Programování v jazyku ILE C pro IBM i

Vladimír Župka

Obsah

Předmluva

Kurs je určen zájemcům o programování v jazyku ILE C pro systém IBM i se zvláštním zřetelem na jeho použití pro funkce operačního systému. Kurs je určen lidem, kteří alespoň trochu znají jazyk C, např. ze školy. Po co možná nejstručnější rekapitulaci základů jazyka C se kurs věnuje hlavně příkladům.

Tato publikace slouží jen jako pomůcka ke školení, nesnaží se vyložit všechny prostředky jazyka, uvádí jen ty nejpoužívanější. Další prostředky a přesné definice je nutno hledat v oficiálních příručkách IBM.

Probírají se témata, která mohou být užitečná v praxi. Jde zejména o programování databázových souborů s původním přístupem k záznamům, programování obrazovkových souborů, programování tiskových souborů, použití API (Application Program Interface) k získání informací z operačního systému, použití API typu UNIX k programování komunikací (sockets) apod.

V rámci příkladů se příležitostně uvádějí některé další pojmy jazyka C, které nejsou probírány v rekapitulační části, nebo se znovu vysvětlují některé obtížnější pojmy.

Podrobné informace o jazyku ILE C a dalších tématech můžeme nalézt příručkách

ILE C/C++ Language Reference
ILE C/C++ Runtime Library Functions
ILE C/C++ Programmer's Guide
ILE C/C++ Compiler Reference
Database Programming
DDS Concepts
Programming DDS for Display files
Programming DDS for Printer files

Obecné informace o API (Application Program Interfaces) nalezneme v příručce
API overview and concepts

a informace o konkrétních API můžeme vyhledat pomocí stránky
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/apifinder/finder.htm

Vytvoření programu

Jeden zdrojový soubor

V tomto příkladu je program tvořen jedním zdrojovým souborem PGM01, z něhož se přímo vytvoří program stejného jména. Zdrojový soubor může být vytvořen a umístěn dvěma způsoby.

První způsob předpokládá, že pomocí PDM (Programming Development Manager) a SEU (Source Entry Utility) zapíšeme text programu do samostatného *členu PGM01* ve zdrojovém databázovém souboru (např. v QCSRC).

Druhý způsob předpokládá, že v integrovaném systému souborů (IFS) vytvoříme adresář, pojmenovaný např. KURS_C a umístíme jej např. do systému souborů ROOT. V tomto adresáři vytvoříme *soubor PGM01.C* libovolným způsobem z PC (např. programem Notepad – Poznámkový blok). Soubor můžeme vytvořit také ze systému IBM i kopírováním zdrojového členu do adresáře příkazem CPYTOSTMF. Kopírovaný zdrojový člen může být i prázdný:

```
CPYTOSTMF
         FROMMBR('/qsys.lib/knihovna.lib/qcsrc.file/pgm01.mbr')
         TOSTMF('/Kurs_C/Pgm01.c')
         STMFOPT(*REPLACE)
```

Lomítko před jménem adresáře Kurs_C (nebo qsys.lib) znamená, že adresář je umístěn v systému souborů ROOT. Malá a velká písmena ve jménech adresářů a souborů se v systému souborů ROOT nerozlišují.

Zdrojový soubor PGM01

```
/* Funkce max */

int max (int a,int b)            /* definice funkce (function definition) */
{
   if ( a > b )
      return (a);
   else
      return (b);
}

/* Funkce main (hlavní funkce) */
                                  /* příkazy předkompilátoru (preprocesoru) */
#include <stdio.h>                   /* deklarace (prototypy) I/O funkcí */
#define ONE   1                      /* definice konstant */
#define TWO   2

extern int max(int, int);         /* deklarace funkce (function declaration) */

main() /* definice funkce (function definition) */
{
   int a, b, c;                   /* deklarace a definice proměnných */

   a = ONE;                       /* příkazy */
   b = TWO;
   c = max(a,b);
   printf ("maximum = %d\n", c);
}
```

Text programu můžeme zapsat buď pomocí SEU (Source Entry Utility) do zdrojového členu PGM01 databázového souboru QCSRC, nebo vhodným editorem v PC do souboru Pgm01.c adresáře /Kurs_C v IFS. Podle toho, kde máme zdrojový soubor, zvolíme způsob kompilace.

Máme-li zdrojový text v databázovém členu PGM01 souboru QCSRC, použijeme ke kompilaci tento příkaz:

```
CRTBNDC
      PGM(knihovna/PGM01)
      SRCFILE(knihovna/QCSRC)     zdrojový soubor
      SRCMBR(*PGM)                zdrojový člen stejného jména jako program
      SHOWINC(*YES)               tisk prototypo v protokolu
      OUTPUT(*PRINT)              tisk protokolu o kompilaci
      DBGVIEW(*LIST)              umožnit ladicí režim
```

Máme-li zdrojový text v adresáři IFS, použijeme tento příkaz:

```
CRTBNDC
      PGM(knihovna/PGM01)
      SRCSTMF('/Kurs_C/Pgm01.c')  adresář, v němž je umístěn zdrojový text
      SRCMBR(*PGM)                zdrojový člen stejného jména jako program
      SHOWINC(*YES)               tisk prototypu v protokolu
      OUTPUT(*PRINT)              tisk protokolu o kompilaci
      DBGVIEW(*LIST)              umožnit ladicí režim
```

Program pak můžeme spustit obvyklým příkazem CALL z příkazového řádku CL:

```
CALL knihovna/PGM01.
```

Dva zdrojové soubory

V tomto příkladu je program rozdělen do dvou zdrojových souborů, které se kompilují samostatně a pak se spojí do programu.

Zdrojový soubor 1 (PGM01_1)

```
/* Funkce main (hlavní funkce) */
                                   /* příkazy předkompilátoru (preprocesoru) */
#include <stdio.h>                    /* deklarace (prototypy) I/O funkcí */
#define ONE   1                       /* definice konstant */
#define TWO   2

extern int max(int, int);          /* deklarace funkce (function declaration) */

main() /* definice funkce (function definition) */
{
   int a, b, c;                    /* deklarace a definice proměnných */

   a = ONE;                        /* příkazy */
   b = TWO;
   c = max(a,b);
   printf ("maximum = %d\n", c);
}
```

Zdrojový soubor 2 (PGM01_2)

```
/* Funkce max */

int max (int a,int b)             /* definice funkce max se dvěma argumenty */
{
   if ( a > b )
      return (a);
   else
      return (b);
}
```

V tomto příkladu se program skládá ze dvou zdrojových souborů. Zdrojovému souboru se někdy říká *kompilační* či překladová *jednotka* (compilation unit, translation unit), protože se kompiluje (překládá) samostatně. Program se tvoří z kompilačních jednotek tak, že nejprve se

každá zkompiluje do modulu (objektu typu *MODULE), načež se moduly spojí *spojovacím programem* (příkazem CRTPGM) do objektu typu *PGM.

Máme-li zdrojový text v databázovém členu PGM01_1 souboru QCSRC, použijeme ke kompilaci tento příkaz CRTCMOD:

```
CRTCMOD
      MODULE(knihovna/PGM01_1)   výsledný modul
      SRCFILE(knihovna/QCSRC)    zdrojový soubor
      SRCMBR(*MODULE)            zdrojový člen stejného jména jako modul
      SHOWINC(*YES)              tisk prototypu v protokolu
      OUTPUT(*PRINT)             tisk protokolu o kompilaci
      DBGVIEW(*LIST)             umožnit ladicí režim
```

Stejně vytvoříme modul PGM01_2.

Máme-li zdrojový text v souboru PGM01_1.c v adresáři /Kurs_C v IFS, zkompilujeme jej do modulu příkazem CRTCMOD stejně jako výše, ale s jiným označením zdroje:

```
CRTCMOD
      MODULE(knihovna/PGM01_1)
      SRCSTMF('/Kurs_C/PGM01_1.c')       adresář, v němž je umístěn zdrojový text
      OUTPUT(*PRINT)
      OPTION(*SHOWINC)
      DBGVIEW(*LIST)
```

Stejně vytvoříme modul PGM01_2.

Z modulů vytvoříme program PGM01 příkazem CRTPGM, nyní již nezávisle na tom, kde byl umístěn zdrojový text:

```
CRTPGM
      PGM(knihovna/PGM01)                          jméno spustitelného programu
      MODULE(knihovna/PGM01_1 knihovna/PGM01_2)    moduly tvořící program
      ENTMOD(*FIRST)                               vstupní modul
```

Program pak můžeme spustit normálním příkazem CALL z příkazového řádku CL:

```
CALL knihovna/PGM01.
```

(Argumenty ani návratová hodnota funkce main se zde nevyužívají.)

Rekapitulace základů jazyka C

Identifikátory

Identifikátory jsou jména proměnných. Musí vyhovovat těmto pravidlům:

- jsou neomezeně dlouhé, ale rozlišuje se prvních 255 znaků,
- skládají se z písmen a až z, A až Z, číslic 0 až 9 a znaku _ (podtržítko),
- musí začínat písmenem nebo podtržítkem,
- rozlišují se malá a velká písmena.

Typy dat a deklarace proměnných

Jazyk ILE C rozeznává tyto (jednoduché, základní, primitivní) typy dat:

- celá čísla - integer numbers (integers),
- znaky - characters,
- čísla v pohyblivé řádové čárce - floating-point numbers,
- pakovaná dekadická čísla - packed decimal numbers (zvláštnost IBM i),
- void - prázdná množina dat v deklaraci funkce nebo typ ukazatele na libovolný typ dat.

Z těchto typů jsou odvozeny další typy dat:

- pole - arrays,
- ukazatele - pointers,
- struktury - structures,
- unie - unions,
- přejmenování typu - typedef,
- výčtové typy - enumerations.

Stručný přehled primitivních typů je uveden v následující tabulce:

Typ dat	**Označení typu**	**Velikost**	**Příklady konstant**
integer nejběžnější krátké neliší se od int není standardní varianty bez znaménka	 int short long long long unsigned	 4 B 2 B 4 B 8 B	 17 dekadicky 17 0x11 hexadecimálně 17 021 oktalově 17 75L typ long
floating-point	float double long double	4 B 8 B 8 B	25.5L 450E-12L -34879.789e+12
character	char unsigned char signed char	1 B	'a', '1', '.', \n
packed decimal	decimal (n, p)	n = 1 až 31 p = 0 až n (1 až 16 B)	-123.25d

Upozornění: Znaková data, včetně řídicích (escape) znaků, jako \n, \r apod., jsou kódována v soustavě EBCDIC (nikoliv ASCII)! Např. \n má hexadecimální vyjádření 0x15, nikoliv 0x0a, jak se uvádí v učebnicích jazyka C.

Typ integer (celá čísla)

Celočíselný typ je založen na binárním vyjádření čísel v rozsahu 2 až 8 bajtů. Celé číslo může být se znaménkem nebo bez znaménka (unsigned).

Možné kombinace celočíselných typů (tučně jsou označeny ty častěji používané):

- short int, **short**, signed short, signed short int (2 bajty),
- signed, signed int, **int** (4 bajty),
- long int, **long**, signed long, signed long int (4 bajty),
- unsigned short int, **unsigned short** (2 bajty),
  unsigned, unsigned int (4 bajty),
- unsigned long int, **unsigned long** (4 bajty),
- long long int, long long, signed long long, signed long long int (8 bajtů),
- unsigned long long int, unsigned long long (8 bajtů).

Deklarace bez inicializace vyhrazuje paměť, je zároveň *definicí* proměnné a jejího *typu*:

```
int              i;
int n1, n2, n3;
unsigned m, n;
```

Deklarace s inicializací je rovněž *definicí*, navíc však přiděluje proměnné *hodnotu*:

```
int  a = 1;
int  b1 = 1, b2 = 2, c = 10 ;
unsigned long pocet = 100;
const int konstanta = 100;
```

Symbol *const* je tzv. kvalifikátor. Umožňuje definovat proměnné, které se chovají v programu jako konstanty, tj. jejich obsah nelze výpočtem změnit. Kvalifikátor const lze použít u každé deklarace dat, nejen celočíselného typu.

Znakový typ (character type)

Typy *char* a *unsigned char* jsou stejné. Typ signed char může sloužit jako celé číslo se znaménkem o velikosti jednoho bajtu. Znakové *konstanty* mají typ *int*, nikoliv char. Lze s nimi tedy provádět výpočty.

Příklady znakových konstant:

'a' znak a,
'0' znak nula,
'\' znak zpětné lomítko,
'(' znak levá kulatá závorka,

Tzv. *escape* (změnové, únikové) sekvence jsou dvou- nebo víceznaky začínající zpětným lomítkem. Používají se k vyjádření znaků, které nejsou viditelné, nebo které by v určitém kontextu znamenaly něco jiného než zamýšlíme, nebo které chceme vyjádřit číselně.

Speciální znaky vyjádřené escape sekvencemi:

Escape sekvence	**Odpovídající znak**
\a	alert (bell) - poplach
\b	backspace - znak zpět
\f	form feed (new page) - nová stránka
\n	new-line, line feed - nový řádek (parametr kompilace SYSIFCOPT (*IFSIO) mění hodnotu 0x15 na 0x25)
\r	carriage return - návrat vozíku
\t	horizontal tab - horizontální tabulátor
\v	vertical tab - vertikální tabulátor
\'	single quotation mark - jednoduchá uvozovka - apostrof
\"	double quotation mark - dvojitá uvozovka - uvozovka
\?	question mark - otazník
\\	backslash - zpětné lomítko
\0	NULL = 0x00

Poznámka: Následující sekvence

```
\
"pokračování zdrojového textu na novém řádku"
```

tj. zpětné lomítko bezprostředně následované neviditelným znakem "nový řádek" se používá ve znakových řetězcích nebo v příkazech předkompilátoru (např. #define) k oznámení, že současný zdrojový řádek pokračuje na dalším řádku. Tento "dvojznak" se nepovažuje za escape sekvenci.

Příklady speciálních znaků:

'\n' znak nový řádek používaný často v tiskových funkcích, např. printf,
'\\' znak zpětné lomítko,
'\0' znak NULL nebo prázdná adresa.

Následující tři řádky vyjadřují tutéž konstantu:

'\118' znak vyjádřený escape sekvencí s dekadickou konstantou,
'0x76' znak vyjádřený hexadecimální konstantou,
'v' znak vyjádřený písmenem.

Deklarace znaků bez inicializace

```
char a, b, c;
unsigned char d;
signed char e;
```

Deklarace znaků s inicializací

```
char znak = 'A';

#define A   'A'
char pismeno = A;

const char jednicka = '1';
const char hvezdicka = '*';
```

Typ packed decimal (pakovaná dekadická čísla)

K použití dekadických čísel je třeba v programu použít hlavičkový soubor <decimal.h>.

Deklarace pakovaných čísel bez inicializace

```
decimal (15, 2) ucet_MD;
decimal (25, 2) ucet_MD_ITL;
```

Deklarace pakovaných čísel s inicializací

```
decimal (15, 5)  kurs_meny = 37.85d;
const decimal (10,2)  sto = 100.D;
decimal (4,0)  citac = 99;         /* může být bez označení D nebo d */
```

Příklady pakovaných dekadických konstant

Pakovaná dekadická konstanta	**(velikost, přesnost)**
1234567890123456D	(16,0)
12345678.12345678D	(16,8)
12345678.d	(8,0)
.1234567890d	(10,10)
-12345.99d	(7,2)
000123.990d	(9,3)
0.00D	(3,2)
200.425	(6,3)

Operátory

Aritmetické operátory

Binární operátory stojí mezi dvěma číselnými operandy:

- \+ plus
- \- minus
- * krát
- / děleno (pro celá čísla dělí celočíselně: 5 / 2 dá hodnotu 2)
- % "modulo", zbytek po dělení jen pro celá čísla: 3 % 2 dá hodnotu 1)

Operátory + a – mohou být také před jedním operandem. Pak se nazývají unární.

Relační operátory

== rovno (Pozor, dvě rovnítka! Jedno rovnítko by znamenalo operátor přiřazení)
!= nerovno (vykřičník a rovnítko)
< menší
> větší
<= menší nebo rovno
>= větší nebo rovno

Logické operátory

&& "a" – logický součin (AND)
|| "nebo" – logický součet (OR)
! "ne" – logická negace (NOT)

Bitové operátory

Bitové operátory provádějí také logické operace, ale jen s odpovídajícími jednotlivými bity v celočíselných operandech.

```
&     bitový logický součin (AND):  0 & 1 = 0
                                    1 & 0 = 0
                                    1 & 1 = 1
                                    0 & 0 = 0

|     bitový logický součet (OR):   0 | 1 = 1
                                    1 | 0 = 1
                                    1 | 1 = 1
                                    0 | 0 = 0

~     bitový doplněk (NOT):         ~1 = 0
                                    ~0 = 1

^     bitová nonekvivalence (XOR – exclusive OR)
                                    0 ^ 1 = 1
                                    1 ^ 0 = 1
                                    1 ^ 1 = 0
                                    0 ^ 0 = 0
```

Příklady s hexadecimálními konstantami:

```
0xFFFB & 0x000F = 0x000B     odmaskování poslední hexadecimální číslice B
0x000A & 0x0002 = 0x0002     odmaskování bitu 2 (= hexadecimální číslice 2)
0x0001 | 0x00F0 = 0x00F1     přimaskování číslice F na druhé místo zprava
                        (horní půlbajt)
```

Bitové posuvy

Tyto operátory mají dva operandy: první je celé číslo v binární reprezentaci a druhý je celé číslo udávající počet bitových posuvů.

>> bitový posuv vpravo (rovnocenný dělení mocninou 2^1)
<< bitový posuv vlevo (rovnocenný násobení mocninou 2^1)

Například:

```
0x0008 >> 2 = 0x0002 neboli 2 (= 8 / 2^2),
0x0008 << 2 = 0x0020 neboli 32 (= 8 * 2^2)
```

Operátory přiřazení

```
=       pravá strana se překopíruje do levé strany (nejčastější přiřazení)
a += b  je stejné jako  a = a + b
a -= b                  a = a - b
a *= b                  a = a * b
a /= b                  a = a / b
a %= b                  a = a % b
a &= b                  a = a & b
a |= b                  a = a | b
a >>= b                 a = a >> b
a <<= b                 a = a << b
```

Operátory přičtení a odečtení jedničky (inkrementace a dekrementace)

```
a++    je stejné jako.  a = a + 1    po vyhodnocení ve výrazu
a—                      a = a — 1    po vyhodnocení ve výrazu
++a                     a = a + 1    před vyhodnocením ve výrazu
—a                      a = a - 1    před vyhodnocením ve výrazu
```

Operátor adresy a dereference

Operáror adresy je unární operátor získání adresy proměnné a označuje se znakem & před proměnnou.

```
b = &a;
```

znamená, že do *ukazatele b* se dosadí *adresa* proměnné *a*.

Operátor dereference je unární operátor vyhodnocení ukazatele a označuje se * (hvězdičkou) před ukazatelem. Ve výrazu získá z ukazatele adresu a z ní vyjme hodnotu. Dělá vlastně obrácenou činnost než operátor adresy.

```
a = *b;
```

znamená získání hodnoty z paměti, kam ukazuje ukazatel b a její dosazení do proměnné a.

Další operátory

[]	operátor přístupu k položkám pole,
()	operátor volání funkce,
.	operátor tečka – přímý přístup ke členům struktury nebo unie,
->	operátor šipka – nepřímý přístup (přes ukazatel) ke členům struktury nebo unie,
()	operátor změny typu, přetypování, konverze, casting,
sizeof()	operátor získání délky proměnné nebo typu v bajtech.

Výrazy

Volně řečeno, výraz je kombinace operandů a operátorů doplněná podle potřeby uzavíráním do závorek. Operandy mohou být proměnné, volání funkce nebo konstanty rozmanitých typů. Výrazy se používají v příkazech. Výraz, za nímž následuje středník, se stává příkazem.

Nejčastěji používaný je *přiřazovací výraz*, který má tento obecný tvar:

```
lvalue = výraz
```

Zde je *lvalue* označení pro výraz schopný přijímat hodnotu, nejčastěji identifikátor proměnné. Nemůže to tedy být např. konstanta. Výraz na pravé straně ovšem může být i konstanta. Po doplnění středníku se přiřazovací výraz stává přiřazovacím příkazem:

```
lvalue = výraz;
```

Pozoruhodnou (nepříliš zřejmou) vlastností jazyka C je to, že *přiřazovací výraz* nabývá *hodnotu*. Například přiřazovací výraz

a = 0

má hodnotu nula (stejnou jako levá strana) a přiřazovací výraz

a = 1

má hodnotu 1 (stejnou jako levá strana). To by samo o sobě nebylo nic divného, ale nabývá to významu v logických operacích. Celé číslo 0 se považuje za logickou hodnotu nepravda (false), každé jiné číslo za hodnotu pravda (true). Pak tedy výraz

a = 1 je vždy "pravdivý" z definice, a výraz

a = 0 je vždy "nepravdivý" z definice,

zatímco relační výraz

a == 1 nebo

a == 0

může a nemusí být pravdivý, podle toho jaká je hodnota proměnné a. Hodnotou relačního výrazu (a obecně logického výrazu) je 0 (nepravda) nebo 1 (pravda). Je vždy nutné dávat pozor a rozlišovat operátor = od operátoru ==.

Přesná obecná definice výrazu je mnohem složitější a většinou se v praxi nepotřebuje.

Příkazy

Příkazy jsou tyto:

- výrazový příkaz (výraz ukončený středníkem)
- složený příkaz čili blok (deklarace a příkazy uzavřené ve složených závorkách)
- prázdný příkaz (samotný středník)
- **if**
- **while**
- **do**
- **for**
- **goto** a příkaz s návěštím
- **break**
- **continue**
- **return**
- **switch**

Výrazový příkaz

Výrazový příkaz je výraz následovaný středníkem. Například

```
a = b; a = b = c;
printf ( "Hello world" );
```

Složený příkaz

Složený příkaz, čili blok se skládá z deklarací a příkazů uzavřených ve složených závorkách. Není za ním středník. (Nezaměňovat se seznamem hodnot u inicializace hodnot polí nebo struktur, za nímž je středník.) Používá se většinou u příkazu if, while, do a u definice funkce. Příklad samostatného bloku:

```
{
   int     a, b, c;
   double  g = 3.01;

   a = (b + c) * g;
}
```

Příklad bloku v definici funkce:

```
int max (int a,int b)
{
   if ( a > b )
      return (a);
   else
      return (b);
}
```

Příklad bloku v příkazu if:

```
if ( a > b ) {
   a = 1;
   b = 0;
}
```

Příkaz if

Příkaz if má dva základní tvary:

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

Zde může být za příkaz opět dosazen opět příkaz *if* nebo jiný příkaz. Výraz je zpravidla logický výraz, ale také přiřazovací výraz nebo volání funkce nebo konstanta nebo prázdný příkaz atd. Výraz v závorkách tedy znamená podmínku. Je-li výraz pravdivý (čili hodnota výrazu je různá od 0), je podmínka splněna a výpočet pokračuje příkazem za kulatou závorkou. Není-li výraz pravdivý (hodnota výrazu je 0), příkaz za kulatou závorkou se neprovede; je-li zadáno slovo *else*, provede se příkaz zapsaný za ním.

```
if (a == 1) c = 500;
else if (a == 2) c = 300;
     else if (a == 3) c = 100;
          else c = 0;
```

Tato konstrukce se nazývá *else-if* a často se používá, když se podmínky u jednotlivých výrazů vzájemně vylučují. Jde vlastně o vnořování příkazů *if* do příkazů *else* ve více úrovních. Obvykle se však vnořené příkazy neodsazují a příkaz by vypadal spíše takto:

```
if      (a == 1) c = 500;
else if (a == 2) c = 300;
else if (a == 3) c = 100;
else             c = 0;
```

Poznámka: Existuje tzv. *podmínkový výraz* (contitional expression), který má tento tvar:

```
výraz1 ? výraz2 : výraz3
```

kde *výraz1* je podmínka, *výraz2* se vyhodnotí, je-li podmínka splněna, jinak se vyhodnotí *výraz3*. Vyhodnocený výraz se stává hodnotou celého podmínkového výrazu. Používá se jako zjednodušený příkaz *if*. Například

```
c = (a == 1) ? 1 : 0;
```

znamená: "jestliže a se rovná jedné, c bude 1, jinak 0". Je to totéž, jako bychom napsali

```
if (a == 1) c = 1; else c = 0;
```

Příkaz while

Příkaz *while* je obecný příkaz cyklu (smyčky). Má tento obecný tvar:

```
while (výraz) příkaz
```

Výraz v kulatých závorkách má stejný význam podmínky jako v příkazu if. Je-li podmínka splněna, *příkaz* se provede a znovu se zkoumá platnost podmínky. *Příkaz* se opakuje tak dlouho, dokud podmínka platí. Jestliže podmínka neplatí, příkaz se přeskočí. Není-li podmínka splněna hned po prvé, *příkaz* se neprovede ani jednou.

```
dbfbP  = _Rreadn ( dbP, &dbb, sizeof(dbb), __DFT ); /* číst první záznam */
while (dbfbP->num_bytes != EOF) {
   /* ... zpracování záznamu */
   dbfbP  = _Rreadn ( dbP, &dbb, sizeof(dbb), __DFT ); /* číst další záznam */
}
```

Nekonečný cyklus by vypadal takto:

```
while (1) { /* ... */ }
```

Příkaz for

Příkaz *for* je příkaz cyklu, který se používá nejčastěji, když je třeba provádět určitý známý počet průchodů a zároveň používat "proměnnou cyklu". Má tento obecný tvar:

```
for ( počáteční-výraz; podmínka-opakování; opakovaný-výraz )  příkaz
```

Typické použití je asi toto:

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

```
   b -= i;
}
```

Nekonečný cyklus by vypadal např. takto:

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

Zde jsou všechny tři výrazy prázdné, tzn. že i příkazy jsou prázdné. Podmínka pro opakování cyklu je v tomto případě splněna, protože prázdný příkaz je zde "pravdivý", má totiž z definice hodnotu 1 (je stejný jako u while(1)).

Příkaz do

Příkaz do je příkaz cyklu a používá se, když je potřeba provést "tělo cyklu" alespoň jednou. Má tento obecný tvar:

```
do příkaz while (výraz);
```

Příklad:

```
do {
   a++;
   b--;
}
while (a <= 1000);
```

Nekonečný cyklus by vypadal takto:

```
do { /* ... */ }
while (1);
```

Příkaz skoku goto

Obecný tvar příkazu goto je:

```
goto návěští;
```

kde *návěští* je identifikátor (jméno). Identifikátor následovaný dvojtečkou se musí vyskytovat ještě jinde ve funkci:

```
návěští: příkaz
```

Za návěštím musí následovat příkaz, který může být i prázdný.

Příklad:

```
main() {
   ...

   if (F3) goto konec;
   ...

   konec:;
}
```

Příkazy break a continue

Jsou to vlastně příkazy pro přerušení sledu příkazů v těle cyklu a lze je nahradit příkazem goto. Dává se jim přednost před goto, protože nevyžadují návěští, a nadto je příkaz goto často zcela zavrhován.

Následující tabulka ukazuje, jak lze příkazy break a continue v nekonečných cyklech nahradit příkazem goto s návěštím.

for (;;) {
...
break;
...
}
	for (;;) {
	...
	goto ven;
	...
	}
	ven:;

while (1) {
...
continue;
...
}
	while (1) {
	...
	goto pokracovat;
	...
	pokracovat:;
	}

do { ... **continue;** ... } while (1);	do { ... goto pokracovat; ... pokracovat:; } while (1);

Poznámka: příkazy break a continue lze použít i v příkazu switch, který zde však neprobíráme.

Příkaz return

Příkaz *return* ukončí výpočet těla funkce a případně dosadí funkční hodnotu na místo volání funkce. Příkaz return má obecný tvar

`return` *`výraz`*`;`

kde výraz může být prázdný. Hodnota výrazu musí mít typ určený v definici funkce pro vracenou hodnotu. Kupříkladu v následujícím programu má vracená hodnota typ int.

```
#include <stdio.h>

int  soucet ( int a, int b ) {
  return a + b;
}

main() {
  int a;
  a = soucet ( 2, 3 );
  printf("Součet dvou čísel je %d \n", a );
}
```

Příkaz return nemusí být vůbec použit, nevrací-li funkce hodnotu a když výpočet funkce končí za posledním příkazem. Příkaz return bez výrazu lze použít tam, kde výpočet končí jinde než za posledním příkazem.

Pole (arrays)

Pole je český odborný výraz pro anglický *array*, ačkoliv pole je anglicky field. (To také někdy způsobuje různá nedorozumění.) Pole je série paměťových míst stejné délky a typu. Pole se deklarují s pomocí hranatých závorek, v nichž je uveden počet položek. Počet položek (prvků, složek) pole je pevně dán při kompilaci. Položky se číslují *od nuly*. První položka pole je tedy vlastně nultá.

Deklarace pole bez inicializace

```
int           pole1[10];        /* deset položek typu int */
char          pole2 [100];      /* sto položek typu char */
decimal(10,2) pole3 [12];       /* dvanáct položek typu decimal(10,2) */
```

Deklarace pole s inicializací seznamem

Pole *pole1* bude mít 10 položek s hodnotou 10:

```
int           pole1[10] = {10,10,10,10,10,10,10,10,10,10};
```

Pole *pole2* bude mít 5 položek s hodnotou 10 a 95 položek s hodnotou 0:

```
int           pole2[100] = {10,10,10,10,10};
```

Znakové řetězce (strings)

Znakový řetězec je pole složené z jednoznakových položek, z nichž poslední je nulová, tj. NULL, neboli 0x00 neboli '\0'. Řetězcová konstanta (literál) se vyjadřuje textem uzavřeným mezi dvojitými uvozovkami.

Deklarace pole s inicializací řetězcem

Pole text bude mít 5 položek | A | B | C |D | \0 |

```
char text [5] = "ABCD";
```

Pole *pole4* bude mít tolik položek, kolik je v řetězci znaků, *plus jedna*; v hranatých závorkách nemusí být počet položek uveden:

```
char          pole4 []  = "toto je řetězec, jehož jednotlivé znaky se \
stávají položkami pole";
```

Upozornění: 'x' a "x" neznamená totéž; první je jeden znak x, druhý je řetězec | x | \0 |.

Stanovení velikosti typu

```
sizeof (long);                /* počet bajtů v typu long */
sizeof (double);              /* počet bajtů v typu double */

sizeof (decimal(10,2));       /* počet bajtů v typu decimal(10,2) = 6 */
                              /* je to celá část čísla (10+2)/2 */
digitsof(decimal(10,2);       /* počet číslic = 10 */
precisionof(decimal(10,2);    /* počet desetinných míst = 2 */
```

Stanovení velikosti proměnné

```
#define  7   SEDM
int     delka_pole5, pocet_polozek_pole5;
int     pole5 [SEDM];

delka_pole5 = sizeof (pole5);  /* počet bajtů celého pole = 7*sizeof(int) = 28 */

pocet_polozek_pole5 = sizeof(pole5)/sizeof(pole5[0]);  /* počet položek pole = 7 */
```

Poznámka: *sizeof* není jméno funkce, ale operátoru. Proto jej lze použít i pro specifikaci hodnot konstant v době kompilace. Chceme-li například, aby počet položek *pole6* byl stejný jako u *pole5*, můžeme napsat deklaraci:

```
int  pole6 [sizeof(pole5)/sizeof(pole5[0])];
                                          /* délka pole5 dělená délkou položky */
```

Použití polí ve výrazech

Jednotlivé položky pole označujeme s pomocí hranatých závorek, stejně jako v deklaraci. Zatímco v deklaraci musí být konstanta nebo konstantní výraz, ve výrazu může být v hranatých závorkách i proměnná nebo výraz.

```
#include <decimal.h>                /* prototypy pro dekadické výpočty */

decimal(10,2)    soucet = 0;        /* deklarace s inicializací */
decimal(10,2)    array[12];         /* pole s dekadickými položkami */
int              i;

for (i = 0;                         /* počítá se od nuly do počtu položek */
     i <= sizeof(array)/sizeof(array[0]);  /* menší než počet položek */
     i++)
    soucet += array[i];             /* sečtení všech položek pole */
```

Ukazatele (pointers)

Ukazatel (*směrník, pointer*) je proměnná, která obsahuje adresy. Lze s ním provádět omezené výpočty: přičíst nebo odečíst celé číslo, spočítat rozdíl dvou ukazatelů, porovnat velikost ukazatelů. Každý *ukazatel* má stejný *typ* jako proměnná, na niž ukazuje. Ukazatel se označuje hvězdičkou v deklaraci. Obsahem ukazatele je adresa proměnné nebo "prázdná adresa" NULL (lze ji také kódovat jako '\0' nebo 0x00). Přičtením celého čísla k ukazateli se ukazatel zvýší o násobek čísla a délky typu ukazatele. U odečtení je pravidlo podobné. Přičteme-li tedy k ukazateli číslo 1, zvětší se ukazatel o jedničku jen tehdy, je-li typu char (protože typ char má délku 1). U typu int se přičtením jedničky ukazatel zvětší o 4 (délku typu int).

```
decimal(10,2) * pointer;       /* prázdný ukazatel na typ decimal(10,2) */
int    *intptr;
char   *p;
double *dp;
```

Inicializace ukazatele adresou proměnné:

```
int      i;
int      cislo = 3;
int     *ptr = &cislo;   /* ptr je ukazatel na proměnnou cislo */
```

Vyhodnocení ukazatele - *dereference* - **se označuje stejně jako deklarace** - předřazením hvězdičky před jméno ukazatele:

```
i = *ptr;               /* i bude obsahovat číslo 3 */
```

Ukazatele a pole

Pole (arrays) a ukazatele (pointers) jsou v jazyku C těsně propojeny.

```
pointer = array;              /* dosazení adresy začátku pole do ukazatele */
pointer = &array[0];          /* dosadí stejnou adresu */
```

Jméno pole (zde array) je ve skutečnosti konstantní ukazatel na začátek pole, není to proměnná a nelze proto měnit jeho hodnotu. K ukazateli lze přičítat nebo od něj odečítat celá čísla. Přičteme-li k ukazateli jedničku, zvětší se o délku jedné položky, tj. bude ukazovat na další položku pole. Dva ukazatele lze odečíst, jsou-li stejného typu. Ukazatel lze porovnávat s ukazatelem stejného typu (nebo s konstantou NULL).

```
for (pointer = &array[0];
     pointer < &array[0] + sizeof(array)/sizeof(array[0];
     pointer += 1)
    soucet += * pointer;       /* dereference – vyřešení nepřímého odkazu */
```

Výraz

```
pointer < &array[0] + sizeof(array)/sizeof(array[0]
```

znamená, že porovnáváme ukazatel s adresou, která vznikne, když k adrese první položky přičteme počet položek (12), čili s adresou ukazující těsně za poslední položku.

Upozornění 1: Ve výrazu nelze zapsat

pointer < &array

místo *pointer < &array[0]*, protože operandy nemají slučitelné typy: proměnná *pointer* je ukazatel, kdežto *&array* je adresa ukazatele, jméno *array* je totiž chápáno jako ukazatel na první položku pole *array*. Lze však zapsat

pointer < array,

protože oba operandy jsou ukazatele na stejný typ proměnné.

Upozornění 2: Zápis * *pointer* znamená dvě různé věci, podle kontextu. Je-li uveden v deklaraci, např.

```
decimal(10,2) * pointer;
```

jde o *deklaraci ukazatele*. Je-li uveden ve výrazu v příkazu, např.

```
soucet += * pointer;
```

jde o *dereferenci*, tj. vyhodnocení obsahu proměnné, na niž ukazuje ukazatel, tedy přičtení obsahu proměnné, na niž právě ukazuje pointer, k proměnné soucet.

Program PGMPTR1 ilustruje ukazatele a pole:

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

main()
{
  decimal(10,2)    soucet = 0;  /* deklarace s inicializací */
  decimal(10,2)    array[12];   /* pole s dekadickými položkami */
  int              i;
  int              elements;    /* počet položek pole = 12 */
  decimal(10,2)  * pointer;     /* prázdný ukazatel na typ decimal(10,2) */

  i=sizeof (array);
  printf ("Délka pole v bajtech = %d \n", i);

  i = sizeof (decimal(10,2));
  printf ("Délka decimal(10,2)  = %d \n", i);

  i = sizeof (array[0]);
  printf ("Délka položky pole   = %d \n", i);

  i = sizeof (array) / sizeof(array[0]);
  printf ("Počet položek pole   = %d \n", i);

  elements = sizeof (array) / sizeof(array[0]); /* počet položek pole */

  /* Cyklus naplní pole samými jedničkami */
  for (i = 0;                   /* počítá se od nuly do počtu položek - 1! */
       i < elements;                /* menší než počet položek */
       i++ )
      array[i] = 1;

  /* Cyklus sečte položky pole */
  for (pointer = &array[0];
       pointer < &array[0] + elements;
       pointer += 1)            /* přičte se délka položky */
      soucet += * pointer;      /* dereference - zrušení nepřímého odkazu */

  printf ("Součet = %D(10,2)", soucet);
}
```

Výsledkem spuštění programu je následující výpis na standardním výstupu:

```
Délka pole v bajtech = 72
Délka decimal(10,2)  = 6
Délka položky pole   = 6
Počet položek pole   = 12
Součet = 12.00
```

Stojí za povšimnutí, že tisk dekadického čísla se formátuje symbolem %D(d,p), kde d je celkový počet dekadických číslic a p je počet desetinných míst (ten může být i 0).

Vícerozměrná pole

V jazyku C se vícerozměrná pole deklarují jako "pole polí". Dvojrozměrné pole deklarujeme jako R řádků, z nichž každý má S sloupců. Jako příklad si uvedeme pole obsahující čtyři řádky a tři sloupce (matici). Deklarujeme ji jako pole o čtyřech položkách, z nichž každá je

pole o třech znakových položkách. Program PGMPTR2 ilustruje různé způsoby odkazů na znakové položky.

```
#include <stdio.h>
#define R 4        /* počet řádků */
#define S 3        /* počet sloupců */

main() {
  /* matice 4 řádky x 3 sloupce */
  char matrix [R] [S] = { 'A', 'B', 'C',
                          'D', 'E', 'F',
                          'G', 'H', 'I',
                          'J', 'K', 'L' };
  char a1, a2, a3, a4;
  int i, j;

  for (i = 0; i < R; i++) {
    for (j = 0; j < S; j++) {
      a1 = matrix [i] [j];          /* oba indexy */
      a2 = (*(matrix + i)) [j];     /* první ukazatel, druhý index */
      a3 = *(matrix [i] + j);       /* první index, druhý ukazatel */
      a4 = *(*(matrix + i) + j);    /* oba ukazatele */

      printf ("i = %d  j = %d \n\n" , i, j);
      printf ("a1 = %c \n",    a1);
      printf ("a2 = %c \n",    a2);
      printf ("a3 = %c \n",    a3);
      printf ("a4 = %c \n\n" , a4);
    }
  }
}
```

Zde je `matrix + i` ukazatel na začátek i-tého řádku pole. Pole `matrix` je totiž *polem ukazatelů*. i-tá položka tohoto pole

`matrix[i] = &matrix[0]+i = matrix + i`

obsahuje ukazatel na i-té pole druhé úrovně. Tento ukazatel získáme vyhodnocením - dereferencí: `*(matrix + i)`.

Struktury (structures)

Struktura se skládá z několika datových proměnných různého typu spojených dohromady pod jedním názvem. Struktury se hojně vyskytují v parametrech rozhraní API (volání systémových funkcí). Příkladem může být struktura údajů, které vrací API QUSRUSAT (Retrieve User Space Attributes):

```
struct         recv  {        /* receiver variable */
   int         bytesret;      /* bytes returned */
   int         bytesavl;      /* bytes available */
   int         spacesize;     /* space size */
   char        autext;        /* automatic extensibility */
   char        initvalue;     /* initial value */
   char        libname[10];   /* user space library name */
};
```

Uvnitř složených závorek jsou deklarace jednotlivých položek (členů, polí, složek - members, fields). Toto je *deklarace struktury*, nikoliv její definice. Nepřiděluje paměť, jen ohlašuje, že proměnná typu struct recv se bude používat, jestliže bude dále definována. Všimněme si středníku za složenou závorkou na konci. Proměnnou pro strukturu můžeme definovat např. takto:

```
struct recv     receiver;
```

Místo jména receiver by klidně mohlo být i recv (stejné jméno jako má typ struktury). Obě deklarace můžeme spojit do jedné tak, že před koncový středník vepíšeme jméno proměnné:

```
struct         recv  {        /* receiver variable */
   int         bytesret;      /* bytes returned */
   int         bytesavl;      /* bytes available */
   int         spacesize;     /* space size */
   char        autext;        /* automatic extensibility */
   char        initvalue;     /* initial value */
   char        libname[10];   /* user space library name */
} receiver;
```

Na jednotlivé položky (členy, složky, pole, members, fields) struktury se odvoláváme jménem proměnné s operátorem tečka, např:

```
receiver.spacesize
receiver.libname[i]
```

Kdyby však struktura nebyla definována přímo v programu, ale data struktury by vytvořil jiný program nebo operační systém a předal je jako výsledek volání funkce (např. API) prostřednictvím ukazatele, pak se na položky odvoláváme operátorem šipka (->). Tato situace se vyskytuje velmi často v systémových programech. Například struktura hostent (host entry) se používá ve funkci (API) gethostbyaddr (get host by address), která podle IP adresy získá údaje o odpovídajícím počítači (host), mj. jeho jméno. Struktura je deklarována takto:

```
struct hostent {
  char         *h_name;              /* host name          */
  char        **h_aliases;           /* NULL-terminated list of
                                        host aliases               */
  int           h_addrtype;          /* address family of address   */
  int           h_length;            /* length of each address in
                                        h_addr_list                */
  char        **h_addr_list;         /* NULL-terminated list of
                                        host addresses             */
};
```

Dvě hvězdičky znamenají ukazatel na ukazatel. Abychom se strukturou mohli pracovat, potřebujeme pro ni proměnnou. Definujeme proto proměnnou hostp, což je ukazatel typu struct hostent (všimněme si hvězdičky před hostp):

```
struct hostent *hostp;   /* Pointer to a host entry */
```

Např. na položku h_name struktury hostp se odvoláme výrazem s šipkou

```
hostp->h_name
```

nebo s vyhodnoceným ukazatelem (s předřazenou hvězdičkou) v závorce a tečkou

```
(*hostp).h_name
```

Bez ukazatele se zde nemůžeme obejít, protože data jsou mimo náš program. Pojítko na ně máme jen přes ukazatel, který získáme jako hodnotu vrácenou po vyvolání funkce

```
hostp = gethostbyaddr(&in_addr, sizeof(in_addr), AF_INET);
```

Podrobnosti o funkci gethostbyaddr() budeme probírat později, zde nám stačí, že vrací ukazatel předepsaného typu.

Přejmenování typů (typedef)

Jazyk C umožňuje přejmenovat existující typy. Hlavní smysl tohoto prostředku je ten, že zlepšuje přenosnost C programů mezi různými operačními systémy. Například následující *definice typu **long** pod novým jménem* time dovoluje všechny deklarace proměnných označujících čas deklarovat typem ***time***.

```
typedef  long  time;            /* definice typu time jako long */
time t1;                        /* deklarace proměnné t1 typu time (= long) */
```

Kdyby se však ukázalo, že v jiném operačním systému jsou časové údaje typu int (s kratší délkou než long), stačilo by v programu změnit jen definici typu time:

```
typedef  int   time;            /* definice typu time jako int */
```

Přejmenovaný typ může být dále přejmenován, např.:

```
typedef  time  daytime;
```

Přejmenování typu struktury

```
typedef struct recv {      /* receiver variable */
   int         bytesret;      /* bytes returned */
   int         bytesavl;      /* bytes available */
   int         spacesize;     /* space size */
   char        autext;        /* automatic extensibility */
   char        initvalue;     /* initial value */
   char        libname[10];   /* user space library name */
}  RECEIVER;
```

Nový datový typ se jmenuje RECEIVER a není novou definicí struktury. Je jen novým jménem pro typ struct recv definovaný uvnitř (mezi slovem typedef a slovem RECEIVER. Všimněme si, že mezi pravou složenou závorkou a slovem RECEIVER není středník. Ten je až na konci příkazu typedef, který ukončuje. Definice proměnné pro strukturu by mohla být např.

```
RECEIVER data_returned;
```

Přejmenování typu ukazatele

```
typedef int * ptr_int;       /* hvězdička může být bez mezer */
```

Zde je typ ukazatele int * přejmenován na ptr_int. Můžeme tedy deklarovat ukazatel typu int např. takto:

```
ptr_int  p1;               /* p1 je proměnná typu ptr_int = ukazatel na int */
```

Změna typu ukazatele

Přejmenování struktur pomocí příkazu typedef se hojně používá zejména proto, abychom nemuseli při každé definici proměnné typu struktura psát

```
struct recv data_returned;
```

nebo i při jejím použití, třeba při změně typu ukazatele:

```
char     *pointer;        /* ukazatel typu char */
RECEIVER *ptr_str;        /* ukazatel typu struct recv neboli RECEIVER */

pointer = (char*)&data_returned;  /* změna typu ukazatele na typ char* */
ptr_str = (RECEIVER*)pointer + 1; /* změna typu ukazatele na typ RECEIVER* */
```

kde výraz se závorkami (char*) a (RECEIVER*) znamenají *přetypování* (casting) ukazatele.

Poznámka: Příkaz

```
   pointer = (char*)&data_returned;
```

nelze zjednodušit na

```
   pointer = &data_returned;
```

protože ukazatele mají různé typy, ani nelze obrátit pořadí znaku & a výrazu (char*)

```
   pointer = &(char*)data_returned;
```

protože struktuře nejde dát typ char, ani jiný typ. V obou těchto případech kompilátor hlásí chybu.

Unie (union)

Často je potřeba definovat několik proměnných různých typů dat, které leží ve stejné paměti, tj. překrývají se. K tomu lze použít unie bez jména typu (mezi slovem union a levou složenou závorkou není žádné jméno):

```
union {                   /* union of 2 byte binary and character values */
   unsigned short  reladr;
   char            reladrchar[2];
} relad;                  /* to manipulate relative address in subfile */
```

V tomto příkladu se definuje unie a zároveň její proměnná *relad*. Zde ve stejné paměti leží krátké celé číslo bez znaménka *relad.reladr* a dvoupoložkové znakové pole *relad.reladrchar*. Každý z obou údajů zabírá dva bajty. Smyslem takové definice je schopnost použít stejnou hodnotu v různých příkazech jako různé typy, někdy jako číslo, jindy jako dva znaky.
K proměnné reladr můžeme např. přičíst číslo 1, proměnnou reladrchar můžeme používat jako jednotlivé bajty, třeba pro jejich vyjádření ve znakové hexadecimální podobě (např. číslo 0x1A převést na dva znaky '1' a 'A' na obrazovce).

V unii mohou být sjednoceny nejen jednoduché proměnné nebo pole, ale také (a hlavně) struktury. Následuje složitější ukázka, která je zato převzata ze skutečnosti.

Struktura typu in_addr definuje paměť jedné (nejjednodušší) složky IP adresy, čtyřbajtové celé číslo bez znaménka. Některé API pro obsluhu soketů vyžadují tuto definici IP adresy.

```
typedef unsigned long             u_long;

struct in_addr {                      /* internet address */
   u_long           s_addr;           /* IP address */
};
```

Typ u_long je zde definován zdánlivě zbytečně, může však sloužit pro snadnější přenos programů do jiného prostředí, v němž typ long má jinou délku než 4 bajty. Pak stačí změnit jen definici typu u_long (např. na unsigned int).

Struktura typu sockaddr definuje IP adresu používanou v jiných API pro sokety:

```
typedef unsigned short           u_short;

struct sockaddr {                       /* generic socket address      */
    u_short       sa_family;            /* address family              */
    char          sa_data[14];          /* address                     */
};
```

Struktura typu sockaddr_in definuje IP adresu ještě jiným způsobem ve stejně dlouhé paměti, ale v jiném členění:

```
struct sockaddr_in {                    /* socket address (internet)   */
   short           sin_family;          /* address family (AF_INET)    */
   u_short         sin_port;            /* port number                 */
   struct in_addr  sin_addr;            /* IP address                  */
   char            sin_zero[8];         /* reserved - must be 0x00's   */
};
```

Každý z obou typů `sockaddr` a `sockaddr_in` se používá v jiném kontextu, ale musí zabírat stejnou paměť , protože jde stále o tutéž IP adresu. Všimněme si, že ve struktuře `sockaddr_in` je jedním členem opět struktura (typu `in_addr`). Různá API vyžadují jednou typ `sockaddr`, jindy `sockaddr_in`. Členy těchto struktur lze používat s operátorem tečka. O definici a sjednocení paměti se postará unie typu `socket_addr`, jejíž jméno proměnné je `usckaddr`:

```
union socket_addr {   /* Union of two structures */
  struct sockaddr
      IPaddress;       /* socket address info structure (simpler) */
  struct sockaddr_in
      IPaddr;          /* socket IP address info structure (detailed) */
} usckaddr;
```

Struktura `IPaddress` a struktura `IPaddr` zabírají stejné místo v paměti. Na jednotlivé položky těchto struktur se odvoláváme operátorem tečka, např. takto:

```
usckaddr.IPaddress.sa_family    = 0;
usckaddr.IPaddr.sin_port = 0;
usckaddr.IPaddr.sin_addr.s_addr = 0;
```

Kdybychom unii přidělili *ukazatel*, např. `uptr`, naplnili jej adresou proměnné usckaddr, mohli bychom ji používat i jiným způsobem. Odkazy na členy (položky) unie by musely obsahovat ukazatel s operátorem šipka:

```
union socket_addr * uptr;   /* deklarace ukazatele typu socket_addr (unie) */
...

uptr = &usckaddr;           /* naplnění ukazatele adresou proměnné unie */

uptr->IPaddress.sa_family    = 0;
uptr->IPaddr.sin_port        = 0;
uptr->IPaddr.sin_addr.s_addr = 0;
```

Právě tak, jako struktura může být členem unie, může být unie členem struktury.

Příklad na ukazatele a struktury

Program GETBYADR získává a tiskne údaje z "host table" nebo DNS pro IP adresu zadanou v tečkové formě (např. 127.0.0.1):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <arpa/nameser.h>
#include <resolv.h>

/*********************************************************************/
/*   System structures                                               */
/*********************************************************************/

struct in_addr in_addr;          /* IP address - long unsigned */        (1)
/***
struct hostent {                 // host entry. Must be 16 byte aligned
  char         *h_name;              // host name                        (2)
  char        **h_aliases;           // NULL-terminated list of          (3)
                                     // host aliases
  int           h_addrtype;          // address family of address
  int           h_length;            // length of each address in
                                     // h_addr_list
  char        **h_addr_list;         // NULL-terminated list of
                                     // host addresses
};
***/
/*********************************************************************/
/*   Function prototypes                                             */
/*********************************************************************/

// struct hostent  * gethostbyaddr(struct in_addr *, int, int);          (5)

/*********************************************************************/
/*   Main function                                                   */
/*********************************************************************/

void main(void)
{
  struct hostent *hostp;   /* Pointer to a host entry */
  char   servername[256];
  long   unsigned
         binaddr;          /* Binary IP address */
  int    i;

/*   Get host by address - from 127.0.0.1 to LOOPBACK               */

  binaddr = inet_addr("127.0.0.1");   /* convert dotted to binary */     (6)
  memcpy(&in_addr.s_addr, &binaddr, sizeof(binaddr));                   (7)
                                     /* put it to in_addr structure */

  hostp = gethostbyaddr(&in_addr, sizeof(in_addr), AF_INET);            (8)

  if (!hostp)
  {
    printf("Error number = %d\n", h_errno );                            (9)
    printf("Text for h_errno = %s\n", hstrerror(h_errno));

    return;
  }
```

```
/*   Print address type and length of binary IP address             */
  printf(" Address family  = %d \n", hostp->h_addrtype);
  printf(" Address length  = %d \n", hostp->h_length  );

/*   Print host name                                                */
  printf(" Host name       = %s \n", hostp->h_name);                    (10)

/*   Retrieve and print aliases                                     */
  for (i = 0; *((hostp->h_aliases) + i) != NULL; i++)                  (11)
  {
    printf(" Alias host name = %s \n", *((hostp->h_aliases) + i));
  }

}
```

Vysvětlivky k programu GETBYADR

(1) Jméno `in_addr` představuje strukturu (deklarovaou v hlavičkovém souboru *<netinet/in.h>*).

```
typedef unsigned long           u_long;
struct in_addr {                      /* internet address           */
   u_long          s_addr;            /* IP address                 */
};
```

Abychom mohli pracovat se strukturou `in_addr`, musíme si v programu definovat proměnnou, např. stejného jména:

```
struct in_addr in_addr;          /* IP address - long unsigned */
```

Prototyp funkce *`gethostbyaddr`* je obsažen v hlavičkovém souboru `<netdb.h>`, zde je uveden jen jako komentář, abychom věděli, jak funkci volat.

```
struct hostent  * gethostbyaddr(struct in_addr *, int, int);
```

Hvězdička před jménem funkce znamená, že vracená hodnota je ukazatel typu `struct hostent`. První argument je ukazatel typu `struct in_addr`, další dva jsou celá čísla. Podrobné informace o tom, co argumenty znamenají, lze najít na stránce *https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/apis/ghosta.htm*.

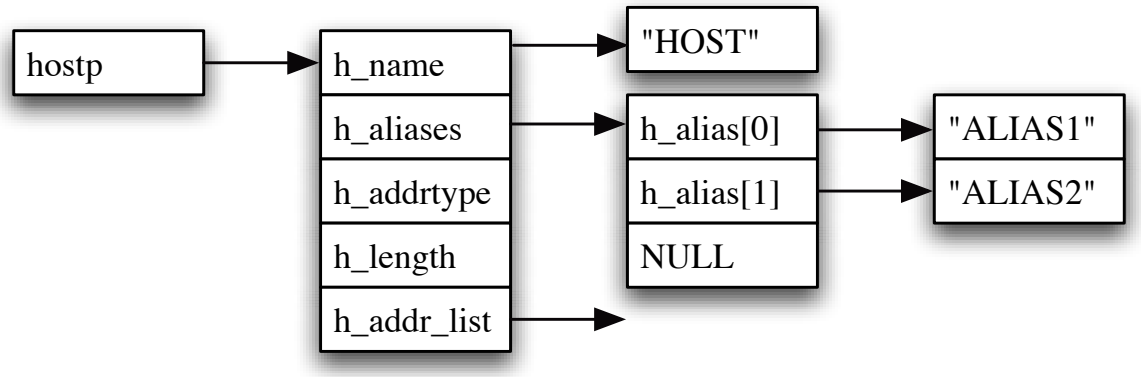
(2) Např. položku *`h_name`* struktury *`hostp`* získáme výrazem s šipkou (operátorem šipka)

```
hostp->h_name
```

Tato položka je ukazatel *`h_name`*, který ukazuje na jméno počítače. Jméno počítače je vlastně pole znaků o nespecifikovaném počtu prvků, ale je ukončené znakem NULL, tedy 0x00 neboli '\0'. Jde tedy o tzv. "null terminated string" neboli znakový řetězec ukončený nulovým znakem, viz též **(10)**:

```
printf(" Host name = %s \n", hostp->h_name);
```

(3) Symbol *`**h_aliases`* znamená totéž jako *`*(*h_aliases)`*, tedy "ukazatel na ukazatel". Z dokumentace plyne, že alternativní jména počítače (alias jména, přezdívky) jsou k dispozici ve znakových řetězcích, na něž ukazují adresy obsažené v poli ukazatelů, jak ukazuje následující obrázek.



Ukazatel *h_aliases* tedy obsahuje adresu prvního ukazatele z pole ukazatelů. Ten se dá interpretovat jako ukazatel na pole znaků (v obrázku *h_alias[0]*), který obsahuje adresu prvního znaku řetězce (v obrázku "ALIAS1"). Každý řetězec je ukončen nulovým znakem (NULL = '\0' = 0x00). Pole ukazatelů je také ukončeno nulovým ukazatelem NULL (který se sice kóduje stejně jako NULL u řetězců, ale má jiný význam).

(5) Deklarace (prototyp) funkce `gethostbyadr()` je zde uvedena jen jako komentář, protože je obsažena v hlavičkovém souboru. Je potřebná k použití funkce v programu, viz také (9).

(6) Funkce *inet_addr* dosadí do proměnné binaddr binární podobu IP adresy vyjádřené "tečkovou" formou v argumentu.

(7) Funkce *memcpy* je hojně používána pro kopírování paměti bez ohledu na typ jejího obsahu. Je proto založena na použití ukazatelů (libovolného typu). Oba první argumenty "kam" a "odkud" jsou proto ukazatele (v našem případě vyjádřené přímo jako adresy). Cílový operand je obsažen ve struktuře *in_addr* definované přímo v programu, používá se tedy operátor tečka. Místo příkazu *memcpy* lze ovšem použít přiřazovací příkaz

```
in_addr.s_addr = inet_addr("127.0.0.1");
```

(8) Přiřazovací příkaz používá funkci *gethostbyaddr*.

```
hostp = gethostbyaddr(&in_addr, sizeof(in_addr), AF_INET);
```

Ta vrací jako hodnotu ukazatel na strukturu, kam dosadil výsledky. Jméno *AF_INET* představuje konstantu 2 definovanou v hlavičkovém souboru *<unistd.h>* (obsaženém také v hlavičkovém souboru *<sys.socket.h>*). Všechny definice konstant lze vidět v protokolu o kompilaci, uvedeme-li argumenty OPTION(*SHOWINC) a OUTPUT(*PRINT). Jestliže ve funkci *gethostbyaddr* dojde k chybě, vrátí nulový ukazatel NULL, což lze zjistit v příkazu *if* takto:

```
if (!hostp) ...
```

nebo s přetypováním ukazatele NULL

```
if (hostp == (struct hostent *)NULL) ...
```

(9) Číslo chyby můžeme zjistit ze systémové proměnné *h_errno*. Chceme-li ještě zjistit text chyby, použijeme funkci *hstrerror(h_errno)*.

Další příkaz vytiskne

```
Error number = 5
```

Význam chybových kódů je patrný z následujících definic (hlavičkového souboru *<netdb.h>*).

```
#define HOST_NOT_FOUND   5           /* host not found              */
#define NO_DATA          10          /* no data of requested type   */
#define NO_ADDRESS       NO_DATA     /* name valid, but no address  */
```

```
#define NO_RECOVERY      15           /* unrecoverable error         */
#define TRY_AGAIN        20           /* try again                   */
```

(10) Výraz *hostp->h_name* odkazuje na položku *h_name* struktury *hostp*. Identifikátor *h_name* je ukazatel na řetězec obsahující jméno počítače.

```
printf(" Host name       = %s \n", hostp->h_name);
```

(11) Příkaz cyklu

```
for (i = 0; *((hostp->h_aliases) + i) != NULL; i++)
{
  printf(" Alias host name = %s \n", *((hostp->h_aliases) + i));
}
```

zkoumá a tiskne alternativní jména počítače. Proměnná cyklu i se zvyšuje od nuly po jedničce; je potřebná k získání dalšího ukazatele v poli ukazatelů. Ukazatel *hostp* obsahuje adresu struktury *hostp*, ukazatel *h_aliases* obsahuje adresu pole ukazatelů.

Výraz **((hostp->h_aliases) + i)* se vyhodnotí takto:

1) Na adrese *hostp->h_aliases* se získá ukazatel *h_aliases*, který obsahuje adresu pole ukazatelů.
2) K takto získané adrese se přičte *i* krát velikost ukazatele (*i * 16*), po prvé nula. Na takto získané adrese je umístěn *i-tý ukazatel* v poli ukazatelů, který obsahuje adresu řetězce nebo NULL.
3) Adresa řetězce se získá operátorem hvězdička.

Takto vyhodnocená adresa se porovná s konstantou NULL a nerovná-li se jí, vytiskne se řetězec nalezený na této adrese (která se v argumentu funkce *printf* znovu vyhodnocuje). Rovná-li se adresa konstantě NULL, cyklus se ukončí.

Tento zápis cyklu je sice krátký, ale na první pohled poněkud nesrozumitelný. Delší, ale srozumitelnější zápis by mohl vypadat takto:

```
char  **ptrptr;                    /* průběžný ukazatel na ukazatel na char */
char   *ptr;                       /* průběžný ukazatel na char */
...
ptrptr = hostp->h_aliases;  /* nastavení ukazatele na první ukazatel v poli */
while (1) {
  ptr = *ptrptr;                          /* ukazatel na řetězec */
  if (ptr == NULL) break;
  printf(" Alias host name = %s \n", ptr);
  ptrptr += sizeof(hostp->h_aliases);     /* přičtení velikosti ukazatele */
}
```

Funkce

Funkce v jazyku C jsou totéž, co se v jiných jazycích jmenuje podprogramy, procedury, podprocedury, subrutiny apod. Program v jazyku C musí obsahovat hlavní funkci zvanou *main*. Každá funkce musí být definována.

Obecný tvar definice funkce

```
paměťová-třída  vracený-typ  jméno-funkce  ( seznam-argumentů )
nepovinná-deklarace-argumentů
{ tělo-funkce }
```

Paměťová třída je nepovinná a může být **extern** nebo **static**. Třída extern se rozumí i bez zápisu (je předvolená) a dovoluje volat funkci zvenčí (z jiných modulů). Třída static se používá, když funkce nemá být viditelná zvenčí.

Vracený typ je nepovinný (když se nezadá, je **`int`**) a může být libovolný kromě pole (array). Může to však být ukazatel na pole. Nemá-li funkce vracet žádnou hodnotu, měl by být zadán typ **`void`**.

Seznam argumentů může být

- prázdný nebo `void` (což je totéž),
- prototypový, tvořený pouhými specifikacemi typů oddělenými čárkou, např. `(char*, int, double)`,
- prototypový, tvořený dvojicemi – typu a jména proměnné – oddělenými čárkou, např. `(char* a, int b, double c)`,
- neprototypový, tvořený jmény proměnných oddělenými čárkou, např. `(a, b, c)`, argumenty bez typu mají typ `int`.

Deklarace argumentů je nutná jen pro neprototypový seznam argumentů a určuje typy jednotlivých argumentů, např.

```
char* a; double c;.
```

Argumenty typu `int` se zde nemusí deklarovat.

Tělo funkce se skládá z deklarací lokálních proměnných a z příkazů, nebo může být prázdné.

Poznámka: V definici jazyka C se používá pojem *argument*. Formální argument se vyskytuje v deklaraci nebo definici funkce, aktuální argument ve volání funkce. V různých jiných souvislostech se používá místo pojmu argument pojem *parametr* ve stejném významu. Oba pojmy lze libovolně zaměnit.

Příklady definice funkce

```
int max (int a, int b)   /* prototypová definice funkce */
{
   if ( a > b )
      return (a);
   else
      return (b);
}
...

max (3, 6);              /* volání funkce */
```

Slovo max je *jméno* funkce, v kulatých závorkách je definice *formálních argumentů* (parametrů). Ve složených závorkách je *tělo* funkce. Před jménem funkce je `int` – *typ vracené hodnoty*.

Argumenty můžeme deklarovat také alternativně za kulatými závorkami:

```
int max (a, b)           /* neprototypová definice funkce */
long a;                  /* b je typu int */
{
   if ( a > b )
      return (a);
   else
      return (b);
}
...
max (3L, 6);             /* volání funkce */
```

Deklarace funkce

Deklarace funkce s prototypy argumentů musí být uvedena před voláním funkce, je-li její definice až za voláním:

```
int max (long, int)       /* prototypová deklarace funkce */
```

```
/* int max (long aa, int bb)  alternativa – libovolná jména aa a bb se ignorují */
...
max (3L, 6);             /* volání funkce */
...
int max (long a, int b)  /* prototypová definice funkce */
{
   if ( a > b )
      return (a);
   else
      return (b);
}
```

Deklarace funkce je také nutná pro kontrolu typů argumentů a vracené hodnoty, je-li funkce definována vně programu (v jiném modulu).

Ukazatel na funkci

Někdy je vhodné předávat ukazatel na funkci jako argument při volání jiné funkce, protože jméno funkce samotné nelze jako argument předávat. Často se uvádí třídicí funkce (sort), která v jisté fázi třídicího algoritmu vyvolá funkci (např. compare) poskytnutou uživatelem. Funkce *compare* provede porovnání dvou hodnot a vrátí třídicí funkci informaci o tom, která hodnota byla menší. Funkce *sort* pak pokračuje v třídění. Volání funkce compare z funkce sort se někdy nazývá zpětné volání (*callback*), protože uživatel volá funkci sort a ta volá zpět uživatele prostřednictvím funkce compare.

Deklarace ukazatele na funkci má tvar

typ (* *jméno-ukazatele*) ()

např.

```
decimal(5,2) (*funcptr)()
```

kde hvězdička znamená, že jde o ukazatel a prázdné závorky, že jde o funkci. Přitom hvězdička se jménem ukazatele musí být uzavřena v závorkách, protože zápis

```
decimal(5,2) *funcptr ()
```

by znamenal deklaraci funkce bez argumentů vracející ukazatel typu *decimal(5,2)*.

Volání funkce podle ukazatele má tvar

(* *jméno-ukazatele*) (*seznam-aktuálních-argumentů*)

např.

```
(*funcptr)(a, b).
```

Příklad ukazatele na funkci

Program MINIMUM určuje minimální číslo z pole dekadických čísel a vytiskne ho.

Funkce *compare* porovná dvě dekadická čísla, na něž dostane ukazatele v argumentech. Vrátí celé číslo 0, když první argument není větší než druhý, jinak vrátí číslo 1.

Funkce *min* vrátí nejmenší číslo z pole dekadických čísel, k porovnávání použije funkci *compare*. Jako první argument dostává ukazatel na pole a jako druhý argument *ukazatel na funkci* vracející hodnotu typu *int* (což bude ukazatel na funkci *compare*).

Funkce *main* jen vyvolá funkci *min* s aktuálními argumenty.

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

#define  N  5

/* Deklarace funkcí - prototypy */
```

```
/* ---------------------------- */
decimal(15,0)  min     ( decimal(15,0)[], int(*)() );                    (1)
int            compare ( decimal(15,0)*, decimal(15,0)* );               (2)

/* Hlavní program */
/* -------------- */
main() {
  decimal (15,0) array [] = { 5, 4, 3, 2, 1 };
  decimal (15,0) a;
  a = min(array, compare);                                               (3)
  printf("Minimum ze všech čísel je %D(15,0) \n", a );
}

/* Definice funkce min */
/* ------------------- */
decimal(15,0)  min( decimal(15,0) array[N], int (*funct)() ) {           (4)
  decimal(15,0) a;
  int           i;
  a = array[0];
  for (i = 1; i < N; i++) {
     if ( (*funct)(&a, &array[i]) == 1 ) /* vede porovnání k výměně? */     (5)
        a = array[i];                    /* ano, nahradit i-tou položkou */
     printf ("mezivýsledek %d je %D(15,0) \n", i, a );
  }
  return a;                          /* vrátí se nejmenší položka */
}

/* Definice funkce compare */
/* ----------------------- */
int            compare ( decimal(15,0)* a, decimal(15,0)* b ) {
  if (*a <= *b) return  0;
  else          return  1;
}
```

Ukazatel na funkci můžeme zadat také s operátorem adresy, jestliže definice funkce je k dispozici ve stejném modulu:

```
min(array, &compare);
```

Vysvětlivky k programu MINIMUM

(1) Prototyp funkce *min* obsahuje na místech paramtetrů jen označení typů. První argument

```
decimal(15,0)[]
```

je pole dekadických čísel bez udání počtu položek. Všimněme si, že v prototypu nemusíme zadávat počet položek pole. Druhý argument

```
int(*)()
```

je ukazatel na funkci vracející typ *int* s libovolnými argumenty. Hvězdička označující, že jde o ukazatel, musí být v závorkách. Kdybychom vynechali závorky, znamenal by argument funkci vracející ukazatel na typ *int*. Funkci však nelze zadat jako argument, lze zadat jen ukazatel na funkci. Všimněme si, že v seznamu argumentů chybějí jména proměnných, jde tedy o prototypové deklarace. Jména proměnných bychom sice mohli zadat také, ale nebrala by se v úvahu.

(2) Funkce *compare* bude použita jako argument volání funkce *min*, ovšemže prostřednictvím ukazatele. Zde je opět prototypová deklarace. Prototypové deklarace jsou nutné, protože definice funkcí jsou umístěny v programu až za jejich voláním.

(3) V hlavním programu (funkci *main*) se volá funkce *min*, která dostává aktuální argumenty, tedy jméno pole *array*, tedy ukazatel na jeho začátek, a jméno funkce *compare*, které zde představuje *ukazatel* na funkci *compare*.

(4) V definici funkce *min* jsou argumenty již pojmenovány. Druhý argument je *ukazatel* *funct*, který ukazuje na funkci (kterou, to se určí až při vyvolání funkce *min*).

(5) Funkce určená druhým argumentem se volá pomocí dereference, tedy s předřazenou hvězdičkou. Hvězdička se jménem ukazatele *funct* musí být v závorkách, protože jinak by to znamenalo, že funkce *funct* vrací ukazatel a my ho chceme vyhodnotit dereferencí.

Argumenty funkce main

Každý program musí obsahovat definici funkce *main*. Funkce *main* může mít dva argumenty (nebo žádný argument) a může (nemusí) vracet hodnotu. Zatímco vracenou hodnotu v IBM i stěží využijeme (po návratu z programu vyvolaného příkazem CALL), argumenty využijeme častěji. Nejširší tvar funkce *main* je tento:

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

Vracená hodnota je typu *int* (mohla by být i jiného typu), argumenty jsou dva. První argument *argc* je pojmenován konvenčně podle výrazu "argument count" – počet argumentů a představuje počet složek (hodnot) druhéno argumentu. Druhý argument *argv* je pojmenován rovněž konvenčně, podle výrazu "argument values" – hodnoty argumentů. Pojmy argument a parametr lze klidně zaměnit.

Této sestavě argumentů se říká "command line arguments" neboli argumenty příkazového řádku. Název je odvozen od příkazového řádku systémů UNIX, kde se program volá jménem a s argumenty oddělenými čárkou. V IBM i je volání obdobné, až na jméno příkazu CALL, které je zde navíc, a argumenty v závorkách bez čárek.

Řetězcové argumenty bez konverze

Program PGMPAR vyžaduje předání vstupních argumentů:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   while (--argc >= 0)
   printf("%s ", argv[argc]);
}
```

Program spustíme CL příkazem CALL se dvěma parametry:

```
CALL PGM(PGMPAR) PARM('řetězec1' 'řetězec2')
```

a dostaneme výstup

```
řetězec2 řetězec1 knihovna/PGMPAR
```

tedy argumenty v obráceném pořadí.

Operační systém spočítá argumenty a zjistí, že jsou tři (jméno programu a dva další argumenty `'řetězec1'` a `'řetězec2'`), dosadí tedy do proměnné `argc` číslo 3. Znakové řetězce jsou ukončené nulovým znakem 0x00 = NULL, který systém sám dosadí. Argument `argv` představuje pole (array) ukazatelů (směrníků) na znakové řetězce. V našem případě má pole čtyři prvky, tedy čtyři ukazatele, které obsahují adresy začátku každého řetězce:

Proměnná	**Hodnota**
argc	3
argv[0]	ukazatel na řetězec "knihovna/PGMPAR"
argv[1]	ukazatel na řetězec "řetězec1"
argv[2]	ukazatel na řetězec "řetězec2"

argv[3]	ukazatel NULL

Stojí za pozornost, že poslední prvek (zde čtvrtý) obsahuje vždy prázdný ukazatel NULL.

Poněkud jiná situace je při volání programu PGMPAR z jiného programu, např z CL programu. Příkaz CALL sám nedosadí ukončovací nulový znak na konec řetězců. O to se musí postarat programátor. CL program PGMPARCL, který vyvolá program PGMPAR, vypadá takto:

```
DCL &arg1 *char 100
DCL &arg2 *char 100
DCL &NULL *char 1  X'00'   /* NULL = 0x00 = '\0' v jazyku C */

CHGVAR  &ARG1  VALUE('řetězec1' *CAT &NULL)
CHGVAR  &ARG2  VALUE('řetězec2' *CAT &NULL)

call pgmpar (&arg1 &arg2)
```

Výsledek je stejný jako předtím.

Konverze typů argumentů

Velmi často se však vyskytne předávání argumentů, které nejsou znakovými řetězci. V IBM i to budou nejčastěji dekadická čísla, ale i celá čísla (int, short) aj.

Program PGMPAR2 vyžaduje předání sedmi argumentů (`argc` se nepočítá):

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

int main(int argc, char *argv[])
{
/*   Musíme vědět, jaké typy mají jednotlivé parmetry */
  printf ("Počet argumentů = %d\n",                          argc   );
  printf ("Jméno programu  = %s\n",                          argv[0]);
  printf ("decimal(15,5)   = %D(15,5)\n", *(decimal(15,5)  *) argv[1]);
  printf ("short int       = %d\n",       *(short          *) argv[2]);
  printf ("int             = %d\n",       *(int            *) argv[3]);
  printf ("string21        = %s\n",                          argv[4]);
  printf ("decimal(5,0)    = %D(5,0)\n",  *(decimal(5,0)   *) argv[5]);
  printf ("decimal(10,0)   = %D(10,0)\n", *(decimal(10,0)  *) argv[6]);
}
```

Všimněme si, že musíme *přetypovat ukazatele*, chceme-li z argumentů získat správné hodnoty (hodnoty přetypovat nemůžeme) a ještě podle potřeby ukazatele vyhodnotit (dereferencí). Hvězdička uvnitř závorek značí "je to ukazatel". Hvězdička před závorkami značí "vezmi hodnotu" dereferencí. Bez přetypování ukazatelů by se argumenty chápaly jako řetězce ukončené nulovým znakem, což je správné jen u jednoho.

CL program, PGMPAR2CL, který vyvolá program PGMPAR2, vypadá takto:

```
dcl &dec155   *dec (15 5)  1111111111.11111

dcl &char20   *char 20    '44444444444444444444'
dcl &string21 *char 21
dcl &NULL     *char 1    X'00'
dcl &short    *char 2    X'0003'
dcl &int      *char 4    X'00000004'
dcl &dec5     *dec  5      00005
dcl &dec10    *dec  10     0000000006

chgvar &string21 (&char20 *TCAT &NULL)  /* přidání nulového znaku */
call pgmpar2 (&dec155 &short &int &string21 &dec5 &dec10)
```

Výsledek spuštění CL programu je tento:

```
Počet argumentů = 7
Jméno programu  = knihovna/PGMPAR2
decimal(15,5)   = 1111111111.11111
short int       = 3
int             = 4
string21        = 4444444444444444444
decimal(5,0)    = 5
decimal(10,0)   = 6
```

Přehled ukazatelů a odvozených typů

Použití odvozených typů napodobuje jejich deklaraci, což může, ale nemusí přispívat čitelnosti programu.

```
int arc[10];       /* deklarace pole */
i = arc[2];        /* použití položky pole */
int* ptr;          /* deklarace ukazatele */
i = *ptr;          /* použití hodnoty, na niž ukazatel ukazuje – dereference */
```

Potíže při komponování vznikají tím, že operátor * je prefixový, kdežto operátory [] a () jsou postfixové. Proto je zapotřebí v určitých případech použít závorky kvůli prioritám operátorů. Operátor * má nižší prioritu než operátory [] a (). Je tedy

```
char* arc[10];   /* pole ukazatelů */
```

ale

```
char (*arc)[10]; /* ukazatel na pole */
```

Další příklady:

```
int*     i;             /* i je ukazatel na celé číslo */
int     *i;

int**    j;             /* j je ukazatel na ukazatel na celé číslo */
int    **j;

char*    a[10];         /* a je pole 10 ukazatelů na znaky */
char    *a[10];

char   (*b)[10];        /* b je ukazatel na pole 10 znaků */

void*    f(void);       /* f je funkce bez argumentu, vrací ukazatel na void */
void    *f(void);

void   (*g)(void);      /* g je ukazatel na funkci bez argumentu, která nevrací
                           hodnotu */
```

Rozsah platnosti identifikátorů čili viditelnost (scope)

Rozsah platnosti identifikátoru je určen tím, kde a jak je identifikátor deklarován. Rozsahy platnosti jsou tyto:

- soubor,
- blok,
- prototyp funkce,
- funkce.

V rámci zdrojového *souboru* jsou platné (viditelné) identifikátory deklarované mimo bloky. Jsou viditelné od místa deklarace do konce souboru. Soubory přikopírované příkazem #include se považují za součást zdrojového souboru.

V rámci *bloku* (mezi levou a pravou složenou závorkou) deklarovaný identifikátor je platný (viditelný) jen v tomto bloku. Bloky lze do sebe vnořovat. Identifikátor deklarovaný ve vnořeném bloku není viditelný ve vnějším bloku. Příkladem bloku je tělo definice funkce.

Identifikátor deklarovaný v seznamu argumentů u *prototypu funkce* je viditelný jen od místa jeho deklarace do koncové kulaté závorky deklarace fukce.

Jediný identifikátor s platností v rozsahu *funkce* je návěští (deklarované svým výskytem). Příkaz goto, který se na ně odvolává, musí být ve stejné funkci.

Spojení (linkage)

- externí,
- interní,
- žádné.

Externí spojení znamená, že stejné identifikátory v odděleně kompilovaných souborech označují stejný datový objekt nebo funkci. Je-li identifikátor deklarován s klíčovým slovem **`extern`**, má externí spojení. Je-li však stejný identifikátor deklarován předtím s jiným spojením (např. s klíčovým slovem `static`), platí předchozí spojení. Identifikátor viditelný v rozsahu souboru, deklarovaný *bez* klíčových slov `extern` a `static`, má také externí spojení.

Interní spojování znamená, že stejné identifikátory ve stejném zdrojovém souboru označují stejný datový objekt nebo funkci. Je-li identifikátor viditelný v souboru deklarován s klíčovým slovem **`static`**, má interní spojení.

Žádné spojení nemá identifikátor, jestliže

- nereprezentuje žádný objekt ani funkci (např. návěští),
- představuje argument funkce,
- je deklarován uvnitř bloku bez klíčového slova extern.

Životnost paměti (storage duration, lifetime)

- statická (globální),
- automatická (lokální).

Paměťový objekt má *statickou* životnost, jestliže má interní nebo externí spojení, nebo jestliže je označen klíčovým slovem **`static`**. Statická paměť existuje již při spuštění programu a trvá stále až do jeho ukončení.

Všechny ostatní objekty mají *automatickou* životnost. Paměť s automatickou životností se vytvoří při výpočtu v okamžiku vstupu do bloku a zruší se v okamžiku opuštění bloku. Automatickou životnost mají všechny objekty deklarované uvnitř bloku s klíčovým slovem **`auto`**. Toto slovo je však nepovinné, protože uvnitř bloku mají všechny objekty automatickou životnost.

Ještě se uvádí *dynamická* životnost, kterou má paměť obstaraná z operačního systému v průběhu výpočtu, např. standardní funkcí `malloc`. Dynamickou paměť lze také zrušit (uvolnit), např. standardní funkcí `free`.

Prostory jmen (name spaces)

Identifikátory ve stejném prostoru jmen musí být jedinečné, tzn, že jeden identifikátor označuje jeden objekt. Stejný identifikátor použitý v různých prostorech jmen však může označovat různé objekty. Jazyk C rozeznává následující prostory jmen:

- obyčejné identifikátory - proměnné, funkce a výčtové konstanty,
- struktury, unie a výčtové typy,
- položky struktur a unií,
- identifikátory vzniklé přejmenováním typů (`typedef`),
- návěští.

Klíčová slova

auto	**for**	**union**
break	**goto**	**unsigned**
case	**if**	**short**
char	**int**	**signed**
const	**long**	**sizeof**
continue	**precisionof**	**static**
default	**register**	**struct**
decimal	**return**	**switch**
digitsof	**short**	**typedef**
do	**signed**	**union**
double	**sizeof**	**unsigned**
else	**static**	**void**
enum	**struct**	**volatile**
extern	**switch**	**while**
float	**typedef**	

Poznámka: Kompilátor ILE C rozeznává klíčová slova `_Decimal`, `__digitsof` a `__precisionof`. Slova `decimal`, `digitsof` a `precisionof` jsou však definována jako *makropříkazy* v hlavičkovém souboru `<decimal.h>` pro kompatibilitu s jinými kompilátory jazyka C.

Trojznaky (trigraphs)

V některých prostředích nelze některé znaky jazyka C použít, proto jsou definovány tzv. trojznaky, které začínají dvěma otazníky a končí znakem napodobujícím nahrazovaný znak. Někdy je vhodné je použít, když přenášíme zdrojový kód z české kódové stránky do jiné kódové stránky, která by mohla tyto znaky interpretovat zcela jinak. Hodně potíží bývá s hranatými závorkami (brackets).

Trojznak	**Znak**	**Název znaku (angl.)**
??=	#	pound sign
??(	[	left bracket
??)	]	right bracket
??<	{	left brace
??>	}	right brace
??/	\	backslash
??'	^	caret
??!	|	vertical bar
??-	~	tilde

Poznámka: IBM i překóduje a uloží trojznak `??'` jako symbol NOT.

Některé knihovní funkce normy ANSI

Knihovní funkce jsou součástí jazyka C. Některé jsou definovány normou ANSI, jiné definicí v knize *ILE C/C++ Runtime Library Functions*. Probereme jen některé funkce, které se nejvíce uplatňují v našich příkladech. Vybrané funkce z normy ANSI budou tyto:

printf pro ladicí tisky do standardního výstupu (probíráme pouze zhruba),
memset pro inicializaci paměti v určené délce,
memcpy pro kopírování paměti v určené délce,
memmove pro přesun paměti v určené délce,
memcmp pro porovnání paměti v určené délce,
malloc pro obstarání dynamické paměti z operačního systému,
free pro zrušení dynamické paměti.

Funkce printf

Tisková funkce *printf* (a od ní odvozené funkce) je nejsložitější ze všech standardních funkcí, co se týká množství možných argumentů a jejich druhů. Prototyp má tento obecný tvar:

```
#include <stdio.h>
int printf(const char *formát, seznam-argumentů);
```

Formát má tvar znakového řetězce obsahujícího formátové specifikace. Kolik je formátových specifikací uvnitř řetězce, tolik musí být argumentů v seznamu argumentů.

Formátová specifikace je strukturovaný údaj. Obecný tvar je tento:

```
%příznaky šířka . přesnost velikost typ
```

Povinný je jen znak % a *typ*. Ostatní součásti formátové specifikace jsou nepovinné.

Poznámka 1: V následujících tabulkách jsou uvedeny jen některé možnosti.

Příznaků může být vedle sebe několik. Jednotlivé příznaky jsou tyto:

Příznak	**Význam**	**Předvolba**
-	Zarovnat doleva v rámci délky pole	Zarovnání vpravo
+	Tisknout znaménko zleva	Pouze pro záporná čísla
mezera (' ')	Tisknout mezeru zleva u kladných čísel	Bez mezery
#	U typu o, x, X tiskne předponu 0, 0x, 0X	Bez předpony
	U typu f, D(n,p), e, E tiskne vždy desetinnou tečku	Desetinná tečka jen pro čísla s desetinnými místy
0	U typu d, i, D(n,p) o, u, x, X, e, E, f tiskne vedoucí nuly. Příznak 0 se neuplatní, je-li zadána přesnost u celého čísla nebo je-li zadán příznak -.	Tisk vedoucích mezer. Bez vedoucích mezer u D(n,p).

Šířka je buď číslo určující minimální počet tištěných znaků nebo je to hvězdička. Je-li šířka dána hvězdičkou, číselnou hodnotu minimálního počtu určuje argument ze seznamu argumentů (předchází argument odpovídající tištěnému číslu a musí mít typ int). Případný delší výsledek se nezkracuje.

Přesnost je buď číslo zapsané za tečkou a určuje počet tištěných znaků nebo desetinných míst nebo je to hvězdička. Je-li přesnost dána hvězdičkou, číselnou hodnotu minimálního počtu

určuje argument ze seznamu (předchází argument odpovídající tištěnému číslu a musí mít typ int). Případný delší výsledek může být zkrácen.

Velikost je znak upřesňující chápání odpovídajícího argumentu.

Velikost	**Argument se chápe jako**	**Platí pro typy**
h	short int, unsigned short int	d, i, o, u, x, X
l	long int, unsigned long int	d, i, o, u, x, X
ll	long long int, unsigned long long int	d, i, o, u, x, X
L	long double	e, E, f

Typ je dán znakem z následující tabulky. Seznam není úplný, tvar výstupu je vyznačen jen schematicky.

Typ	**Argument**	**Tvar výstupu**
d, i	Celé číslo	Dekadické celé číslo se znaménkem
u	Celé číslo	Dekadické celé číslo bez znaménka
o	Celé číslo	Osmičkové číslo bez znaménka
x	Celé číslo	Hexadecimální číslo bez znaménka s použitím abcdef
X	Celé číslo	Hexadecimální číslo bez znaménka s použitím ABCDEF
D(n,p)	Pakované dekadické číslo místo n i p může být hvězdička	[-]dddddd**.**ddddd
f	Double	[-]dddddd**.**ddddd
e	Double	[-]d**.**dddd **e**[zn.] ddd
E	Double	[-]d**.**dddd **E**[zn.] ddd
g	Double	jako f nebo e, podle vhodnosti
G	Double	jako f nebo E, podle vhodnosti
c	Znak (bajt)	Jeden znak
s	Řetězec zakončený nulou	Znaky před \0 nebo do dosažení *přesnosti*
p	Ukazatel	Obsah ukazatele v tisknutelné formě

Příklad – program PRINTF

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

main()
{
   int           i1 = 111,       i2 = -222;
   double        d1 = 1.1e+3,    d2 = -2.2e-3;
   decimal(8,3)  D1 = 11111.111, D2 = -22222.222;
   int           dp1 = 1,        dp2 = 0;          /* počty des. míst */
   int          *ptr = &i1;                        /* ukazatel na číslo i1 */

 printf ( "D1 (%%010D(8,3))   = |%010D(8,3)|\                                  (1)
  D2 (%% 10D(8,3))   = |% 10D(8,3)|\
\n"                                                      , D1, D2 );
 printf ( "D1 (%%010.2D(8,3)) = |%010.2D(8,3)|\
  D2 (%% 10.2D(8,3)) = |% 10.2D(8,3)|\
\n\n"                                                    , D1, D2 );
 printf ( "D1 (%%010.*D(8,3)) = |%010.*D(8,3)|\                               (2)
  D2 (%% 10.*D(8,3)) = |% 10.*D(8,3)|\
\n\n"                                                 , dp1, D1, dp2, D2 );

 printf ( "i1 (%%015d) = |%015d|   i2 (%% 15i) = |% 15i| \n" ,  i1, i2 );
 printf ( "i1 (%%015o) = |%015o|   i2 (%% 15o) = |% 15o| \n" ,  i1, i2 );
 printf ( "i1 (%%015u) = |%015u|   i2 (%% 15u) = |% 15u| \n" ,  i1, i2 );
 printf ( "i1 (%%015x) = |%015x|   i2 (%% 15X) = |% 15X| \n\n", i1, i2 );

 printf ( "d1 (%%015f) = |%015f|   d2 (%% 15f) = |% 15f| \n" ,  d1, d2 );
 printf ( "d1 (%%015e) = |%015e|   d2 (%% 15e) = |% 15e| \n" ,  d1, d2 );
 printf ( "d1 (%%015E) = |%015E|   d2 (%% 15E) = |% 15E| \n" ,  d1, d2 );
 printf ( "d1 (%%015g) = |%015g|   d2 (%% 15g) = |% 15g| \n" ,  d1, d2 );
 printf ( "d1 (%%015G) = |%015G|   d2 (%% 15G) = |% 15G| \n\n", d1, d2 );

 printf ( "ptr (%%p)   = |%p| \n",                                  ptr );    (3)
}
```

Výstup z programu PRINTF:

```
D1 (%010D(8,3))   = |011111.111|  D2 (% 10D(8,3))   = |-22222.222|
D1 (%010.2D(8,3)) = |0011111.11|  D2 (% 10.2D(8,3)) = | -22222.22|

D1 (%010.*D(8,3)) = |00011111.1|  D2 (% 10.*D(8,3)) = |    -22222|          (2)

i1 (%015d) = |000000000000111|   i2 (% 15i) = |           -222|
i1 (%015o) = |000000000000157|   i2 (% 15o) = |    37777777442|
i1 (%015u) = |000000000000111|   i2 (% 15u) = |     4294967074|
i1 (%015x) = |00000000000006f|   i2 (% 15X) = |       FFFFFF22|

d1 (%015f) = |00001100.000000|   d2 (% 15f) = |      -0.002200|
d1 (%015e) = |0001.100000e+03|   d2 (% 15e) = |  -2.200000e-03|
d1 (%015E) = |0001.100000E+03|   d2 (% 15E) = |  -2.200000E-03|
d1 (%015g) = |000000000001100|   d2 (% 15g) = |        -0.0022|
d1 (%015G) = |000000000001100|   d2 (% 15G) = |        -0.0022|

ptr (%p)   = |SPP:0000 :1aefTCP01     QPGMR     166153 :140:0:24032|        (3)
```

Vysvětlivky k programu PRINTF

(1) Dva znaky % za sebou znamenají tisk znaku % (není to tedy součást formátové specifikace). Jsou použity pro tisk tvaru formátové specifikace na výstup.

(2) Hvězdičky ve formátových specifikacích na místě *přesnosti* způsobují tisk jednoho a žádného desetinného místa, jak je zadáno v proměnných `dp1` a `dp2`.

Obsah ukazatele je podivně strukturovaný, obsahuje mj. identifikaci úlohy (job). Ve skutečnosti ovšem (podíváme-li se před tiskem do paměti pomocí ladicího programu, je obsah ukazatele `ptr` na proměnnou `i1` např. tento:

```
SPP:C34844A8EE001140
```

přesněji

```
80000000 00000000 C34844A8 EE001140.
```

Hodnota ukazatele je ovšem při spuštění v jiné úloze jiná. Koncovka je 001140 je stále stejná, není to ovšem *1aef*, jak bychom čekali z formátového výtisku. Není tedy jasné, co tedy tento výtisk vlastně znamená.

Poznámka: Funkce *`sprintf`* funguje stejně jako `printf`, ale výsledek umístí do paměti podle ukazatele `buffer` jako řetězec ukončený nulovým znakem. Má tento prototyp:

```
#include <stdio.h>
int sprintf(char *buffer, const char *format-string, argument-list);
```

Vrácená hodnota je počet bajtů výsledku nepočítaje v to koncový znak \0.

Funkce memset

Funkce *`memset`* slouží k inicializaci paměti zvoleným znakem. Má tento prototyp:

```
#include <string.h>
void *memset(void *dest, int c, size_t count);
```

Paměť od adresy `dest` se naplní znakem `c` v délce `count`. Funkce vrací ukazatel na `dest`. (Typ `size_t` je `int`.)

Funkce memcpy

Funkce *`memcpy`* kopíruje paměť v určené délce z jedné adresy na jinou. Funguje dobře, jestliže se výchozí paměť nepřekrývá s cílovou pamětí. Jinak jsou výsledky neurčité. Funkce má tento prototyp:

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t count);
```

Paměť od adresy `src` se kopíruje na adresu `dest` v délce `count`. Funkce vrací ukazatel na *`dest`*.

Funkce memmove

Funkce *`memmove`* kopíruje paměť v určené délce z jedné adresy na jinou, i když se výchozí a cílová paměť překrývá. Má tento prototyp:

```
#include <string.h>
void *memmove(void *dest, const void *src, size_t count);
```

Paměť od adresy `src` se kopíruje na adresu `dest` v délce `count`. Při tom se to jeví tak, jako kdyby se nejdříve celá výchozí paměť zkopírovala do pomocné paměti a ta pak do cílové paměti. Funkce vrací ukazatel na `dest`.

Funkce memcmp

Funkce *`memcmp`* porovnává paměť na jedné adrese s pamětí na jiné adrese v určené délce a výsledek porovnání vrací jako celé číslo.

```
#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t count);
```

Paměť na adrese `buf1` se porovná s pamětí `buf2` v délce `count`. Výsledek porovnání je

záporný, je-li	`buf1 < buf2`,
nula, je-li	`buf1 = buf2`,
kladný, je-li	`buf1 > buf2`.

Kolekce funkcí isxxx pro testování dat

Sem patří funkce jako *`isalnum`* nebo *`isdigit`*, tedy funkce jejichž jméno začíná písmeny **`is`**. Všechny mají jediný argument typu *`int`*. Vracejí hodnotu 0, je-li výsledek testu negativní, nebo hodnotu různou od 0, je-li výsledek testu pozitivní. Uvedeme jen některé.

```
#include <ctype.h>

int isalnum(int c);  /* Test for upper- or lowercase letters, or decimal digit */
int isalpha(int c);  /* Test for alphabetic character */
int isdigit(int c);  /* Test for decimal digit */
int islower(int c);  /* Test for lowercase */
int isupper(int c);  /* Test for uppercase */
int isxdigit(int c); /* Test for hexadecimal digit */
```

Aktuálním argumentem je zpravidla znakový typ, který je ovšem před provedením testu převeden na celočíselný a pak porovnán s číslem odpovídajícím znaku v kódu EBCDIC. Příklad testuje tříznakové pole, zda první znak je písmeno a druhé dva znaky jsou dekadické číslice.

```
char a[] = "A01";
int  i;

if ( isalpha(a[0]) && isdigit(a[1]) && isdigit(a[2]) ) i = 1;
else i = 0;
```

Funkce malloc

Funkce *`malloc`* získá paměť z operačního systému v určené délce. Funkce má tento prototyp:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Funkce obstará paměť v délce `size` bajtů a vrátí ukazatel na začátek této paměti. Ukazatel je typu `void`, proto je nutné jej před použitím přetypovat. Vrácený ukazatel je NULL, jestliže není dostatek paměti v systému, nebo je-li zadána nulová velikost `size`.

Poznámka: Existují dvě další paměťové funkce. Funkce *`calloc`* funguje stejně jako `malloc`, ale inicializuje paměť binárními nulami. Funkce *`realloc`* mění velikost paměti.

Funkce free

Funkce *`free`* zruší paměť získanou funkcí malloc. Má tento prototyp:

```
#include <stdlib.h>
void free(void *ptr);
```

Zruší (vrátí) se paměť, na niž ukazuje *`ptr`*, a to v délce, která byla naposledy použita ve funkci `malloc`. Je-li `ptr` nulový (NULL), funkce neprovede nic.

Poznámka: Jestliže ukazatel `ptr` obsahuje jinou adresu než tu, která byla získána z operačního systému funkcí `malloc` (`calloc`, `realloc`), je výsledek nejistý.

Vybrané knihovní funkce ILE C

Konverze dat

Často je zapotřebí konverze celého čísla typu *int* do zónového dekadického čísla a naopak, nejčastěji při práci s obrazovkovými soubory.

QXXITOZ (Convert Integer to Zoned Decimal) - konverze *do zónového* dekadického čísla,
QXXZTOI (Convert Zoned Decimal to Integer) - konverze *ze zónového* dekadického čísla.

Práce s datovou oblastí

Čtení z datové oblasti (*data area*) a zápis do ní je užitečný pro globální data, která není vhodné zařazovat do programu jako argumenty nebo proměnné s externím spojením (`extern`), ani jako databázový soubor.

QXXRTVDA (Retrieve Data Area) - čtení datové oblasti,
QXXCHGDA (Change Data Area) - změna datové oblasti,

Obsluha souborů

Pro obsluhu souborů slouží funkce, jejichž jméno začíná **`_R`**. Jde o soubory databázové, obrazovkové, tiskové a komunikační ICF soubory (APPC, asynchronní apod.). Ty jsou většinou spjaty s externím popisem souboru DDS (Data Description Specifications).

_Ropen (Open a Record File for I/O Operations) - otevření pro všechny soubory,
_Rclose (Close a File) - uzavření pro všechny soubory,
_Rreadn (Read the Next Record) - čtení dalšího záznamu pro databázové soubory,
_Rreadk (Read a Record by Key) - čtení záznamu podle klíče pro databázové soubory,
_Rwrite (Write the Next Record) - zápis nového záznamu pro všechny soubory včetně tiskových,
_Rupdate (Update a Record) - přepis záznamu pro databázové a obrazovkové soubory,
_Rdelete (Delete a Record) - výmaz záznamu pro databázové soubory,
_Rwrited (Write a Record Directly) - zápis záznamu podle pořadového čísla pro databázové a obrazovkové soubory (subfiles),
_Rwriterd (Write and Read a Record) - zápis a čtení pro obrazovkové soubory,
_Rreadnc (Read the Next Changed Record) - čtení dalšího změněného záznamu pro obrazovkové soubory (subfiles).

Některé funkce API

Funkcím API (Application Programming Interface) se také říká volání API (API calls). Ty nejsou součástí knihovny jazyka C. Jde o služby operačního systému IBM i, které lze volat z programů v různých jazycích, nejen C. Je jich velmi mnoho, ale my uvedeme jen některé z nich.

Editace čísel

Editaci čísel lze sice provádět pomocí funkce `printf()` na standardní výstup a pomocí funkce `sprintf()` do paměti, ale editace podle konvencí IBM i vypadá jinak. Používají se tzv. *ediční kódy* (edit codes) nebo *ediční slova* (edit words). Takovou editaci můžeme docílit voláním dvojice API:

QECCVTEC (Convert Edit Code) - konverze edičního kódu do masky *nebo QECCVTEW* (Convert Edit Word) - konverze edičního slova do masky a
QECEDT (Edit) – editace čísla podle zkonvertované masky.

Program EDTCDE ilustruje použití edičního kódu k úpravě dekadického čísla pro výstup.

Zasílání programových zpráv

Uvedeme jen dvě volání API, a to zasílání programových zpráv (tj. z programu do fronty zpráv jiného programu nebo jiné fronty zpráv) a přijímání programových zpráv (tj. do programu z programové či jiné fronty zpráv).

QMHSNDPM (Send Program Message) – zaslání programové zprávy,
QMHRCVPM (Receive Program Message) – přijetí programové zprávy.

Práce s databázovým souborem

K obsluze databázových souborů s přístupem k záznamům je zapotřebí tento hlavičkový soubor:

```
#include <recio.h>
```

Vytvoří se mimo jiné typ pro "feedback" informační strukturu:

```
typedef struct {
  unsigned char               *key;
  _Sys_Struct_T               *sysparm;
  unsigned long               rrn;
  long                        num_bytes; /* testuje se na konec dat EOF = -1 */
  short                       blk_count;
  char                        blk_filled_by;
  int                         dup_key   : 1;
  int                         icf_locate: 1;
  int                         reserved1 : 6;
  char                        reserved2[20];
} _RIOFB_T;
```

Údaj *num_bytes* se testuje, zda je menší než počet bajtů, který byl zadán; v tom případě je to chybná operace (např. čtení z uzavřeného souboru apod.). Dále se vytvoří definice typu pro ukazatel na soubor (file pointer). Z ostatních položek se normálně nic nepotřebuje.

```
typedef _Packed struct {
  char                        reserved1[16];
  volatile void  *const *const in_buf;
  volatile void  *const *const out_buf;
  char                        reserved2[48];
  _RIOFB_T                    riofb;
  char                        reserved3[32];
  const unsigned int          buf_length;
  char                        reserved4[28];
  volatile char  *const       in_null_map;
  volatile char  *const       out_null_map;
  volatile char  *const       null_key_map;
  char                        reserved5[48];
  const int                   min_length;
  short                       null_map_len;
  short                       null_key_map_len;
  char                        reserved6[8];
} _RFILE;
```

Do programu musíme napsat deklaraci:

```
#pragma mapinc("database","*LIBL/DATA(*ALL)","both key",,,"A" )
#include "database"
```

Příkaz preprocesoru *#pragma mapinc* je vlastně kopírování údajů z DDS databáze DATA a vytvoří definici typu struktury:

```
typedef struct {
   char KEY[5];                          /*  Key
   char DESCRIPT[30];                    /*  Description
   decimal( 7, 2) NUMBER;                /*  Number
                                         /*  PACKED SPECIFIED IN DDS
}A_DATAR_both_t;

typedef struct {
   char KEY[5];
                                         /*  DDS - ASCENDING
                                         /*  STRING KEY FIELD
                                         /*  ALTERNATE SEQUENCE
}A_DATAR_key_t;
```

Dále je třeba napsat deklarace proměnných k uvedeným typům, z nichž první dvě definují paměť dat *dbb*, a paměť klíče *key*, a další dvě definují ukazatel *dbf* na soubor a ukazatel *dbfb* na stavovou strukturu (feedback structure):

```
A_DATAR_both_t     dbb;    /* database data buffer */
A_DATAR_key_t      key;    /* database key buffer */
_RFILE   *dbf;  /* Database file  pointer */
_RIOFB_T *dbfb; /* Pointer to the file's feedback structure */
```

Teď již můžeme psát příkazy pro zpracování databázového souboru DATA. V první řadě je nutné otevřít soubor:

```
/*   Open the database file for reading, writing and updating       */
dbf  = _Ropen ( "DATA", "rr+" );
```

Pak můžeme číst záznam podle klíče atd.

```
/*   Read database record by key                                   */
dbfb = _Rreadk(dbf, &dbb, sizeof(dbb), __KEY_EQ, &key, sizeof(key));

if (dbfb -> num_bytes == 0) {  /* not found */
  ...
}
if (dbfb -> num_bytes != 0) {  /* found */
  ...
}
```

Na konci práce se souborem je nutno uzavřít soubor:

```
_Rclose ( dbf );
```

Program DQDBMAINT ilustruje zpracování databázového souboru zároveň s obrazovkovým souborem. Tento program je nutné vytvořit jako spojení dvou modulů – DQDBMAINT a PACKUNPK. Druhý modul realizuje převod dekadického čísla z pakovaného do zónového tvaru funkcí *pack()* a obráceně funkcí *unpk()*.

Práce s obrazovkovým souborem

K obsluze obrazovkových souborů je zapotřebí stejný hlavičkový soubor jako pro databáze:

```
#include <recio.h>
```

kde *typy* pro "feedback" strukturu _RIOFB_T a pro strukturu údajů o souboru _RFILE jsou totožné s databázovými.

Do programu musíme napsat deklaraci:

```
#pragma mapinc("display","*LIBL/DISPLAY(*ALL)","both indicators",,,"B"  )
#include "display"
```

Příkaz preprocesoru *#pragma mapinc* je vlastně kopírování údajů z DDS obrazovkového souboru DISPLAY a vytvoří definice různých typů struktur podle toho, kolik formátů obsahuje popis DDS:

```
typedef struct {
   char KEY[5];                         /*  Key
}B_DISPLAY1_both_t;

typedef struct {
   char IN01_IN02[2];                   /*  UNUSED INDICATOR(S)
   char IN03;
   char IN04_IN99[96];                  /*  UNUSED INDICATOR(S)
}B_DISPLAY1_indic_t;

typedef struct {
   char KEY[5];                         /*  Key
   char DESCRIPT[30];                   /*  Description
   char NUMBER[7];                      /*  Number
                                        /*  ZONED SPECIFIED IN DDS
                                        /*  REPLACED BY CHARACTER TYPE
}B_DISPLAY2_both_t;

typedef struct {
   char IN01_IN02[2];                   /*  UNUSED INDICATOR(S)
   char IN03;
   char IN04_IN11[8];                   /*  UNUSED INDICATOR(S)
   char IN12;
   char IN13_IN80[68];                  /*  UNUSED INDICATOR(S)
   char IN81;
   char IN82_IN99[18];                  /*  UNUSED INDICATOR(S)
}B_DISPLAY2_indic_t;
```

Dále je třeba napsat deklarace proměnných k uvedeným typům. Jednak proměnné pro informační struktury:

```
_RFILE   *dspf;  /* Display file pointer */
_RIOFB_T *dspfb; /* Pointer to the file's feedback structure */
```

jednak pro datové oblasti:

```
B_DISPLAY1_both_t     dsp1;   /* display buffer 1 */
B_DISPLAY2_both_t     dsp2;   /* display buffer 2 */
```

I když indikátory jsou popsány v typu struktury *B_DISPLAY1_indic_t* a *B_DISPLAY2_indic_t*, je někdy výhodné mít indikátory definované speciálním typem pole *_SYSindara*:

```
typedef char _SYSindara[99];
_SYSindara             ind;    /* indicator area */
```

Na tyto indikátory se lze odvolávat jako na položky pole. Například *ind[0]* znamená indikátor 01, ind[98] znamená indikátor 99. V popisu DDS je třeba zadat klíčové slovo INDARA (indicator area).

Teď již můžeme psát příkazy pro zpracování obrazovkového souboru DISPLAY. V první řadě je nutné otevřít soubor:

```
dspf = _Ropen ( "DISPLAY", "rr+ indicators=Y");
```

Dále musíme přiřadit pole indikátorů *ind[]* obrazovkovému souboru:

```
_Rindara ( dspf, &ind[0]);
```

Pak již lze zobrazit data obsažená v paměti dsp1 na obrazovce a čekat na vstup z klávesnice:

```
_Rformat ( dspf, "DISPLAY1" );    /* Select display format 1 */
dspfb = _Rwriterd ( dspf, &dsp1, sizeof(dsp1) );
```

Ihned potom je nutné testovat indikátory funkčních kláves, v našem případě klávesy F3:

```
if (ind[02] == '1') goto end;
```

pak připravit data pro druhý formát (číst z databáze podle klíče), zobrazit je (s čekáním na vstup s klávesnice) a testovat funkční klávesy (F3 a F12):

```
_Rformat ( dspf, "DISPLAY2" );        /* Select display format 2 */
dspfb = _Rwriterd ( dspf, &dsp2, sizeof(dsp2) );
if ( ind[02] == '1' ) goto end;    /* F3 - end program */
if ( ind[11] == '1' ) goto repeat; /* F12 - repeat format 1 */
```

Nakonec je třeba uzavřít obrazovkový soubor příkazem

```
_Rclose ( dspf );
```

Program DQDBMAINT ilustruje zpracování obrazovkového souboru zároveň s databázovým souborem. Tento program je nutné vytvořit jako spojení dvou modulů – DQDBMAINT a PACKUNPK. Druhý modul realizuje převod dekadického čísla z pakovaného do zónového tvaru funkcí *`pack()`* a obráceně funkcí *`unpk()`*.

Práce s datovými frontami (data queues)

Datové fronty umožňují efektivní komunikaci mezi programy, ať běží ve stejné úoze nebo v různých úlohách (ať už v interakčních nebo dávkových). Datová fronty je objekt typu *DTAQ, vytváří se příkazem CRTDTAQ a ruší příkazem DLTDTAQ. K práci s nimi slouží následujcící funkce API:

QCLRDTAQ (Clear Data Queue) – vyčistit datovou frontu,
QRCVDTAQ (Receive Data Queue) – vyzvednout (přijmout) zprávu z datové fronty,
QSNDDTAQ (Send Data Queue) – zaslat zprávu do datové fronty.

Datová fronta může mít jeden ze tří typů *FIFO, *LIFO, *KEYED, který se určuje při vytváření. Podle typu se řadí zprávy do fronty a vybírají zprávy z fronty. U typu *FIFO (First In First Out) se první zaslaná zpráva vyzvedne jako první. U typu *LIFO (Last In First Out) se poslední zaslaná zpráva vyzvedne jako první. U typu *KEYED se zprávy řadí a vybírají podle klíče, podobně jako u databázového souboru.

Programy DQSERVER a DQCLIENT ilustrují komunikaci přes datové fronty typu *FIFO a *KEYED. V programu DQCLIENT je použita též funkce API

QUSRJOBI (Retrieve Job Information) – získat informace o úloze (zde jméno a čislo úlohy a jméno uživatele);

Práce s tiskovými soubory

Tiskové soubory bývají často popsány pomocí DDS. V tom případě se opět, podobně jako u ostatních souborů zadávají příkazy

```
#pragma mapinc("printer","*LIBL/PRINTER(*ALL)","output indicators","_P",,"PR" )
#include "printer"

_RFILE   *prtf;       /* Printer file pointer */
_RIOFB_T *prtfb;      /* Pointer to the file's feedback structure */

PR_DETAIL_o_t         det;     /* printer detail line */

prtf = _Ropen ( "PRINTER", "wr" );

/*   .Print header 1                                                  */
  _Rformat ( prtf, "HEADER1" );    /* Select printer format */
      prtfb = _Rwrite (prtf, "", 0 );

/*   .Print header 2                                                  */
  _Rformat ( prtf, "HEADER2" );    /* Select printer format */
      prtfb = _Rwrite (prtf, "", 0 );

/*   .Print detail line                                               */
  _Rformat ( prtf, "DETAIL" );     /* Select printer format */
  prtfb = _Rwrite (prtf, &det, sizeof(det) );

_Rclose ( prtf  );
```

Navíc bývá potřeba definovat dvě informační oblasti pro zjištění počtu řádků na stránce a čísla běžného řádku.

```
_XXIOFB_T *iofb;      /* Pointer to the file's feedback area */
_XXOPFB_T *opnfb;     /* Pointer to the file's open feedback area */
```

Počet řádků na stránce (overflow line) se zjistí příkazy

```
opnfb = _Ropnfbk( prtf );
ovfline = (*opnfb).overflow_line_num;
```

a číslo právě vytištěného řádku se zjistí příkazy

```
iofb = _Riofbk ( prtf );
curline = *((short *)((char *)iofb + iofb->file_dep_fb_offset));
```

Program DQPRINT tiskne obsah databázového souboru DATA s dvěma hlavičkovými řádky na každé stránce.

Práce s proudovými soubory (stream files) v IFS

Proudové soubory (stream files) jsou objekty typu *STMF a jsou umístěny v integrovaném systému souborů IFS (Integrated File System). Z hlediska obsahu se dělí na dva základní typy:

- textový,
- binární.

Z hlediska zpracování se tyto typy dělí na

- bajtový (stream I/O),
- záznamový (record I/O).

Bajtové soubory nemají žádnou strukturu (skládají se z proudu bajtů), zatímco záznamové jsou členěny na záznamy pevné nebo proměnné délky, blokované nebo neblokované.

Základní příkazy pro zpracování proudových souborů v IFS jsou realizovány jako funkce API typu UNIX. Těch je mnoho, jsou popsány v dokumentaci uvedené na začátku. Zde uvedeme jen čtyři.

close() – Close file descriptor
open() – Open file
read() – Read from file
write() – Write to file

Příkazy pro zpracování proudových souborů podle definice ANSI začínají písmenem **`f`**. Jsou popsány v knize *ILE C/C++ Runtime Library Functions*. Zde je výběr některých nejdůležitějších příkazů.

fopen() – **Open Files**
fclose() – **Close Stream**
fread() – **Read Items**
fgetc() – Read a Character
fgets() – Read a String
feof() – Test End-of-File Indicator
fwrite() – **Write Items**
fputc() – Write Character
fputs() – Write String
fprintf() – Write Formatted Data to a Stream
fscanf() – Read Formatted Data
fseek() – Reposition File Position
fsetpos() – Set File Position
ftell() – Get Current Position

Příkaz *fopen* může definovat bajtový nebo záznamový režim, příkazy *fread*, *fwrite* umožňují záznamové zpracování, příkazy *fgetc*, *fputc* aj. zase umožňují zpracování po znacích. Tučně vytištěné příkazy lze použít také ke zpracování databázových souborů.

Tyto příkazy jsou realizovány jako nadstavba základních funkcí pro IFS (open, read, write, close aj.), jsou-li použity na soubory v IFS. Jsou-li použity na databázové soubory, jsou realizovány jako nadstavba obvyklých databázových operací IBM i.

Program IFS1 ilustruje kopírování zdrojového členu do IFS souboru a zpět.

```
/*****************************************************************/
/*   Program kopíruje zdrojový člen do adresáře IFS a zpět.      */
/*     Překódování z EBCDIC do ASCII a zpět.                     */
/*****************************************************************/

#include <string.h>
#include <fcntl.h>
#include <decimal.h>
#include <H/QDCXLATE>

   int  fdin, fdout;
   size_t  size;

   char          inpath  [256] = "/qsys.lib/vzc.lib/qcsrc.file/IFS1.mbr";
   char          outpath [256] = "/home/ilec00/IFS1.txt";

   char          text [100];
   decimal (5,0) textl = sizeof(text);

   char  toASCII  [10] = "QASCII    ";
   char  toEBCDIC [10] = "QEBCDIC   ";
   char  lib      [10] = "*LIBL     ";

 main () {

/*   Kopírování ze zdrojového členu do adresáře IFS                */
/*   ----------------------------------------------                */

   fdin  = open (inpath,  O_RDONLY | O_TEXTDATA, S_IRUSR );
   fdout = open (outpath, O_CREAT | O_RDWR | O_TRUNC | O_TEXTDATA, S_IWUSR );

   size = read  ( fdin, &text, sizeof(text) );

   while ( size == sizeof(text)) {
     QDCXLATE ( &textl, text, toASCII, lib );
     size = write ( fdout, &text, strlen(text) );

     size = read  ( fdin, &text, sizeof(text) );
   }
   if (size > 0)
     QDCXLATE ( &textl, text, toASCII, lib );
     size = write ( fdout, &text, size );

   close (fdin);
   close (fdout);


/*   Kopírování z adresáře IFS do zdrojového členu                 */
/*   ----------------------------------------------                */
/*   (člen IFS1X musí existovat - třeba prázdný)                   */

   strcpy (inpath, "/home/ilec00/IFS1.txt");
   strcpy (outpath, "/qsys.lib/vzc.lib/qcsrc.file/IFS1X.mbr");
   textl = 92d;

   fdin  = open (inpath,  O_RDONLY | O_TEXTDATA, S_IRUSR );
   fdout = open (outpath, O_CREAT | O_WRONLY | O_TRUNC | O_TEXTDATA, S_IWUSR );

   size = read  ( fdin, &text, 92 );

   while ( size == 92 ) {
     QDCXLATE ( &textl, text, toEBCDIC, lib );
     size = write ( fdout, &text, 92 );
```

```
      size = read  ( fdin, &text, 92 );
    }
    if (size > 0)
      QDCXLATE ( &textl, text, toEBCDIC, lib );
      size = write ( fdout, &text, size );

    close (fdin);
    close (fdout);
}
```

Program IFS2 ilustruje kopírování ukládacího souboru (save file) do IFS souboru a zpět pomocí funkcí `open`, `read`, `write`, `close` a `fopen`, `fread`, `fwrite`, `fclose`.

```
/*****************************************************************/
/*                                                               */
/*   Program kopíruje save file do IFS (binárně)                 */
/*                                                               */
/*   Kompilace musí probíhat s parametrem SYSIFCOPT(*NONE)       */
/*   kvůli jménu souboru ve tvaru IBM i (lib/obj)               */
/*                                                               */
/*****************************************************************/
#include <stdio.h>
#include <fcntl.h>
#include <decimal.h>

   int     fdin;
   int     fdout;
   FILE   *instream;
   FILE   *outstream;
   int     size;

   char    outpath [] = "/home/ILEC00/SAVF.SAVF";
   char    inpath  [] = "/home/ILEC00/SAVF.SAVF";

   char    text [528];  /* 528 je povinná délka záznamu */

 main () {

   instream  = fopen ( "vzc/savf",
                       "rb lrecl=528 type=record" );
   fdout = open ( outpath, O_CREAT | O_WRONLY | O_TRUNC, S_IWUSR );

   size = fread ( text, 1, 528, instream );
   while ( size != 0 ) {
     size = write ( fdout, &text, 528 );
     size = fread ( text, 1, 528, instream );
   }

   fclose (instream);
   close (fdout);


   fdin = open ( inpath, O_RDONLY, S_IRUSR );
   outstream = fopen ( "vzc/savf2",
                       "wb lrecl=528 type=record" );

   size = read ( fdin, &text, 528 );
   while ( size != 0 ) {
     size = fwrite ( text, 1, 528, outstream );
     size = read ( fdin, &text, 528 );
   }

   close (fdin);
   fclose (outstream);
}
```

Program IFS3 ilustruje kopírování ukládacího souboru (save file) do IFS souboru a zpět pomocí funkcí `open`, `read`, `write`, `close` a `_Ropen`, `_Rreadn`, `_Rwrite`, `_Rclose`.

```
#include <recio.h>
#include <fcntl.h>

   int     fdin;
   int     fdout;
   int     size;

   _RFILE   *fp;
   _RIOFB_T *fb;

   char          inpath  [] = "/home/ILEC00/SAVF.SAVF";
   char          outpath [] = "/home/ILEC00/SAVF.SAVF";

   char          text [528];

 main () {

   fp    = _Ropen ( "VZC/SAVF", "rr lrecl=528 riofb=N" );
   fdout = open (outpath, O_CREAT | O_RDWR | O_TRUNC, S_IRWXU );

   fb   = _Rreadn ( fp, text, 528 , __DFT );
   while ( fb -> num_bytes != EOF ) {
     size = write ( fdout, &text, 528 );
     fb   = _Rreadn ( fp, text, 528 , __DFT );
   }

   _Rclose (fp);
   close (fdout);


   fdin  = open (inpath, O_RDONLY, S_IRUSR );
   fp    = _Ropen ( "VZC/SAVF2", "wr lrecl=528 riofb=N" );

   size = read ( fdin, &text, 528 );
   while ( size != 0 ) {
     fb   = _Rwrite ( fp, text, 528 );
     size = read ( fdin, &text, 528 );
   }

   close (fdin);
   _Rclose (fp);
}
```