Stručný přehled základní syntaxe jazyka C

Text předkládá základní pojmy jazyka C, předpokládané pro čtení skripta
Šusta, R. -Programování pro řízení ve Windows, ČVUT-FEL Praha 1999

Obsah

1. Číselné konstanty:

```
123, 123.45, 1.7e+308, .5   // zápisy celých a reálných čísel
12345678L                   // konstanta typu long - příponu lze psát jako malé l i velké L
65000u                      // konstanta typu unsigned int, přípona je malé u nebo velké U
65000ul                     // konstanta typu unsigned long
1.1e+4932L                  // reálné číslo typu long double (přípona je malé l i velké L)
123, 123.45, 1.7e+308, .5   // běžné zápisy čísel
0x123, 0x8000u              // čísla typu  int a unsigned
0x7FFFFFl, 0xFFFFFFFFul     // long a long unsigned
0123                        // Pozor, oktalové konstanty začínají číslicí 0.
```

2. Znakové konstanty

```
'A', 'AB'                   // znak A = 0x41 a dvojznak AB = 0x4241
'\n' '\r'                   // znaky LF = 10 a CR  = 13 (klávesa Enter)
'\t'                        // tabelátor = 9
'\\' , '\"' , '\''          // znaky \ " '
'\41' /*nebo*/  '\041'      // znak s oktalovým ekvivalentem 041 = '!'
'\x41' /*nebo*/  '\x041'    // znak s hex. ekv. 0x41 = 'A' (uvažují se vždy 3 číslice)
"Alfa\tBeta\n"              // dvě slova Alfa a Beta oddělená tabulátorem a ukončené CR
"Alfa\t" "Beta\n"           // řetězec hodnotou totožný s řetězcem z předchozí řádky
„X:\\VYUKA\\RTIME\\PRJ“      // adresář - Pozor, všimněte si dvojitých lomítek!
```

3. Příkazy preprocesoru

```
#include <ctype.h>                  /* preprocesor nahradí řádek obsahem souboru ctype.h, který
                                       bude hledat v adresářích seznamu INCLUDE překladače */
#include "type.h"                   /* řádka se nahradí obsahem souboru type.h, hledaným napřed
                                       v okamžitém adresáři a poté v adresářích INCLUDE */
#include "def\data.h"               /* Pozor, jména souborů v #include mají jednoduchá lomítka,
                                       protože nejde o řetězce jazyka C, ale jeho preprocesoru!
                                       Do nich nelze vkládat speciální znaky, jako např. '\n' */

#define PI 3.1415926                // preprocesor bude nahrazovat slovo PI znaky 3.141592
#define moc(x) (x*x)                /* moc(var) se nahradí (var*var), tj. zápis moc(2)/a bude (2*2)/a,
                                       všimněte si, že za deklarací moc není ; Proč? */
#define cti(x,y) fce(x, data##y)    // cti(1,2) se nahradí fce(1, data2)
#define vloz(x) cti(#x)             // příkaz vloz(jmeno) se nahradí cti("jmeno")
#undef PI                           // zruš definici symbolu PI pro následující řádky programu

#ifdef _DATA_H_                     /* překládej, je-li definován symbol _DATA_H_ příkazem
                                       #define. (Pozor, _DATA_H_ není proměnná programu, ale
                                       symbol preprocesoru). Není-li symbol definován, pak se
                                       vynechají následující řádky programu až k #else či #endif */
#ifndef _DATA_H_                    // překládej, není-li definován symbol _DATA_H_
#if TEST==10                        // překládej, rovná-li se symbol TEST číslu 10
#else                               // sekce else podmíněného překladu (může být vynechána)
#endif                              // konec podmíněného překladu zahájeného #ifdef, #ifndef, #if
```

- ***Poznámky k preprocesoru***

  Seznam není úplný. Chybějí v něm direktivy #line, #error a #pragma (pro řízení překladu).

⇒ Preprocesor nahrazuje řetězce podle předefinovaného schématu (provádí například odstranění komentářů z programu). Jeho činnost dále ovlivňují příkazy tvořené řádkami s #-direktivami. **Překladač zpracovává až výsledek činnosti preprocesoru**. Chceme-li se podívat na výstup preprocesoru, můžeme ho spustit jako samostatný program z příkazové řádky nebo z menu nástrojů překladače (*Tools*).

⇒ **Symboly preprocesoru (definované #define) tvoří samostatnou množinu**, která nemá nic společného s proměnnými definovanými v programu a naopak.

4. Deklarace proměnných

```
char c, bznak = 'b';                /* 8-mi bitová proměnná c a proměnná bznak inicializovaná
                                       na hodnotu znaku 'b' */
unsigned char cu = 0xFF;            //  8-mi bitová proměnná cu (0 až 255) inicializovaná na 255
int index;                          /* proměnná uložená v paměti jako 16-bitů v 16-ti bitových
                                       programech a na 32-bitů ve 32-bitových programech. */
unsigned int data = 0x8000u;        // celé číslo bez znaménka o délce jako int
short int si, sj;                   // vždy 16-ti bitové celé číslo
long len;                           // vždy 32-ti bitové celé číslo len
unsigned long dword;                // jako long, ale tentokrát bez znaménka
```

```
float real;                          // reálná čísla v jednoduché přesnosti
double dreal;                        /* dvojnásobná přesnost  (v jazyce C preferovaný typ
                                        reálných čísel */
long double dk = 1.1e+4932L;         // číslo ve formátu aritmetického kooprocesoru

enum DEN { po = 1, ut, st, ct, pa, so, ne } dnes;  /* definice proměnné den typu DEN a
                                        nabývající hodnot po až ne odpovídajícím číslům 1, 2, až 7 */

const int Delka = 100;               // modifikátor const pro definici konstanty int

typedef unsigned char byte;          // deklarace nového typu se jménem byte
typedef unsigned short int word;     /* deklarace nového typu word.
                                        Řekněte, proč je v definici uvedeno short int a proč nestačí
                                        napsat jen int ? */

char * pznak;                        // pointer typu char, tj. ukazatel na proměnnou typu char

void * pbeztypu;                     /* pointer, který nemá určený typ proměnné, na níž ukazuje.
                                        Používá se hlavně jako parametr funkcí (třeba memset() ),
                                        protože každý jiný pointer se na něho převede bez nutného
                                        přetypování - viz příklad na soubory na konci této kapitoly.
                                        */
char poleznaku[16];                  // pole 16-ti prvků typu char

typedef char POLE16[16];             // nový typ POLE16 - 16 prvků typu char
POLE16 jmeno;                        // pole jmeno 16-ti prvků typu char
```

- ***Poznámky k proměnným***

⇒ Váš překladač nemusí znát analogii typu *boolean*, který existuje v PASCALu, protože ho neobsahuje norma jazyka. Někteří výrobci ho zavádějí jako rozšíření pod názvem **BOOL**, resp. bool. Konstanty TRUE a FALSE, resp. true a false, se obecně definují:

```
#define TRUE 1
#define FALSE 0
```

a samotný typ *boolean* jako:

```
typedef int  BOOL;
```

⇒ Příkaz enum sice nabízí elegantnější definici BOOL:

```
enum BOOL { FALSE=0, TRUE=1 };
```

ale řada systémových funkcí, třeba if(), while(), považují nenulové číslo za splnění podmínky a 0 za nesplnění, a proto se stejný přístup vyžaduje i od jiných funkcí. Definice TRUE jako hodnoty typu enum by to nedovolovala bez přetypování.

⇒ Ve výrazech jazyka C se typy nekontrolují tak přísně jako v PASCALu. Například běžně se s typem char zachází jako s menším bratrem typu int, což sice popuzuje zastánce strohé kontroly typů proměnných, ale na druhou stranu, podíváme-li se na to z hlediska práce procesoru, tak není mezi oběma typy skutečně jiný rozdíl než délka.

5. Struktury

Operace se strukturami se podrobně rozebírají v kapitole 3 skripta při výkladu objektů. Zde jen obecné znalosti.

Struktury dovolují uzavřít několik proměnných pod společný název a zacházet s nimi jako s celistvým objektem. Například struktura se třemi subelementy:

```
struct SDATA
 {
        char clen1;
        int clen2;
        long clen3;
 } ;
```

Všimněte si nutného středníku za poslední závorkou - jeho vynechání naprosto dokonale zmate překladač.

Vlastní definici proměnných, které budou typu námi deklarované struktury, provedeme:

```
struct SDATA sdataA, sdataB;
```

Klíčové slovo struct před názvem SDATA se může vynechat, nehrozí-li záměna s jinou stejnojmennou proměnnou. Překladač totiž hledá jména struktur v samostatném seznamu proměnných (*angl. name space*). Teoreticky může existovat i jiná proměnná téhož jména např. **int SDATA;** (*SDATA nemusí být nadefinována námi, ale třeba některou systémovou knihovnou*).

Deklaraci struktury a datových prvků lze provést i jediným příkazem:

```
struct SDATA
 {
        char clen1;
        int clen2;
        long clen3;
 }
        sdataA, sdataB;
```

Dostup na členy struktury se provádí operátorem tečky. Například: sdataA.clen1 = 0;

Na struktury se často odkazuje pointry:

```
struct SDATA * psdata;                  // neinicializovaný pointer.
struct SDATA * psdataB = &sdataB;       // inicializovaný pointer - ukazuje na sdataB
```

Dostup na členy struktury sdataB vypadá například takto:

```
(*psdataB).clen1 = 0;
```

Závorky jsou nezbytné díky vyšší prioritě operace tečky oproti operaci dereference pointru, a proto pro dostup na členy struktury přes pointer existuje zkratka:

```
psdataB -> clen1 = 0;
```

Dokážete říct, co se stane, pokud napíšeme: (*psdata).clen2 = 0;

6. Operace

Následují tabulky shrnuje nejzákladnější operace jazyka C. Seznam je uspořádaný podle priority operací od nejvyšší priority (1) k nejnižší (16).

Priorita	Znak	Komentář
1.	()	*// závorky*
	[]	*// prvek pole*
	.	*// prvek struktury dat*
	->	*// zkratka pro zápis:* pstruktura->prvek *je totožné s* (*pstruktura).prvek
2.	(typ)	*// přetypování*
	!	*// unární operace logické negace - výsledkem je 1 nebo 0*
	~	*// unární operace bitová negace*
	-	*// unární aritmetická negace*
	++	*// unární operace přičtení 1*
	--	*// unární operace odečtení 1*
	&	*// unární operace zjištění adresy prvku (pointru)*
	*	*// unární operace určení prvku z jeho adresy (dereference pointru)*
	sizeof	*/* unární operace pro zjištění velikosti proměnné v bytech. Lze psát jako operátor či jako funkci. Obě následující použití operandu jsou totožná:* long X,Y; Y=sizeof X; Y=sizeof(X); */*
3.		*// operace této priority se týkají C++ a jsou probrány v kapitole 3 skripta*
4.	*	*// násobení*
	%	*// zbytek po celočíselném dělení,*
	/	*/* dělení. Na rozdíl od PASCALu není speciální operátor pro celočíselné dělení a použitý typ dělení určují operandy výrazu. Jsou-li oba celá čísla, provede se celočíselné dělení, jinak bude reálné. */*
5.	+	*// sčítání*
	-	*// odčítání*
6.	>>	*// x>>n bitový posun x doprava o n bitů, levá bity x se plní 0*
	<<	*// x<<n bitový posun x doleva o n bitů, pravé bity x se plní 0*
7.	<	*// operace logického porovnání, jejichž výsledkem je 1 nebo 0*
	<=	
	>	
	>=	
8.	==	*// operace rovnosti – výsledkem je 1 (rovná se) nebo 0 (nerovná se).*
	!=	*// operace nerovnosti - výsledkem je 1 (nerovná se) nebo 0 (rovná se).*
9.	&	*// bitové AND.*
10.	^	*// bitové XOR*
11.	|	*// bitové OR*
12.	&&	*// logické AND – výsledkem je 1 nebo 0*

13.	**||**	*// logické OR - výsledkem je 1 nebo 0*
14.	**?:**	*/* podmíněný výraz, zápis:* max = i > j ? i : j; *má význam:* if(i>j) max = i; else max = j; **/*
15.	**=**	*/* přiřazení. Na rozdíl od PASCALu ho lze řetězit, např.* x = y = z = 0; *uloží* 0 *do tří proměnných;* y = (x=7)*6; *uloží do* x = 7 *a do* y = 42 *(tj.* x*6*). */*
		// zkratky pro změnu hodnoty proměnné
	+ =	*//* x += 5; *odpovídá:* x = x + 5;
	- =	*//* x -= 5*a; *odpovídá:* x = x - 5*a;
	*** =**	*// analogicky* x *= 5*y - 1; *má význam:* x = x * (5*y-1);
	/ =	*// podobně* x /= 5+c; *odpovídá:* x = x / (5+c)*;*
	% =	*//* x %= 5+c/2*; odpovídá:* x = x % (5+c/2);
	& =	*//* x &= 7; *odpovídá:* x = x & 7;
	| =	*//* x |= 0xF; *odpovídá:* x = x | 0xF;
	>>=	*//* x >>= 5; *odpovídá:* x = x >> 5; *což se někdy rovná* x /= 32; *Řekněte kdy.*
	<<=	*/** x <<= 5; *odpovídá:* x = x << 5; *což může někdy být i* x *= 32, *ale někdy ne. Řekněte kdy. */*
16.	**,**	*/* oddělení výrazů. Levý výraz má jen vedlejší efekt a hodnota* operace *se rovná pravému výrazu. Operátor se používá především u příkazu* for*, kde dovoluje napsat více výrazů pro jeden parametr, např.* for(i=0, j=1, s=0; i<=7; i++, j<<=1) s+= (znak & j) != 0; **/*

- ***Poznámky k operacím:***

⇒ Výčet není úplný. Chybí v něm šest operací, (*jmenovitě:* new delete typeid :: .* ->.), které jsou svázané s C++ objekty a budou se probírat v dalších kapitolách, a dále nejsou uvedené speciální operátory typů, které závisí na použitém překladači a lze je nalézt v jeho dokumentaci

⇒ Převody mezi jednotlivými typy se provádějí automaticky na typ operandu, který má největší rozsah přípustných hodnot. Například - je-li jeden z operandů real a druhý long, oba se převedou na typy real a tohoto typu bude i výsledek operace.

⇒ Překladač vypisuje varování, hrozí-li při převodu proměnné ztráta přesnosti, ale při běhu programu se přetečení povoleného rozsahu proměnné nehlídá. Výjimkou případů testů vestavěných přímo do aritmetické jednotky procesoru, jako třeba dělení 0. Potřebujeme-li kontrolovat meze, musíme si potřebné testy naprogramovat sami.

⇒ V případě potřeby nebo ve sporných případech lze převody čísel řídit přetypováním, tj. požadovaným typem uvedeným v závorkách před proměnnou, například:

```
short int i = 200, j=1234, k;
k = i * j / 100;                  // CHYBA! Stanovte proč!
k = ((long) i * j )/100;          // Výpočet sice správný, ale hlásí se varování. Proč?
k = (int) ((long) i * j)/100);    // Správný výsledek bez zbytečného varování.
```

⇒ Jazyk C rozlišuje logické a bitové AND a OR (*PASCAL nikoliv*). Všimněte si poněkud nevhodně zvolené priority pro tyto operace. Zápis: A & 1 == 0 se provede jako A & (1==0) a jeho výsledkem bude 0. Testy na bity je nezbytné psát jako (A&1)==0. Překladač upozorňuje na vynechání závorky varováním „*Ambiguous operators need parentheses.*“ = víceznačné operace vyžadují používání závorek.

⇒ Operace jazyka C++ mají kromě priority i ***asociativitu***. Operace priority ***2, 14 a 15*** se sdružují ***zprava doleva*** a ostatní operace klasicky ***zleva doprava***. Asociativita se uplatní zejména při kumulaci operací stejné priority, například program:

```
char znak;  int data = 0x1020;
znak = ++ * (char*) & data;
```

dává výsledky: znak = 0x21; (dolní byte + 1) a data = 0x1021; protože operace se provedou v pořadí zprava doleva, čili jako:

```
znak = ++ (  * (  (char *) ( & data ) );
```

a stejně tak skupina operací: * pznak ++; se vykoná jako zápis * (pznak ++);
Naproti tomu operátor [] se sdružuje zleva doprava, a proto příkaz:

```
char pole[10] [80] [5];
```

vytvoří pole o 10-ti prvcích, jehož každý prvek bude mít 80 elementů o délce 5 znaků.

⇒ Pozor, u binárních operací není obecně stanoveno, v jakém pořadí se vyčíslují jejich operandy. Následující program může vypočítat hodnotu x rovnou 10 nebo 20:

```
int x, y = 0;
x = (y=10) + y;
```

Výjimkou z předchozího pravidla jsou logické operace && a || , u nichž je zaručeno, že jejich členy se vyčíslují důsledně zleva doprava. V následujícím ukázce se testy na hodnotu funkce X() vykonají v pořadí testovaných čísel. Výsledek se samozřejmě počítá podle priority operací a závorek.

```
#include <stdio.h>

int X( int cislo ) { printf("%4d", cislo); return cislo;  }

void main(void)
 {      if(  X(1)==1 && X(2)==2 && ( X(3)!=3 ||  X(4)==4 && X(5)!=5 || X(6)!=6 || X(7)!=7 )  )
          { /* příkazy podmínky */ }
  }
```

nám vytiskne výstup: "1 2 3 4 5 6 7". Význam se zaručeného pořadí se projevují v situacích, kdy výpočty operandů podmínky mají vedlejší efekty, například:

```
char znak;
while( (znak = CtiZnak()) != 0  &&  znak != '\n' ) {  /* příkazy cyklu */  }
```

Pořadí vyčíslování operací && a || nám zaručuje, že se napřed zavolá funkce CtiZnak() a přečte znak a teprve poté se znak testuje na hodnotu '\n'.

⇒ Logické podmínky se v jazyce C vyčíslují pouze ke členu, při němž lze už rozhodnout o výsledku celého výrazu. Změníme-li podmínku programu předchozího odstavce na:

```
if(  X(1)==1 && X(2)==2 && ( X(3)==3 ||  X(4)==4 && X(5)!=5 || X(6)!=6 || X(7)!=7 )  ) { }
```

výstup bude: "1 2 3". ***Řeknete proč!***

⇒ Zvláštností jazyka C je operace = , která vrací hodnotu přiřazeného čísla, a pomocí ní lze uložit mezivýsledek do proměnné bez přerušení výpočtu výrazu. Paradoxně, právě tato operace dělá některým začátečníkům skoro stejně velké potíže jako pointry, i když z hlediska činnosti procesoru je velmi přirozená. Umožňuje nejen zjednodušit zápis programu, ale při konstrukci objektů je prakticky nepostradatelná. Nezapomínejte však na její nízkou prioritu, uložení mezivýsledku musí být uzavřeno do závorek.

⇒ Jinou specifikou jazyka C jsou operace ++ a -- pro přičtení, resp. odečtení jedničky. Vyskytují se v prefixové a postfixové variantě, které se liší hodnotou operace. Zápis: y=1; x= y++; dává výsledek x=1, (hodnotu y před přičtením 1), zatímco y=1; x= ++y; dává výsledek x=2; (hodnotu y po přičtením 1). Proměnná y se v obou případech rovná 2. Operace -- se chová analogicky až na odčítání 1 od proměnné y.

7. Podmíněný skok - if

Podmíněný skok if ... else se zapisuje běžným stylem, až na vynechání slova *then*, které jazyk C nezná. Syntaxe příkazu je patrná z následujícího příkladu:

```
char data[10];
int i;
        /*. nějaké příkazy programu... */
if(i<0) data[0] = '-'; else data[0] = '+';          /* lze napsat i pomocí operátoru ? :  jako
                                                      data[0] = i<0 ? '-' : '+'; */

if(i>=10 || i<=-10)
        { data[1] = 'x'; data[2]='2'; data[3]=0; }
 else
        { data[1] = '1'; data[2] = 0; }

if(i==0) data[0] = 0;
```

8. Nepodmíněný skok - goto

Překladače jazyka C bývají víceprůchodové (až 6-ti), a proto nemusí klást žádná omezení na skoky jako jednoprůchodový PASCAL. Návěští se nedefinují v hlavičce funkce, ale rovnou použijí. Platí uvnitř celé funkce bez ohledu na místo definice. Například:

```
#include <stdio.h>

void main(void)
 {      int cislo, delka;

NoveCislo: // návěští NoveCislo
        printf("Zadej cislo:"); scanf("%d", &cislo);      // vstup celého čísla
        delka=0; if(cislo<0) goto Konec;

Opakuj:   // návěští Opakuj
        if(cislo<=0) goto Vysledek;
        cislo /= 2; delka++;
        goto Opakuj;

Vysledek:  // návěští Vysledek
        printf("Delka cisla v bitech=%d\n",delka); goto NoveCislo;

Konec:  // návěští Konec
        printf("Konec programu.");
 }
```

- ***Poznámky:***

⇒ Uvedený příklad slouží pouze jako ukázka použití skoků, a nikoliv jako vzor hodný

následování. Volnost, kterou nám jazyk C poskytuje, neznamená, že budeme programy psát nestrukturovaně (*nebo i hůře*). Právě naopak! Jazyk C neklade žádná omezení, aby nám nekomplikoval práci v situacích, kdy je použití skoků nezbytné. **Stejně jako jinde i v jazyce C platí zásada minima nepodmíněných skoků.**

Dokážete přepsat předchozí příklad pomocí příkazů pro cykly?

⇒ Za jistých podmínek lze naprogramovat i skok ven z funkce - blíže viz. knihovní funkce setjmp() a longjmp().

9. Přepínač - switch

Umožňuje rozvětvit program na několik větví v závislosti na hodnotě parametru. Nepřekládá jako série za sebou jdoucích příkazů if jako analogická operace v PASCALu, ale seznamem hodnot parametru a k nim odpovídajících adres návěští a pomocí nich se skáče přímo na odpovídající úsek programu. Příkaz switch větví velmi rychle i při velkém množství odkazů.

Na druhou stranu skokové tabulky neumožňují zadávat intervaly pro cílové hodnoty parametru a každá cílová hodnota musí být konstantou známou v době překladu programu. Potřebujeme-li předat řízení na větev programu pro několik cílových hodnot parametru, musíme je samostatně vyjmenovat. Příslušné větve ukončuje příkaz break, který předává řízení na instrukci následující za příkazem switch. Pozor, je-li break vynechán, větev pokračuje za ní následující větví, jak ukazuje následující příklad:

```
char cislice;                   // znak načtený ze vstupního souboru
unsigned char priznak;          // bity proměnné definují vlastnosti znaku

#define LICHE 1                 // bit-0 udává, je-li číslice lichá
#define PRVOCISLO 2             // bit-1 udává, je-li číslice prvočíslo

/* ... nějaký program ...*/

switch(cislice)                 // větvení výpočtu podle hodnoty cislice
 {  case '7':
    case '5':
    case '3':
    case '1':                   // společná větev pro hodnoty cislice rovné '7', '5', '3' a '1'
       priznak = PRVOCISLO;     // zapiš priznak (bit-1)
                                // ! vynechán break ! - pokračuj na další větev- case '9'
    case '9':
       priznak |= LICHE;        // nastav bit-0  priznak
       break;                   // konec příkazu switch

    case '2':                   // větev pro hodnotu cislice rovnou  '2'
       priznak = PRVOCISLO;     // zapiš priznak (bit-1)
       break;                   // konec příkazu switch

    default:                    // přejdi sem, pokud není hodnota nalezena v tabulce,
       priznak = 0;             // nuluj příznak
 }

/* operace za příkazem switch */
```

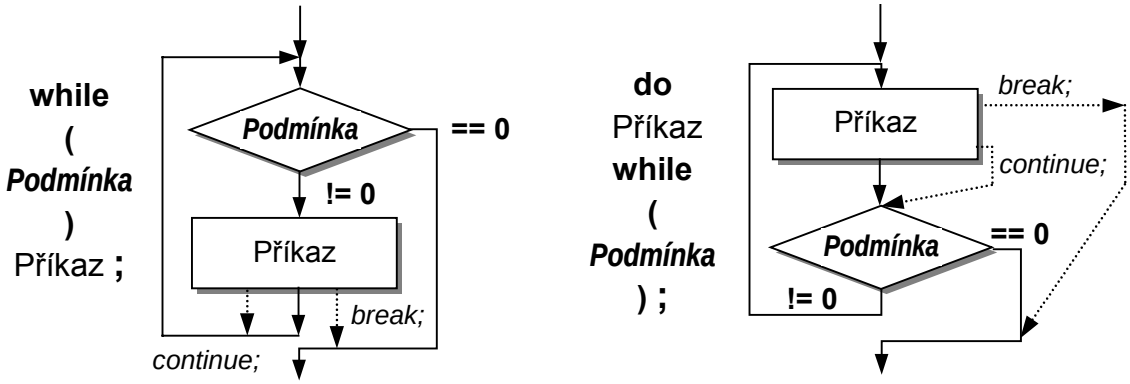
- ***Poznámky***

⇒ Větev default nemusí být nutně poslední, ale lze ji uvést kdekoliv a můžeme ji vynechat. V tom případě se při nenalezení odpovídající hodnoty pokračuje operacemi za příkazem switch.

⇒ Poslední větev příkazu switch nemusí končit break, neboť výpočet pokračuje za ní v každém případě. Některé učebnice ho doporučují uvádět ho. Pozor, je-li použito, některé kompilátory hlásí varování „*Nadbytečná operace*“, které lze vypnout.

10. Programové cykly

Jazyk C zná tři příkazy pro řízení cyklu. První dva jsou while a do...while



Oba podmíněné cykly zapisují podmínku klíčového slova while ve tvaru parametru funkce while(*Podmínka*). Podmínkou může být libovolný celočíselný výraz. Cyklus bude ukončen, bude-li se jeho hodnota rovnat 0. Za příkaz se považuje jeden výraz nebo blok několika příkazů uzavřený do { }.

Sekce Příkaz může obsahovat klíčové slovo break, které přeruší cyklus, nebo klíčové slovo continue, které ukončí vypočet sekce Příkaz a přejde na výpočet podmínky. Cykly lze také přerušit i nepodmíněným skokem z nich ven.

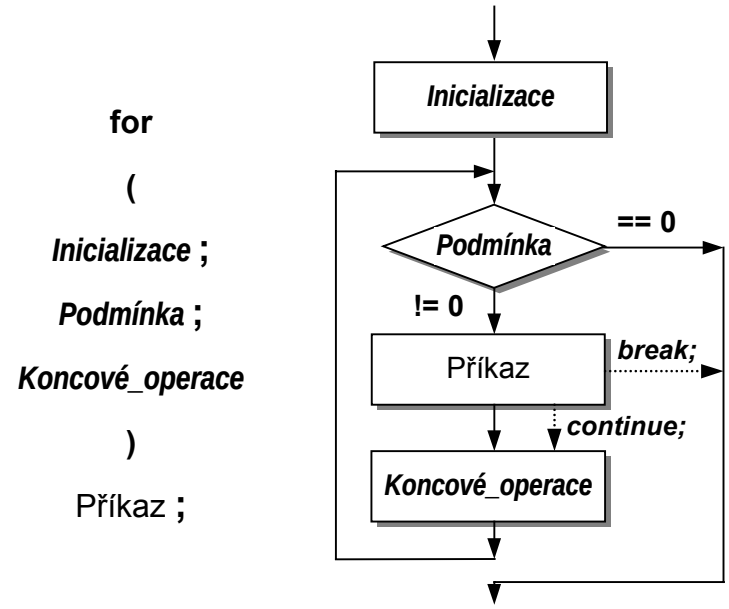
Programátoři PASCALu Nezapomínejte také na vždy pozitivní tvar podmínky while. Příkaz do...while tedy neodpovídá dvojici repeat...until jazyka PASCAL.

Všimněte si umístění středníků v následujících příkladech. Ukončují se jimi příkazy a v jazyce C se musí psát i tam, kde to v PASCALu není nutné.

```
#define POLEMAX 10
int pole[POLEMAX] = { 9,8,7,6,5,4,3,2,1,0 };    // pole inicializované čísly 9 až 0
int n,i,pom, bity;
long faktorial;

i=0; while( i<POLEMAX && pole[i] != 5 ) i++;    // hledání indexu čísla 5

i=0;
while(i<POLEMAX/2)                               // obrácení pořadí prvků uložených poli
 {  pom=pole[i];
    pole[i]=pole[POLEMAX-1-i];                   // Pozor, pole má prvky [0] až [POLEMAX-1] !
    pole[POLEMAX-1-i]=pom; i++;
 }
```

```
n=10; faktorial=1;                          // Vypočti faktoriál čísla 10
do  faktorial *= n--; while(n>0);

bity = 0;                                   // Odhadni počet bitů pro uložení 10!
do { faktorial /= 2; bity++; }              // log2(abs(n))+1>= bity >= log2(abs(n))
    while(faktorial);                       // Výpočet probíhá, pokud je faktorial != 0
```

Další podmíněným cyklem je příkaz for, který je ve své podstatě rozšířením příkazu while o dvě další sekce o ***Inicializaci*** a o ***Koncové_operace***. Sekce příkaz může opět obsahovat break nebo continue (a rovněž i nepodmíněný skok z cyklu ven) se stejnou funkcí jako u while.



Příkaz for nezavádí žádný interní parametr cyklu. Pouze vytváří posloupnost operací, a proto má širší uplatnění, než jemu analogické příkazy jiných programovacích jazyků. Ukázky operací while() z předchozího příkladu lze pomocí příkazu for napsat jako:

```
for( i=0; i<POLEMAX && pole[i] != 5; i++) ;          // hledání indexu čísla 5

for(i=0; i<POLEMAX/2; i++)                  // obrácení pořadí prvků uložených v poli
 {
        pom=pole[i];
        pole[i]=pole[POLEMAX-1-i];
        pole[POLEMAX-1-i]=pom;
 }
```

Příklad hledání indexu nepoužívá sekci Příkaz a na jejím místě nechává volné místo. Obecně můžeme kterýkoliv parametr přesunout vně příkazu for, včetně podmínky. Není-li podmínka použita pokládá se za nenulovou, avšak v tom případě musíme cyklus for ukončit sami pomocí break. Opačně lze kumulovat příkazy uvnitř jednotlivých sekcí for, pokud je od sebe oddělíme operátorem čárky. Následující zápisy příkazu for jsou z hlediska funkce shodné:

```
                    for(i=0, j=1, s=0;   i<=7; s += (znak & j) != 0, i++, j<<=1) ;

                    for(i=0, j=1, s=0;   i<=7; i++, j<<=1  )   s += (znak & j) != 0;
  j=1; s=0;         for(i=0;             i<=7; i++, j<<=1  )   s += (znak & j) != 0;
  j=1; s=0;         for(i=0;             i<=7; i++         ) { s += (znak & j) != 0; j<<=1; }
```

```
i=0; j=1; s=0;   for( ;              i<=7; i++          ) { s += (znak & j) != 0; j<<=1; }
i=0; j=1; s=0;   for( ;              i<=7; i++          ) { s += (znak & j) != 0; j<<=1; }
i=0; j=1; s=0;   for( ;              i<=7;              ) { s += (znak & j) != 0; i++; j<<=1; }

i=0; j=1; s=0;   for( ; ; ) {  if( i>7 ) break;
                               s += (znak & j) != 0;
                               i++; j<<=1;
                            }
```

⇒ Pro cykly platí totéž co pro skoky - volnost, kterou nám poskytuje jazyk C, nesmí být zneužívána. Program musí být v prvé řadě přehledný, a proto jsou z výše uvedených řádků přípustné pouze ty, kterým rozumíme na první pohled. Pokud jste nenašli takový řádek, upravte si příklad tak, aby se vám líbil.

Řekněte, co uvedená operace počítá. Dokážete ji napsat lépe nebo jiným způsobem?

11. Minimum z knihovních funkcí

Jazyk C nemá skoro žádné vestavěné funkce, s výjimkou několika systémových funkcí pro řízení programu, jako while(), if(), main() atd., ostatní se připojují z externích knihovních modulů - soubory *.LIB nebo *.DLL. V čase překladu se jejich definice zavádějí pomocí prototypů, které jsou shrnuty v souborech s příponou *.h. Postup bude probrán v kapitole 2. Počet knihovních funkcí je značný. Knihovny překladače Borland C++ 4.53 obsahují pro DOS skoro 600 funkcí plus několik desítek předdefinovaných konstant a v prostředí Windows je tento počet ještě několikanásobně vyšší.

V jazyce C lze nalézt hodně funkcí se stejnou činností, ale s různým názvem a odlišným pořadím parametrů (např. memset() a setmem()). Tyto duality existují z historických důvodů a pro přenositelnost na různé platformy. Třeba všechny funkce se jmény začínajícími na _f jsou analogie k funkcím se jmény bez _f a byly stvořené pro starší verze Win16 (např. memset() a _fmemset()). (*Pozn. Funkce s* _f *na začátku jména používají důsledně pointry kategorie* far**.** *Rozlišení kategorie pointru nemá význam pro Win32.*)

Většina funkcí má speciální použití, a proto nemá smysl učit se celé knihovny funkcí nazpaměť - při psaní programu máme k dispozici nápovědu, v níž najdeme vše - samozřejmě, pokud víme, že to tam je a jak to najít. Známe-li nejčastější funkce, ušetří nám to hodně času:

V následujícím přehledu bude uveden seznam funkcí rozdělný na tři části:

- **často používané funkce**, které je dobré umět nazpaměť včetně zápisu parametrů
- **užitečné funkce,** u nichž je vhodné vědět jejich jméno, použití a význam parametrů
- **zajímavé funkce**, pro něž stačí znát jen název a uplatnění

Seznam funkcí je doplněný příklady, ale jednotlivé funkce nejsou podrobněji vysvětleny. Chybějící popisy lze nalézt v nápovědě k překladači nebo v literatuře.

♦ *Často používané funkce*

printf() **- formátovaný výstup** (**#include <stdio.h>**). **P**atří mezi hojně využívané funkce a často přebírané jinými jazyky (např. MATLAB).

Příklad uplatnění - program:

```
char * text = "Alfa";       // pointer text ukazující na pole 5-ti znaků 'A', 'l', 'f', 'a', 0
double r = 314.1529;
printf(  "int=%3d uint=%3u long=%ld hex1=%3x hex2=%3X\n",    // popis formátu
          -1, -1, 2L, 200, 200);                            // hodnoty tištěné podle formátu
printf(  "text1=%6s text2=%-6s %c \'%c\'\n", text, text, 'a', 'a');
printf(  "Reálná čísla: "                                   // řetězec pokračuje na další řádce
         "r1=%6.2g r2=%6.2f r3=%+.3f r4=%+.3f",   // konec řetězce
          r, r, r, -r );
```

dává výstup na displej:

```
int= -1 uint=65535 long=2 hex1= c8 hex2= C8
text1=  Alfa text2=Alfa   a 'a'
Reálná čísla: r1=3.1e+02 r2=314.15 r3=+314.153 r4=-314.153
```

⇒ **Pozor**, počet a typ tištěných hodnot funkce printf musí přesně odpovídat %-specifikátorům uvedeným v popisu formátu, jinak hrozí chyba narušení ochrany paměti.

⇒ **Existuje celá řadu funkcí, které používají pro výstup stejný formát jako** printf() - sprintf(), fprintf(), cprintf() a liší se od sebe pouze cílem, kam směřuje výstup, jde o tak zvané přesměrování známé z UNIXu.

sprintf() **-** formátovaný výstup do řetězce (#include <stdio.h>). Provádí stejně formátovaný výstup printf(), ale místo na displej zapisuje výsledek do řetězce. Je snad o nejčastěji používanou funkci v jazyce C. Program:

```
char * text = "Alfa";       // pointer text ukazující na pole 5-ti znaků 'A', 'l', 'f', 'a', 0
char buf[256];   // pointer text ukazující na pole 5-ti znaků 'A', 'l', 'f', 'a', 0

sprintf(buf,"text1=%6s text2=%-6s %c \'%c\'\n", text, text, 'a', 'a');
```

Pole buf bude obsahovat řetězec znaků (zakončený 0), který můžeme pomocí znakových konstant zapsat jako: "Alfa text2=Alfa a \'a\'\n"**.**

⇒ **Pozor**, řetězec, do kterého ukládáme výsledek musí mít dostatečnou délku, jinak hrozí chyba narušení ochrany paměti.

wsprintf() - knihovní funkce systému Windows, částečný ekvivalent funkce sprintf, umí pouze celá čísla

main() - povinný vstupní bod programu v prostředí DOS a ve „*Win32-Console*“. Může být bez parametrů:

```
void main();
```

nebo mít až 0 až 3 parametry a vracet výstup int:

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

Parametry env[ ] se používají zřídka. Obsahují proměnné typu „*enviroment“*, definované obvykle v AUTOEXEC.BAT, jejichž význam ustoupil do pozadí s nástupem Windows. Popis významu env[ ] lze nalézt v nápovědě k překladači.

Veličina argc obsahuje počet parametrů předaných programu v char * argv[ ] (pole mající za prvky pointry na řetězce). Program dostává v argv[0] svoje jméno včetně

úplné cesty, a proto je argc vždy větší nebo rovno jedné. Další členy argv pak obsahují parametry, které programu dostal při spuštění.

Napíšeme-li program:

```
#include <stdio.h>

int main(int argc, char * argv[])
{
   if(argc<4)                          // chceme tři parametry, argc musí být 4
    {  printf("\nSyntax: TEST par1 par2 par3\n");       // zadáno méně parametrů
       return 1;                       // konec s chybou 1
    }
   printf(   "argc=%d\n" "argv[0]: %s\n" "argv[1]: %s\n"     // Pozor,  jeden řetězec!
             "argv[2]: %s\n" "argv[3]: %s\n",                // Pořád jeden řetězec!
             argc, argv[0], argv[1], argv[2], argv[3]  // tištěné parametry
         );
   return 0;        // konec, návratová hodnota 0 značí bez chyby
}
```

a spustíme ho:

```
C:\MOJEPROG\TEST.EXE prvni "text s mezerami" 3
```

bude jeho výstup:

```
argc=4
argv[0]: "C:\MOJEPROG\TEST.EXE"
argv[1]: "prvni"
argv[2]: "text s mezerami"
argv[3]: "3"
```

Parametr argv[argc], tj. v našem příkladu argv[4], je obecně nedefinovaný, ale některé překladače ho nastavují na hodnotu NULL, což je systémová konstanta mající význam nedefinované adresy pointru.

♦ *Užitečné funkce*

Funkce pro práci s řetězci - jejich prototypy jsou v souboru **string.h**

Doporučuji letmo se seznámit se všemi funkcemi ze string.h, s výjimkou funkcí jejichž jména začínají _f (analogie k funkcím se jmény bez _f), a pečlivě si prostudovat zejména tyto funkce:

strlen() - zjištění délky řetězce
strcpy() - uložení (kopírování) jednoho řetězce do druhého (analogii =)
strncpy() - uložení (kopírování) jednoho řetězce do druhého s omezením délky
strcat() - připojení řetězce na konec řetězce (analogie +=)
strncat() - připojení řetězce na konec řetězce s omezením délky
strcmp() - porovnání dvou řetězců (analogie ==)
strncmp() - porovnání dvou řetězců s omezením délky porovnávaného úseku
memset() - uložení znaku do úseku paměti (do celého řetězce)
Pozor! Nezaměňovat memset() s podobnou funkcí setmem(), která má jiné pořadí parametrů.

Napíšeme-li program:

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

void main(void)
{
   char text1[255];          // u znakových polí se vyplácí nešetřit délkou
   char text2[5];            // zde šetříme a to se nám vymstí
   strcpy(text1,"Alfa");     /* uložení řetězce "Alfa" do text1 Pozor, příkaz text1="Alfa"
                             má zcela odlišný význam. Řekněte jaký? */
   strncpy(text2,"DruhyText", sizeof(text2) );   /* do text2 uložíme maximální počet
                                                 znaků, které může pojmout. */
   text2[sizeof(text2)-1]=0; /* Pozor, strncpy() neukládá koncovou 0, došlo-li k omezení, a
                             proto ji musíme uložit do posledního prvku text2. Pozn. Totéž
                             platí i strncat() */
   strcat(text1,text2);      // text1 je dost velké, a proto nemusíme užít strncat()
   printf("Řetězec text1: %s má délku %d. Obsah text2: %s\n",text1, strlen(text1), text2);
}
```

bude jeho výstup: Řetězec text1: AlfaDruh má délku 8. Obsah text2: Druh

⇒ Operace s řetězci se v jazyce C často provádějí pomocí funkce sprintf(), jejíž široké možnosti nahrazují nedefinované aritmetické operace pro řetězce existující v PASCALu. Další možnost nabízejí objekty, ale o nich až později.

Funkce pro převod číselných řetězců na hodnoty

sscanf() (#include <stdio.h>) - formátovaný vstup převod řetězce znaků na hodnot proměnných. Zápis formátu se podobá formátům v printf(), ale není s nimi zcela shodný.

Funkce sscanf() existuje i ve verzi pro přímý převod ze zařízení vstupu

(obvykle klávesnice) scanf() a také ve verzi pro převod údajů ze souboru fscanf(), ale doporučuje se napřed načíst řetězec a ten převádět sscanf().

atol() - (#include <stdlib.h>) - převod řetězce na číslo long
atoi() - (#include <stdlib.h>) - převod řetězce na číslo int
atof() - (#include <math.h>) - převod řetězce na číslo double

Funkce pro čtení a zápis do souborů - (#include <stdio.h>)

fopen() - otevření souboru s možnostmi - pouze čtení, nebo i zápis v režimu binární data nebo text; dále možnost připojení nových dat nakonec starého souboru, nebo přepsání starého souboru, či vytvoření nového souboru.
fclose() - zavření souboru otevřeného fopen()

fseek() - nastavení ukazatele pozice čtení zápisu v otevřeném souboru
ftell() - zjištění pozice ukazatele čtení zápisu v otevřeném souboru

feof() - test, je-li ukazatel pozice čtení zápisu na konci souboru

fgetc() - čtení znaku z otevřeného souboru
fputc() - zápis znaku do otevřeného souboru

fgets() - čtení celé řádky z otevřeného souboru
fputs() - zápis řetězce do otevřeného souboru

fprintf() - analogie printf() a sprintf(), ale s výstupem do souboru

fread() - čtení bloku z otevřeného souboru
fwrite() - zápis bloku do otevřeného souboru

Program používající uvedené funkce pro práci se souborem může vypadat takto:

```
#include <stdio.h>                    // FILE *, fopen(), size_t

/* Popis: Funkce přečte do pole adresa blok dat ze souboru jmeno od jeho pozice zacatek
          a o maximální délce pozadovano bytů. Funkce vrací počet skutečně přečtených
          bytů, nebo 0, nebyl-li soubor nalezen. */
size_t PrectiBlok (          void * adresa,          // pole o délce min. pozadovano bytů
   const char * jmeno,       // soubor, z něhož se má blok načíst
   long zacatek,             // počáteční pozice od začátku souboru
   size_t pozadovano         // požadovaná délka bloku
   )
{
   FILE * in;              //  vstupní soubor
   size_t delka;  /* size_t je typ definovaný v stdio.h jako unsigned int pro vyjádření délky
                                 bloku čteného ze souboru */
   in=fopen(jmeno,"rb");          // otevři soubor pro čtení binárním dat
   if( in==NULL )                 // testuj, zdařilo-li se otevření
            return 0;             // soubor nenalezen

   if(      zacatek != 0          // volání fseek po fopen jen tehdy, je-li zacatek ) != 0 *

        &&  (  0!=fseek(in,zacatek, SEEK_SET) // nastav pozici čtení souboru na zacatek,
            || ftell(in) != zacatek )         /* překontroluj, zdařilo-li se nastaveni,
                                 nutnost kontroly je nedostatkem MS-DOSu, který hlásí
                                 chybu, pokud se změna pozice nezdaří. */
    )
```

```
        {  fclose(in); return 0; }        /* zadaná pozice nenalezena, (soubor kratší) - zavři
                                          soubor a vrať 0 */
    delka = fread( adresa, 1, pozadovano, in );  // čti data o délce 1 byte v počtu pozadovano
    fclose(in);                           // zavři soubor
    return delka;                         // vrat skutečně přečtený počet dat
}
int pamet[2000];                          /* velká datová pole je vhodné deklarovat mimo funkci,
                                          aby nedošlo k přetečení zásobníku */
void main(void)
{  size_t precteno;
   precteno = PrectiBlok(pamet,"X:\\Vyuka\\Rtime\\Prj\\Ecg\\Person1.ecg",
                                          100000L, sizeof(pamet) );
   printf("%d\n", precteno);
}
```

♦ *Zajímavé funkce*

Pomocné funkce pro práci se soubory:

remove() - (#include <stdio.h>) smazání ***zavřeného*** souboru

fnsplit() - (#include <dir.h>) rozklad úplného jména souboru na jméno souboru, příponu, adresář a písmeno disku

fnmerge()- (#include <dir.h>) vytvoří úplné jméno souboru ze jména souboru, přípony, adresáře a písmena disku

findfirst() a **findnext**() - (#include <dir.h>) hledání souboru v adresáři podle jména, které může obsahovat znaky * ?

searchpath() - (#include <dir.h>) hledání souboru v adresářích uvedených v PATH

Čtení znaků z klávesnice *(DOS, Easy Windows a Win32-Console):*

kbhit() - (#include < conio.h>) test na stisknutou klávesu
getch() - (#include < conio.h>) čekej na znak

Testy hodnot *(* #include < conio.h> *):*

isdigit() - test, je-li znak číslice
isalpha() - test, je-li znak písmeno (bohužel neumí české znaky)

a další podobné funkce.