vypisPrevod(int volba)
{
  if(volba == 1)
  {
    float num;
    printf("Zadejte pocet stop: ");
    scanf("%f", &num); // !!!
    printf("%f  stop  je %f metru\n", num, num/KOEF_PREVODU);
  }
  else if(volba == 2)
  {
    printf("Zadejte pocet metru: ");
    float num;
    scanf("%f", &num); // !!!
    printf("%f  metru  je %f stop\n", num, num*KOEF_PREVODU);
  }
```

```
else
  {
    printf("Chybna volba.");
  }
}
int main(void)
{
  int volba = 0;

  printf("Zadejte volbu \n"
         "1: Stopy na metry\n"
         "2: Metry na stopy\n: ");

  if (scanf("%d", &volba) == 1)
    vypisPrevod(volba);
  else
    return EXIT_FAILURE;
  return EXIT_SUCCESS;
}
```

4.3.2 Výraz - příkaz

Výrazem je nejen aritmetický výraz (například `a + b`, či `5 + 1.23 * a`), prostý výskyt konstanty (literálu) či proměnné, ale i funkční volání a přiřazení. Jestliže výraz ukončíme symbolem `;` (středník), získáme výrazový příkaz.

Syntaxe výrazového příkazu [C99]:
expression-statement:
expression$_{opt}$ `;`

Prázdný příkaz
Prázdný příkaz je příkaz, v němž není výrazová část (`;`). Prázdným příkazem je také blok, který neobsahuje žádný příkaz (`{}`). Tato konstrukce není tak nesmyslná, jak se na první pohled může zdát. Často používanou konstrukcí je například cyklus s prázdným tělem, kdy veškerý výkonný kód cyklu je umístěn přímo v podmínce cyklu. Prázdný příkaz nám také dává možnost umístit nadbytečný středník (;) do zdrojového textu.

Příklad: Vzory běžně používaných konstrukcí cyklů s prázdným tělem. Tyto vzory je dobré znát zpaměti.

```
int c;
// přeskočí mezery
while ((c = getchar()) == ' ')
{}

// přeskočí prázdné řádky
while ((c = getchar()) == '\n')
  ;
```

4.3.3 Oblast platnosti identifikátoru

Identifikátor, je platný od místa své deklarace až do konce bloku, funkce, či modulu, v němž tato deklarace proběhla. Na úrovni souboru je rozsah platnosti deklarace vymezena místem, kde je deklarace dokončena, a koncem překládaného modulu (zdrojového souboru).

Deklarace parametru funkce má rozsah od místa deklarace parametru v rámci definice funkce až do ukončení bloku definice funkce. Pokud se nejedná o definici funkce, končí

rozsah deklarace parametru s deklarací funkce. V rámci bloku je deklarace platná v rozsahu od jejího dokončení až do konce bloku.

Jméno (identifikátor) je v tomto rozsahu viditelné, není-li zastíněno. Například lokální proměnná stejného jména jako globální proměnná zastíní ve svém rozsahu viditelnosti tuto globální proměnnou. Podobně to platí i pro lokální identifikátory ve vnořených blocích.

Příklad: Ukázka zastínění identifikátorů proměnných.

```
char znak = 'A';

int main(void)
{
  int cislo = 1;
  printf("%c%d", znak, cislo);     // A1
  char znak = 'B';                 // zastínění
  printf("%c%d", znak, cislo);     // B1
  {
    char znak = 'C';               // zastínění
    printf("%c%d", znak);          // C1
    float cislo = 2.0;             // zastínění
    printf("%c%.2f", znak, cislo); // C2.00
  }
  printf("%c%d", znak, cislo);     // B1
}
```

Jméno makra je platné od jeho definice (direktivou *#define*) až do místa, kdy je definice odstraněna (direktivou *#undef*, pokud vůbec odstraněna je). Jméno makra nemůže být zastíněno.

Příklad: Ukončení platnosti makra – v tomto případě konstanty.

```
#define N 10
int pole[N] = {0};
#undef N
int lany[N*N] = {0}; //!! zde už hodnotu N nelze použít
```

4.3.4 Podmíněné příkazy

Podmíněné příkazy umožňují větvit program. To znamená vybírat z více možností pokračování běhu programu. Jejich syntaktický zápis je následující [C99]:

selection-statement:

`if (` *expression* **`)`** *statement*

`if (` *expression* **`)`** *statement* **`else`** *statement*

`switch (` *expression* **`)`** *statement*

Kromě podmíněných příkazů ještě existuje podmíněný výraz (`? :`), který slouží pro podmíněný výběr části výrazu. V některých speciálních případech lze tímto výrazem nahradit podmíněný příkaz, ale nelze to doporučit obecně. Podmíněným výrazem se budeme zabývat v další kapitole.

Příkaz if

Význam příkazu **if** je následující: Po vyhodnocení výrazu (*expression*, musí být v závorkách) se v případě jeho nenulové hodnoty provede příkaz za závorkou (*statement*). Po jeho

provedení program pokračuje následujícím příkazem (který už ovšem není ovlivněn podmíněným příkazem **if**). V případě nulového výsledku výrazu se řízení programu předá bezprostředně za podmíněný příkaz. Jinak řečeno se první příkaz přeskočí.

Příklad: Je zadané číslo záporné?

```
/*****************************/
/* Zadané číslo je záporné ?  */
/*****************************/

#include <stdio.h>

int main(void)
{
  int num;

  printf("Zadejte cele cislo:  ");
  scanf("%d", &num);

  if(num < 0)
    printf("\nCislo %d je zaporne.\n", num);

  return 0;
}
```

Poznámka k formátování:

```
// pozor na formátování, takto ne!
if (num < 0)
  printf(…);
  printf(…);  // !

// lépe takto
if (num < 0)
{
  printf(…);
}

printf(…);
```

Příkaz if-else

Příkaz **if** můžeme použít i s variantou **else**. Sémantika příkazu **if-else** je v první části totožná se samotným **if**. Je-li výslednou hodnotou výrazu (*expression*) jedna (nenulová hodnota), provede se příkaz za závorkou (*statement*, první příkaz). V opačném případě, kdy výsledkem je nula, se provede příkaz za **else** (*statement*, druhý příkaz). V obou případech se řízení programu po provedení prvního, respektive druhého, příkazu předá za celý podmíněný výraz.

Někdy se výklad sémantiky předkládá způsobem, kdy se nulové hodnotě říká nepravda a nenulové (jedničce) pravda, jak ostatně známe z logických výrazů. Pak lze ve druhé variantě říci, že je-li podmínka splněna, vykoná se první příkaz, jinak příkaz druhý. Zdůrazněme, že oba příkazy jsou ukončeny středníkem, pokud nejde o bloky. Jak už víme, za blokem se středník nepíše!

Příklad: Zadané číslo je záporné nebo přirozené?

```
/*****************************************/
/* Zadané číslo je záporné nebo přirozené?  */
/*****************************************/

#include <stdio.h>

int main(void)
{
  int num;

  printf("Zadejte cele cislo:  ");
  scanf("%d", &num);

  if(num < 0)
    printf("\nCislo %d je zaporne.\n", num);
  else
    printf("\nCislo %d  je prirozene.\n",num);

  return 0;
}
```

Následující příklad nám ukáže použití podmíněného příkazu **if-else**. Máme vypočíst a zobrazit podíl dvou zadaných celých čísel. Protože je známo, že nulou celočíselně dělit nelze, využijeme podmíněný příkaz k ošetření tohoto omezení.

Příklad: Výpočet podílu dvou celých čísel a zbytku po dělení.

```
/*********************************/
/* Podíl dvou celých čísel        */
/*********************************/
#include <stdio.h>

int main(void)
{
 int a, b;
 printf("Zadejte dve cela cisla:");
 scanf("%d %d", &a, &b);

 if (b == 0)
   printf("\nNulou delit nelze!\n");
 else
 {
   int podil, zbytek;
   podil = a / b;
   zbytek = a % b;
   printf("Jejich podil je: %d, zbytek po deleni je:%d\n",
     podil, zbytek);
 }
 return 0;
}
```

Jako první příkaz po testu **if** je jednoduchý výstup řetězce pomocí printf(). Je ukončen středníkem a následuje klíčové slovo **else**. Příkaz v této části je tvořen blokem. Za blokem středník není. Pokud by tam byl, byl by chápán jako prázdný příkaz.

Vnořování příkazů if
Když je příkaz **if** součástí jiného příkazu **if** nebo **else**, říká se, že je vnořen do vnějšího **if**.

Příklad: Vnořený příkaz **if.**
```
if (count > max)  // vnější if
  if (error)      // vnořený if
    printf("Chyba, zkuste to znovu." );
```

Varianta **else** se pojí vždycky s nejbližším předchozím **if** ve stejném bloku, které ještě nemá připojené **else**. Na následujícím příkladu je dobře vidět význam nesprávného a správného formátování zdrojového kódu.

Příklad: Formátování zdrojového kódu.
```
// takto ne
if (p)
  if (q) printf("p  i  q jsou pravdive\n");
else printf("Ke kteremu prikazu patri toto else?\n");

// takto ano
if (p)
  if (q)
    printf("p  i  q jsou pravdive\n");
  else
    printf("Nyni je prislusnost else jasne patrna.\n");
```

Vnořování může mít i složitější strukturu. Příkaz **if** lze použít i na místě příkazu za **else** (tato konstrukce je obvyklejší). Výsledná konstrukce bývá často nazývána **if-else-if** [C99]:

if (*<expression1>*) *<statement1>*
else if (*<expression2>*) *<statement2>*
else if (*<expression3>*) *<statement3>*
...
else if (*<expressionN>*) *<statementN>*
else *<statementN+1>*

Její často využívanou vlastností je skutečnost, že se provede právě jeden z příkazů. Pokud není poslední příkaz bez podmínky *<statementN+1>* uveden, nemusí se provést žádný. Jinak řečeno, provede se nejvýše jeden.

Z prostorových důvodů je zvykem konstrukce **else-if** neodsazovat, respektive odsazovat pouze vzhledem k prvnímu příkazu **if**.

Zapamatujme si dobře tuto konstrukci. Nejen pro její užitečnost. Později ji budeme moci porovnat s konstrukcí zvanou přepínač.

Ukažme si použití této konstrukce na reálné úloze. Naším úkolem je programově ošetřit, jaký alfanumerický znak byl zadán, případně vydat zprávu o zadání znaku jiného. Protože je jisté, že zadaný znak patří právě do jedné ze tříd malá písmena, velká písmena, číslice a jiné, použijeme konstrukci podmíněného příkazu. Při pouhém použití **if** by pro každou třídu byla znovu testována příslušnost načteného znaku. Bez ohledu, zdali znak již některou z předchozích podmínek splnil. Použitá konstrukce **if-else-if** případné nadbytečné testy odstraní. Vždy bude vybrán právě jeden z příkazů.

Logický výraz, který tvoří jednu z podmínek:

```
if ((znak >= 'a') && (znak <= 'z'))
```

Vnější závorky jsou nezbytné, neboť ohraničují podmínku příkazu **if**. Vnitřní závorky naopak uvést nemusíme. Priorita operátoru **&&** je nižší než relačních operátorů **<=** a **>=.** Závorky jsme přesto uvedli, neboť zvyšují čitelnost. Výrazy se budeme zabývat v další kapitole. Celý výpis zdrojového textu následuje.

Příklad: Zobrazí alfanumerický znak.

```
/**********************/
/*   Výpis znaku      */
/**********************/

#include <stdio.h>

int main(void)
{
  printf("Zadejte alfanumericky znak:\n");
  int znak = getchar();

  printf("\nZadali jste  ");
  putchar(znak);

  if ((znak >= 'a') && (znak <= 'z'))
  { // malá písmena
    printf("\nZnak je male pismeno\n");
  }
  else if ((znak >= 'A') && (znak <= 'Z'))
  { // velká písmena
    printf("\nZnak je velke pismeno\n");
  }
  else if ((znak >= '0') && (znak <= '9'))
  { // číslice
    printf("\nZnak je cislice\n");
  }
  else
  { // jiný než alfanumerický znak
    printf("\nNeni zadan alfanumericky znak!\n");
  }

  return 0;
}
```

Takto ne :

```
/*********************************************/
/*   Výpis znaku bez použití else-if-else    */
/*********************************************/
#include <stdio.h>

int main(void)
{
 printf("Zadejte alfanumericky znak:\n");
 int znak = getchar();

 printf("\nZadali jste  ");
 putchar(znak);

 if ((znak >= 'a') && (znak <= 'z'))
   printf("\nZnak je male pismeno\n");
```

```
 if ((znak >= 'A') && (znak <= 'Z'))
   printf("\nZnak je velke pismeno\n");
 if ((znak >= '0') && (znak <= '9'))
   printf("\nZnak je cislice\n");
 if (!((znak >= 'a' && znak <= 'z') ||
        (znak >= 'A' && znak <= 'Z') ||
        (znak >= '0' && znak <= '9')))
   printf("\nNeni zadan alfanumericky znak!\n");
return 0;
}
```

4.3.5 Přepínač

Přepínač slouží k větvení výpočtu podle hodnoty celočíselného výrazu. Syntakticky zapsaný příkaz přepínač má tvar [C99]:

selection-statement:
switch (*expression* **)** *statement*

labeled-statement:
case *constant-expression* **:** *statement*
default : *statement*

Syntaktickou definici přepínače ovšem musíme upřesnit. Po klíčovém slově **switch** následuje celočíselný výraz (*expression*) uzavřený v závorkách. Za ním je příkaz (*statement*) zpravidla tvořený blokem. Blok představuje posloupnost příkazů, v níž mohou být umístěna návěští, jejichž syntaxi vidíme na druhé řádce definice.
Řídicí příkaz tvoří celočíselný konstantní výraz (*constant expression*) uvozený klíčovým slovem case a ukončený dvojtečkou (:). Jedno z návěští může být klíčovým slovem default, přirozeně ukončeným dvojtečkou (:).
Sémantika přepínače je poměrně složitá. Popíšeme si jednotlivá pravidla, jimiž se řídí:

- Program vyhodnotí výraz a jeho hodnotu porovnává s každým z **case** návěští přepínače. Pokud hodnota výrazu odpovídá některému z návěští, předá se řízení programu na toto návěští.
- V jednom přepínači se nesmí používat dvě návěští se stejnou hodnotou.
- Pokud hodnotě výrazu neodpovídá žádné návěští, je řízení předáno návěští **default**. Pokud návěští **default** v přepínači chybí, nevykoná se žádná varianta a řízení se předá bezprostředně za příkaz přepínače.
- Návěští **default** se může v jednom přepínači vyskytovat pouze jednou.
- Návěští **case** může být ukončeno příkazem **break**. Pokud tomu tak není, program bude od aktivního návěští pokračovat přes všechna následující návěští, dokud nenarazí na příkaz **break**, **return** nebo konec přepínače.
- Příkaz **break** může být v rámci přepínače umístěn v libovolně zanořeném bloku. Příkaz **break** se vždy vztahuje k nejbližšímu nadřazenému přepínači nebo cyklu.
- Pro návěští **default** platí stejná pravidla jako pro ostatní návěští. Konvence říká, že by se mělo zapisovat na konec přepínače.

Příklad: Doporučená syntaxe a použití.

```
switch(vyraz)
{
  case 1:
    printf("volba 1");
    break;
  case 2: // volby lze spojovat takto
  case 3:
    printf("volba 2 nebo 3");
    break;
//case 2: - nelze mít dvě stejná návěští
  default:
    printf("vsechny ostatni volby");
    break;
}
```

Příklad: Vnořený přepínač.

```
switch(a)
{
  case 1:
    switch(b)
    {
      case 0:
        printf("b je nula");
        break;
      case 1:
        printf("b je jedna");
        break;
    }

  case 2:
  . . .
```

Příklad: V jednom přepínači se nesmí používat dvě návěští se stejnou hodnotou.

```
switch (getchar())
{
  case 'a'  :  putchar('1');
  case 'b'  :  putchar('2');
  case 'a'  :  putchar('3');  // nelze
  case 'd'  :  putchar('4');
}
```

Příklady: Nastane-li shoda hodnoty case návěští s hodnotou **switch** výrazu *expression*, je přeneseno řízení programu na toto návěští. Jinak je řízení přeneseno na návěští **default** v rámci příslušného přepínače.

Příklad1:

```
switch (getchar())
{
  case 'a'  :  putchar('1');
  case 'b'  :  putchar('2');
  case 'c'  :  putchar('3');
  case 'd'  :  putchar('4');
  default   :  putchar('5');
}
```

Výsledky:

'a' -> 12345, 'b' -> 2345, 'c' -> 345, 'd' -> 45, jakýkoliv jiný znak -> 5

Příklad2:

```
switch (getchar())
{
  case 'a'  :
  case 'b'  :
  case 'c'  :  putchar('1');
  case 'd'  :  putchar('2');
  default   :  putchar('3');
}
```

Výsledky:
'a' -> 123, 'b' -> 123, 'c' -> 123, 'd' -> 23, jakýkoliv jiný znak -> 3

Pro návěští **default** platí stejné zásady jako pro jiná návěští.

Příklad: Pokud nenapíšeme **default** návěští a hodnota výrazu **switch** se nebude shodovat s žádným z návěští, bude řízení přeneseno na příkaz následující za přepínačem.

```
switch (getchar())
{
  case 'a'  :  putchar('1');
  case 'b'  :  putchar('2');
  case 'c'  :  putchar('3');
  case 'd'  :  putchar('4');
}
```

Výsledky:
'a' -> 1234, 'b' -> 234, 'c' -> 34, 'd' -> 4, jakýkoliv jiný znak -> nezobrazí se nic

Příklady: Pokud program při provádění přepínače vykoná příkaz **break**, bude řízení přeneseno na příkaz následující za přepínačem.

Příklad1:

```
switch (getchar())
{
  case 'a'  :  putchar('1'); break;
  case 'b'  :  putchar('2'); break;
  case 'c'  :  putchar('3'); break;
  case 'd'  :  putchar('4'); break;
  default   :  putchar('5');
}
```

Výsledky:
'a' -> 1, 'b' -> 2, 'c' -> 3, 'd' -> 4, jakýkoliv jiný znak -> 5

Příklad2:

```
switch (getchar())
{
  case 'a'  :  putchar('1'); break;
  case 'b'  :  putchar('2'); break;
  case 'c'  :  putchar('3'); break;
  case 'd'  :  putchar('4'); break;

}
```

Výsledky:
'a' -> 1, 'b' -> 2, 'c' -> 3, 'd' -> 4, jakýkoliv jiný znak -> nezobrazí se nic

Návěští **case** může být obsaženo uvnitř jiných příkazů (v rámci přepínače). Tuto možnost ale není příliš rozumné používat, protože jde o stejně špatnou konstrukci jako v případě použití **goto** (viz kap. 4.3.7.4). Takový kód je velmi špatně čitelný a jeho chování může být těžko předvídatelné.

Příklad: Jak nepoužívat **case** uvnitř jiného **case**.

```
switch (znak)
{
  case 'A':
    {
      int tmp = 12;
      case 'B': // !!!
        printf("Kolik je tmp? %d\n", tmp);
    }
}
```

Srovnáme-li přepínač s konstrukcí **if-else-if**, vidíme zásadní rozdíly:

1. (Rozhodovací) výraz v příkazu **if** může testovat hodnoty různých typů, zatímco rozhodovací výraz v příkazu **switch** musí být výhradně celočíselným výrazem.
2. V konstrukci **if-else-if** se provede nejvýše (či podle použití právě) jeden z příkazů. I u přepínače se nemusí provést žádný z příkazů. Může jich být ovšem provedeno více. Konstantní návěští určuje pouze první z nich. Pokud chceme, oddělíme následující příkazy příkazem **break**.
3. V přepínači se návěští **default** může vyskytovat kdekoliv. Odpovídající varianta v konstrukci **if-else-if** může být umístěna pouze na konci.

4.3.6 Cykly

Příkazy pro vytváření cyklů používáme tehdy, když potřebujeme nějakou činnost (akci) provádět opakovaně. Cyklus je část programu, která je v závislosti na podmínce prováděna opakovaně. U cyklu obvykle rozlišujeme **řídicí podmínku** cyklu a **tělo** cyklu. Řídicí podmínka cyklu určuje, bude-li provedeno tělo cyklu, či bude-li řízení předáno za příkaz cyklu. Tělo cyklu je příkaz, zpravidla v podobě bloku.

Cykly můžeme rozdělit podle toho, provede-li se tělo alespoň jedenkrát, a cykly, kde tělo nemusí být provedeno vůbec. Obecně můžeme říci, že i když v jazyce C existují různé typy cyklů, je možné vystačit s jedním z nich. Jejich výběr závisí na vhodnosti použití v dané situaci. Pouze amatéři však používají jen jeden typ cyklu v každé situaci. V části věnované příkazu **while** si popíšeme některé vlastnosti společné všem cyklům. Postupně tedy **while**, **for** a **do-while**.

Syntaxe cyklu [C99]
iteration-statement:

while (*expression* **)** *statement*
do *statement* **while (** *expression* **);**
for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*
for (*declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

4.3.6.1 Cyklus while

Syntaxe:

while (*expression* **)** *statement*

Cyklus **while** bývá také nazýván cyklus s podmínkou na začátku. Používá se v případech, kdy ***dopředu neznáme počet iterací*** (nelze tedy použít cyklus **for**) a kdy se také může stát, že cyklus ***nebude smět proběhnout ani jednou*.**

Příkaz **while** vykonává příkaz (*statement* – tělo cyklu) vícekrát, nebo vůbec, dokud má v daném kontextu testovací výraz (*expression*) nenulovou hodnotu.
Příkaz (*statement*) je velmi často blokem. Zde se zejména začátečníci mylně domnívají, že pokud se kontext kdykoliv během provádění příkazů bloku těla cyklu změní, a podmínka tak není splněna, je cyklus opuštěn. To ovšem není pravda. Pravidlo říká, že se příkaz provede, je-li podmínka *expression* splněna. Je-li příkazem blok, provede se tedy blok celý (pokud není uvnitř bloku příkaz skoku (**break**, **return**, **continue**)). Teprve poté je znovu proveden test (také bezprostředně po použití **continue**).

Příklad: Počítá počet zadaných číselných a nečíselných znaků.

```
// Příkaz cyklu while
int cisla = 0;
int necisla = 0;
int znak;

while ((znak = getchar()) != EOF)
{
  if (znak >= '0' && znak <= '9')
    cisla++;
  else
    necisla++;
}
printf("Zadal jste %d cisel a %d "
       "neciselnych znaku.\n", cisla, necisla);
```

4.3.6.2 Cyklus do-while

Syntaxe:

do *statement* **while (** *expression* **);**

Cyklus **do-while** opakuje příkaz (*statement*) dokud je výraz (*expression*) pravdivý. Ukončí se, když se výraz stane nepravdivým. Cyklus **do-while** je jedinečný v tom, že se jeho příkazová část provede vždy alespoň jednou, jelikož řídicí výraz je testován na konci cyklu.

Skutečnost, že **do-while** provádí příkazovou část cyklu vždy alespoň jednou, se výborně hodí například pro zjišťování volby z menu. Například tato verze kalkulačky se při zadání nesprávného vstupu zeptá uživatele znovu.

Příklad: Program čeká na zadání správné operace a dvou čísel, pak operaci s čísly provede.
Poznámka: V rámci zjednodušení program nemá všude ošetřen chybný vstup od uživatele.

```
/****************/
/*  kalkulačka  */
/****************/
#include <stdio.h>
int main(void)
{
```

```
  int a, b;
  int ch;

  printf("Chcete:\n"
         "Scitat, Odecitat, Nasobit nebo Delit?\n");

  /* přinutí uživatele zadat správnou odpověď */
  do {
    printf("\nZadejte prvni pismeno operace: \n");
    ch = getchar();
  } while(ch!='S' && ch!='O' && ch!='N' && ch!='D');
  printf("\n");

  printf("Zadejte prvni cislo: ");
  scanf("%d", &a); //!
  printf("Zadejte druhe cislo: ");
  scanf("%d", &b); //!

  if(ch=='S') printf("%d\n", a+b);
  else if(ch=='O') printf("%d\n", a-b);
       else if(ch=='N') printf("%d\n", a*b);
            else if(ch=='D' && b!=0) printf("%d\n", a/b);

  return 0;
}
```

Cyklus **do-while** je zvláště užitečný, když program čeká až se stane nějaká událost.

Poznámka k syntaxi: Protože se druhá část příkazu **do-while** velmi podobá cyklu **while** s prázdným tělem, je velmi žádoucí, aby bylo klíčové slovo **while** psáno vždy za uzavírací složenou závorkou na jednom řádku:

Příklad:
```
do {
  // příkazy - tělo cyklu
} while(podmínka); // zde je středník nezbytný!
```

4.3.6.3 Cyklus for

Cyklus for se používá pro zadaný počet opakování příkazu nebo bloku příkazů.
Syntaxe [C99]:

for (*expression*$_{opt}$ **;** *expression*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*
for (*declaration expression*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*

for (*inicializace* **;** *test podmínky* **;** *inkrementace* **)** *příkaz*

Část ***inicializace*** se používá pro zadání počáteční hodnoty proměnné, která řídí průběh cyklu. Tato hodnota se obvykle označuje jako řídicí proměnná cyklu. Podle nové normy (ISO C99) lze v této části provést rovnou i definici této proměnné, která má potom rozsah platnosti omezen pouze na tělo cyklu, což je velmi užitečné. Část inicializace se provádí pouze jednou před začátkem cyklu.

Část ***test podmínky*** testuje řídicí proměnnou cyklu na koncovou hodnotu. Je-li test podmínky vyhodnocen jako pravdivý, cyklus se opakuje. Je-li nepravdivý, cyklus se ukončí a

zpracování programu pokračuje příkazem, který následuje za cyklem. Test podmínky se provádí na začátku cyklu neboli před každým opakováním těla cyklu.

Část ***inkrementace*** cyklu **for** se provádí vždy po každém provedení těla cyklu. To znamená, že část inkrementace se provádí poté, co se provede příkaz nebo blok, který tvoří tělo cyklu. Účelem části inkrementace je zvyšovat nebo snižovat řídicí proměnnou cyklu o určitou hodnotu (ale lze ji použít i pro jiné operace).

Příklad: Zobrazí čísla od 1 do 10. Na nový řádek vypíše řetězec "konec".

```
// Příkaz cyklu for
for(int num = 1; num < 11; num++)
  printf("%d ", num);

printf("\nkonec");
```

Pro opakování několika příkazů je možné použít jako tělo cyklu blok příkazů.

Příklad: Součin a součet čísel od 1 do 6.

```
/***********************/
/* Příkaz cyklu for    */
/***********************/
#include <stdio.h>
#define POCET 6
int main(void)
{
  int sum = 0;
  int soucin = 1;

  for(int num=1; num<=POCET; num++)
  {
    sum += num;
    soucin *= num;
  }
  printf("Soucin a soucet cisel od 1 do %d\n"
         "Soucin: %d soucet: %d \n", POCET, soucin, sum);
  return 0;
}
```

Cyklus **for** může probíhat i naopak. Například tento úsek programu snižuje hodnotu řídicí proměnné cyklu:

```
for(int num=20; num>0; num--)
```

Hodnota řídicí proměnné cyklu může být zvyšována nebo snižována i o jinou hodnotu než 1.

Příklad: Zobrazí čísla od nuly do sta po pěti.

```
for(int i=0; i<101; i+=5)
  printf("%d ", i);
```

Kteroukoli z částí (inicializace, test, inkrementace) lze vynechat. Většinou však jde o konstrukce, které zatemňují podstatu algoritmu, takže je lepší používat cyklus **for** pouze výše prezentovaným způsobem – pro cykly s předem známým počtem iterací. Neúplný cyklus **for** lze většinou přehledněji přepsat pomocí cyklu **while** nebo **do-while**.

Příklad: Zobrazí čísla od nuly do devíti.

```
for(int i=0; i<10; )
{
  printf("%d ", i);
  i++;
}

int i = 0;
while(i < 10)
{
  printf("%d ", i);
  i++;
}
```

Lze také používat více řídicích proměnných cyklu:

Příklad: Vypíše do sloupců dvě posloupnosti, jednu rostoucí a jednu klesající.

```
int i
for(int i = 0, j = 10; i <= j; i++, j--)
{
  printf("%d %d\n", i, j);
}
```

4.3.6.4 Vnořování cyklů

Když tělo cyklu obsahuje jiný cyklus, říká se, že je druhý cyklus vnořen do prvního. Jakýkoliv cyklus může být vnořen do jiného.

Příklad: Zobrazí desetkrát čísla od jedné do deseti.

```
// vnější cyklus
for(int i=0; i<10; i++)
{
  for(int j=1; j<11; j++)
    printf("%d ", j); // vnořený cyklus

  printf("\n");
}
```

4.3.7 Příkazy skoku

Syntaxe:

jump-statement:

goto *identifier* **;**
continue ;
break ;
return $expression_{opt}$ **;**

4.3.7.1 Ukončení cyklu příkazem break

Příkaz **break** umožňuje ukončit cyklus v libovolném místě těla cyklu a obejít tak normální ukončovací výraz. Pokud se uvnitř cyklu narazí na **break**, je cyklus okamžitě ukončen a zpracování programu pokračuje příkazem následujícím za cyklem.

Příklad: Zobrazí jen čísla od 1 do 10.

```
// break v cyklu

for(int i=1; i<100; i++)

{
  printf("%d ", i);

  if(i==10)

    break; /* ukončení cyklu */
}
// bude se pokračovat tady
```

Při vnořování cyklů ukončí **break** jen nejbližší cyklus, ve kterém je použit.

Příklad: Zobrazí čísla nula až pět pětkrát.

```
// break v cyklu

for(int i=0; i<5; i++)
{
  for(int j=0; j<100; j++)
  {
    printf("%d", j);

    if(j==5) break;
  }
  printf("\n");
}
```

Příkaz **break** lze použít v jakémkoli blokovém příkazu. Vždy přeruší vykonávání posloupnosti příkazů a předá řízení na první příkaz za blokem.

4.3.7.2 Příkaz continue

V cyklech **while**, **do-while** způsobí příkaz **continue** přechod řízení na test podmínky a pak (ne)pokračování zpracování cyklu.

V případě cyklu **for** se provede inkrementační část cyklu, pak test podmínky cyklu a cyklus (ne)pokračuje.

Jeden z vhodných případů použití **continue** je nové spuštění posloupnosti příkazů, když nastane chyba.

Příklad: Program, který nikdy nic nevypíše.

```
// continue v cyklu for

for(int x=0; x<100; x++)

{
  continue;

  printf("%d ", x); // nikdy se neprovede
}
```

Příklad: program počítá průběžný součet všech čísel zadaných uživatelem. Před přičtením hodnoty k průběžnému součtu otestuje správnost zadaného čísla tak, že ji uživatel musí zadat znovu. Pokud se tato dvě čísla liší, program použije continue pro restartování cyklu.

```
/*******************/
/* continue v cyklu */
/*******************/

#include <stdio.h>

int main(void)
{
  int i, j;
  int total=0;
  do {
    printf("Zadejte dalsi cislo (0 = konec): ");
    scanf("%d", &i);
    printf("Zadejte cislo znovu: ");
    scanf("%d", &j);
    if(i != j)
    {
      printf("Cisla nesouhlasi\n");
      continue;
    }
    total += i;
  } while(i != 0);

  printf("Soucet je: %d", total);
  return 0;
}
```

4.3.7.3 Příkaz return

Dojde-li provádění programu na příkaz **return**, ukončí se provádění funkce, která tento příkaz obsahuje. Ve funkci **main** ukončí příkaz **return** celý program. Často se pomocí **return** vrací nějaká hodnota, jejíž typ záleží na typu funkce.
Syntaxe [C99]:

`return` *expression*$_{opt}$**`;`**

Příklad: Pokud funkce proběhne v pořádku, vrací hodnotu jedna. V případě, že dojde k chybě, vrací hodnotu nula.

```
#define N 10
int x[N], a[N], b[N]; // definice globálních polí

int vypocet(int a[], int b[], int x[])
{
  for (int i = 0;  i < N;  i++) {
    for (int j = 0;  j < N;  j++) {
      for (int k = 0;  k < N;  k++) {
        if (x[k] == 0)
          return 0;      /* neuspech */
        a[i] += b[j] / x[k];
      }
    }
  }
  return 1;              /* uspech */
}
```

4.3.7.4 Příkaz skoku

Jazyk C podporuje příkaz nepodmíněného skoku nazývaný **goto**. Většina programátorů však příkaz **goto** nepoužívá, protože narušuje strukturu programu a pokud je používán často, může se stát, že bude později velmi obtížné programu porozumět. Neexistuje ani žádný obvyklý postup, který by **goto** potřeboval. Z těchto důvodů nebude ani dále používán mimo tuto část.

Syntaxe [C99]:

goto *identifier*;

labeled-statement:

identifier : *statement*

Příkaz **goto** může provádět skok v rámci funkce. Nemůže přeskakovat mezi dvěma funkcemi. Používá se spolu s návěštím. V jazyce C je návěští platné jméno identifikátoru následované dvojtečkou.

Příklad: Následující **goto** přeskočí příkaz printf().

```
goto navesti;

printf("Toto se nikdy nevytiskne.");

navesti: printf("Toto se vytiskne.");
```

Jediným vhodným použitím **goto** je vyskočení z hluboko zanořené části programu, pokud nastane fatální chyba.

4.4 Shrnutí

Nyní je čtenář již schopen vytvářet podprogramy v jazyce C. Dokáže funkci deklarovat pomocí prototypu a také ji definovat a chápe, jaký je v tom rozdíl. Dále čtenář umí vytvořené funkce volat a používat jejich návratovou hodnotu. Rovněž umí deklarovat parametry funkcí jednoduchých typů a předávat skutečné argumenty při volání.

Při používání funkcí čtenář již chápe rozdíl mezi parametrem funkce, globální a lokální proměnnou a dokáže správně určit, k čemu se používají. Při spojení znalostí o funkcích a o blokových (složených) příkazech také dokáže správně vyhodnocovat rozsah platnosti identifikátorů a využívat toho ve svých programech.

Čtenář již nyní zná řídící struktury pro podmíněná rozhodování i pro tvorbu cyklů. Zná jejich syntaxi v jazyce C a umí je v tomto jazyce použít. U jednotlivých druhů podmíněných příkazů (if-else, switch) dokáže rozlišit situace, kde se jejich použití hodí více a kdy méně. Totéž dokáže vyhodnotit pro jednotlivé druhy příkazů cyklu (while, do-while, for).
Čtenář je v tuto chvíli schopen tvořit jednodušší programy s podprogramy, které využívají jednoduché datové typy.

4.5 Úlohy k procvičení

Zpracování posloupností hodnot.
Vytvořte program v jazyce C pro:

1. součet n hodnot.
2. počet záporných, nulových a kladných hodnot z n hodnot.
3. aritmetický průměr kladných hodnot z neznámého počtu hodnot. Seznam hodnot je ukončený nulou.
4. zjištění největší hodnoty z n hodnot.
5. zjištění nejmenší a největší hodnoty z n hodnot.
6. zjištění 2 největších hodnot z n hodnot.
7. zjištění největší hodnoty a počet jejích výskytů z n hodnot.
8. zjištění nejmenší hodnoty a pořadí jejího prvního výskytu z n hodnot.
9. Jsou dány známky 1-5, koncová hodnota 0. Určete počet výskytů jedniček.
10. Jsou dány kladné hodnoty vyjadřující teplotu, zakončené nulou. Určete nejmenší teplotu.
11. Ze vstupu zadejte hodnotu *limit*. Proveďte součet čísel aritmetické posloupnosti 1+2+ ……n, tak, aby součet nepřesáhl hodnotu *limit*. Vypište maximální hodnotu n, pro kterou součet čísel 1 až n nepřesáhne hodnotu *limit*.
12. Zobrazte tabulku funkcí sin(x), cos(x), tg(x), kde x se mění v intervalu <0,90> stupňů.
13. Ze vstupu zadejte koeficienty a, b, c. Zobrazte tabulku hodnot polynomu ax^2+bx+c, kde x se mění v intervalu <1,10> po jedné. Pro výpočet polynomu definujte funkci.

4.6 Kontrolní otázky

1. Jaký je obecný formát programu v jazyce C?
2. Co je deklarace funkce, kde se uvádí?
3. Co to je prototyp funkce?
4. Co je definice funkce?
5. Jak se definuje funkce, která nevrací hodnotu?
6. K čemu slouží parametry funkce, kde se deklarují?
7. Kde a kdy se používají argumenty funkce?
8. Uveďte rozdíl mezi globální a lokální proměnnou.
9. Co se rozumí pojmem zastínění proměnné?
10. Jaká je struktura složeného příkazu?
11. Jaké jsou druhy podmíněných příkazů, uveďte jejich rozdíl.
12. Uveďte druhy cyklů a v čem se jednotlivé cykly liší?
13. Uveďte jednotlivé příkazy skoku, a kde se používají.