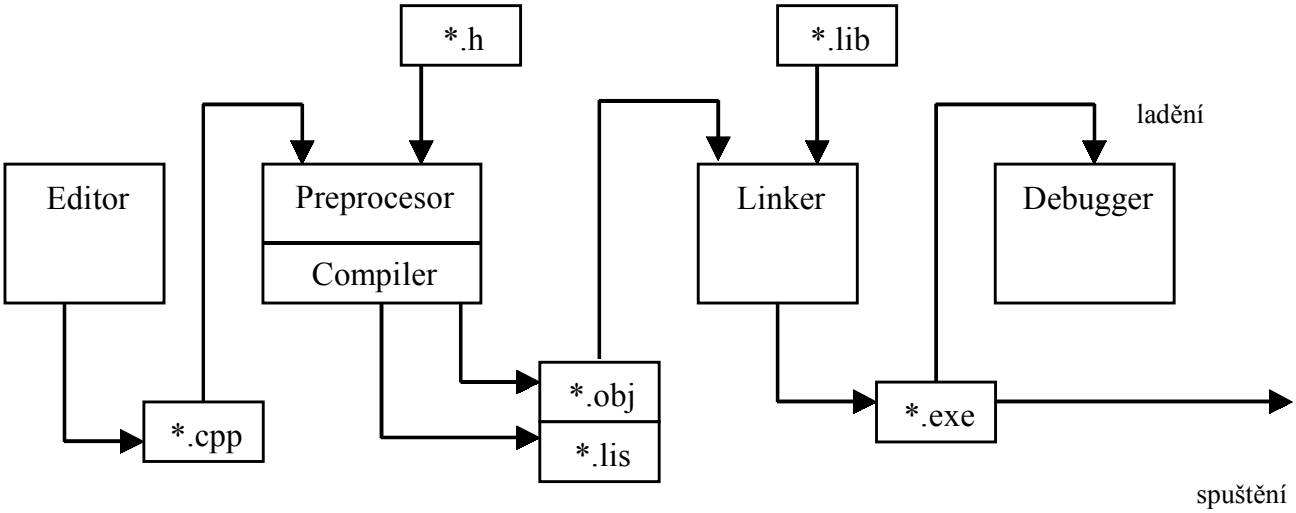
Úvod do jazyka C



Editor Pomocí něj vytváříme a upravujeme zdrojový soubor (*.C, *.CPP).

Preprocesor Jedná se o součást překladače, která předzpracovává zdrojový soubor. Vkládá soubory *.H.

Compiler Kompilátor provádí překlad zdrojového souboru do relativního (objektového) kódu počítače – vznikne *.OBJ soubor. Relativní kód je téměř hotový program. Pouze adresy proměnných nebo funkcí ještě nejsou známy (např. jsou uloženy ještě v knihovně), a jsou tedy uloženy pouze relativně.Vedlejší produkt překladače je tzv. protokol o překladu *.LIS.

Linker Linker je sestavovací program, který přidělí relativnímu kódu absolutní adresy a provede všechny odkazy (najde adresy) na dosud neznámé identifikátory (např. na knihovní funkce z *.LIB souborů). Výsledkem je spustitelný program *.EXE.

Debugger Jedná se o ladící program. Slouží k nalezení chyb, které se vyskytuji při běhu programu.

Úvod

Programovací jazyk C je všeobecně použitelný programovací jazyk známý svou efektivitou a přenositelností. Tato charakteristika jej předurčuje pro prakticky všechny oblasti programování. Obzvláště užitečným je C v systémovém programování, protože umožňuje psaní rychlých, kompaktních programů, které jsou snadno adaptovatelné pro jiné systémy. Dobře napsaný C program je často stejně rychlý, jako program napsaný v assembleru. Navíc je ovšem čitelnější a snadněji udržovatelný.

Jazyk C vznikl roku 1972 u firmy AT&T pod taktovkou Dennise Ritchieho. Jazyk byl vyvinut původně pro Unix (je zajímavé, že vlastně Unix v něm byl i napsán). V roce 1978 přišla další verze, tentokrát s názvem Kernighan-Ritchie C. První norma vznikla v r. 1984 - ANSI C, další pak roku 1990. Jazyk C++ je jazyk vycházející z C, ale je doplněn o plno nových lepších prvků - například možnost používat naplno objektovou technologii programování. Nyní se již nové normy ANSI C neobjevují, zlepšují se totiž normy ANSI C++. C++ dnes již úplně nahradil starý C (vše co jde v C, jde i v C++ a leckdy daleko rychleji).

Program v jazyce C nebo C++ se ukládá do souborů ***.C** nebo ***.CPP**. Do těchto základních souborů se mohou vkládat hlavičkové soubory ***.h**. Ostatní soubory se tak často nepoužívají, nebo jsou používány u složitějších programů. Pro jednoduché programy pro DOS stačí otevřít nový soubor a začít psát program. Programy pro Windows mívají již složitější strukturu, takže jsou vytvářeny pomocí tzv. Project souborů - ***.PRJ**.

První program v jazyce C

Abychom mohli začít s C co nejrychleji experimentovat, pokusíme se ve zbytku kapitoly ukázat nejdůležitější jazykové konstrukce. Nebudeme si činit nároky na úplnost jejich objasnění či definice. Tuto úlohu splníme v dalších kapitolách. Zde nám půjde spíše o vytvoření základní představy o tom, jak program v C vypadá, případně jak použijeme jednoduchý vstup a výstup.

Každý program v C musí obsahovat alespoň jednu funkci. Ta jediná nepostradatelná funkce se musí jmenovat `main`. V dalším textu budeme pro zřetelné odlišení jmen funkcí od ostatních identifikátorů používat kulaté závorky. Každý C program tedy obsahuje funkci `main()`, kterou vykonává při svém spuštění.

Funkce v C má deklarován typ návratové hodnoty a může mít argumenty různých typů. C disponuje datovými typy plně postačujícími pro běžné použití. Pokud to je užitečné, můžeme definovat vlastní nové datové typy. Návratovou hodnotu funkce určuje `return` příkaz.

Další součásti programu si popíšeme na příkladu. Jistě by nám chybělo, kdybychom jako první program napsali něco jiného, než klasické "Hello, world".

```
/************************/
/* HELLO.C              */
/* rychly zacatek       */
/************************/

#include <stdio.h>

int main(void)
{
 printf("Hello, world.\n");
 return 0;
} /* main */
```

O nepostradatelnosti funkce `main()` již víme. Její argument `void` nám říká, že jí nepředáváme žádné argumenty. Úvodní `int` určuje celočíselný typ návratové hodnoty. Tělo funkce je vymezeno složenými závorkami `{}`. Funkce `printf()` umožňuje formátovaný výstup. Ještě ji v této kapitole použijeme. Zde zobrazí pozdrav na monitoru.

Ve zdrojovém textu jsou ještě komentáře. Jsou vymezeny dvojicí `/*` a `*/`. Vše, co se mezi těmito znaky nachází, je komentář. Komentáře budeme v našich programech používat zejména pro zvýšení čitelnosti programu.

Zbývá nám příkaz preprocesoru, začleňující hlavičkový soubor: `#include <stdio.h>`. Zatím jen tolik, že obsahuje všechny údaje potřebné pro správné použití funkce `printf()`.

Tento zdrojový text musíme nyní přeložit překladačem jazyka C. Překladač musí mít k dispozici i začleněný hlavičkový soubor. Pokud jsme se nedopustili žádné chyby, získáme přeložený tvar. K němu musíme

připojit kód související s výstupem. Je uložen ve standardních knihovnách. Spojovací program se jmenuje linker. Často se volá příkazem link. Teprve po spojení modulů a připojení knihovních funkcí můžeme spustit proveditelný tvar našeho prvního zdrojového textu v jazyce C. Teď je jistě zřejmé, proč začínáme tak jednoduchým programem.

Jednoduchý vstup a výstup.

První program nám umožnil zvládnout překlad zdrojového textu. Teď si ukážeme některé číselné datové typy a jednoduchý vstup a výstup dat. Naším úkolem je načíst celočíselnou a racionální hodnotu, zobrazit výsledek jednoduchého početního výkonu, který s těmito hodnotami provedeme.

```
/*****************************/
/* SIMPLEIO.C                */
/* jednoduchy vstup a vystup  */
/*****************************/

#include <stdio.h>

int main(void)
{
 int i, j;
 float x, y;
 printf("zadej dve cela cisla: \n");
 scanf("%d %d", &i, &j);
 printf("zadej racionalni cislo: \n");
 scanf("%f", &x);
 printf("%5d + %5d = %5d\n", i, j, i+j);
 y = i * x;
 printf("%d * %f = %f\n", i, x, y);
 return 0;
} /* main */
```

```
zadej dve cela cisla:
4 6
zadej racionalni cislo:
8.9
    4 +     6 =    10
4 * 8.900000 = 35.599998
```

Začátek programu je stejný, jako v předchozím případu. V těle funkce `main()` dále vytvoříme dvě celočíselné `int` proměnné `i, j` a pak dvě racionální `float` proměnné `x, y`. Poté pomocí známého výstupu `printf()` zobrazíme výzvu. Na následujícím řádku pomocí formátovaného vstupu načteme dvě celočíselné hodnoty:

```
scanf("%d %d", &i, &j);
```

Hodnoty načítáme do proměnných `i` a `j`. Nesmíme před ně ovšem zapomenout umístit adresový operátor `&`. Ve formátovacím řetězci musíme pro celočíselnou hodnotu umístit `%d`. Obdobně čteme i racionální hodnotu, formát `%f`.

Při formátovaném výstupu používáme stejné formátovací symboly. Navíc můžeme přidat i další specifikace formátu. Podrobněji se formátům vstupu a výstupu budeme věnovat v kapitole *Vstup a výstup*. V příkladu vidíme i přiřazení výsledku do proměnné. Navíc se jedná o smíšený výraz. Jeden z operandů je typu `int`, druhý `float`. Výsledek je přiřazen proměnné typu `float`.

Možný chod programu máme zobrazen v rámečku. Číselné hodnoty, které zadáváme, můžeme oddělit nejen klávesou **Enter**, ale i mezerou či tabelátorem. Výledek násobení racionální hodnoty vidíme zaokrouhlen. Vzniklo omezeným rozsahem platných míst při uložení hodnoty ve vnitřním tvaru.

První seznámení s C máme za sebou. Další výklad povedeme systematičteji.

Konstanty, proměnné a deklarace.

V této kapitole se seznámíme s klíčovými slovy, identifikátory, komentáři. Poznáme rozdíl mezi konstantou a proměnnou a naučíme se nejen základní datové typy, ale i tvorbu nových typů dat.

Identifikátory, klíčová slova a komentáře.

Klíčová slova mají speciální význam pro překladač C. Žádný identifikátor nemůže mít ve fázi překladu stejné znění jako klíčové slovo. ANSI norma určuje následující klíčová slova:

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

Identifikátory jsou jména, která dáváme například proměnným, funkcím a typům. Identifikátor se musí lišit od kteréhokoliv klíčového slova. Nejvyšší počet znaků identifikátoru je implementačně závislý. ANSI říká, že interní identifikátor může být dlouhý 31 znaků, externí 6.

Identifikátor je tvořen posloupností alfanumerických znaků a podtržítka, přičemž musí být splněny následující podmínky:

- prvním symbolem smí být písmeno nebo podtržítko
- následuje libovolná kombinace písmen, číslic a podtržítek (nejvýše však do maximální délky identifikátoru - viz výše).

Jazyk C v identifikátorech rozlišuje malá a velká písmena. Následující identifikátory jsou tedy navzájem odlišné:

identifikator	Identifikator	IDENTIFIKATOR	IdEnTiFiKaToR

Komentář je část programu umístěná mezi dvojici párových symbolů `/*` a `*/`. Komentář může vypadat například takto:

```
/* Toto je komentář,
 a toto je jeho pokračování na druhém řádku. */
...
/*
  if (uk->chyba)
    {
     ts->pom_info++;
    }
  else
 nepripustim chybu, ladim stejne nasucho
*/
...
```

Komentáře obvykle umisťujeme do zdrojového textu z důvodu jeho lepší čitelnosti. Často popisujeme některé důležité vlastnosti zdrojového textu právě v komentáři. Komentářem si můžeme rovněž přechodně vypomáhat ve fázi tvorby a ladění programu.

Bílý znak (white space) je jeden z následujících symbolů: mezera, tabelátor, nový řádek, posun řádku, návrat vozíku, nová stránka a vertikální tabelátor. Bílé znaky spolu s operátory a oddělovači stojí mezi identifikátory, klíčovými slovy, řetězci a konstantami použitými ve zdrojovém textu. Překladač považuje rovněž komentář za bílý znak.

Základní typy dat

Základní typy dat dělíme na celočíselné, racionální, společně je nazýváme aritmetické datové typy, znaky a ukazatele. Celočíselné datové typy mohou obsahovat modifikátory *unsigned* respektive *signed*, čímž můžeme požadovat hodnoty příslušného typu *bez znaménka*, resp. *se znaménkem*. V céčku máme tedy k dispozici všechny potřebné základní typy i s jejich případnými modifikacemi. Jejich přehled spolu s paměťovými nároky a jejich českým významem následuje:

datový typ	počet bitů	význam
char, unsigned char, signed char	8	znak
short, unsigned short, signed short	16	krátké celé číslo
int, unsigned int, signed int	16 nebo 32	celé číslo
long, unsigned long, signed long	32	dlouhé celé číslo
enum	8|16|32	výčtový typ
float	32	racionální číslo
double	64	racionální číslo s dvojitou přesností
long double	80	
pointer	16|32	ukazatel

Pokud nás překvapí více možných hodnot ve sloupci počet bitů, pak vězme, že tuto hodnotu určuje jak překladač a případně u některých OS i paměťový model, tak skutečnost zásadnějšího významu. Totiž jedná-li se o překladač generující cílový kód šestnáctibitový či třicetidvoubitový (ovlivní typ `int`).

Nebudete-li si jisti rozsahem hodnot jednotlivých aritmetických typů, podívejte se do souboru `LIMITS.H` (pro celočíselné typy), respektive `FLOAT.H` (pro typy racionální). V nich najdete nejmenší případně i největší možné hodnoty, které příslušný překladač připouští.

Při našich prvních krocích vycházejme z následujících zásad, které jsou součástí ANSI C. Jak celočíselné, tak racionální typy je možno co do počtu obsazených bitů (a z toho vyplývajícího rozsahu možných hodnot) uspořádat takto:

`short`	`<=`	`Int`	`<=`	`long`
`float`	`<=`	`double`	`<=`	`double float`
a dále platí, že `char` vyžaduje 8 bitů.				

Na tomto místě věnujme několik slov konverzím aritmetických typů. Výraz složený z operandů různého aritmetického typu bude vyhodnocen a příslušné typové konverze proběhnou automaticky. Tím ovšem není řečeno, že výsledek bude takový, jaký očekáváme na základě znalostí naší školní matematiky. Konverzím aritmetických typů se podrobněji budeme věnovat později.

Konstanty a proměnné.

Většina objektů (entit), označených identifikátorem musí být deklarována dříve, než je použita. Konstanty, typy, proměnné a funkce k takovým objektům patří.

S konstantami a proměnnými jsou úzce spojeny dva pojmy, ***deklarace*** a ***definice***. Deklarací určujeme typ objektu. Deklarace nepřiděluje žádnou paměť. V místě definice definujeme hodnotu proměnné či posloupnost příkazů funkce. To je – proměnné se přidělí určité jméno a alokuje paměť.

Konstanty a proměnné mohou nabývat hodnot jak základních datových typů, tak typů uživatelsky definovaných. Přirozeně mohou tvořit i struktury typu pole. V této části se poli konstantních i proměnných vektorů nebudeme zabývat podrobněji, omezíme se na jejich definice.

Konstanty

Konstanty jsou symboly, reprezentující neměnnou číselnou nebo jinou hodnotu. Překladač jazyka jim přiřadí typ, který této hodnotě odpovídá. Z konstant odpovídajících si typů můžeme vytvářet *konstantní výrazy*. Tyto výrazy musí být regulární (zjednodušeně musí být snadno vyhodnotitelné během překladu). Nesmí obsahovat žádný z následujících operátorů (nejsou-li použity mezi operandy operátoru `sizeof`):

- přiřazení
- inkrementace a dekrementace
- funkční volání
- čárka

Konstantám s vhodně zvolenými identifikátory dáváme přednost například před přímým uvedením konstantní hodnoty jako meze cyklu nebo dimenze pole. Modifikace programu pak probíhá velmi snadno změnou hodnoty konstanty. Odpadá obtížné uvažování, zdali ta či jiná hodnota má býti modifikována či nikoliv. Konstanty mají rovněž určen typ. Tím je umožněna typová kontrola.

Konstanty definujeme po klíčovém slově `const` následovaném typem konstanty, jejím identifikátorem a po rovnítku její hodnotou ukončenou středníkem. Jedná-li se o vektor, následuje za identifikátorem dvojice hranatých závorek, zpravidla obsahující jeho dimenzi. První prvek pole má vždy index 0. Konstanty můžeme definovat třeba takto:

```
const int konstanta = 123;
const celociselna = -987;
const float CPlanck = 6.6256e-34;
const char male_a = 'a';
const char *retezec = "Konstantni retezec."
const float meze[2] = {-20, 60};
const char rimska_znaky[] = {'I', 'V', 'X', 'L', 'C', 'D', 'M'};
const int rimska_hodn[] = {1, 5, 10, 50, 100, 500, 1000};
```

Neuvedeme-li typ, jako v případě druhé konstanty, je implicitně chápán typ `int`. Můžeme definovat konstanty všech základních datových typů. Proměnná `meze` představuje dvouprvkové pole konstant typu `float`. Prvky pole nabývají se vzrůstajícím indexem hodnoty v pořadí, jak jsou v definici zapsány. Poslední dvě konstanty jsou pole konstantních hodnot. Uvedeme-li všechny požadované hodnoty na pravé straně definice, nemusíme v hranatých závorkách uvádět dimenzi pole.

Celočíselné konstanty

Celočíselné konstanty jsou tvořeny zápisem celého čísla. Mohou být zapsány v desítkové, osmičkové případně v šestnáctkové číselné soustavě. Nejprve tedy jednoduché zápisy:

`123 -987 255 4567 -4567`

které představují celočíselné konstanty. V pořadí druhá a poslední představují záporné hodnoty, ostatní jsou kladné. Nyní si uvedeme pravidla, podle nichž určujeme základ číselné soustavy konstanty:

- `0` (číslice nula) uvádí konstanty v osmičkové soustavě
- `0x` nebo `0X` (číslice nula následovaná znakem x) uvádí konstanty v šestnáctkové soustavě
- libovolná číslice s výjimkou nuly je součástí konstanty v desítkové soustavě (viz příklad výše)

Následuje několik desítkových konstant zapsaných ve třech možných číselných soustavách:

desítkový	**osmičkový**	**šestnáctkový**
123	0173	0x7b
-987	0176045	0xfc25
255	0377	0xff
4567	010727	0x11d7
-4567	0167051	0xee29

Jestliže prefix zápisu celočíselné konstanty určoval základ číselné soustavy, pak sufix, pokud je uveden, určuje celočíselný datový typ. A to následovně: `u` nebo `U` modifikuje typ na `unsigned`, zatímco `l` nebo `L` říká, že jde o typ `long`. Oba sufixy je možno spojit, takže například:

`123UL` je (desítková) konstanta 123 typu `unsigned long`.

Racionální konstanty

Racionální konstanty umožňují zapsat číselnou konstantu, která nemusí být celočíselná. Vnitřně je reprezentována ve tvaru, který obsahuje mantisu a exponent, obojí s případným znaménkem.
Implicitní typ racionální konstanty je `double`. Například `12.34e5`. Chceme-li, aby konstanta byla typu `long double`, připojíme k zápisu písmeno L, tedy například
`12.34e5L`.
Pro lepší představu dává následující tabulka přehled některých vlastností racionálních datových typů:

typ	**bitů**	**mantisa**	**exponent**	**rozsah absolutních hodnot (přibližně)**
float	32	24	8	$3.4\ 10^{-38}$ až $3.4\ 10^{+38}$
double	64	53	11	$1.7\ 10^{-308}$ až $1.7\ 10^{+308}$
long double	80	64	15	$3.4\ 10^{-4932}$ až $1.1\ 10^{+4932}$

Znakové konstanty

Znakové konstanty jsou tvořeny požadovaným znakem, respektive posloupností znaků, uzavřeným mezi apostrofy. Na následujícím řádku je zapsáno několik znakových konstant:

```
'a', 'A', 'Š', 'ň', '}', '#', '"'
```

První dvě nás jistě nepřekvapí, další dvě nemusí být nutně k dispozici na všech systémech (byť podpora národního prostředí je čím dál větší samozřejmostí). Další dva znaky zase nejsou k dispozici na standardních českých klávesnicích, ale céčko si bez nich představit nedovedeme. No a poslední znaková konstanta jsou uvozovky. Jimi si připravujeme následující otázku.

Jak zapíšeme znakovou konstantu apostrof? A co případně jiné speciální znaky (řídící symboly, znaky nenacházející se na klávesnici, ...). Zde si pomáháme symbolem opačné lomítko a nejméně jedním dalším znakem. Těmto posloupnostem říkáme *escape sequence*. Ty mohou být jednoduché, kdy opačné lomítko následuje jediný znak. Nebo následuje osmičkový či (po x) šestnáctkový kód znaku. Tak jsme schopni zadat i znak, který se na klávesnici nenachází, ale jehož kód je nám znám. Přehled escape sequencí obsahuje tabulka:

posloupnost	**jméno**	**Ctrl-znak**	**význam**
\a	Alert (Bell)	G	pípnutí
\b	Backspace	H	návrat o jeden znak
\f	Formfeed	L	nová stránka nebo obrazovka
\n	Newline	J	přesun na začátek nového řádku
\r	Carriage return	M	přesun na začátek aktuálního řádku
\t	Horizontal tab	I	přesun na následující tabelační pozici
\v	Vertical tab	K	stanovený přesun dolů
\\	Backslash		obrácené lomítko
\'	Single quote		apostrof
\"	Double quote		uvozovky
\?	Question mark		otazník
\OOO			ASCII znak zadaný jako osmičková hodnota
\xHHH			ASCII znak zadaný jako šestnáctková hodnota

Konstantní řetězce

Konstantní řetězce jsou na rozdíl od znakových konstant tvořeny více než jedním znakem, zpravidla slovem či větou (tedy znakovou posloupností, řetězcem). Začátek a konec řetězce jsou vymezeny úvozovkami. Následující řetězce jsou úmyslně psány *cesky*:

```
"dve slova" "Cela tato veta tvori jeden retezec." "a" "Ahoj!"
```

Na předposlední řetězec musíme upozornit. Jedná se o řetězcovou konstantu tvořenou písmenem `a`, tedy řetězcem délky jeden znak. Nesmíme ji zaměňovat se znakovou konstantou. Ta je vymezena dvěma apostrofy. Navíc, znaková konstanta, bez ohledu na její zápis, představuje jeden jediný znak, zatímco řetězcová konstanta může mít i značnou délku.

Pokud konstantní řetězec obsahuje speciální symboly, zapisujeme je obdobně, jako jsme to činili u znakových konstant. Například:

```
"\tPo uvodnim tabelatoru prejdeme na novy radek\na pipneme\a."
```

Použili jsme jednoduché escape sequence. Pokud bychom po nich udělali mezeru, byla by tato rovněž obsažena i ve výsledném řetězci. A to nechceme. A ještě jedna ukázka delší řetězcové konstanty:

```
"Tato delsi retezcova konstanta obsahuje opacne lomitko \\, \
a pokracuje na dalsim radku od jeho zacatku. " "Navic je takto \
rozdelena."
"Jednodussi pokracovani na dalsim radku "
   "vypada takto, pak nemusime zacinat hned na zacatku."
```

Prvni dva řádky jsou ukončeny opačným lomítkem. Tím je řečeno, že řetězec pokračuje na následujícím řádku. Jednodušší je ovšem konstrukce, při níž ukončit řetězec uvozovkami, ukončíme řádek a pokračujeme novým řetězcem kdekoliv na novém řádku. Překladač totiž dva řetězce, oddělené pouze bílými znaky, spojí v řetězec jeden.

Proměnné

Proměnné jsou paměťová místa přístupná prostřednictvím identifikátoru. Hodnotu proměnných můžeme během výpočtu měnit. Tím se proměnné zásadně odlišují od konstant, které mají po celou dobu chodu programu hodnotu neměnnou - konstantní.

Proměnné deklarujeme uvedením datového typu, jež je následován identifikátorem, nebo seznamem identifikátorů, navzájem oddělených čárkami. Deklarace končí středníkem. Současně s deklarací proměnné můžeme, ale nemusíme, definovat i její počáteční hodnotu:

```
int a, b, c, pocet = 0;
float x, prumer = 0.0, odchylka = 0.0;
float y;
```

ANSI C považuje konstanty za proměnné s neměnnou hodnotou. V závislosti na použitém překladači může být tato hodnota umístěna ve výrazu přímo. Pak jí nemusí být vyhrazeno paměťové místo tak, jak by bylo vyhrazeno pro proměnnou.

Ukazatelé.

Ukazatel představuje adresu paměťového místa. Jeho hodnota říká, kde je uložen nějaký objekt. Součástí deklarace ukazatele je i informace o typu dat, které jsou na získané adrese očekávány.

Obvyklou chybou začátečníků je použití ukazatele bez jeho předchozí inicializace (alokace paměťového místa). Neinicializovaný ukazatel může ukazovat na kritické oblast paměťi a jeho použití může vést i k havárii systému. Ale o ukazatelích až pozdeji.

Operátory a výrazy

Operand, operátor, výraz.

Zapíšeme-li v matematice například `a + b`, hovoříme o **výrazu**. Ten má dva **operandy** `a` a `b` a jeden **operátor** + . Jedná se sice o výraz velmi jednoduchý, nicméně nám umožnil zopakování potřebných termínů.

Rozdělení operátorů.

Operátory rozdělujeme podle počtu operandů (arity) na operátory **unární**, **binární** a **ternální**. Binární operátory jsou aritmetické, relační, logické, bitové a operátory přiřazení a posuvu. Aritmetické operátory jsou aditivní a multiplikativní. Operátory mají svou prioritu a asociativitu. Priorita určuje, že například násobení se vyhodnotí dříve, než třeba sčítání. Asociativita říká, vyhodnocuje-li se výraz zleva doprava, nebo naopak.

Operátory rovněž dělíme podle pozice jejich zápisu vzhledem k operandu(-ům). Takto rozlišujeme operátory prefixové, infixové a postfixové. Operátory v jednotlivých případech zapisujeme před operandy, mezi operandy, respektive za operandy.

Poznamenejme, že v C uplatníme všechny zmíněné varianty operátorů. Základní přehled operátorů jazyka C, rozlišených podle arity, následuje:

Unární operátory:

`+, -`	aritmetické plus a mínus
`&`	reference (získání adresy objektu)
`*`	dereference (získání objektu dle adresy)
`!`	logická negace
`~`	bitová negace
`++, --`	inkrementace resp. dekrementace hodnoty, prefixový i postfixový zápis
`(typ)`	přetypování na typ uvedený v závorkách
`sizeof`	operátor pro získání délky objektu nebo typu

Binární operátory:

`=`	přiřazení, možná je i kombinace s jinými operátory, např. `+=, -=, *=, /=, <<=, ^=`
`+`	sčítání
`-`	odčítání
`*`	násobení
`/`	dělení
`%`	zbytek po celočíselném dělení (modulo)
`<<, >>`	bitový posun vlevo resp. vpravo
`&`	bitový součin (and)
`|`	bitový součet (or)
`^`	bitový vylučovací součet (xor)
`&&`	logický součin (and)
`||`	logický součet (or)
`.`	tečka, přímý přístup ke členu struktury
`->`	nepřímý přístup ke členu struktury
`,`	čárka, oddělení výrazů
`<`	menší než
`>`	větší než
`<=`	menší nebo rovno
`>=`	větší nebo rovno
`==`	rovnost
`!=`	nerovnost

Ternální operátor:

`? :`	podmíněný operátor

Při podrobnějším pohledu na přehled operátorů záhy objevíme některé z operátorů, které jsou uvedeny jako unární i binární současně. Příkladem uveďme −, které může vystupovat jako unární mínus i jako binární operátor odčítání.

Operátory s uvedením priority (v tabulce jsou řazeny sestupně od priority nejvyšší k prioritě nejnižší) a asociativity:

operátor	**typ operátoru**	**asociativita**
`()  []  ->  .`	výraz	zleva doprava
`! ~ ++ -- + -` přetypování `* & sizeof`	unární	zprava doleva
`* / %`	násobení	zleva doprava
`+ -`	sčítání	zleva doprava
`<< >>`	bitového posunu	zleva doprava
`< > <= >=`	relační	zleva doprava
`== !=`	rovnosti	zleva doprava
`&`	bitové AND	zleva doprava
`^`	bitové vylučovací OR (XOR)	zleva doprava
`|`	bitové OR	zleva doprava
`&&`	logické AND	zleva doprava
`||`	logické OR	zleva doprava
`?:`	podmíněné vyhodnocení	zprava doleva
`= *= /= %= += -= <<= >>= &= |= ^=`	přiřazení a přiřazení s výpočtem	zprava doleva
`,`	postupné vyhodnocení	zleva doprava

Unární operátory jsou prefixové s možným postfixovým použitím dekrementace a inkrementace. Binární operátory jsou infixové.

Operátory jsou rovněž `[]`, `()` ohraničující indexy resp. argumenty a `#`, `##`, které zpracovává již preprocesor. Preprocesoru v tomto textu věnujeme celou kapitolu. Užitečným operátorem je `sizeof`, který v průběhu překladu vyhodnotí paměťové nároky svého argumentu. Tento operátor je nezbytný zejména při dynamické alokaci paměti, případně při operacích čtení/zápis z binárních souborů.

Operátor přiřazení, l-hodnota a p-hodnota.

Výrazy, jak již víme, jsou tvořeny posloupností operátorů a operandů. Výraz předepisuje výpočet adresy nebo hodnoty. Upravíme-li například známý vztah

$$c = \sqrt{a^2 + b^2}$$

do syntakticky správného zápisu v jazyce C

```
c = sqrt(a*a + b*b);
```

můžeme zřetelně ukázat některé významné vlastnosti ***operátoru přiřazení*** **=**. Na pravé straně operátoru přiřazení se nachází výraz, jehož vyhodnocením získáme hodnotu tohoto výrazu. Ovšem na levé straně se nachází výraz (v našem případě je to "jen" proměnná), jehož vyhodnocením získáme adresu. Na tuto adresu, představující začátek paměťového místa pro umístění hodnoty proměnné `c`, je umístěna hodnota z pravé strany přiřazovacího operátoru.

Ještě než si zadefinujeme zmíněné pojmy, nesmíme zapomenout na důležitou skutečnost. Výsledkem výrazu přiřazení je hodnota. Co to znamená? Například možnost elegantně řešit inicializaci více proměnných stejnou hodnotou, například

```
int a, b, c;
```

a = b = c = -1;

Nezapomínejme, že přiřazovací operátor je asociativní zprava doleva. Nejprve se tedy vyhodnotí `c = -1`, výsledkem je hodnota `-1`, ta tvoří pravou stranu přiřazení `b =`, jehož výsledkem je opět `-1`. A jak se uvedená hodnota dostane do proměnné `a` není jistě třeba popisovat. Vraťme se však k nastíněným pojmům.

Adresový výraz (*lvalue* - **l-hodnota**) je výraz, jehož výpočtem se získá adresa v paměti. Například: `j=i*2`. Pak `j` je l-hodnotou a výraz `i*2` l-hodnotou není. Stručně řečeno l-hodnota je to, co může být na levé straně přiřazení.

Hodnotový výraz (*rvalue* - **p-hodnota**) je výraz, jehož výpočtem se získá hodnota jistého typu. Typ je jednoznačně určen typem operandů.

Aritmetické operátory - aditivní a multiplikativní.

Aritmetické operátory `+ - * / %` představují základní matematické operace sčítání, odčítání, násobení, dělení a zbytku po (celočíselném) dělení.
Nejlepší ukázkou bude jistě příklad.

```
/*****************************************************************/
/* program op_int01.c                                            */
/* celociselne nasobeni, deleni, zbytek po deleni                */
/*****************************************************************/

#include <stdio.h>

int main()
{
 int o1 = 123, o2 = 456, o3 = 295, v1, v2, v3;
 int c1 = 20000, c2 = 20001, vc;
 v1 = o1 * o2;
 v2 = o3 / 2;
 v3 = o3 % 2;
 printf("%d * %d = %d\n", o1, o2, v1);
 printf("%d / %d = %d\n", o3, 2, v2);
 printf("%d %% %d = %d\n", o3, 2, v3);
 vc = c1 + c2;
 printf("\nnyni pozor:\n\t");
 printf("%d + %d = %d\n", c1, c2, vc);

 return 0;
}
```

Příklad ukazuje nejen inicializaci hodnot proměnných "operandů" `o1` `o2` a `o3`, ale po prvních očekávaných výsledcích i neočekávané hodnoty. Ty jsou způsobeny faktem aritmetického přetečení.
`Podívejme se nyní na výsledky aritmetických operací, v nich argumenty jsou opět celočíselné, ale levá strana je racionálního typu.`

```
/*****************************************************************/
/* program op_int_f.c                                            */
/* zakladni aritmeticke operace a prirazeni vysledky jsou sice i float*/
/* ale vypocty jsou provadeny jako int a teprve pote prevedeny   */
/*****************************************************************/

#include <stdio.h>

int main()
{
 int i, j;
 float r, x;
 j = i = 5;
 j *= i;
 r = j / 3;
 x = j * 3;
 printf("i=%d\tj=%d\tr=%f\tx=%f\n", i, j, r, x);

 return 0;
}
```

```
/* vystup BC31
i=5 j=25 r=8.000000 x=75.000000
*/
```

Rovněž v tomto příkladu vidíme, že výpočet probíhá s hodnotami typů podle zmíněných pravidel a teprve poté je získaná p-hodnota konvertována do typu odpovídajícího l-hodnotě. Proto podíl `25/3` dává `8.0` a nikoliv `8.333`.

Stejným způsobem probíhají aritmetické operace s racionálními hodnotami.

Chceme-li změnit pořadí vyhodnocení jednotlivých částí výrazu, použijeme k tomuto "pozměnění priority" kulatých závorek.

Logické operátory.

Logické operátory představují dvě hodnoty, **pravda** a **nepravda**. ANSI norma C říká, že hodnota nepravda je představována **0** (nulou), zatímco pravda **1** (jedničkou). Ve druhém případě se ovšem jedná o doporučení. Neboť užívaným anachronismem je považovat jakoukoliv nenulovou hodnotu za pravdu.

Logické operátory jsou **`&& || !`**, postupně **and or** a **not**. Provádí výpočet logických výrazů tvořených jejich operandy. Pravidla pro určení výsledku známe z Booleovy algebry. Logické výrazy často obsahují i stanovení (a ověření) podmínek tvořených relačními operátory.

Relační operátory.

Relační operátory jsou **`<, >, <=, >=, ==, !=`**. Pořadě menší, větší, menší nebo rovno, větší nebo rovno, rovno a nerovno. Jsou definovány pro operandy všech základních datových typů. Jejich výsledkem jsou logické hodnoty pravda a nepravda tak, jak jsou popsány v předchozím odstavci.

Bitové operátory.

Jak sám název napovídá, umožňují provádět operace nad jednotlivými bity. Tuto možnost zdaleka nemají všechny programovací jazyky označované jako *vyšší*. Jazyk C jí oplývá zejména proto, že byl vytvořen jako nástoj systémového programátora (OS Unix). Použití bitových operátorů vyžaduje znalosti o uložení bitů v paměti, způsobu kódování čísel, ...

Bitové operátory jsou: **`<< >> & | ~ ^`** , tedy posun vlevo, posun vpravo, and, or, not a xor. Bitové operace jsou možné pouze s celočíselnými hodnotami. Podívejme se nyní na jednotlivé zástupce bitových operátorů.

Při *bitovém posunu vlevo (vpravo)* `<<` , (`>>`) se jednotlivé bity posouvají vlevo (vpravo), tedy do pozice s (binárně) vyšším (nižším) řádem. Na nejpravější (nejlevější) posunem vytvořenou pozici je umístěna nula. Posuny ovšem probíhají aritmeticky. To znamená, že uvedené pravidlo neplatí pro posun vpravo hodnoty celočíselného typu se znaménkem. V takovém případě se nejvyšší bit (znaménkový), zachovává. Takto se při posunu doplňuje do bitového řetězce nový bit. Naopak před posunem nejlevější (nejpravější) bit je odeslán do "říše zapomění".

Bitový posun o jeden (binární) řád vpravo, respektive vlevo, má stejný význam, jako celočíselné dělení, respektive násobení, dvěma. Je-li bitový posun o více než jeden řád, jedná se o násobení (dělění) příslušnou mocninou dvou.

Bitové and `&` , *or* `|`, a *xor* `^` provádí příslušnou binární operaci s každým párem odpovídajících si bitů. Výsledek je umístěn do pozice stejného binárního řádu výsledku. Výsledky operací nad jednotlivými bity jsou stejné, jako v Booleově algebře. *Bitové not* ~ je operátorem unárním, provádí negaci každého bitu v bitovém řetězci jediného operandu. Tomuto operátoru se často říká bitový doplňek.

```
/*************************************************************/
/* program op_bit01.c                                        */
/* ukazuje bitove posuny, a zakladni bitove operace and, or, xor   */
/* a bitovy doplnek                                          */
/*************************************************************/

#include <stdio.h>

int main()
{
 printf("1 << 1 = \t%d\t%#x\n", 1 << 1, 1 << 1);
 printf("1 << 7 = \t%d\t%#x\n", 1 << 7, 1 << 7);

 printf("-1 >> 1 = \t%d\t%#x\n", -1 >> 1, -1 >> 1);
 printf("1024 >> 9 = \t%d\t%#x\n", 1024 >> 9, 1024 >> 9);

 printf("13 & 6 = \t%d\t%#x\n", 13 & 6, 13 & 6);
 printf("13 | 6 = \t%d\t%#x\n", 13 | 6, 13 | 6);
```

```
 printf("13 ^ 6 = \t%d\t%#x\n", 13 ^ 6, 13 ^ 6);

 printf("2 & 1 = \t%d\t%#x\n", 2 & 1, 2 & 1);
 printf("2 | 1 = \t%d\t%#x\n", 2 | 1, 2 | 1);
 printf("2 ^ 1 = \t%d\t%#x\n", 2 ^ 1, 2 ^ 1);

 return 0;
}
```

```
/* BC31 - 16-ti bitovy kod
1 << 1 =        2       0x2
1 << 7 =        128     0x80
-1 >> 1 =       -1      0xffff
1024 >> 9 =     2       0x2
13 & 6 =        4       0x4
13 | 6 =        15      0xf
13 ^ 6 =        11      0xb
2 & 1 =         0       0
2 | 1 =         3       0x3
2 ^ 1 =         3       0x3
*/
```

Adresový operátor.

Adresový operátor **&** je unární. Jak již název adresový operátor napovídá, umožňuje získat adresu objektu, na nějž je aplikován. Adresu objektu můžeme použít v nejrůznějších situacích, obvykle je to ale v souvislosti s ukazateli. Bez tohoto operátoru bychom nebyli schopni pracovat se soubory a ani standardní vstup bychom nebyli schopni číst jinak, než po znacích. Takto například můžeme přečíst hodnoty dvou proměnných jedinou funkcí pro formátovaný vstup:

```
int i;
float f;
scanf("%d %f", &i, &f);
```

Podmíněný operátor.

Podmíněný operátor **?** je poměrně nezvyklý. Proto bude vhodné, objasníme-li si jeho význam. Mějme například výpočet, který potřebujeme provést, v závislosti na nějaké podmínce, jednou ze dvou variant (pochopitelně odlišných). Výsledek výpočtu přiřazujeme vždy stejné proměnné. Pokud navíc je část výrazu, popisující výpočet obou variant, shodná, jedná se o typický příklad využití podmíněného výrazu.

Buďme však raději konkrétnější. Chceme-li vypočíst absolutní hodnotu nějakého čísla, použijeme velmi pravděpodobně podmíněný operátor. Výpis zdrojového textu takového výpočtu následuje:

```
/*********************************/
/* soubor op_cond.c              */
/* ukazuje pouziti podmineneho   */
/* operatoru - absolutni hodnota */
/*********************************/

#include <stdio.h>

int main(void)
{
 int i, abs_i;

 printf("\nZadej cele cislo: ");
 scanf("%d", &i);

 abs_i = (i < 0) ? -i : i;
 printf("abs(%d) = %d\n", i, abs_i);

 return 0;
}
```

Další použití (ternálního) podmíněného operátoru ukazuje následující řádek.

```
return((znak == 0x0) ? getch() + 0x100 : znak);
```

Tento řádek zajišťuje případné načtení a překódování libovolné stisknuté klávesy na tzv. rozšířené klávesnici IBM PC AT. Základní klávesy této klávesnice produkují kód odpovídající jejich pozici v ASCII tabulce.

Rozšířené klávesy produkují dvojici celočíselných kódů, z nichž první je nula. Než si popíšeme jeho činnost, doplňme i předcházející příkaz pro načtení znaku:

```
znak = getch();
return((znak == 0x0) ? getch() + 0x100 : znak);
```

Pomocí podmínky `(znak == 0x0)` otestujeme, jednalo-li se o rozšířenou klávesu. Jestliže ano, je proveden první příkaz následující za podmíněným operátorem, `getch() + 0x100` . Je jím načten tzv. scan kód rozšířené klávesy, který je dále zvětšen o hodnotu 100hex. Jestliže podmínka splněna nebyla, je vykonán příkaz za dvojtečkou. Návratovou hodnotou je pak neupravená hodnota, načtená do proměnné `znak` těsně před ternálním operátorem.

Operátor čárka.

Čárka je operátorem postupného vyhodnocení. Má nejnižší prioritu ze všech operátorů a vyhodnocuje se zleva doprava. Čárkou můžeme oddělit jednotlivé výrazy v místě, kde je očekáván jediný výraz. Například v cyklu `for` :

```
for (i = 0, j = 0; i < MAX; i++, j++)
```

Přetypování výrazu.

Jazyk C nám umožňuje přetypovat výrazy podle naší potřeby. Přetypování má přirozeně svá omezení, ta však často plně odpovídají zdravému rozumu. Těžko se například vyskytne potřeba přetypovat znak na typ ukazatel na double.

Úvodní úvaha postrádá konkrétní ukázku, z níž by vyplýval syntaktický zápis přetypování. Připomeňme si příklad `op_int_f.c`, v němž se vyskytoval příkaz

```
r = j / 3; ,
```

který byl ovšem vyhodnocen celočíselně a teprve poté konvertován na `float`. Chceme-li, aby již podíl proběhl v racionálním oboru, musíme pravou stranu upravit. S přetypováním může vypadat pravá strana takto:

```
r = (float) j / 3;
```

Přetypování provádíme tak, že před hodnotu, kterou chceme přetypovat, napíšeme typ, který chceme získat, v kulatých závorkách. Syntakticky tedy zapíšeme přetypování takto:

```
(type) expression
```

Řízení chodu programu

V dosavadním výkladu jsme se dozvěděli, že se program skládá z funkce `main()` a z příkazů, které tato funkce obsahuje. Své intuitivní představy o programu rozšíříme nejen o případné další funkce, ale detailněji se podíváme i na obsah funkcí. Tělo funkce obsahuje řadu příkazů. Naším cílem je provádět právě ty příkazy, které odpovídají zvolenému záměru. Výběr z příkazů je určen stavem dosavadního běhu programu, vstupními údaji a řídícími strukturami, které jsme použili.

Program, provádějící příkazy v pevném a neměnném pořadí, které odpovídá jejich umístění ve zdrojovém textu a navíc bez možnosti jejich výběru, jistě není naším ideálem. A to bez ohledu na zvolený vyšší programovací jazyk. Cílem této kapitoly je proto seznámení s řídícími strukturami.

Před podrobným přístupem nejprve uveďme přehledně příkazy, které máme v C k dispozici:

- výrazový příkaz
- blok
- podmíněný příkaz
- přepínač
- cyklus
- skok

Výrazový příkaz

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

Prázdný příkaz

Prázdný příkaz je výrazový příkaz, v němž není výrazová část. Tato konstrukce není tak nesmyslná, jak se na první pohled může zdát. Dává nám v některých případech možnost umístit nadbytečný středník **;** do zdrojového textu. Například i za vnořený blok. Protože se o prázdném příkazu můžeme těžko dozvědět něco dalšího, podívejme se raději na další příkazy jazyka C.

Bloky.

Všude v C, kde se může vyskytovat příkaz, se může vyskytovat i ***složený příkaz***. Složený příkaz je posloupností příkazů. Konstrukce, která složený příkaz vymezuje, začíná levou a končí pravou složenou závorkou **{ }**, a nazývá se ***blok***. V bloku můžeme, kromě již zmíněné realizace složeného příkazu, provádět *lokální deklarace a definice*. Ty ovšem pouze na začátku bloku. Jejich platnost je omezena na blok a případné další vnořené bloky. Vnořený blok nemusí být ukončen středníkem. Představuje složený příkaz a jeho konec je jasně určen. Není na škodu si uvědomit, že tělo každé funkce je blokem. Proto jsme mohli v těle funkce `main()` definovat a používat lokální proměnné. Syntakticky můžeme blok popsat následovně:

```
{
 [declaration_1[; declaration_2 ... ];]
 [statement_1 [; statement_2 ... ] ]
}
```

Blok tedy může obsahovat žádnou, jednu či více deklarací. Pokud deklaraci obsahuje, musí být od další části bloku oddělena středníkem. Dále blok může obsahovat jednotlivé příkazy, rovněž oddělené středníkem. Povšimněme si, že poslední z příkazů nemusí být od uzavírací složené závorky bloku středníkem oddělen.

Oblast platnosti identifikátoru

Identifikátor, který deklarujeme či definujeme, si ponechává svou platnost v programu, v němž je deklarován či definován. Jeho jméno je v tomto rozsahu viditelné, není-li maskováno:

- Na úrovni souboru je rozsah platnosti deklarace vymezena místem, kde je deklarace dokončena, a koncem překládané jednotky.
- Deklarace na úrovni argumentu funkce má rozsah od místa deklarace argumentu v rámci definice funkce až do ukončení vnějšího bloku definice funkce. Pokud se nejedná o definici funkce, končí rozsah deklarace argumentu s deklarací funkce.
- V rámci bloku je deklarace platná v rozsahu jejího dokončení až do konce bloku.

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

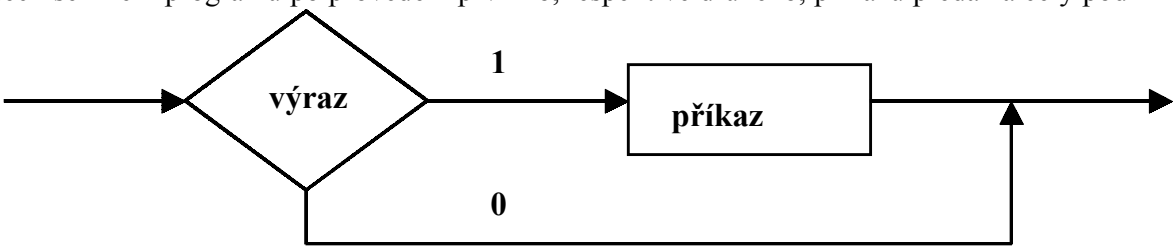
Podmíněný příkaz `if-else`.

Operátor podmíněného výrazu `? :` používáme pro výběr části výrazu. Pro výběr z příkazů máme k dispozici podmíněný příkaz. Mohli bychom říci, že se jedná o příkazy dva. Jejich syntaktický zápis je následující

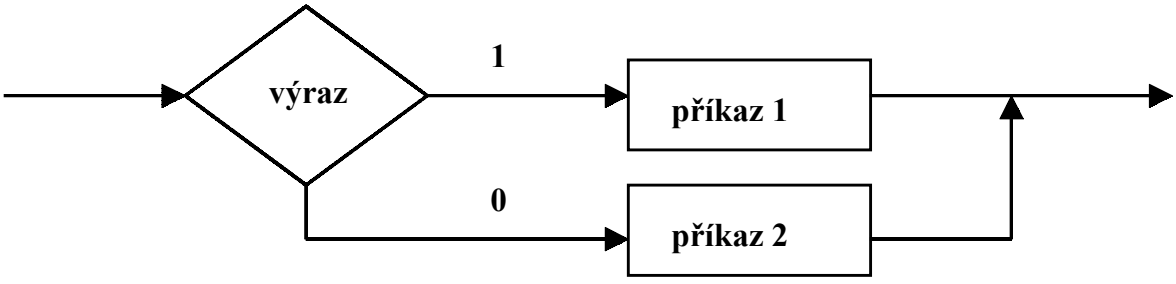
```
if ( <expression> ) <statement1>;

if ( <expression> ) <statement1>;
else <statement2>;
```

Význam příkazu `if` je následující. Po vyhodnocení výrazu `expression` (musí být v závorkách) se v případě jeho nenulové hodnoty provede příkaz `statement1`. Po jeho provedení pokračuje program za tímto příkazem. V případě nulového výsledku výrazu se řízení programu předá bezprostředně za podmíněný příkaz. Jinak řečeno se příkaz `statement1` přeskočí.

Příkaz `if` můžeme použít i s variantou `else`. Sémantika takového příkazu `if-else` je v první části totožná se samotným `if`. Je-li výslednou hodnotou výrazu `expression` jedna (nenulová hodnota), provede se příkaz `statement1`. V opačném případě, kdy výsledkem je nula, se provede příkaz `statement2`. V obou případech se řízení programu po provedení prvního, respektive druhého, příkazu předá za celý podmíněný výraz.



Příkaz **if**



Příkaz **if -else**

Někdy se výklad sémantiky předkládá způsobem, kdy se nulové hodnotě říká nepravda a nenulové (jedničce) pravda. Jak ostatně známe z logických výrazů. Pak lze ve druhé variantě říci, že je-li podmínka splněna, vykoná se první příkaz `statement1`, jinak příkaz druhý `statement2`.

Zdůrazněme, že oba příkazy jsou ukončeny středníkem.

Z předchozích odstavců víme, že místo příkazu může být umístěn blok. V takovém případě je jasně příkaz vymezen a **středník za blokem před** `else` **nepíšeme**!

Příklad nám ukáže použití podmíněného příkazu `if-else`. Máme vypočíst a zobrazit podíl dvou zadaných racionálních čísel. Pro ošetření dělení nulou využijeme podmíněný příkaz.

```
/*******************************/
/* Příkaz if-else if-else1.c   */
/*******************************/

#include <stdio.h>

int main(void)
{
 float a, b;
 puts("Zadej dve racionalni cisla:");
 scanf("%f %f", &a, &b);

 if (b == 0.0)
       printf("\b\nNulou delit nelze!\n");
 else
 {
       float podil;
       podil = a / b;
       printf("Jejich podil je: %10.2f\n", podil);
 }
 return 0;
}
```

Jako první příkaz po testu `if` je jednoduchý výstup `printf()`. Je ukončen středníkem a následuje klíčové slovo `else`. Příkaz v této části je tvořen blokem. Za blokem středník není. Pokud by tam byl, byl by chápán jako prázdný příkaz. Tak je interpretován i středník za posledním příkazem v bloku.

Další možný tvar podmíněného příkazu je umístění příkazu `if` jako jednoho z příkazů `if-else`. Zpravidla se umisťuje na místo druhého příkazu. Výsledná konstrukce bývá často nazývána `if-else-if`:

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

Často využívanou vlastností této konstrukce je skutečnost, že se provede právě jeden z příkazů. Pokud není poslední příkaz bez podmínky `statementN+1` uveden, nemusí se provést žádný. Jinak řečeno, provede se nejvýše jeden.

Zapamatujme si dobře tuto konstrukci. Po příkladu ji budeme moci porovnat s konstrukcí zvanou přepínač.

Naším dalším úkolem je programově ošetřit, jaký alfanumerický znak byl zadán. Případně vydat zprávu o zadání znaku jiného. Protože je jisté, že zadaný znak patří právě do jedné z tříd malá písmena, velká písmena, číslice a jiné, použijeme konstrukci podmíněného příkazu. Při použití `if-else` by pro každou třídu byla znovu testována příslušnost načteného znaku. Bez ohledu, zdali znak již některou z předchozích podmínek splnil. Použitá konstrukce `if-else-if` případné nadbytečné testy odstraní. Vždy bude vybrán právě jednen z příkazů. Připomeňme si rovněž logický výraz, který tvoří podmínku

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

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

```
/*******************************/
/* Příkaz if-else if-else2.c    */
/*******************************/

#include <stdio.h>

int main(void)
{
 char znak;
 printf("Zadej alfanumericky znak:");
 scanf("%c", &znak);

 printf("\nZadal jsi ");
 if ((znak >= 'a') && (znak <= 'z'))
       printf("male pismeno\n");
 else if ((znak >= 'A') && (znak <= 'Z'))
       printf("velke pismeno\n");
 else if ((znak >= '0') && (znak <= '9'))
       printf("cislici\n");
 else
       printf("\nNezadal jsi alfanumericky znak!\n");

 return 0;
}
```

Přepínač

Přepínač slouží k rozdělení posloupnosti příkazů na části, následné vybrání a provedení některé, či některých z nich. Pokud nám tato formulace přepínač příliš nepřiblížila, pokusme se jinak.

Přepínač slouží k větvení výpočtu podle hodnoty celočíselného výrazu. Syntakticky zapsaný příkaz přepínače má tento tvar:

```
switch ( <expression> ) <statement>
case <constant expression> :
default :
```

Syntaktickou definici přepínače ovšem musíme upřesnit. Po klíčovém slově `switch` následuje celočíselný výraz `expression` uzavřený v závorkách. Za ním je příkaz `statement`, zpravidla tvořený blokem. Blok představuje posloupnost příkazů, v níž mohou být umístěna návěští, jejichž syntaxi vidíme na druhé řádce definice.

Řídící příkaz tvoří celočíselný konstantní výraz `constant expression` uvozený klíčovým slovem `case` a ukončený dvoutečkou `:`. Jedno z návěští může být klíčovým slovem `default`, přirozeně ukončeným dvoutečkou `:`.

Sémantika přepínače je poměrně složitá. Popíšeme si jednotlivá pravidla, jimiž se řídí:

Program vyhodnotí konstantní výraz a jeho hodnotu porovnává s každým z `case` návěští přepínače. Návěští `case` může být obsaženo uvnitř jiných příkazů (v rámci přepínače), kromě případného vnořeného přepínače.

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

Nastane-li shoda hodnoty `case` návěští s hodnotou `switch` výrazu `expression`, je přeneseno řízení programu na toto návěští. Jinak je řízení přeneseno na návěští `default` v rámci příslušného přepínače. Pro návěští `default` platí stejné zásady jako pro jiná návěští.

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

Pokud program při provádění přepínače vykoná příkaz `break`, bude řízení přeneseno na příkaz následující za přepínačem. Příkaz `break` může být v rámci přepínače umístěn v libovolných příkazech, kromě případných vnořených `do`, `for`, `switch` nebo `while` příkazech.

Přepínač `switch` může mít mnoho forem. Podívejme se na příklad, kdy naším úkolem je podat informaci o tom, jaká hodnota padla při simulovaném hodu kostky. Místo kostky použijeme generátor pseudonáhodných čísel

(dále jim budeme říkat jen náhodná). Protože takovou elektronickou kostku budeme používat i v příkladu v následujícím odstavci, věnovaném cyklům, probereme si funkce s ní spojené podrobněji.

Pomocí funkce `srand()` nastavíme posloupnost náhodných čísel. Jako počáteční hodnotu nemůžeme zvolit konstantu, neboť pak by byla generovaná posloupnost náhodných veličin vždy stejná. Proto potřebujeme hodnotu, která bude při každém spuštění programu jiná. Takto vhodně se mění například čas. Proto použijeme jako argument návratovou hodnotu funkce `time()`. Jiné záměry s funkcí `time()` nemáme, proto použijeme jako její argument `NULL`. Nesmíme jen zapomenout čas správně přetypovat. A generátor máme připraven:

```
srand((unsigned) time(NULL) );
```

Teď musíme umět kostkou házet. Při volání funkce `rand()` dostáváme náhodné celé číslo v rozsahu 0 až `RAND_MAX`. 16-ti bitové BC31 definuje tuto hodnotu jako `MAXINT`, konkrétně 32767. Pro naši potřebu stačí číslo v rozsahu 0 až 5, k němuž přičteme jedničku. Tuto práci odvede operace modulo šesti.

```
switch (rand() % 6 + 1)
```

Získaná hodnota 1 až 6 je výrazem přepínače `switch`. V závislosti na této hodnotě se řízení programu přenese na konstantní návěští. Problém je v tom, že pokud bychom takto označenou posloupnost příkazů neukončili příkazem `break`, pokračoval by od návěští program dále, až po konec přepínače. I této vlastnosti se dá využít. Ukázku vidíme ve spojeném hlášení pro hodnoty 2 a 3. Umístění návěští `default` v přepínači je libovolné. Zpravidla se píše na konec, což je poměrně zakořeněný zvyk. Budeme jej rovněž dodržovat.

```
/*******************************/
/* Příkaz switch. switch-1.c    */
/*******************************/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(void)
{
 char *s;

 printf("\nHazim kostkou...\n");
 srand((unsigned) time(NULL) );

 switch (rand() % 6 + 1)
       {
        case 1: s = "jednicka";
                break;
        case 2:
        case 3: s = "dvojka nebo trojka";
                break;
        case 4: s = "ctyrka";
                break;
        case 5: s = "petka";
                break;
        default: s = "sestka";
                break;
        }
 printf("\nPadla %s.\n\n", s);

 return 0;
}
```

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

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

Cykly

Cyklus je část programu, která je v závislosti na podmínce prováděna opakovaně. U cyklu obvykle rozlišujeme ***řídící podmínku cyklu*** a ***tělo cyklu***. Řídící podmínka cyklu určuje, bude-li provedeno tělo cyklu, či bude-li řízení předáno za příkaz cyklu. Tělo cyklu je příkaz, zpravidla v podobě bloku.

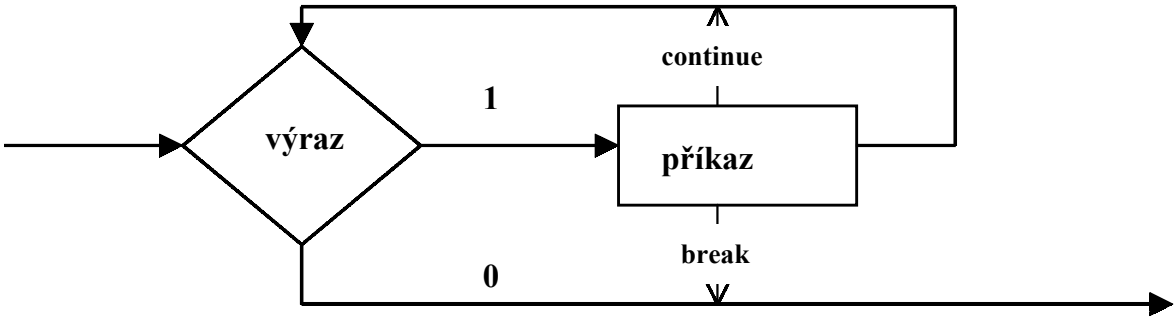
Cykly můžeme rozdělit podle toho, provede-li se tělo alespoň jedenkrát, a cykly, kde tělo nemusí být provedeno vůbec. Rozhodně můžeme říci, že i když v C existují různé typy cyklů, je možné vystačit s jedním z nich. Přesto si v této podkapitole povíme o všech. Jejich výběr ponecháme na vhodnosti použití v dané situaci i na jejich individuální oblibě. V části věnované příkazu `while` si popíšeme některé vlastnosti společné všem cyklům. Postupně tedy **`while`**, **`for`** a **`do`**.

Cyklus while

Cyklus `while`, vykonává příkaz `statement` (tělo cyklu) vícekrát, nebo vůbec, dokud má v daném kontextu testovací výraz `expression` nenulovou hodnotu. Příkaz testuje podmínku před průchodem cyklem. Cyklus tedy nemusí proběhnout ani jednou. Syntaxe příkazu `while` je následující:

```
while ( <expression> ) <statement>
```

Příkaz je velmi často blokem. Zde se zejména začátečníci mylně domnívají, že pokud se kontext kdykoliv během provádění příkazů bloku těla cyklu změní, a podmínka tak přestane být splněna, je cyklus opuštěn. To ovšem není pravda. Pravidlo říká, že se příkaz `statement` provede, je-li podmínka `expression` splněna. Je-li příkazem blok, provede se tedy blok celý. Teprve poté je znovu proveden test. Názornější bude, podívat se na vývojový diagram.



Příkaz **while**

Ve vývojovém diagramu se vyskytují nepovinné příkazy `break` a `continue`. S prvním z nich jsme se již setkali u přepínače. Druhý zatím neznáme. Jejich sémantiku, zakreslenou ve vývojovém diagramu, si nyní popíšeme. Předem poznamenejme, že oba jsou pokládány za prostředky strukturovaného programování.

Příkaz **`break`** má ve všech cyklech stejný význam. Ukončí provádění příkazů těla cyklu a předá řízení prvnímu příkazu za příkazem `while`. Tímto způsobem můžeme bezprostředně ukončit průběh cyklu bez ohledu na hodnotu podmínky.

Příkaz **`continue`** rovněž ukončí provádění příkazů těla cyklu. Řízení ovšem předá těsně před příkaz cyklu. Proběhne tedy vyhodnocení testovacího výrazu a podle jeho hodnoty pokračuje program stejně, jako by bylo tělo cyklu vykonáno do konce (tedy bez předčasného ukončení vykonáním `continue`). Podobně jako `break`, má příkaz `continue` ve všech cyklech stejný význam.

Použití příkazu `while` ukazuje příklad. Opět v něm používáme kostku. S ní související funkce jsme popsali v předchozím příkladu. Tentokrát je naším cílem sečíst počet pokusů potřebných k tomu, abychom hodili šestku `POCET`-krát. Současně s novou látkou si připomeneme některé poznatky z předchozích kapitol.

Konstanta `POCET`, která nemá uveden typ, má implicitně typ int.

Cyklus `while` samotný probíhá tak dlouho, dokud platí podmínka `(pocet < POCET)`. Za blokem, tvořícím příkaz cyklu, nemusíme psát ukončovací středník. Následuje formátovaný standardní výstup výsledku a závěr funkce `main()`.

```
/*****************************/
/* Cyklus while. while-1.c     */
/*****************************/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

const POCET = 10;

int main(void)
{
 int celkem = 0, pocet = 0;

 printf("\nHazim kostkou dokud mi nepadne %d-krat sestka. ...\n", POCET);
 srand((unsigned) time(NULL) );

 while (pocet < POCET)
  {
   celkem++;
   if ((rand() % 6 + 1) == 6)
        pocet++;
  }
 printf("\nA je to! Hodu bylo celkem %d.\n\n", celkem);
 return 0;
}
```

Následující příklad čte znaky z klávesníce, tisknutelné znaky opisuje na obrazovku, neviditelných si nevšímá a zastaví se po přečtení znaku „z“.

```
/****************/
/* Prikaz while */
/****************/
#include <stdio.h>

main()
{
  int c;

  while ((c = getchar()) < 'z') {
    if (c >= ' ')
      putchar(c);               /* tisk znaku */
  }
  printf("\nCteni znaku bylo ukonceno. \n");
}
```

Následuje tentýž příklad s ukázkou nekonečného cyklu `while` s použitím příkazů `break` a `continue`.

```
/*
 * Prikazy break a continue a nekonecna smycka pomoci while  * bc_while.c
 */
#include <stdio.h>

main()
{
  int c;

  while (1) {            /* nekonecna smycka */
    if ((c = getchar()) < ' ')
      continue;          /* zahozeni neviditelneho znaku */
    if (c == 'z')
      break;             /* zastaveni po nacteni znaku 'z' */
    putchar(c);          /* tisk znaku */
  }
  printf("\nCteni znaku bylo ukonceno. \n");
}
```

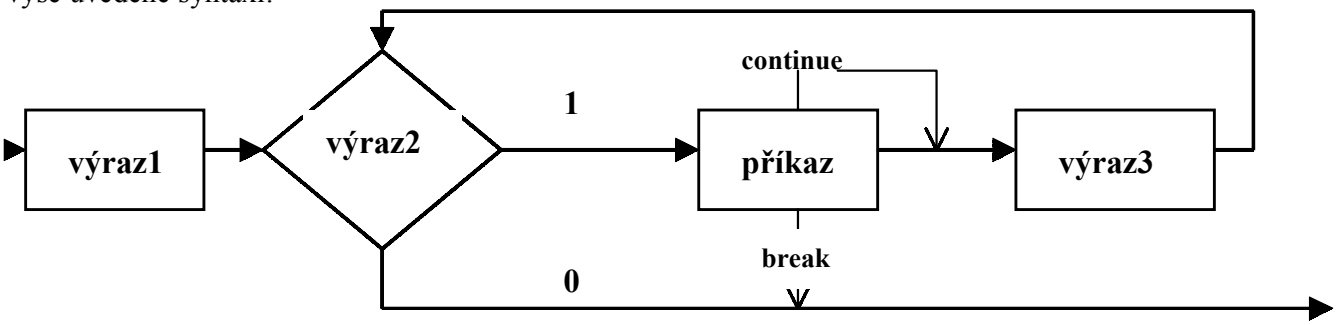
Cyklus for

Příkaz `for` provádí příkaz `statement` vícekrát, nebo vůbec, dokud je hodnota nepovinného testovacího výrazu `expr2` nenulová. V příkazu `for` můžeme ještě napsat dva další výrazy s vedlejším efektem, `expr1` a `expr3`. Syntaxe `for`:

`for ( [<expr1>] ; [<expr2>] ; [<expr3>] ) <statement>`

Nepovinný výraz `expr1` je proveden před prvním vyhodnocením testu. Typicky se používá pro inicializaci proměnných před cyklem. Po každém provedení těla cyklu `statement`, provede program nepovinný výraz `expr3`. Typicky se jedná o přípravu další iterace cyklu. Pokud neuvedeme testovací výraz `expr2`, použije překladač hodnotu `1`, a tedy bude provádět nekonečný cyklus. Naštěstí můžeme použít příkazy `break` a `continue` se stejným významem, jaký jsme popsali u `while`.

Vývojový diagram úplné varianty příkazu `for` je na obrázku, v němž jsme ponechali značení odpovídající výše uvedené syntaxi:



Příkaz **for**

Pro tisk ASCII tabulky znaků s kódy 32 až 127 použijeme příkaz `for` v úplné variantě:

```
for (znak = ' '; znak < 128; znak++)
```

V tomto případě můžeme nazývat proměnnou `znak` *řídící proměnnou cyklu*. Před testem ji inicializujeme mezerou, tedy prvním zobrazitelným ASCII znakem. Následuje test na hodnotu 128. Provádí se před každým průchodem tělem cyklu. Není-li podmínka splněna, pokračuje se za cyklem.

Po každém průchodu tělem cyklu je řídící proměnná cyklu `znak` zvýšena o jedničku. Tak je připraven další průchod cyklem s hodnotou následného ASCII znaku. Samotné tělo cyklu vlastně jen zobrazuje jednotlivý znak. Je-li jeho ASCII hodnota dělitelná 16 beze zbytku, zajistí vysláním konce řádku tvar tabulky.

```
/*****************************/
/* Cyklus for. for-1.c       */
/*****************************/

#include <stdio.h>

int main(void)
{
 int znak;

 putchar('\n');
 for (znak = ' '; znak < 128; znak++)
  {
   if (znak % 16 == 0)
        putchar('\n');
   putchar(znak);
   putchar(' ');
  }
 putchar('\n');
 return 0;
}
```

Výstup programu:

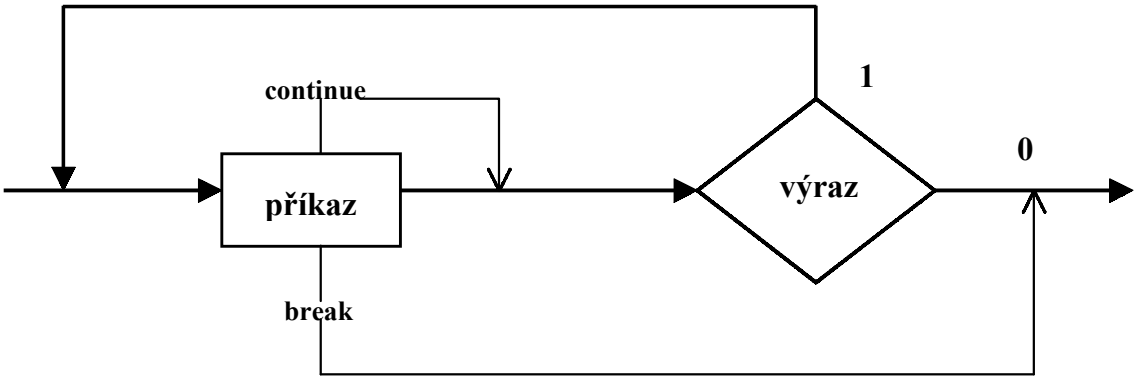
```
  ! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M A O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m a o
p q r s t u v w x y z { | } ~ &127
```

Cyklus do

Příkaz `do` je jediným z cyklů, který zajišťuje alespoň jedno provedení těla cyklu. Jinak řečeno, jeho testovací příkaz `statement` je testován až po průchodu tělem cyklu. Pokud je test splněn, provádí se tělo cyklu. Po syntaktické stránce tvoří tělo cyklu opět výraz `expression`:

`do <statement> while ( <expression> );`

Vývojový diagram příkazu `do` tuto vlastnost ukazuje ještě názorněji:



Příkaz **do**

Příkazy `break` a `continue` v těle cyklu se chovají stejně, jako v cyklech ostatních. Po `break` je cyklus opuštěn, zatímco po `continue` je proveden výraz `expression` a podle jeho výsledku se pokračuje dále.

Následující program je podobný jako u cyklu `while`. Opět opisuje všechny tisknutelné znaky zadané z klávesnice na obrazovku, neviditelných si nevšímá a zastaví se po stisku znaku „z“. Rozdíl je v tom, že vypíše i tento ukončovací znak.

```
/* Cyklus do-while.  do_while.c */

#include <stdio.h>

main()
{
  int c;

  do {
    if ((c = getchar()) >= ' ')
      putchar(c);               /* tisk znaku */
  } while (c != 'z');
}
```

Příkaz skoku

Příkaz skoku se v dobře napsaných programech používá málokdy, protože ve strukturovaném jazyku, jako je C, se mu lze vždy vyhnout. Skutečnost, že jsme si příkaz skoku nezatajili má dvě příčiny:

1. Jazyk C prostě příkaz skoku obsahuje. Jeho neuvedení by nebylo fér.
2. Skokové příkazy `break`, `continue` a `return` jsou označovány jako strukturované skoky. Jejich použití teorie (ani praxe) programování neodmítá.

Zůstává nám tedy jen příkaz skoku `goto`. Jeho syntaxe je jednoduchá:

`identifier: <statement> ;`
`goto <identifier> ;`

Uvedli jsme si nejen syntaxi `goto` samotného, ale i souvisejícího návěští `identifier`. Návěští neprovádí žádnou akci a nemá jiný vliv na řízení běhu. Jeho identifikátor jen označuje nějaké místo ve zdrojovém textu. Příkaz `goto` přenáší řízení na příkaz, označený návěštím. V rámci jedné funkce nesmí být dvě stejná návěští.

PREPROCESOR

Název kapitoly napovídá, že se budeme věnovat prostředku, který předchází překladač. My jsme zatím využívali (víceméně nevědomky) nejčastěji používaný příkaz preprocesoru `#include`. Činnost preprocesoru se dá shrnout do několika základních bodů:

- Zpracovává zdrojový text programu před použitím překladače.
- Nekontroluje syntaktickou správnost programu.
- Provádí pouze záměnu textů, např. identifikátorů konstant za odpovídající číselné hodnoty (*tzv. zpracování maker – macro processing*).
- Vypustí ze zdrojového textu všechny komentáře.
- Provádí podmíněný překlad.

C preprocesor přijímá tyto direktivy:

`#define`	`#elif`	`#else`	`#endif`
`#error`	`#if`	`#ifdef`	`#ifndef`
`#include`	`#line`	`#pragma`	`#undef`

Jednotlivé direktivy popíšeme v rámci následujících podkapitol.

Direktiva preprocesoru musí být vždy uvozena znakem #. # navíc musí být na řádku prvním jiným znakem, než jsou oddělovače. Od direktivy samotné jej opět mohou oddělovat oddělovače.

Zdůrazněme ještě jednu důležitou skutečnost. Direktiva preprocesoru není příkaz jazyka C. Neukončujme ji proto středníkem.

Definice maker.

Definice maker ve významu rozsahů polí je snad typickým příkladem použití preprocesoru. Ve zdrojovém textu se neodvoláváme na „magická čísla“, ale na vhodně symbolicky pojmenovaná makra. Program to nejen zpřehlední, ale případnou změnu hodnoty makra provedeme na jednom místě. Pomocí preprocesoru a maker můžeme vytvářet konstrukce, které zvýší čitelnost programu.

Pro větší přehlednost si makra rozdělme na symbolické konstanty a makra. Klíčem nechť je skutečnost, že makro na rozdíl od symbolické konstanty má argumenty.

Symbolické konstanty – makra bez parametrů

Symbolické konstanty zbavují program „magických čísel“, tj. nejrůznějších konstant, které se v programu objevují. Většinou jsou konstanty definovány na začátku programu (modulu).

Pro psaní konstant platí následující pravidla:

- Jména konstant jsou z konvence psána vždy VELKÝMI PÍSMENY.
- Jméno konstanty je od její hodnoty odděleno alespoň jednou mezerou.
- Za hodnotou může být (a měl by být) komentář.
- Nové konstanty mohou využívat dříve definovaných konstant.
- Pokud se text makra nevejde na jeden řádek, můžeme jej rozdělit na více následujících řádků. Skutečnost, že makro pokračuje na následujícím řádku, se určí umístěním znaku „\“ jako posledního znaku na řádku.

Definování a oddefinování symbolických konstant můžeme syntakticky popsat takto:

```
#define macro_id [token_sequence]
#undef macro_id
```

kde

`macro_id` představuje jméno (identifikátor) makra

`token_sequence` je nepovinný souvislý řetězec

Při své činnosti prohledává preprocesor vstupní text a při výskytu řetězce `macro_id` provádí jeho nahrazení řetězcem token_sequence. Této činnosti se říká rozvoj (expanze) makra. Z tohoto popisu je jasné, proč se preprocesoru někdy zjednodušeně říká makroprocesor.

Příklady:

```
#define MAX     1000    /*max. hodnota */
#define PI      3.14
#define DVE_PI (2*PI)
#define AND     &&
#define DLOUHA_KONSTANTA      Toto je tak dlouha konstanta,\
                              ze se nevejde na jednu radku.
```

Použití:

```
/* Symbolicka konstanta DVE_PI.  dve_pi.c
#include <stdio.h>
#define DVE_PI   (2*3.14)
main()
{
  double r;
  printf("Zadej polomer:");
  scanf("%lf", &r);
  printf("Obvod kruhu s polomerem %f je %f \n", r, r * DVE_PI);
}
```

Makra

Makra již podle našeho dělení mají argumenty. Definujeme je takto:

`#define macro_id([arg_list]) [token_sequence]`

kde (ostatní položky jsou stejné, jako jsme již uvedli u symbolických konstant):
`arg_list` představuje seznam argumentů navzájem oddělených jen čárkou.

Jako klasický příklad makra si uveďme vrácení maximální hodnoty ze dvou:

```
#define max(a,b) ((a>b)?a:b)
```

Výhodou i nevýhodou je, že nepracuje s typy. Výhodou proto, že pokud bychom chtěli definovat podobnou funkci, museli bychom napsat tolik jejích verzí, kolik by bylo navzájem neslučitelných variant datových typů argumentů. Nevýhodou je netypovost makra tehdy, uvedeme-li třeba omylem jako argumenty řetězce (pak by se porovnávaly adresy jejich prvních znaků) nebo dva argumenty neporovnatelných typů (struktura a číslo, ...). Takové chyby pak (někdy) odhalí až překladač.

Při definici makra `max` nás možná překvapí zdánlivě nadbytečné závorky oddělující `token_sequence`. Musíme jen připomenout, že makra nejsou příkazy jazyka C. Jejich rozvoj probíhá na textové úrovni. Preprocesor tedy nemůže v závislosti na kontextu jednou nadbytečné závorky vypustit, jindy chybějící přidat. Proto raději sami nadbytečné závorky nevypouštíme.

Z textovosti rozvoje makra mohou plynout i nečekané problémy. Podívejme se na makro, počítající druhou mocninu argumentu:

```
#define SQR(x)    (x*x)
```

a představme si jejich použití:

```
y = SQR(n+1);     /* (n+1*n+1) t.j. (4+1*4+1) = 9 */       ,
```

což nám dává zcela jiný (nesprávný) výsledek, než jsme očekávali. Pokud opravíme `(x*x)` na správnější `((x)*(x))`, dostaneme tentokrát sice výsledek správný, ale opět najdeme příklad, kdy správný nebude.

Právě z důvodů popsaných vedlejších efektů a netypovosti maker se nedoporučuje používat makra pro náhradu funkcí. Doporučuje se použití funkcí.

Standardní předdefinovaná makra

Soubor, ve kterém je uvedeno množství užitečných maker, je soubor **`ctype.h`**. Makra v něm definovaná pracují se znaky a dělí se do dvou skupin:

1) Makra pro určení typu znaku.
   Makra začínající **`is`**. Např. `isdigit(c)`, které vrací znak v `c` (jeho nenulovou hodnotu), je-li v `c` číslice. Jinak vrací nulu (FALSE).

vrací	**jméno**	**rozsah použití**
znak	`isalnum`	číslice a písmena (malá i velká)
znak	`isalpha`	malá a velká písmena
1	`isascii`	ASCII znaky (0 až 127)
znak	`iscntrl`	Ctrl znaky (1 až 26)
znak	`isdigit`	číslice
znak	`islower`	malá písmena
1	`isprint`	tisknutelné znaky
znak	`ispunct`	interpunkční znaky (čárka, tečka, lomítko, …)
znak	`isspace`	bílé znaky (mezera, tabulátor, nový řádek, …)
znak	`isupper`	velká písmena
znak	`isxdigit`	hexadecimální číslice (‘0’-‘9’, ‘A’-‘F’, ‘a’-‘f’)
1	`isgraph`	znak s grafickou podobou (33 až 126)

2) Makra pro konverzi znaku.
   Makra začínající to. Např. `b = tolower(c)`, které konvertuje velké písmeno v `c` na malé.

jméno	**rozsah použití**
tolower	konverze na malá písmena
toupper	konverze na velká písmena
toascii	převod na ASCII

Podmíněný překlad

V mnoha případech se složitější programy píší tak, že obsahují ladící části. To jsou nejčastěji pomocné výpisy, které mají ladění usnadnit.

Po odladění programu ale nastává typický problém - jak tyto již nepotřebné části odstranit. Lze samozřejmě pomocí editoru projít celý program a tyto pomocné časti vymazat ručně. Tímto způsobem však mnohdy smažeme i to, co jsme vymazat neměli.

Jazyk C umožňuje řešit tento problém pomocí příkazu preprocesoru, jež určuje, které části programu se mají překládat podmíněně. To znamená, že všechny ladící části již při vytváření programu označíme jako podmíněně překládané. Při ladění tyto části překládáme a po odladění nepřekládáme. Ladící části jsou tak trvalou součástí zdrojového souboru, ale volitelnou součástí vlastního přeloženého programu.

Podmíněný překlad může být řízen konstantním výrazem (číslo, symbolická konstanta, podmíněný výraz) a má tuto syntaxi:

```
#if konstantní_výraz
      část_1
#else
      část_2
#endif
```

Překladač bude tuto část zpracovávat tak, že je-li hodnota `konstantní_výraz` rovna 0 (FALSE), překládá se pouze část_2, v opačném případě (TRUE) se překládá pouze část_1.

Častěji než podmíměný překlad řízený hodnotou konstantního výrazu se používá podmíněného překladu, který závisí na tom, zda byla definována určitá symbolická konstanta. Syntaxe je velmi podobna předchozí syntaxi, takže rovnou příklad:

```
/*
 * Podmineny preklad rizeny definici makra
 */

#define PCAT

#ifdef PCAT
  #include <conio.h>      /* consol input/output */
#else
  #include <stdio.h>      /* standard input/output */
#endif

main()
{
  int i;
  for (i = 1;  i <= 100;  i++) {
#ifdef PCAT
      cprintf("\r%d", i);      /* tisk stale na jedno misto */
#else
      printf("\n%d", i);
#endif
  }
}
```

Budeme-li program používat na PC/AT stačí příkaz: `#define PCAT /* prazdný, ale definovaný */`
Na ostatních počítačích pak příkaz změníme na: `#undef  PCAT /* zrušena def. makra PCAT */`
nebo jednodušeji symbolickou konstantu vůbec nedefimujeme.

Zbývající direktivy.

`#include`
je direktivou naprosto nepostradatelnou. Používáme ji pro včlenění zdrojového textu jiného souboru. Tento soubor může být určen více způsoby. Proto má direktiva `#include` tři možné formy (pro snadnější odkazy je očíslujme):

1. `#include <header_name>`
2. `#include "header_name"`
3. `#include macro_identifier`

Ty postupně znamenají:

- soubor `header_name` je hledán ve standardním adresáři pro `include`. Takto se zpravidla začleňují standardní hlavičkové soubory. Není-li soubor nalezen, je ohlášena chyba.
- soubor `header_name` je hledán v aktivním (pracovním) adresáři. Není-li tam nalezen, postupuje se podle první možnosti. Takto se zpravidla začleňují naše (uživatelské) hlavičkové soubory.
- `macro_identifier` je nahrazen. Další činnost podle 1. nebo 2. varianty.

`#error`
je direktivou, kterou můžeme zajistit výstup námi zadaného chybového hlášení. Nejčastěji se používá v souvislosti s podmíněným překladem. Má formát:
`#error chybové hlášení`
kde `chybové hlášení` bude součástí protokolu o překladu.

`#line`
Používá se zejména u strojově generovaných zdrojových textů.
`#pragma`
je speciální direktivou, která má uvozovat všechny implementačně závislé direktivy.

FUNKCE

V této kapitole si podrobněji rozebereme základní stavební kámen jazyka C - funkci. Z předchozího textu víme, že každý C program obsahuje alespoň jednu funkci - `main()`. A máme co vysvětlovat. Proč je za `main` napsána dvojice závorek? Tak poznáme funkci od identifikátoru jiného typu (nebo od výrazu představujícího adresu vstupního bodu funkce). Z ANSI C vyplývá ještě jedna povinnost, funkce `main()` vrací hodnotu typu `int`.

Funkce v sobě zahrnuje takové příkazy, které se v programu často opakují, a proto se vyplatí je vyčlenit, pojmenovat a volat. Takto byla funkce chápána zejména v dobách, kdy byla paměť počítačů malá a muselo se s nízacházet velmi hospodárně. I dnes můžeme používat funkce ze stejných důvodů. Nicméně se na funkce díváme poněkud jinak.

Především rozlišujme standardní funkce, uživatelské funkce a podpůrné a nadstavbové funkce.

Standardní funkce jsou definovány normou jazyka a výrobce překladače je dodává jako součást programového balíku tvořícího překladač a jeho podpůrné programy a soubory. Tyto standardní funkce zpravidla dostáváme jako součást standardních knihoven (proto se jim někdy říká knihovní funkce) a jejich deklarace je popsána v hlavičkových souborech. Nemáme k dispozici jejich zdrojový kód. Ostatně nás ani moc nezajímá. Tyto funkce se přeci mají chovat tak, jak definuje norma.

Uživatelské funkce jsou ty funkce, které jsme napsali a máme jejich zdrojové texty. Pokud jsme profesionály, vyplatí se nám naše funkce precizně dokumentovat a archivovat. Když už jsme je jednou napsali a odladili, můžeme je příště s důvěrou již jen používat. Může být účelné sdružovat více uživatelských funkcí do jednoho, případně několika, souborů, případně z nich vytvořit knihovnu, abychom je při každém použití nemuseli znovu překládat. Hlavičkové soubory (prototypy funkcí, uživatelské typy, definice maker, ...) jsou opět nezbytností.

Podpůrné a nadstavbové funkce jako položka v rozdělení funkcí nám takové pěkné rozdělení okamžitě zničí. Sem totiž zařadíme nejen funkce od tzv. třetích výrobců (podpora pro spolupráci s databázemi, volání modemu, faxu, uživatelské rozhraní, ...), ale i rozšíření překladače o funkce nedefinované normou jazyka, funkce implementačně závislé a podobně. Nakonec sem patří i naše funkce, které používáme jako podpůrné. Jako poslední bod si můžeme představit týmovou práci, kdy používáme funkce kolegů. O jejich zdrojový text se zajímáme až v případě, kdy se funkce nechovají tak, jak mají.

Vraťme se ale k poslání funkcí. Funkce odpovídají strukturovanému programování. Představují základní jednotku, která řeší nějaký problém. Pokud je problém příliš složitý, volá na pomoc jinou (jiné) funkce. Z toho plyne, že by funkce neměla být příliš rozsáhlá. Pokud tomu tak není, nejenže se stává nepřehlednou, ale je i obtížně modifikovatelnou. Jak se například dotkne změna části rozsáhlé funkce její jiné části?

Deklarace a definice funkce.

Definice funkce určuje jak hlavičku funkce, tak její tělo. **Deklarace** funkce specifikuje pouze hlavičku funkce (tj. jméno funkce, typ návratové hodnoty a případně typ a počet jejích parametrů).

Nyní se podívejme, jak se prakticky s funkcemi pracuje. Začněme klasicky, uvedením syntaktického zápisu deklarace funkce:

```
typ jméno([seznam argumentů]);
```

kde

- `typ` představuje typ návratové hodnoty funkce
- `jméno` je identifikátor, který funkci dáváme
- `()` je povinná dvojice omezující deklaraci argumentů (v tomto případě prázdnou)
- `seznam argumentů` je nepovinný - funkce nemusí mít žádné argumenty, může mít jeden nebo více argumentů, nebo také můžeme určit, že funkce má proměnný počet argumentů.

Než se v jednotlivých podkapitolách budeme podrobněji věnovat každé položce z deklarace funkce, popišme si jméno funkce hned.

Identifikátor funkce je prostě `jméno`, pod kterým se budeme na funkci odvolávat. Závorkami za identifikátorem (toto se týká výskytu identifikátoru funkce mimo deklaraci či definici) dáváme najevo, že funkci voláme. Pokud závorky neuvedeme, jde o adresu funkce. Adresou funkce rozumíme adresu vstupního bodu funkce.

V této kapitole jsme zatím popsali jen deklaraci funkce. Ostatně středník, který ji ukončuje, nám nedává prostor pro samotnou definici seznamu deklarací a příkazů, které tělo funkce obsahuje.

Definice funkce, na rozdíl od její deklarace, nemá za závorkou ukončující seznam argumentů středník, ale blok. Tento blok se nazývá tělo funkce. V úvodu může, jako jakýkoliv jiný blok, obsahovat definice proměnných (jsou v rámci bloku lokální, neuvedeme-li jinak, jsou umístěny v zásobníku). Pak následuje posloupnost příkazů. Ty definují chování (vlastnosti) funkce. Při definici funkce vytváříme (definujeme) její kód, který obsazuje paměť.

Deklarace funkce popisuje vstupy a výstupy, které funkce poskytuje. Funkce nemá provádět akce s jinými daty, než která jí předáme jako argumenty. Současně výstupy z funkce mají probíhat jen jako její návratová hodnota a nebo (myšleno případně i současně) prostřednictvím argumentů funkce. Pokud se funkce nechová uvedeným způsobem říkáme, že má vedlejší účinky. Těch se vyvarujme. Často vedou k chybám.

Pokud uvedeme pouze definici funkce, na kterou se později v souboru odvoláváme, slouží tato definice současně jako deklarace.

V případě neshody deklarace a definice funkce ohlásí překladač chybu.

Návratová hodnota funkce.

Pojem datový typ intuitivně známe. Chceme-li získat návratovou hodnotu funkce, musíme určit, jakého datového typu tato hodnota je. Na typ funkce nejsou kladena žádná omezení. Pokud nás návratová hodnota funkce nezajímá, tak ji prostě nepoužijeme. Pokud ovšem chceme deklarovat, že funkce nevrací žádnou hodnotu, pak použijeme klíčové slovo **`void`**, které je určeno právě pro tento účel.

Kromě určení datového typu musíme také určit hodnotu, která bude návratovou. Je to snadné, tato hodnota musí být výsledkem výrazu uvedeného jako argument příkazu **`return`**. Typ `return` výrazu se pochopitelně musí shodovat, nebo být alespoň slučitelný, s deklarovaným typem návratové hodnoty funkce.

Výraz, následující za `return`, může být uzavřen v závorkách. To je klasický K&R styl. ANSI C norma závorky nevyžaduje, z konvence je však většinou píšeme:

```
return (výraz_vhodného_typu);
```

Spojeno s definicí funkce, může funkce celočíselně vracející druhou mocninu svého celočíselného argumentu vypadat takto:

```
int isqr(int i)
{
return (i * i);
}
```

Další příklad uvádí funkci `max()`, krerá vrátí větší ze svých dvou parametrů:

```
/*
 * Funkce vracejici maximum ze dvou parametru. f_max.c
 */

#include <stdio.h>

int max(int a, int b)
{
  return (a > b ? a : b);
}

main()
{
  int i = 3,
      j = 7;

  printf("Z cisel %d a %d je vetsi %d \n", i, j, max(i, j));
}
```

Příkaz `return` se v těle funkce nemusí vyskytovat pouze jedenkrát. Pokud to odpovídá větvení ve funkci, může se vyskytovat na konci každé větve. Rozhodně však příkaz `return` ukončí činnost funkce, umístí návratovou hodnotu na specifikované místo a předá řízení programu bezprostředně za místem, z něhož byla funkce volána.

Další příklady funkcí

Funkce bez parametrů musí být definována i volána včetně obou kulatých závorek. Příkladem je funkce, která přečte dvě celá čísla z klávesnice a vrátí jejich součet (f_nonpar.c). Aby byl překladač ujištěn o tom, že funkce nemá žádné formální parametry, používá se typ void (f_void.c).

```
/*
 * Funkce bez parametru.  f_nonpar.c
 * ================================
 */

#include <stdio.h>

int secti()
{
  int a, b;

  scanf("%d %d", &a, &b);
  return (a + b);
}

main()
{
  int j;

  printf("Zadej dve cela cisla : ");
  j = secti();
  printf("Sectena cisla = %d \n", j);
}
```

```
/*
 * Funkce bez parametru s pouzitim void
 *==================================
 * f_void.c
 */

#include <stdio.h>

int secti(void)
{
  int a, b;

  scanf("%d %d", &a, &b);
  return (a + b);
}

main()
{
  printf("Zadej dve cela cisla : ");
  printf("Sectena cisla = %d \n", secti());
}
```

Funkce nemusí samozřejmě vracet jenom typ `int`. V tomto případě se nezmění nic jiného než návratový typ funkce. Na rozdíl od funkcí s návratovým typem `int` není možné u funkcí s jiným návratovým typem označení typu vynechat, protože by pak hodnota byla implicitně `int`.

Příklad vynásobí parametr číslem 3,14 :

```
 /*
 * Funkce nevracejici typ int - uplny funkcni prototyp  f_nonint.c
 * ==============================================================
 */

#include <stdio.h>

double pikrat(double x);        /* uplny funkcni prototyp */

main()
{
  double r;

  printf("Zadej polomer : ");
  scanf("%lf", &r);
  printf("Obvod kruhu je %f\n", 2 * pikrat(r));
}

double pikrat(double x)
{
  return (x * 3.14);
}
```

Rekurse.

Díky umístění kopií skutečných argumentů na zásobník a umístění lokálních proměnných tamtéž se nám nabízí možnost rekurse.

Rekurse znamená volání funkce ze sebe samé, ať přímo, nebo prostřednictvím konečné posloupnosti voláním funkcí dalších. K tomu je nutné, aby programovací jazyk měl výše uvedené vlastnosti.

Předností rekurse je přirozené řešení problémů takovým způsobem, jaký odpovídá realitě či použitému algoritmu. Nevýhodou jsou často vysoké paměťové nároky na zásobník. S předáváním hodnot do zásobníku a následným obnovením jeho stavu souvisí i značná, zejména časová, režie rekurse.

Je ponecháno na naší úvaze, použijeme-li rekursi, či problém vyřešíme klasickými postupy. Platí totiž věta, že každá rekurse jde převést na iteraci.

Častým příkladem rekurse je výpočet hodnoty faktoriálu přirozeného čísla či určení hodnoty zvoleného členu Fibonacciho posloupnosti. Podívejme se na první z nich. V úloze je použit úmyslně jiný datový typ pro argument (`long`) a pro výsledek (`double`). Rozšíření rozsahu hodnot, pro něž faktoriál počítáme je vedlejší efekt. Chtěli jsme zejména ukázat použití různých datových typů.

```
/*
 * Rekurzivni funkce - vypocet faktorialu rek-fakt.c
 * ================================================
 */

#include <stdio.h>

double fakt(long n)
{
  return ((n <= 0)  ?  1  :  n * fakt(n - 1));
}

main()
{
  int i;

  printf("Zadej cele cislo : ");
  scanf("%ld", &i);
  printf("Faktorial je %lf\n", fakt(i));
}
```

Parametry funkcí

Jazyk C umožňuje pouze jeden způsob předávání parametrů a to předávání *hodnotou*. To v praxi znamená, že skutečné parametry nemohou být ve funkci měněny, ale pouze čteny. Jakákoliv změna parametrů uvnitř funkce je pouze dočasná a po opuštění funkce se ztrácí. Tento nedostatek řeší jazyk C využitím pointerů, pomocí nichž řeší volání *odkazem*.

V jayzce C lze psát i funkce s proměnným počtem parametrů (např. `printf()`). To však vyžaduje detailní znalost implementace – zejména způsobu předávání parametrů.

Podstatný je způsob předávání argumentů funkci. Hodnoty skutečných argumentů jsou předány do zásobníku. Formální argumenty se odkazují na odpovídající místa v zásobníku (kopie skutečných argumentů). Změna hodnoty formálního argumentu se nepromítne do změny hodnoty argumentu skutečného! Tomuto způsobu předávání argumentů se říká *předávání hodnotou*. Problém vzniká v okamžiku, kdy potřebujeme, aby funkce vrátila více něž jednu hodnotu. Řešení pomocí globální proměnné jsme již dříve odmítli (vedlejší účinky) a tvorba nového datového typu jako návratové hodnoty nemusí být vždy přirozeným řešením.

Řešení je předávání nikoli hodnot skutečných argumentů, ale jejich adres. Formální argumenty pak sice budou ukazateli na příslušný datový typ, ale s tím si poradíme. Podstatnější je možnost změny hodnot skutečných argumentů, způsob nazývaný třeba v Pascalu volání odkazem. V C se hovoří o *volání adresou*, což je zcela jasné a vystihující.

Ukažme si jednoduchý program, který volá různé funkce jak hodnotou, tak adresou.

```
/*********************************************************************/
/* soubor fn_argho.c                                                 */
/* pro procviceni predavani argumentu hodnotou i "odkazem" v jazyce C.  */
/*********************************************************************/

#include <stdio.h>

int nacti(int *a, int *b)
{
 printf("\nzadej dve cela cisla:");
 return scanf("%d %d", a, b);
}                       /* void nacti(int *a, int *b) */
float dej_podil(int i, int j)
{
 return((float) i / (float) j);
}                       /* float dej_podil(int i, int j) */

int main(void)
{
 int c1, c2;
 if (nacti(&c1, &c2) == 2)
       printf("podil je : %f\n", dej_podil(c1, c2));
 return 0;
}                       /* main */
```

Funkce `nacti()` musí vracet dvě načtené hodnoty. Předává proto argumenty adresou. Funkce `dej_podil()` naopak skutečné argumenty změnit nesmí (nemá to ostatně smysl). Proto jsou jí předávány hodnotou. Povšimněme si, že návratový výraz je v ní uzavřen v závorkách. Nevadí to, i když jsou závorky nadbytečné. Výraz je přeci ukončen středníkem.

Povšimněme si ještě jednou pozorněji funkce `nacti()`. Její návratová hodnota slouží k tomu, aby potvrdila platnost (respektive neplatnost) argumentů, které předává adresou. To je v C velmi praktikovaný obrat (viz `scanf()`).

Oblast platnosti identifikátorů – globální a lokální definice

Globální definice jsou definice proměnných, jejichž rozsah platnosti je od místa definice do konce souboru – ne programu!!! Tyto definice se vyskytují vně definic funfcí.

```
int i;

prvni()
{
  ...   //telo funkce
}
```

Lokální definice definují proměnné, jejichž rozsah platnosti je od místa definice do konce funkce, ve které jsou definovány. Tyto definice se tedy vyskytují uvnitř funkcí.

```
prvni()
{
  int i;
  ...   //telo funkce
}
```

V jazyce C mohou být některé globální identifikátory překryty (zastíněny) identifikátory lokálními.

Matematické funkce - výběr

typ	funkce	význam	knihovna
int	**abs(n)**	absolutní hodnota čísla	stdlib.h
double	**ceil(cislo)**	Zaokrouhlí číslo na nejbližší větší celé číslo	math.h
double	**floor(cislo)**	Zaokrouhlí číslo na nejbližší menší celé číslo	math.h
double	**fmod(a, b)**	zbytek po dělení dvou čísel a, b	math.h
long	**labs(n)**	absolutní hodnota čísla typu long	stdlib.h
double	**modf(cislo, ukazatel)**	rozdělí číslo na celou a desetinnou část	math.h
double	**cos(a)**	cosinus	math.h
double	**sin(a)**	sinus	math.h
double	**tan(a)**	tangens	math.h
double	**acos(a)**	arccos	math.h
double	**asin(a)**	arcsin	math.h
double	**atan**	arctan	math.h
double	**exp(x)**	exponenciální funkce při základu e	math.h
double	**log(x)**	přirozený logaritmus (ln x)	math.h
double	**log10(x)**	dekadický logaritmus (log x)	math.h
double	**pow(a, x)**	mocniná fukce (a^x)	math.h
double	**sqrt(x)**	druhá odmocnina	math.h
int	**rand()**	generování pseudonáhodného čísla	stdlib.h
void	**srand(cislo)**	nastavení výchozí základny pro výpočet pseudonáhodného čísla	stdlib.h

Příklad:

```
/* Rozdeleni cisla na celou a desetinnou část.  f_rozdel.c
 *=========================================================
 */
#include <math.h>
#include <stdio.h>
main()
{
  double x, y, n;
  x = -999.12345678;
  y = modf(x, &n);
  printf("%g = %g + %g\n", x, n, y);
}
```

```
/* Goniometricke funkce.  f_gonio.c
 *=========================================================
 */
#include <math.h>
#include <stdio.h>
#include <conio.h>

main()
{
  double uhel;

  clrscr();
  printf("Zadej uhel (v radianech): ");
  scanf("%lf", &uhel);
  printf("\nsinus    %.5le = % .5le", uhel, sin(uhel));
  printf("\ncosinus  %.5le = % .5le", uhel, cos(uhel));
  printf("\ntangens  %.5le = % .5le", uhel, tan(uhel));
  getch();
}
```

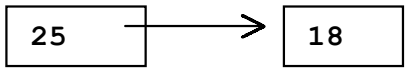
UKAZATELE, POLE A ŘETĚZCE

Ukazatele mohou být používány v souvislosti se statickými daty, například s proměnnými. Mnohem elegantnější použití poskytují v souvislosti s poli (a zejména s poli znaků - řetězci). Zcela nezastupitelné místo mají ukazatele v souvislosti s dynamickými datovými strukturami.

Protože zmíněné použití ukazatelů je příliš rozsáhlé pro jednu kapitolu, popíšeme jej v kapitolách dvou. Téma této kapitoly napovídá její název. Následující kapitola bude obsahovat podrobný přehled vstupu a výstupu. V další kapitole se budeme věnovat dynamickým datovým strukturám.

Ukazatele - pointery

Pointer (ukazatel) představuje adresu paměti, na které je uložena příslušná hodnota. Jinak řečeno pointer je proměnná uchovávající paměťovou adresu. Např.:



Obrázek má následující význam:

- proměnná **poi** je pointer (leží na symbolické adrese **poi**)
- hodnota poi je **25** (číslo uložené na symbolické adrese **poi** je 25)
- číslo **25** se nevyužije přímo k výpočtu, ale představuje absolutní adresu v paměti
- na sbsolutní adrese **25** v paměti je hodnota **18** (**18** je hodnota použitelná k výpočtu)
- **poi** ukazuje na hodnotu **18**, ale sám má hodnotu **25**

Absolutní adresa **poi** není v našem případě důležitá, protože používáme symbolické adresy. Dejme tomu, že poi bude ležet v paměti na adrese **87**. Pak říkáme, že obsah adresy **87** ukazuje na adresu **25**, kde je uložena hodnota **18**.

Tato odlišná interpretace obsahu proměnné nás staví před nutnost, jak překladači sdělit, že hodnota proměnné je adresa a ne už vlastní cílová hodnota. To se provede pomocí operátoru *****. To je druhý význam operátoru „hvězdička" – první je násobení.

Pomocí operátoru `*` můžeme:

1. získat obsah na adrese, na níž ukazuje pointer (např. `i = *poi`)
2. zapsat hodnotu na tuto adresu (např. `*poi = 5`)

Pointer je vždy svázán s nějakým datovým typem. Správně by se tedy mělo místo „*pointer"* uvádět „*pointer na typ* ..".

Na ukázku definujeme pointer na typ int. Na jiné datové typy je definice analogická.

```
int  *p_i;
```

Je možné uvést definici proměnné a pointeru najednou:

```
int  *p_i, i;
```

Identifikátory pointerů je vhodné začínat jednotně znaky **p_**. Je to sice maličkost, ale výrazně zvyšuje čitelnost programu.

V následujících příkladech platí definice: `int i, *p_i;`

`i = 3;`	do `i` dá hodnotu `3`
`*p_i = 4;`	na adresu v `p_i` dá hodnotu `4`
`i = *p_i;`	do `i` dá obsah z adresy v `p_i`
`*p_i = i;`	na adresu v `p_i` dá obsah `i`
`p_i = &i;`	naplní `p_i` adresou `i` – přiřazuje adresu do proměnné `i` – nesmí zde být `*`

Příklad:

Předpokládejme: `int i, *p_i1, *p_i2;`

A dále předpokládejme, že proměnná `i` leží na absolutní aderse `10`, `p_i1` na adrese `20` a `p_i2` na adrese `30`.

přiřazení	obsah na adrese 10 (&i)	obsah na adrese 20 (&p_i1)	obsah na adrese 30 (&p_i2)
`i = 1;`	1	?	?
`p_i1 = &i;`	1	10	?
`*p_i1 = 2;`	2	10	?
`i = *p_i1 + 1;`	3	10	?
`p_i2 = &i;`	3	10	10

Následující program čte dvě čísla (int) a zobrazí větší z nich:

```
/*
 * Prirazeni adresy promenne pointeru.  p_adrpro.c
 */

#include <stdio.h>

main()
{
  int i, j, *p_i;

  printf("Zadej dve cisla oddelena mezerou: \n");
  scanf("%d %d", &i, &j);
  p_i = (i > j) ? &i : &j;
  printf("Vetsi je %d \n", *p_i);
}
```

Pointery na funkce – volání odkazem

Jednou z důležitých vlastností pointerůje, že umožňují volání parametrů odkazem. Pointery v tomto případě použijeme, když chceme ve funkci trvale změnit hodnotu skutečného parametru. To znamená, že nepředáváme hodnotu proměnné, ale její adresu.

```
/*
 * Volani odkazem f_volod.c
 */

#include <stdio.h>

void vymen(int *p_x, int *p_y)
{
  int pom;

  pom = *p_x;
  *p_x = *p_y;
  *p_y = pom;
}
main()
{
  int i, j;
  printf("Zadej dve cela cisla:\n");
  printf("i = ");scanf("%d",&i);
  printf("j = ");scanf("%d",&j);
  printf("Pred vymenou: i = %d, j = %d \n", i, j);
  vymen(&i, &j);
  printf("Po vymene   : i = %d, j = %d \n", i, j);
}
```

Časté chyby při volání:

1. `vymen(i, j);`
   způsobí, že se bude zapisovat na adresy dané obsahem proměnných i a j, tj. na absolutní adresy 3 a 5
2. `vymen(*i, *j);`
   zápis bude proveden na adresy adres z obsahů i aj. Např. z absolutní adresy 3 se vezme hodnota, která se použije jako adresa, na které se ncosi změní.

Obě tato chybná volání vedou velice často ke zhroucení programu.

Pole

Pole je množina proměnných stejného typu, které mohou být označovány společným jménem. K prvkům pole přistupujeme prostřednictvím identifikátoru pole a indexu. Pole v C obsazuje spojitou oblast operační paměti tak, že první prvek pole je umístěn na nejnižší přidělené adrese a poslední na nejvyšší přidělené adrese. Pole je v C výhradně jednorozměrné.

Povšimněme si skutečnosti, že na prvky pole neklademe žádné omezení. To nám umožňuje zavést vícerozměrná pole jako pole polí.

Definice pole je jednoduchá:

```
typ jméno[rozsah];
```

kde

typ určuje typ prvků pole (bázový typ)

jméno představuje identifikátor pole

rozsah je kladný počet prvků pole (celočíselný konstantní výraz, který musí být vyčíslitelný za překladu)

Na tomto místě zdůrazněme význam položky `rozsah`. Znamená počet prvků pole. V souvislosti se skutečností, že pole začíná vždy prvkem s indexem `0`, to znamená, že poslední prvek pole má index `rozsah-1`.

Ještě jedna důležitá vlastnost C. Překladač nekontroluje rozsah použitého indexu. Pokud se o tuto skutečnost nepostaráme sami, můžeme číst nesmyslné hodnoty při odkazu na neexistující prvky pole.

Proti některým jiným vyšším programovacím jazykům na nás klade C vyšší nároky. Pokud se ovšem zamyslíme, zmíněná kontrola mezí se stejně použije pouze při ladění, hotový program by možnost nedodržení mezí neměl obsahovat.

Ukažme si několik definic polí, v nichž budeme postupně ukazovat možnosti definice proměnných typu pole:

```
#define N 10
int a[N];
```

`a` je deklarováno jako pole o `N` prvcích, každý typu `int`. Na jednotlivé prvky se můžeme odkazovat indexy 0 až 9, t.j.

```
a[0], a[1], ... , a[N-1]
```

paměťový prostor přidělený poli a můžeme zjistit výpočtem:

```
počet obsazených byte = N * sizeof(int)
```

Proč musíme potřebný paměťový prostor počítat právě tímto způsobem? Pouhé `sizeof(a)` vrátí velikost paměti pro proměnnou typu ukazatel. `sizeof(*a)` vrátí velikost prvku pole, tedy velikost `int`.

Jazyk C sice netestuje indexy polí, ale umožňuje nám současně s definicí pole provést i jeho inicializaci:

	b[0]	b[1]	b[2]	b[3]	b[4]	
`static double b[5] = {`	`1.2,`	`3.4,`	`-1.2,`	`123.0,`	`4.0`	`};`

Princip je jednoduchý. Před středník, jímž by definice končila, vložíme do složených závorek seznam hodnot, jimiž chceme inicializovat prvky pole. Hodnoty pochopitelně musí být odpovídajícího typu. Navíc jich nemusí být stejný počet, jako je dimenze pole. Může jich být méně. Přiřazení totiž probíhá následovně: první hodnota ze seznamu je umístěna do prvního prvku pole, druhá hodnota do druhého prvku pole, ... Pokud je seznam vyčerpán dříve, než je přiřazena hodnota poslednímu prvku pole, zůstávají odpovídající (závěrečné) prvky pole neinicializovány.

Při inicializaci popsané výše jsme museli uvádět dimenzi pole. Tuto práci ovšem můžeme přenechat překladači. Ten poli vymezí tolik místa, kolik odpovídá inicializaci, jako například:

```
int c[] = { 3, 4, 5};
```

Další příklad definice statického pole v C:

```
#define MAX 10
int x[MAX], y[MAX*2], z[MAX+1];
```

byla definována tři pole:

1. x o 10 prvcích s indexy od 0 do 9
2. y o 20 prvcích s horním idexem 19
3. z o 11 prvcích s indexy 0 až 10

Následující program zjistí počet jednotlivých písmen v řetězci zadaném z klávesnice. Malá a velká písmena se nerozlišují.

```
/*
 * Pocet pismen v retezci.  p_pism.c
 * ================================
 */

#include <stdio.h>
#include <ctype.h>
#include <conio.h>

#define POCET   ('Z' - 'A')

main()
{
  int c, i;
  int pole[POCET];

  clrscr();
  printf("Zadavej pismena, vstup ukonci nulou:\n");
  for (i = 0;  i < POCET;  i++)
    pole[i] = 0;                /* nulovani pole */

  while ((c = getchar()) != '0') {
    if (isalpha(c))             /* precteny znak je pismeno */
      pole[toupper(c) - 'A']++;
  }

  printf("\n  V retezci byl tento pocet jednotlivych pismen:\n");
  printf("  ---------------------------------------------\n");
  for (i = 0;  i < POCET;  i++) {
    printf("     %c - %d ", i + 'A', pole[i]);
    i++;                        /* druhy sloupec */
    printf("          %c - %d \n", i + 'A', pole[i]);
  }
  getch();
}
```

Řetězce

Řetězec je (jednorozměrné) pole znaků ukončené speciálním znakem ve funkci zarážky. Na tuto skutečnost musíme myslet při definici velikosti pole. Pro zarážku musíme rezervovat jeden znak. Nyní se zamysleme nad zarážkou. Ta je součástí řetězce (bez ní by řetězec byl jen pole znaků). Musí tedy být stejného typu, jako ostatní prvky řetězce - `char`. A při pohledu na ASCII tabulku nám zůstane jediná použitelná hodnota - znak s kódem nula: `'\0'`.

Podívejme se již na definici pole, které použijeme pro řetězec:

```
char s[SIZE];
```

Na identifikátor `s` se můžeme dívat ze dvou pohledů:

1. Jako na proměnnou typu řetězec (pole znaků, zarážku přidáváme někdy sami) délky `SIZE`, jehož jednotlivé znaky jsou přístupné pomocí indexů `s[0]` až `s[SIZE-1]`.
2. Jako na konstantní ukazatel na znak, t.j. na první prvek pole `s`, `s[0]`.

Řetězcové konstanty píšeme mezi dvojici uvozovek, uvozovky v řetězcové konstantě musíme uvodit speciálním znakem \ - opačné lomítko. Tak to jsme si připomněli obsah jedné z úvodních kapitol. Pokračujme ještě chvíli v jednoduchých ukázkách:
`"abc"` je konstanta typu řetězec délky 3+1 znak (zarážka), tedy 3 znaky, 4 bajty paměti (konstantní pole 4 znaků).
`"a"` je rovněž řetězcovou konstantou, má délku 1+1 znak
`'a'` je znaková konstanta (neplést s řetězcem!)

Nyní si spojíme znalosti řetězcových konstant a polí. Například zápis
```
char pozdrav[] = "hello";
```
je ekvivalentní zápisu
```
char pozdrav[] = {'h','e','l','l','o','\0'};
```
V obou případech se délka (dimenze) pole použije z inicializační části. Druhý příklad je zejména zajímavý nutností explicitního uvedení zarážky. Jinak lze říci, že se tento způsob zadání řetěce nepoužívá.

Čtení řetězce z klávesnice a tisk řetězce na obrazovku

Čtení řetězce se provádí pomocí funkce `scanf()`. Vlastní přikaz vypadá takto:
```
scanf("%s", s1);
```
Formát pro čtení řetězce je **`"%s"`**, stejně jako pro tisk řetězce. Příkaz přečte zadaný řetězec z klávesnice tak, že přeskočí všechny bílé znaky, přečte řetězec, který je ukončen bílým znakem a a uloží ho na adresu `s1`. `s1` je adresa, a proto „chybí“ operátor **`&`**. Potřebujeme-li přečíst celou řádku (tj. i případné mezery), použijeme funkci `gets()`.

Tisk řetězce se provádí nejčastěji pomocí funkce `printf()`. Příkaz vypadá následovně:
```
printf("%s", *s1);
```
Tento příkaz tiskne znaky od aresy `s1` až do okamžiku, kdy narazí na ukončovací znak `'\0'`.

Následující program přečte prvních maximálně 10 znaku zadaného řetězce. Definovaná délka je sice 11, ale řetězec nesmí přesáhnout délku 10, protože pak by začal přepisovat paměť ležící za `str`. V tomto případě by bylo vhodnější číst řetězec pomocí `scanf("%10s", str);`

```
/*
 * Nacteni a tisk statickeho retezce.   ss_read.c
 * ==============================================
 */

#include <stdio.h>

main()
{
  char str[11];

  printf("Zadej retezec : ");
  scanf("%s", str);
  printf("Retezec je : %s \n", str);
}
```

Pokud chceme definovat dynamický řetězec, bude program vypadat takto:

```
/*
 * Nacteni a tisk dynamickeho retezce.  ds_read.c
 *  ============================================
 */

#include <stdio.h>
#include <stdlib.h>

main()
{
  char *str;

  if ((str = (char *) malloc(11)) == NULL) {
    printf("Malo pameti \n");
  }
  printf("Zadej retezec : ");
  scanf("%s", str);
  printf("Retezec je : %s \n", str);
}
```

Funfce `malloc()` je funkce pro přidělení paměti. Jediný parametr této funkce je typu `unsigned int` a udává počet bytů, které chceme alokovat. Vrací pointer na `void`, který představuje adresu prvního přiděleného prvku. Není-li v paměti dost místa pro přidělení požadovaného úseku, vrací `malloc()` hodnotu `NULL`.

Protože řetězec je normální jednorozměrné pole, neměl by přístup k jednotlivým jeho znakům dělat problémy. Náskedující část programu vyplní s1 hvězdičkami:

```
for (i = 0; i < 10 - 1; i++)
  s1[i] = '*';
s1[10 - 1] = '\0';   //ukonceni retezce
```

Na ukončovací znak se nesmí nikdy zaponenout. Je-li řetězec definován jako char `s[MAX]`, pak využitelná délka je `MAX - 1`.

Jestliže jsme pracovali s polem znaků, proč to nyní nezkusit s ukazatelem na znak. Následující definice je velmi podobná těm předcházejícím:

```
char *ps = "retezec";
```

přesto se liší dosti podstatně. `ps` je ukazatel na znak, jemuž je přiřazena adresa řetězcové konstanty `retezec`. Tato konstanta je tedy umístěna v paměti (ostatně, kde jinde), ale proměnná `ps` na ni ukazuje jen do okamžiku, než její hodnotu třeba změníme. Pak ovšem ztratíme adresu konstantního řetězce `"retezec"`.

Jednoduchý program popisovanou situaci ozřejmí (v něm jsme pochopitelně neměnili hodnotu konstantního ukazatele na pole):

```
/*************************************/
/* soubor str_ptr.c                  */
/* ukazuje jednoduchou praci s retezci  */
/*************************************/

#include <stdio.h>

int main()
{
 char text[] = "world", *new1 = text, *new2 = text;
 printf("%s\t%s\t%s\n", text, new1, new2);
 new1 = "hello";
 printf("%s\t%s\t%s\n", text, new1, new2);
 printf("%s\n", "hello world" + 2);
 return 0;
}
```

Výstup programu:

```
world world world
world hello world
llo world
```

Nejdůležitějším bodem programu je přiřazení

```
new1 = "hello",
```

v němž pouze měníme hodnotu ukazatele `new1`. Nedochází při něm k žádnému kopírování řetězců. To by se musel změnit obsah obou dalších řetězců `text` i `new2`, vždyť původně byly umístěny na stejné adrese.

Podobné věci, které můžeme dělat s ukazateli, můžeme dělat i s konstantními řetězci, například:

```
"ahoj světe" + 5
```

představuje ukazatel na

```
"světe"
```

Prostě jsme použili aritmetiku ukazatelů a zejména fakt, že hodnota řetězcové konstanty je stejná jako jakéhokoliv ukazatele na řetězec (a nejen na řetězec) - ukazatel na první položku.

Nadále si tedy pamatujme přibližně toto. Při práci s řetězci nesmíme zapomínat na skutečnost, že jsou reprezentovány ukazateli. Pouhou změnou či přiřazením ukazatele se samotný řetězec nezmění. Proto s těmito ukazateli nemůžeme provádět operace tak, jak jsme zvyklí třeba v Turbo Pascalu.

Další řetězcové funkce

Řetězce zřejmě budeme v našich programech používat velmi často. Jistě by nám nebylo milé, kdybychom museli psát pro každou operaci nad řetězci svou vlastní funkci. Jednak by to vyžadovalo jisté úsilí, jednak bychom museli vymýšlet již vymyšlené. Tuto práci naštěstí dělat nemusíme. Zmíněné funkce jsou součástí standardní knihovny funkcí a jejich prototypy jsou obsaženy v hlavičkovém souboru `string.h`.

Následující řádky dávají přehled o nejpoužívanějších řetězcových funkcích. Protože jsme se řetězcům věnovali dostatečně, uvedeme pouze deklarace těchto funkcí s jejich velmi stručným popisem.

`int strcmp(const char *s1, const char *s2);`

Lexikograficky porovnává řetězce, vrací hodnoty.

```
< 0 je-li s1 < s2
0 s1 == s2
> 0 s1 > s2
```

`int strncmp(const char *s1, const char *s2, unsigned int n);`

Jako předchozí s tím, že porovnává nejvýše `n` znaků.

`unsigned int strlen(const char *s);`

Vrátí počet významných znaků řetězce (bez zarážky).

Příklad: `strlen("ahoj")` vrátí hodnotu 4

`char *strcpy(char *dest, const char *src);`

Nakopíruje `src` do `dest`.

Příklad: `strcpy(str,"ahoj");` v `str` bude „ahoj“

`char *strncpy(char *dest, const char *src, unsigned int n);`

Jako předchozí, ale nejvýše n znaků (je-li jich právě n, nepřidá zarážku).

`char *strcat(char *s1, const char *s2);`

Připojí `s2` za `s1`.

Příklad: `strcat(str, "+ nazdar");` v `str` bude „ahoj + nazdar“

`char *strncat(char *s1, const char *s2, unsigned int n);`

Jako předchozí, ale nejvýše `n` znaků.

`char *strchr(const char *s, int c);`

Vyhledá první výskyt (zleva) znaku `c` v řetězci `s`. V případě neúspěch vrátí `NULL`.

Příklad: z předchozího řetězce `strchr(str, 'x');` vrátí `NULL`

`char *strrchr(const char *s, int c);`

Vyhledá první výskyt (zprava) znaku `c` v řetězci `s`.

`char *strstr(const char *str, const char *substr);`

Vyhledá první výskyt (zleva) podřetězce `substr` v řetězci `str`. V případě neúspěch vrátí `NULL`.

Vícerozměrná pole, ukazatele na ukazatele

Po delší vsuvce, která ukazovala podrobněji vlastnosti znakových polí - řetězců, se dostáváme k problematice vícerozměrných polí. Nejčastěji nám zřejmě půjde o matice.

Jazyk C umožňuje deklarovat pole pouze jednorozměrné. Jeho prvky ovšem mohou být libovolného typu. Mohou tedy být opět (například) jednorozměrnými poli. To však již dostáváme vektor vektorů, tedy matici. Budeme-li uvedeným způsobem postupovat dále, vytvoříme datovou strukturu prakticky libovolné dimenze. Pak již je třeba mít jen dostatek paměti.

Definice matice (dvourozměrného pole) vypadá takto:
`type jmeno[5][7];`
kde
`typ` určuje datový typ položek pole
`jmeno` představuje identifikátor pole
`[5][7]` určuje rozsah jednotlivých vektorů, 5 řádků a 7 sloupců

Pamatujme si zásadu, že poslední index se mění nejrychleji (tak je vícerozměrné pole umístěno v paměti). S vícerozměrným polem můžeme pracovat stejně, jako s jednorozměrným. To se přirozeně týká i možnosti inicializace. Jen musíme jednotlivé "prvky" - vektory uzavírat do složených závorek. Tento princip je ovšem stejný (což se opakujeme), jako pro vektor. Proto si přímo uvěďme ukázkový program, převzatý od klasiků - K&R.
Naším úkolem je pro zadaný den, měsíc a rok zjistit pořadové číslo dne v roce. Ve výpisu je uvedena i funkce, mající opačný význam. Pro den v roce a rok (co když je přestupný) určí měsíc a den. Řešení je velmi elegantní.

```
/*************************************/
/* soubor yearday.c                  */
/* urci cislo dne v roce a naopak    */
/* motivace K&R                      */
/*************************************/

static int day_tab[2][13] =
      {{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
       {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

int day_of_year(int year, int month, int day)
{
 int i, leap;
 leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
 for (i = 1; i < month; i++)
   day += day_tab[leap][i];
 return day;
}

int month_day(int year, int yearday, int *pmonth, int *pday)
{
 int i, leap;
 leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
 for (i = 1; yearday > day_tab[leap][i]; i++)
   yearday -= day_tab[leap][i];
 *pmonth = i;
 *pday = yearday;
 return i;
}

int main()
{
 int year = 1993, month = 11, day = 12, yearday, m, d;
 yearday = day_of_year(year, month, day);
 month_day(year, yearday, &m, &d);
 return 0;
}
```

Program nemá vstup ani výstup. Představuje prostě definice funkcí a ukázku jejich volání. Hned v úvodu je definováno statické pole, obsahující počty dní v měsících roku. Prvky s indexem nula mají nulovou hodnotu. Díky tomuto posunu vůči standardnímu C indexování má pátý měsíc index pět. Pole je dvourozměrné. Jeho první vektor odpovídá běžnému roku, druhý vektor odpovídá roku přestupnému. Toto řešení je velmi rychlé a jeho paměťové nároky nejsou přemrštěné.

Funkce `day_of_year()` určí pro vstupní `year`, `month` a `day` číslo dne v roce. Funkce obsahuje rozsáhlejší logický výraz, který určí, je-li rok přestupný či nikoliv. Pak se pouze k zadanému dnu v zadaném měsíci postupně přičítají počty dnů v měsících předcházejících.

Funkce `month_day()` má opačnou úlohu. Ze vstupních údajů `year` a `yearday` (číslo dne v roce) vypočte měsíc a den. Problémem je, že výstupní hodnoty jsou dvě. Řešením jsou formální argumenty `pmonth` a `pday`, předávané adresou (tj. jsou to ukazatele). Postup algoritmu výpočtu je opačný, než v předchozím případě. Postupně se odečítají délky měsíců (počínaje lednem), dokud není zbytek menší, než je počet dnů měsíce. Tento měsíc je pak výsledným měsícem. Zbytek dnem v měsíci. Návratová hodnota funkce nemá prakticky žádný význam. Je prostě „do počtu“.

Ukazatele na ukazatele a pole ukazatelů

Ukazatel je proměnná jako každá jiná. Můžeme na něj tedy ukazovat nějakým jiným (dalším) ukazatelem. Ostatně, ve vícerozměrném poli provádíme v podstatě totéž, jen jednotlivé vektory nejsou pojmenované. Lze říci, že pole ukazatelů je v jazyce C nejen velmi oblíbená, ale i velmi často používaná konstrukce.

Mějme jednoduchou úlohu. Pole řetězců (t.j. ukazatelů na řetězce), které máme setřídit. Výhodou našeho řešení je, že řetězce nebudou měnit svou pozici v paměti. Zaměňovat se budou jen ukazatele na ně.

Naše pole nechť se jmenuje `v` a nechť má nějaký počet ukazatelů na `char`.

```
char *v[10];
```

Pokud umístíme do jednotlivých ukazatelů pole například ukazatele na načtené řádky textu (nemusí mít stejnou délku), můžeme princip třídění ukázat na záměně dvou sousedních ukazatelů (na řetězce) v poli.

Důležité je povšimnout si, že text (řetězce) svou pozici v paměti nezměnily. To by bylo poměrně komplikované, uvědomíme-li si, jak probíhá záměna dvou proměnných. Zaměnily se pouze adresy (hodnoty ukazatelů).

Po vysvětlení principu se můžeme podívat na program, který lexikograficky setřídí jména měsíců v roce. Protože standardní knihovny C netřídí česky, nejsou ani jména měsíců česky, ale jen "cesky". Povšimněme si i inicializace vícerozměrného pole a zjištění počtu jeho prvků.

```
/*********************************************/
/* soubor str_sort.c                         */
/* ukazuje trideni pole ukazatelu na retezce */
/*********************************************/

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

void sort(char **v, int n)
{
 int gap, i, j;
 char *temp;
 for (gap = n/2; gap > 0; gap /= 2)
   for (i = gap; i < n; i++)
     for (j = i - gap; j >= 0; j -= gap)
     {
      if (strcmp(v[j], v[j+gap]) <= 0)
        break;
      temp = v[j]; v[j] = v[j+gap]; v[j+gap] = temp;
     }
}
```

```
int main()
{
 char *name[] =
  {
   "chyba", "leden", "unor", "brezen", "duben", "kveten", "cerven",
   "cervenec", "srpen", "zari", "rijen", "listopad", "prosinec"
  };
 int i;
 puts("jmena mesicu v roce (podle poradi):");
 for (i = 0; i < 13; i++)
   printf("%2d. mesic je : %s\n", i, name[i]);
 puts("\na ted po setrideni:");
 sort(name, sizeof(name)/sizeof(char *));
 for (i = 0; i < 13; i++)
   printf("%2d. mesic je : %s\n", i, name[i]);
 return 0;
}
```

Nevýhoda zobrazení jmen všech měsíců před i po seřazení se projeví na monitorech s nejvýše 25-ti textovými řádky. Nechtěli jsme však program činit méně přehledným, proto jsme takový výstup ponechali.

NÁSLEDUJÍCÍ TEXT NEPROŠEL JAZYKOVOU ÚPRAVOU

Ukazatele na funkce.

Ukazatele na funkce jsme mohli začlenit do předchozí podkapitoly. Jedná se často o ukazatele nebo o pole ukazatelů. Funkce ovšem nejen vrací hodnotu jistého typu, ale mohou mít i různý počet argumentů různého typu. Protože je použití ukazatelů na funkce nejen velmi mocným prostředkem C, nýbrž i prostředkem velmi nebezpečným, rozhodli jsme se popsat ukazatele na funkce v samostatné podkapitole.

Definujme například ukazatel na funkci, která nemá argumenty a vrací hodnotu zvoleného datového typu:

```
typ  (*jmeno)();
```

Poznamenejme, že závorky okolo identifikátoru jmeno a úvodní hvězdičky jsou nutné, neboť zápis

```
typ *jmeno();
```

představuje funkci vracející ukazatel na typ.

Příkladem použití ukazatelů na funkce může být knihovní funkce qsort(). Její deklarace je následující:

```
void qsort(void *base, size_t nelem, size_t width, int (*fcmp)(const void *,
const void *));
```

kde

`base` je začátek pole, které chceme setřídit

`nelem` je počet prvků, které chceme setřídit (rozsah pole)

`width` je počet bajtů, který zabírá jeden prvek

`fcmp` je ukazatel na funkci, provádějící porovnání, ta má jako argumenty dva konstantní ukazatele na právě porovnávané prvky (nutno přetypovat)

Prototyp funkce `qsort()` je v hlavičkovém souboru `stdlib.h`.

Funkce `qsort()` je napsána obecně, a tak nemá žádné konkrétní informace o typech hodnot, které třídí. Tuto záležitost řeší naše uživatelská funkce, která správně porovná hodnoty. Neboť my víme, co třídíme. Ukázku použití funkce spolu s měřením času a použitím generátoru náhodných čísel nám dává program:

```
/*****************************************************/
/* soubor fn_qsrt.c                                  */
/* ukazuje pouziti qsort() a definici a pretypovani  */
/* uzivatelske srovnavaci funkce                     */
/*****************************************************/

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

```
#define      POCET          1000
#define      RND_START      1234

int float_sort (const float *a, const float *b)
{
 return (*a - *b);      /* <0, ==0, >0      */
} /* int float_sort (const float *a, const float *b) */

void test (float *p, unsigned int pocet)
/* otestuje zda dane pole je setrideno vzestupne     */
{
 int chyba = 0;
 for (; !chyba && --pocet > 0; p++)
   if (*p > *(p+1))
     chyba=1;
 puts ((chyba) ? "\npole neni setrideno\n"
             : "\npole je setrideno\n");
} /* void test (float *p,unsigned int pocet) */

void vypln(float *p, int pocet)
/* vypni pole dane adresou a poctem prvku nahodnymi */
/* cisly pomoci generatoru nahodnych cisel          */
{
 srand(RND_START);
 while (pocet-- > 0)
    *p++ = (float) rand();
} /* void vypln(float *p, int pocet) */

int main (void)
{
 static float pole [POCET];
 clock_t start, end;
 vypln (pole, POCET);
 start = clock();
 qsort(pole, POCET, sizeof(*pole),
       (int (*) (const void *, const void *)) float_sort);
 end = clock();
 printf ("Trideni qsort trvalo %fs", (end - start) / CLK_TCK);
 test (pole, POCET);
 getc (stdin);
 return 0;
}   /* int main (void)  */
```

Při práci s ukazateli na ukazatele, a tedy i na funkce, je velmi užitečné zavést nový typ, který definujeme podle našeho záměru. Odkaz na tento nový typ výrazně zpřehlední náš program. Před uvedením další ukázky poznamejme, že odkazy na funkce mohou být i externí. Znamená to, že máme možnost nezávisle na zdrojovém textu definovat funkci s požadovanými vlastnostmi, kterou pak připojíme do výsledného programu. Naše ukázka ovšem používá jen ukazatele na v souboru definované funkce.

```
/*******************************/
/* soubor PTR_FN01.C           */
/* pole ukazatelu na funkce    */
/*******************************/

#include <stdio.h>
#include <process.h>

typedef void (*menu_fcn) (void); /* definice typu                */
menu_fcn command[3];             /* pole tri ukazatelu na funkce */

void fn_1(void)                  /* definice prvni funkce        */
/**************/
```

```
{
 puts("funkce cislo 1\n");       /*puts() je kratsi, nez printf() */
} /* void fn_1(void) */

void fn_2(void)                  /* definice druhe funkce         */
/**************/
{
 puts("funkce cislo 2\n");
} /* void fn_2(void) */

void fn_3(void)                  /* definice treti funkce         */
/**************/
{
 puts("funkce cislo 3\nKONEC\n");
 exit(0);                       /* ukonceni chodu programu        */
} /* void fn_3(void) */

void assign_fn(void)            /* prirazeni ukazatelu na fce     */
/******************/            /* do pole ukazatelu              */
{
 command[0] = fn_1;
 command[1] = fn_2;
 command[2] = fn_3;
} /* void assign_fn(void) */

void menu(void)                  /* funkce s nabidkou            */
/*************/
{
 int choice;
 do {
     puts("1\tpolozka\n2\tpolozka\n3\tukonceni\n");
     putchar('>');
     scanf("%d", &choice);
     if (choice >= 1 && choice <= 3)
       command[choice - 1]();
       /* ^ volani funkce pomoci pole ukazatelu, () jsou nutne  */
    } while (1);                  /* nekonecny cyklus           */
} /* void menu(void) */

int main(void)                   /* hlavni funkce programu     */
/*************/
{
 assign_fn();     /* volani spravne inicializace pole ukazatelu */
 menu();          /* volani nabidky, obsahuje i volbu ukonceni  */
 return 0;        /* tento radek je vlastne zbytecny, program   */
                  /* ukonci exit() ve treti funkci - fn_3       */
} /* int main(void) */
```

Argumenty příkazového řádku

Pokud ještě spouštíme programy z příkazového řádku, asi rovněž víme, že jim můžeme předat argumenty. Často to bývají jména vstupního a výstupního souboru, přepínače ... . Stejná situace je i v případě, kdy spouštíme program z jiného programu.

Aby nám byly argumenty příkazového řádku v programu dostupné, stačí rozšířit definici funkce main o dva argumenty. Situace je podobná, jako u funkce s proměnným počtem argumentů.

První argument funkce `main()` udává počet argumentů příkazové řádky (1 = jen jméno programu, 2 = jméno programu + jeden argument, ...). Je celočíselného typu.

Druhý argument funkce `main()` představuje hodnoty těchto argumentů. Jeho typ je pochopitelně pole ukazatelů na řetězce, neboť jimi argumenty příkazové řádky skutečně jsou.

Je dobrým zvykem popsané argumenty funkce `main()` pojmenovat `argc` a `argv`, kde `c` znamená count a `v` znamená value.
Program zobrazující argumenty, s nimiž byl spuštěn následuje:

```
/*********************************************/
/* soubor CMD_LN04.C                         */
/* pomoci ukazatelu, dva formaty printf      */
/*********************************************/

#include <stdio.h>

int main(int argc, char **argv)
{
 while (argc-- > 0)
   printf((argc > 0) ? "%s " : "%s\n", *argv++);
 return 0;
}
```

Na závěr rozsáhlé kapitoly věnované polím a ukazatelům si zopakujme hlavní zásady pro práci s nimi:

- nesmíme se odkazovat na neinicializovaný ukazatel
- je-li `p` ukazatel na nějaký typ, pak `*p` je právě ta hodnota, na kterou ukazuje
- identifikátor pole je jen konstantní ukazatel na toto pole
- nebojme se aritmetiky ukazatelů

Vstup a výstup

Každý program zpracovává nějaká vstupní data a sděluje nám výsledky touto činností získané. Pokud by tomu tak nebylo, neměli bychom zřejmě důvod takový program vůbec aktivovat.

Vstup a výstup probíhá z různých vstupně výstupních zařízení. Jejich nejjednodušší rozdělení je na *znaková* a *bloková* (blokově orientovaná). Znakové vstupní zařízení typicky představuje klávesnice, výstupní pak monitor či tiskárna. Blokové vstupně/výstupní zařízení je velmi často pružný či pevný disk. Rozdíl mezi nimi spočívá zvláště v možnostech přístupu k datům. Z klávesnice čteme sekvenčně znak po znaku (*sekvenční přístup*), zatímco u diskového souboru můžeme libovolně přistupovat ke zvolené části dat (používaný termín popisující tuto činnost je *náhodný přístup*).

Vstup a výstup budeme často zkráceně zapisovat I/O, nebo, nebude-li možnost omylu, jen IO.

Ještě než se pustíme do podrobného výkladu, definujme si některé základní pojmy.

Řádek textu je posloupnost znaků ukončená symbolem (symboly) přechodu na nový řádek. Zdůrazněme, že v uvedené definici není žádná zmínka o délce řádku. Tuto skutečnost je třeba mít na paměti.

Soubor je posloupnost znaků (bajtů) ukončená nějakou speciální kombinací, která do obsahu souboru nepatří - konec souboru. Tuto hodnotu označujeme symbolicky **`EOF`**. **Textový soubor** obsahuje řádky textu. **Binární soubor** obsahuje hodnoty v témže tvaru, v jakém jsou uloženy v paměti počítače. Binární soubor obvykle nemá smysl vypisovat na terminál.

Standardní vstup a výstup

Každý program v jazyce C má standardně otevřen standardní vstup `stdin`, standardní výstup `stdout` a standardní chybový výstup `stderr`. Ty jsou obvykle napojeny na klávesnici a terminál. Na úrovni operačního systému máme ovšem možnost vstup a výstup přesměrovat.

Několik poznámek ke standardnímu I/O:

Pro ukončení vstupu z klávesnice použijeme v MS-DOSu kombinaci `ctrl-z`, v Unixu `ctrl-d`.

Standardní vstup a výstup používá vyrovnávací paměť obsahující jeden textový řádek.

Při volání funkcí standardního vstupu/výstupu musíme použít hlavičkový soubor `stdio.h`.

Standardní vstup a výstup znaků

Standardní vstup a výstup znaků představuje zcela základní možnost I/O. Funkce

```
int getchar(void);
```

přečte ze standardního vstupu jeden znak, který vrátí jako svou návratovou hodnotu. V případě chyby vrátí hodnotu `EOF`.

Funkce

```
int putchar(int c);
```

má zcela opačnou úlohu, znak, který je jejím argumentem, zapíše na standardní výstup. Zapsaná hodnota je současně návratovou hodnotou, nenastane-li chyba. Pak vrací `EOF`.

Zdůrazněme podstatný fakt, typ hodnoty, kterou čteme/zapisujeme, je `int`, nikoliv `char`, jak bychom v první chvíli očekávali. Tuto záležitost nám objasní opětné přečtení definice souboru z úvodu této kapitoly. Soubor obsahuje znaky. To odpovídá naší představě. Ale konec souboru je představován hodnotou, která do souboru nepatří. Tato hodnota ovšem musí být odlišná od ostatních znaků. A proto je typu `int`.

Čtení znaků ze standardního vstupu a jejich zápis na standardní výstup ukazuje program, představující jednoduchou variantu příkazu kopírování souboru (nesmíme ovšem zapomenout přesměrovat vstup a výstup).

```
/**************************************/
/* COPY.C                             */
/* CoPY character                     */
/**************************************/

#include <stdio.h>

int main(void)
{
 int c;

 while ((c = getchar()) != EOF)
   putchar(c);
 return 0;
}
```

Standardní vstup a výstup řetězců

Standardní vstup a výstup řetězců je jednoduchou nadstavbou nad čtením znaků. Obě funkce,

`char *gets(char *s);`
`int puts(const char *s);`

pracují s řetězci. `gets` načte do znakového pole vstupní řetězec až do konce řádku, symbol `'\n'` není do znakového pole zapsán. Ukazatel na pole (načtený řetězec) je rovněž návratovou hodnotou. Chybu signalizuje návrat `NULL`. `puts` zapíše řetězec na výstup a přidá přechod na nový řádek `'\n'`. Chybu představuje návratové `EOF`, jinak vrací kladné celé číslo.

Jednoduchost použití skrývá velké nebezpečí. Funkce `gets()` nemá informaci o délce oblasti vymezené pro čtený řetězec. Je-li oblast kratší, než vstupní řádek, dojde jeho načtením velmi pravděpodobně k přepsání paměťové oblasti související s vyhrazenou pamětí. A to se všemi důsledky z toho vyplývajícími.

Následující program je modifikací předchozího kopírování. Je ovšem možno jej použít pouze pro textové soubory. Navíc mohou vzniknout odlišnosti mezi originálem a kopií - např. končí-li poslední řádek originálu přímo `EOF` a ne `'\n'`, je konec řádku `'\n'` připojen díky vlastnosti funkce `puts()`.

```
/**************************************/
/* GETS.C                             */
/* GET String function                */
/**************************************/

#include <stdio.h>
#define MAX_STR 512

int main(void)
{
 char s[MAX_STR];

 while (gets(s) != NULL)
   puts(s);
 return 0;
}
```

Formátovaný standardní vstup a výstup

V *úvodu* jsme se povrchně seznámili s funkcemi pro formátovaný vstup a výstup `printf()` a `scanf()`. Tyto funkce jsou základními představiteli funkcí pro formátovaný I/O. Proto na tomto místě podrobněji popíšeme vlastnosti (způsob volání), na něž se dále budeme odvolávat, nebo které jsou v textu použity.
Pro formátovaný standardní výstup používáme funkci:

```
int printf (const char *format [, argument, ...]);
```

První parametr `format` určuje formátovací řetězec. Ten může obsahovat popis formátu pro každý argument, nebo text, který bude zapsán do výstupu. Popis formátu vždy začíná znakem `%`. Chceme-li znak `%` použít jako text, zdvojíme jej: `%%`. Návratová hodnota reprezentuje počet znaků zapsaných do výstupu, nebo `EOF` v případě chyby.
Specifikace formátu má poměrně velmi rozsáhlé možnosti:

```
% [flags] [width] [.prec] [F|N|h|l|L] type_char
```

Každá specifikace formátu začíná symbolem `%`. Po něm mohou v uvedeném pořadí následovat další položky:

položka	význam
`flags`	zarovnání výstupu, zobrazení znaménka a desetinných míst u čísel, úvodní nuly, prefix pro osmičkový a šestnáctkový výstup
`width`	minimální počet znaků na výstupu, mohou být uvedeny mezerami nebo nulami
`.prec`	maximální počet znaků na výstupu, pro celá čísla minimum zobrazených znaků, pro racionální počet míst za desetinnou tečkou
`F|N|h|l|L`	`l` indikuje dlouhé celé číslo, `L long double`, ostatní mají význam v MS-DOSu
`type_char`	povinný znak, určuje typ konverze

Specifikujme si nyní typ konverze (`type_char`):

symbol	význam
`d, i`	desítkové celé číslo se znaménkem
`u`	desítkové celé číslo se bez znaménka
`o`	osmičkové celé číslo
`x, X`	šestnáctkové celé číslo, číslice ABCDEF malé (`x`) nebo velké (`X`)
`f`	racionální číslo (`float`, `double`) bez exponentu, implicitně 6 desetinných míst
`e, E`	racionální číslo (`float`, `double`) v desetinném zápisu s exponentem, implicitně jedna pozice před des. tečkou, 6 za. Exponent uvozuje e, resp. E (dle použitého symbolu).
`g, G`	racionální číslo (`float`, `double`) v desetinném zápisu s exponentem nebo bez něj (podle absolutní hodnoty čísla). Nemusí obsahovat desetinnou tečku (nemá-li desetinnou část). Pokud je exponent menší, než -4, nebo větší, než počet platných číslic, je použit.
`c`	znak
`s`	řetězec

Příznak (`flag`) může být:

příznak	význam
`-`	výsledek je zarovnán zleva
`+`	u čísla bude vždy zobrazeno znaménko (i u kladného)
mezera	pro kladná čísla vynechá prostor pro znaménko
`#`	pro formáty `o, x, X` výstup jako konstanty jazyka C, pro formáty `e, E, f, g, G` vždy zobrazí desetinnou tečku, pro `g, G` ponechá nevýznamné nuly, pro `c, d, i, s, u` nemá význam.

Šířka (`width`) může být:

šířka	význam
`n`	je vytištěno nejméně n znaků zarovnaných zleva či zprava (viz příznak), doplněno mezerami
`0n`	jako předchozí, doplněno zleva nulami
`*`	jako šířka pole bude použit následující parametr funkce `printf()`

Přesnost (`.prec`) může být:

přesnost	význam
`.0`	pro `e, E, f` nezobrazí desetinnou tečku, pro `d, i, o, u, x` nastaví standardní hodnoty
`.n`	pro `d, i, o, u, x` minimální počet číslic, pro `e, E, f` počet desetinných číslic. Pro `g, G` počet platných míst, pro `s` maximální počet znaků
`*`	jako přesnost bude použit následující parametr funkce `printf()`

Pro **formátovaný standardní vstup** použijeme funkci:

```
int scanf (const char *format [, address, ...]);
```

První argument je opět formátovací řetězec. Ten může obsahovat tři typy zpracování objektů. Jsou to:

- přeskok bílých znaků (oddělovačů), t.j. mezery, tabulátoru, nového řádku a nové stránky
- srovnání znaků formátovacího řetězce se vstupními. Je-li na vstupu jiný, než určený znak, je čtení ukončeno.
- specifikace formátu vstupní pro hodnoty, je vždy uvozena znakem %.

Druhý (případně třetí, čtvrtý, ...) argument je úmyslně nazván `address`. Jde o to, že musí určovat paměťovou oblast, do níž bude odpovídající vstupní hodnota uložena. V praxi jde nejčastěji o adresu proměnné, nebo o ukazatel na pole znaků.

Čtení ze vstupu probíhá tak, že první formátovací popis je použit pro vstup první hodnoty, která je uložena na první adresu, druhý formátovací popis je použit pro vstup druhé hodnoty uložené na druhou adresu, ...

Návratová hodnota nás informuje kladným celým číslem o počtu bezchybně načtených a do paměti uložených položek (polí), nulou o nulovém počtu uložených položek a hodnotou `EOF` o pokusu číst další položky po vyčerpání vstupu.

Formát vstupní hodnoty popíšeme následovně:

```
% [*] [width] [F|N] [h|l|L] type_char
```

Po znaku `%` mohou následovat jednotlivé položky v uvedeném pořadí:

položka	význam
`*`	přeskoč popsaný vstup
`width`	maximální počet vstupních znaků
`F|N`	blízký nebo vzdálený ukazatel (jen MS-DOS)
`h|l|L`	modifikace typu
`type_char`	(povinný) typ konverze

Hvězdička (`*`) ve specifikaci formátu umožňuje příslušnou vstupní hodnotu sice ze vstupu načíst, ale do paměti nezapisovat. Šířka (`width`) určuje maximální počet znaků, které budou použity při vstupu. Modifikátory typu (`h|l`) týkající se celočíselné konverze (`d`) určují typ `short` resp. `long`, (`l|L`) modifikuje racionální typ `float` na `double` na `long double`.

Specifikujme si nyní typ konverze (`type_char`):

symbol	význam
`d`	celé číslo
`u`	celé číslo bez znaménka
`o`	osmičkové celé číslo
`x`	šestnáctkové celé číslo
`i`	celé číslo, zápis odpovídá zápisu konstanty jazyka C (např. 0x uvozuje šestnáctkové č.)
`n`	počet dosud přečtených znaků probíhajícím voláním funkce `scanf()`
`e, f, g`	racionální číslo typu `float`, lze modifikovat pomocí `l|L`
`s`	řetězec (i zde jsou úvodní oddělovače přeskočeny!), v cílovém poli je ukončen `'\0'`,
`c`	vstup znaku, je-li určena šířka, je čten řetězec bez přeskočení oddělovačů!
`[search_set]`	jako `s`, ale se specifikací vstupní množiny znaků (je možný i interval, např. `%[0-9]`, i negace, např. `%[^a-c]`.

Popis formátovaného standardního vstupu a výstupu je značně rozsáhlý. Některé varianty kombinací paramertů formátu se vyskytují poměrně exoticky. Častěji se setkáme s využitím návratové hodnoty funkce `scanf()`.

Příkladem může být cyklus, v němž načteme číselnou hodnotu (zadat můžeme celé i racionální číslo) a vypočteme a zobrazíme její druhou mocninu. Cyklus končí teprve ukončením vstupu - symbolem `EOF`. Tento symbol je v MS-DOSu `^Z`, v Unixu `^D`. Program můžeme chápat i jako malé ohlédnutí k cyklům:

```
/*********************************************************/
/* SCANWHIL.C                                            */
/* program je ukazkou nacitani pomoci scanf  v cyklu while  */
/*********************************************************/

#include <stdio.h>

int main(void)
{
 float f;

 printf("zadej x:");
 while (scanf("%f", &f) != EOF)
   {
    printf("x=%10.4f x^2=%15.4f\nzadej x:", f, f*f);
   }
 return 0;
} /* int main(void) */
```

```
zadej x:2
x=    2.0000 x^2=         4.0000
zadej x:3
x=    3.0000 x^2=         9.0000
zadej x:4
x=    4.0000 x^2=        16.0000
zadej x:^Z
```

Práce se soubory.

V odstavci věnovaném standardnímu vstupu a výstupu jsme si ukázali, že na této úrovni můžeme vlastně pracovat i se soubory. Stačí jen na úrovni operačního systému přesměrovat vstup a respektive výstup. Taková práce se soubory má velikou výhodu. Vždy probíhá na textové úrovni, která je implementačně nezávislá.

Soubor, s nímž můžeme v jazyce C pracovat, má své **jméno**. Tím rozumíme jméno na úrovni operačního systému. Programátor může určit, zda obsah souboru bude interpretován textově, nebo binárně. Uvedené vlastnosti odlišují pojmenovaný soubor od standardního vstupu a výstupu.

Jazyk C umožňuje se soubory pracovat na dvou odlišných úrovních.

První z nich je **přímé volání** . Tím se rozumí přímé volání služeb jádra systému. Poskytuje sice maximální možnou rychlost díky vazbě na jádro systému, současně však může být systémově závislý. Tento způsob práce se soubory je definován v K&R. Soubory s přímým voláním nejsou součástí ANSI normy.

Druhým přístupem je **datový proud,** kterým se budeme dále zabývat. Způsob práce s ním definuje ANSI norma jazyka C. Pro manipulaci s datovým proudem C disponuje řadou funkcí, které poskytují vysoký komfort.

Poznamenejme, že počet souborů, které můžeme v programu současně otevřít, je omezen OS (nejčasteji jeho konfigurací). Pro zjištění, jaký limit máme k dispozici, slouží makro `FOPEN_MAX`. Operační systém rovněž omezuje délku jména souboru. („Někdy dokonce na neuvěřitelných 8 + 3 znaky.“) Rovněž tuto hodnotu můžeme zjistit pomocí makra, tentokrát `FILENAME_MAX`.

Datové proudy

Pro práci s datovými proudy musíme používat funkční prototypy umístěné v `stdio.h`. Základem pro přístup k proudu je datový typ `FILE`. Při každém spuštení programu máme otevřeny proudy, které bychom mohli deklarovat takto:

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

ANSI norma definuje dva režimy proudů - textový (rozlišuje řádky) a binární. Režim stanovíme při otevírání souboru. Pro další výklad bude zřejmě nejvhodnější, uvedeme-li si nejprve nejdůležitější funkce a konstanty pro práci s proudy, a teprve poté několik příkladů, které výčet osvětlí. Pro přehlednost uvedeme skupiny funkcí, rozdělené podle možnosti jejich použití.

Pro práci s proudy máme k dispozici značné množství funkcí, maker a globálních proměnných. Jejich plný výčet je nad rámec tohoto textu. Proto se omezíme na ty z nich, které považujeme za významné. Ostatní můžeme vyhledat v referenčních příručkách C, nebo v elektronické dokumentaci přímo u počítače.

Otevření a zavření proudu

Každý proud (soubor) máme možnost otevřít a uzavřít. Při otevření určujeme režim našeho přístupu k datům v proudu. Ve speciálních případech oceníme i možnost proud znovu otevřít či spojit s novým souborem.

`FILE *fopen(const char *filename, const char *mode);`

Je funkce vracející ukazatel na strukturu `FILE` v případě úspěšného otevření proudu. Při neúspěchu vrací hodnotu `NULL`. Konstantní řetězec `filename`, označuje jméno souboru podle konvencí příslušného OS. Řetězec `mode` určuje režim práce se souborem i jeho typ. Základní možnosti jsou uvedeny v tabulce:

řetězec	význam (otevření pro:)
r	čtení
w	zápis
a	připojení
r+	aktualizace (update) - jako rw
w+	jako výše (r+), ale existující proud ořízne na nulovou délku, jinak vytvoří nový
a+	pro aktualizaci, pokud neexistuje, vytvoří
t	textový težim
b	binární režim

Poznamenejme, že otevření proudu v binárním režimu vyžaduje vždy písmeno `b` v řetězci `mode`, například: `"rb"`, `"wb"`, ... Obdobně pro textový režim je vhodné uvést jako součást řetězce `mode` znak `t`.

`int fclose(FILE *stream);`

Je funkce uzavírající určený proud. V případě úspěchu vrátí hodnotu `0`, jinak `EOF`. Uvolní paměť vyhrazenou pro strukturu `FILE *` a vyprázdní případnou vyrovnávací paměť.

`FILE *freopen(const char *filename, const char *mode, FILE *stream);`

Funkce uzavře soubor asociovaný s proudem `stream` (jako při volání `fclose()`). Pak otevře soubor jménem `filename` (jako při `fopen(filename, mode)`). Návratovou hodnotou je v případě úspěchu otevřený proud, jinak nulový ukazatel `NULL`.

Proudy a vstup/výstup znaků

Znak je po načtení z proudu konvertován bez znaménka na typ `int`. Obdobně je při zápisu do proudu konvertován opačným postupem. Tak máme ponechánu možnost rozlišit konec souboru od dalšího načteného znaku.

Připomeňme si čtení a zápis znaku z/do standardního vstupu/výstupu. Při pohledu na následující funkce je význam maker `getchar()` a `putchar()` zcela zřejmý. Stačí prostě doplnit druhý argument odpovídající funkce hodnotou `stdin` či `stdout`.

`int getc(FILE *stream);`

V případě úspěšného načtení znaku z proudu jej bez znaménka převede na typ `int`. Takto získaná hodnota je hodnotou návratovou. V případě chyby, nebo dosažení konce proudu pro `getc()`, vrací `EOF`.

`int ungetc(int c, FILE *stream);`

Je-li `c` různé od `EOF`, uloží jej funkce `ungetc()` do datového objektu s adresou `stream` a případně zruší příznak konce souboru. Následným čtením z tohoto proudu získáme námi zapsanou hodnotu. Je-li `c` rovno `EOF` nebo nemůže-li zápis proběhnout, vrací funkce `EOF`. Jinak vrací `c` (přesněji `(unsigned char) c`).

`int putc(int c, FILE *stream);`

Zapíše znak `c` do proudu `stream`. Vrátí stejnou hodnotu, jako zapsal. V případě chyby nebo dosažení konce proudu pro `getc()`, vrací `EOF`.

Proudy a vstup/výstup řetězců

Z proudu nemusíme číst pouze jednotlivé znaky. Můžeme načítat i celé řádky. Ty jsou ukončeny přechodem na nový řádek. Pro vyšší bezpečnost musíme při čtení uvést velikost vyrovnávací paměti. Při zápisu to pochopitelně nutné není. Do proudu se zapíše celý řetězec až po koncovou zarážku (ovšem bez ní).

`char *fgets(char *s, int n, FILE *stream);`

Načte řetězec (řádek až po jeho konec) z proudu `stream` do vyrovnávací paměti `s`, nejvýše dlouhý `n-1` znaků. Vrátí ukazatel na tento řetězec (vyrovnávací paměť), nebo, při chybě, `NULL`.

`int fputs(const char *s, FILE *stream);`

Zapíše do proudu řetězec ukončený zarážkou. Ani zarážku, ani případný konec řádku (obsažený na konci řetězce) do proudu nezapíše. V případě úspěchu vrátí počet zapsaných znaků (délku řetězce), jinak `EOF`.

Formátovaný vstup/výstup z/do proudu

Jestliže jsme zvládli standardní formátovaný vstup a výstup, nemůže nám činit formátovaný vstup a výstup z a do proudu potíže. Funkce se navzájem liší pouze úvodním `f` a identifikací proudu jako prvního argumentu. Právě srovnání je důvodem, proč uvádíme odpovídající funkce ve dvojici.

```
int fprintf (FILE *stream, const char *format [, argument, ...]);
int printf ( const char *format [, argument, ...]);
int fscanf (FILE *stream, const char *format [, address, ...]);
int scanf ( const char *format [, address, ...]);
```

Proudy a blokový přenos dat – práce s binárním souborem

Blokový přenos dat je nezbytný při práci s binárním proudem. Pokud srovnáme tyto funkce s odpovídajícími funkcemi pro soubory s přímým přístupem, pochopíme, jak se navzájem liší. Funkcí pro přímý přístup je poměrně málo. Představují základní operace, které při práci programátor potřebuje. A to je vše. Funkce pro práci s proudy umí všelijaké jemnůstky. Je jich díky tomu poměrně mnoho a programátor se musí umět rozhodnout, kterou z nich pro daný účel použije. Některé jejich argumenty můžeme dokonce považovat za nadbytečné. Zmiňujeme se o tom právě zde, neboť to můžeme názorně dokumentovat. Při blokovém proudovém IO musím určit, kolik položek chci přenést a jaká je velikost jedné položky. Prostý součin těchto hodnot by jistě každý zvládnul. Počet argumentů by se o jeden snížil.

Typ `size_t`, použitý v následujících funkcích, je zaveden pro určení velikosti paměťových objektů a případně počtu. Dává nám rovněž najevo, že je implementačně závislý. Kdyby místo něj bylo pouze `unsigned int`, nemuseli bychom si hned uvědomit, že v 16-ti bitovém systému je velikost položky omezena na 65535 bajtů.

`size_t fread(void *ptr, size_t size, size_t n, FILE *stream);`

Přečte z proudu `stream` položky o velikosti `size` v počtu `n` jednotek do paměťové oblasti určené ukazatelem `ptr`. V případě úspěchu vrátí počet načtených položek. Jinak vrátí počet menší (pravděpodobně to bude nula). Menší návratovou hodnotu, než je zadaný počet položek, můžeme získat například při dosažení konce proudu před načtením požadovaného počtu položek. Načtené položky jsou platné.

`size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);`

Tato funkce má argumenty obdobného významu, jako funkce předchozí. Pochopitelně s tím, že provádí zápis položek do proudu.

Další užitečné funkce

Protože je počet dosud neuvedených funkcí pro práci s proudy velmi značný, shrneme pod názvem *Další užitečné funkce* alespoň některé z nich.

`int feof(FILE *stream);`

Je funkce, umožňující zjistit dosažení konce proudu. Její návratová hodnota je - true (t.j. různá od nuly), nacházíme-li se na konci proudu, nebo - false (nula) - jinak.

`int fflush(FILE *stream);`

Má-li proud přiřazenu vyrovnávací paměť, provede její vyprázdnění (zápis) do souboru. Jinak neprovede nic. Návratová nula svědčí o úspěchu, `EOF` signalizuje chybu.

`int fseek(FILE *stream, long offset, int whence);`

Přenese aktuální pozici CP v proudu `stream` na stanovené místo. To je určeno posunem `offset` vzhledem k počátku, aktuální pozici, nebo konci souboru. Vztažný bod určuje argument `whence`. Předdefinované konstanty jsou stejné, jako v případě přímého přístupu - tj. `lseek()` (jsou to `SEEK_SET`, `SEEK_CUR`, `SEEK_END`). Poznamenejme, že použití této funkce zruší účinek bezprostředního předešlého `ungetc()`.

`long ftell(FILE *stream);`

Vrátí aktuální pozici v proudu. To je pro binární proud počet bajtů vzhledem k začátku. V případě chyby vrátí `-1L` a nastaví globální proměnnou `errno` na kladné celé číslo.

Jen zkratkovitě uvedeme ještě některé další užitečné funkce:

`ferror()` informuje o případné chybě při práci s proudem.

`clearerr()` ruší nastavení příznaku chyby a konce proudu.

`perror()` zobrazí řetězec chybového hlášení na standardní chybové zařízení (obvykle je to konsola).

`tmpfile()` otevře přechodný soubor v binárním režimu pro aktualizaci. S přechodnými soubory jsou spjaty ještě funkce `tmpnam()` a `tempnam()`.

Dvojice `fgetpos()` a `fsetpos()` umožňuje uchovat (získat) pozici v proudu a pak ji (opětně) nastavit.

`setbuf()` a `setvbuf()` umožňují nastavit a případně modifikovat velikost vyrovnávací paměti pro určený proud.

Příklady práce s proudy

Program vytvoří soubor POKUS.TXT a zapíše do něj čísla od 1 do 10, každé na novou řádku:

```
/* Zapis cisel do souboru.  w_cis.c */

#include <stdio.h>
main()
{
  FILE *fw;
  int  i;

  fw = fopen("POKUS.TXT", "w");
  for (i = 1;  i <= 10;  i++)
    fprintf(fw, "%d \n", i);
  fclose(fw);
}
```

Program přečte tři double čísla ze souboru DATA.TXT (soubor potřeba nejprve vytvořit např. editorem) a vypíše na obrazovku jejich součet:

```
/* Cteni cisel ze souboru.  r_cis.c */

#include <stdio.h>
main()
{
  FILE *fr;
  double x, y, z;

  fr = fopen("DATA.TXT", "r");
  fscanf(fr, "%lf %lf %lf", &x, &y, &z);
  printf("%f\n", x + y + z);
  fclose(fr);
}
```

Funkce `fscanf()` vrací počet úspěšně přečtených položek (na konci souboru vrací `EOF`). Následuje modifikace předchozího příkladu s testem, zda soubor obsahuje ři čísla:

```
if (fscanf(fr, "%lf %lf %lf", &x, &y, &z) == 3)
    printf("%f\n", x + y + z);
  else
    printf("Soubor DATA.TXT neobsahuje 3 realna cisla\n");
```

Program přečte dva znaky ze souboru ZNAKY.TXT a zapíše je do souboru KOPIE.TXT:

```
/* Cteni ze souboru a zapis do souboru  */

#include <stdio.h>
main()
{
  FILE *fr, *fw;
  int c;

  fr = fopen("ZNAKY.TXT", "r");
  fw = fopen("KOPIE.TXT", "w");

  c = getc(fr);                 /* cteni prvniho znaku */
  putc(c, fw);                  /* zapis prvniho znaku */
  putc(getc(fr), fw);           /* cteni a zapis druheho znaku */

  fclose(fr);
  fclose(fw);
}
```

Následující program přečte jednu řádku ze souboru DOPIS.TXT a opíše ji na obrazovku včetně znaku nové řádky. Program používá základního triku pro čtení až do konce rádky `if ((c = getc(fr)) != '\n')`:

```
/* Cteni radky ze souboru.  r_line.c */

#include <stdio.h>

main()
{
  int c;
  FILE *fr;

  fr = fopen("DOPIS.TXT", "r");
  while ((c = getc(fr)) != '\n')
    putchar(c);
  putchar(c);                   /* vypis '\n' - odradkovani */
  fclose(fr);
}
```

Program zkopíruje soubor ORIG.TXT do souboru KOPIE.TXT. Program používá základního triku pro čtení až do konce rádky `if ((c = getc(fr)) != EOF)`:

```
/* Kopirovani souboru pomoci EOF.  Cp1_EOF.c */

#include <stdio.h>

main()
{
  FILE *fr, *fw;
  int c;
  fr = fopen("ORIG.TXT", "r");
  fw = fopen("KOPIE.TXT", "w");

  while ((c = getc(fr)) != EOF)
    putc(c, fw);

  fclose(fr);
  fclose(fw);
}
```

Ještě jednou tentýž program tak, jak by měl vypadat po ošetření všech souborových operací:

```
/*
 * Kopirovani souboru s osetrenim otevirani a uzavirani souboru. Cp2_EOF.c
 * ========================================================================
 */

#include <stdio.h>

main()
{
  FILE *fr, *fw;
  int c;

  if ((fr = fopen("ORIG.TXT", "r")) == NULL) {
    printf("Soubor ORIG.TXT se nepodarilo otevrit\n");
    return;            /* ukonceni programu */
  }

  if ((fw = fopen("KOPIE.TXT", "w")) == NULL) {
    printf("Soubor KOPIE.TXT se nepodarilo otevrit\n");
```

```
    return;            /* ukonceni programu */
  }

  while ((c = getc(fr)) != EOF)
    putc(c, fw);

  if (fclose(fr) == EOF) {
    printf("Soubor ORIG.TXT se nepodarilo uzavrit\n");
    return;
  }

  if (fclose(fw) == EOF) {
    printf("Soubor KOPIE.TXT se nepodarilo uzavrit\n");
    return;
  }
}
```

Práce s binárními soubory

Doposud jsme pracovali pouze s textovými soubory. Textové soubory mají totiž obrovskou výhodu – jejich obsah si můžeme kdykoliv prohlédnout, vytvořit nebo opravit běžným editorem. Jejich nevýhodou je ale to, že pro uchování stejného množství informací potřebují mnohem více prostoru než soubory binární.Další výhodou binárních souborů je, že se s nimi pracuje mnohem rychleji než se soubory textovými.

Z těchto důvodů se binární soubory v profesionálních programech používají poměrně často. Nejvýhodnější je jejich využití pro ukládání rozměrných dat – velkých polí atd. Pro pouhé ukládání znaků nemají význam, protože znak v textovém souboru zabírá jeden Byte stejně jako v souboru binárním.

Následující příklad zapíše do binárního souboru POKUS.DAT hodnoty dvou proměnných (`i` a `d`). Soubor je otevřen jako binární pro čtení i zápis, což umožňuje, aby se po zápisu (`fwrite`) obou proměnných a přesunu ukazatele (`fseek`) dalo ze souboru hned číst (`fread`).

```
/*
 * Prace s binarnim souborem.  bin_file.c
 * ======================================
 */

#include <stdio.h>

main()
{
   FILE *f;                    /* pro cteni i pro zapis */
   int i = 5;
   double d = 3.14159;

   f = fopen("POKUS.DAT", "wb+");
   fwrite(&i, sizeof(i), 1, f);  /* zapis dat do souboru */
   fwrite(&d, sizeof(d), 1, f);

   printf("Pozice v souboru je %ld \n", ftell(f));
   fseek(f, 0, SEEK_SET);        /* posun na zacatek souboru */

   i = 0; d = 0.0;               /* nulovani promennych */
   fread(&i, sizeof(i), 1, f);   /* cteni a zobrazeni dat */
   fread(&d, sizeof(d), 1, f);
   printf("Nactena data: i = %d, d = %f \n", i, d);
   fclose(f);
}
```

DYNAMICKÉ DATOVÉ STRUKTURY

Dynamické datové struktury představují jakýsi protiklad ke statickým datovým strukturám. Povězme si tedy nejprve něco o nich. Porovnáním jejich vlastností lépe vyniknou přednosti těch či oněch pro jisté třídy úloh.

Statická data (SD) mají svou velikost přesně a neměnně určenou v okamžiku překladu zdrojového textu. Proto se s nimi pracuje jednoduše. Odpadá možnost práce s nepřidělenou pamětí. Potud klady. Nyní zápory. Ať náš program potřebuje dat méně či více, nedá se s tím nic dělat. Proto často raději volíme SD poněkud naddimenzovaná, aby náš program zvládl "většinu" úloh, jejichž třídu je schopen řešit. Pro příklady nemusíme chodit daleko. Napíšeme-li program pro práci s maticemi postavený na SD, musíme určit jejich dimenzi. Dimenze 3 x 3 asi nepostačí, tak raději sáhneme k dimenzi 10 x 10, nebo raději k 20 x 20. A budeme mít dobrý pocit, že naše úloha vyřeší vše. Pocit je pochopitelně mylný. Úloha si neporadí už s maticí 21 x 21. Navíc klademe na OS vždy stejné, a to většinou přemrštěné, požadavky na operační paměť. V jednoúlohovém systému to obvykle nevadí. Ve víceúlohovém, natož ještě víceuživatelském prostředí je takové plýtvání téměř neodpustitelné. SD jsou pochopitelně statická ve smyslu velikosti paměti, kterou mají přidělenu. Jejich hodnoty se během programu mohou měnit.

Dynamická data (DD) nemají velikost při překladu určenou. Při překladu jsou vytvořeny pouze proměnné vhodného typu ukazatel na, které nám za chodu slouží jako pevné body pro práci s DD. O požadovaný paměťový prostor ovšem musíme požádat OS. Ten naši žádost uspokojí, nebo nikoliv. Rozhodně však žádáme vždy jen tolik paměti, kolik se za chodu ukázalo být potřeba. Popravdě řečeno, požadujeme jí trošku víc. Ta trocha navíc je dána jednak nutnou režií, jednak našimi schopnostmi. Záleží jen na nás, jak vhodné dynamické datové struktury vybereme (nebo si oblíbíme). Pochopitelně vznikají mnohá nebezpečí, zejména při opomenutí požádání o paměť a zápisu do nepřidělených prostor.

DD mají ještě jednu přednost proti SD. Pokud požadavky na DD vystupují v na sobě nezávislých částech programu, může paměťová oblast být menší, než prostý součet těchto požadavků. Jakmile totiž končí část programu, vrátí přidělenou část DD zpět. Naopak nový úsek programu podle potřeby o paměť požádá.

DD tedy snižují paměťové nároky ze dvou důvodů. Jednak proto, že požadují tolik paměti, kolik je třeba. Jednak díky možnému vícenásobnému používání přidělěné dynamické paměti.

SD jsou umístěna v datovém segmentu. Klasický program je totiž rozdělen na datový segment a kódový segment.

Kódový segment obsahuje kód, který překladač vytvoří na základě našeho zdrojového textu. Tento kód by se za chodu programu tedy neměl měnit. Neměnnost kódového segmentu operační systémy obvykle kontrolují a zaručují. MS-DOS ovšem nikoli. Právě z toho plynou prakticky jisté havárie dokonce celého OS při chybné práci s ukazateli.

Datový segment je opět vytvořen překladačem. Jsou v něm umístěna všechna statická data programu. Tato část programu se během chodu programu mění.

Kromě kódového a datového segmentu přiděluje při spuštění (zavádění) programu OS ještě jisté prostředí. To má buď nějakou standardní velikost, nebo je tato velikost určena na základě požadavků zaváděného programu. Zde je umístěn zásobník. O něm víme, že je nezbytný při předávání argumentů funkcím, návratové hodnoty od funkcí a také pro nalezení návratové adresy po ukončení funkce. Tato návratová adresa je na zásobník zapsána při volání funkce.

Zásobník však zpravidla neobsadí celou zbývající paměť, kterou má program (proces) k dispozici. Část přidělené paměti zůstává volná. Té se zpravidla říká hromada či halda (heap). Její velikost můžeme při tvorbě programu určit. A právě halda představuje oblast paměti, do které se umisťují DD.

Dynamická alokace paměti.

Aby části programu mohly jak žádat o přidělení dynamické paměti, tak již nepotřebnou paměť vracet, musí existovat alespoň základní programová podpora. Tu si ovšem nebudeme vytvářet sami. Je definována ANSI normou jazyka a tudíž ji dostáváme spolu s překladačem.

Souhrně se programová podpora pro dynamickou alokaci paměťi nazývá podle oblasti, jíž se přidělování týká, správce haldy (heap manager). Součástí správce haldy jsou nejen potřebné funkce, ale i datové struktury. Jako uživatele nás pochopitelně zajímají především funkce správce haldy. Popišme si tedy některé z nich. Náš výběr určen zejména jejich přenositelností. Deklarace funkcí správce haldy jsou umístěny v `alloc.h`, případně ve `stdlib.h`.

```
void *malloc(size_t size);
```
představuje požadavek o přidělení souvislého bloku paměti o velikosti `size`. Je-li úspěšný, dostáváme ukazatel na jeho začátek, jinak `NULL`.
```
void *calloc(size_t nitems, size_t size);
```
jako předchozí s tím, že náš požadavek je rozložen na `nitems` položek, každá o `size` bytech. Navíc je přidělená paměť vyplněna nulami.
```
void free(void *block);
```
je naopak vrácení dříve alokované paměti, na kterou ukazuje `block`[1].
```
void *realloc(void *block, size_t size);
```
umožňuje změnit velikost alokované paměti, na kterou ukazuje `block` na novou velikost určenou hodnotou `size`. V případě potřeby (požadavek je větší, než původní blok) je obsah původního bloku překopírován. Vrací ukazatel na nový blok.
UNIX a MS-DOS definují pro dynamickou změnu velikosti haldy dvojici funkcí. První z nich,
```
int brk(void *addr);
```
nastaví hranici haldy programu na hodnotu danou `addr`. Druhá v dvojice,
```
void *sbrk(int incr);
```
umožní zvýšit tuto hodnotu o `incr` bajtů.
MS-DOS přidává ještě funkci, která vrací počet volných bajtů na hromadě (v závislosti na paměťovém modelu vrací `unsigned int` nebo `unsigned long`):
```
unsigned coreleft(void);
```
Krátkou ukázku přidělení a navrácení paměti pro řetězec představují dvě následující funkce. Rozsáhlejší a úplné programy jsou součástí každé z následujících podkapitol.

```
char *newstr(char *p)
{
 register char *t;
 t = malloc(strlen(p) + 1);
 strcpy(t, p);
 return t;
}

void freestr(char *p)
{
 free(p);
}
```

První funkce vytvoří na hromadě dostatečný prostor a nakopíruje do něj předávaný řetězec. Ukazatel na tuto kopii vrací jako svou funkční hodnotu. Druhá funkce provádí činnost opačnou. Oblast určenou ukazatelem vrátí správci haldy.

Detailní vlastnosti správce haldy mohou být různé. Čemu se však obvykle nevyhneme je situace zvaná segmentace haldy. Nejsou-li úseky z haldy vráceny v opačném pořadí, než byly přidělovány (LIFO), vzniknou na hromadě střídavě úseky volné a obsazené paměti. Celková velikost volné paměti, daná součtem všech volných úseků, je pak větší než velikost největšího souvislého volného bloku. Může pak nastat situace, kdy volné paměti je sice dost, ale náš požadavek na její přidělení není uspokojen, protože není k dispozici dostatečně velká souvislá oblast.

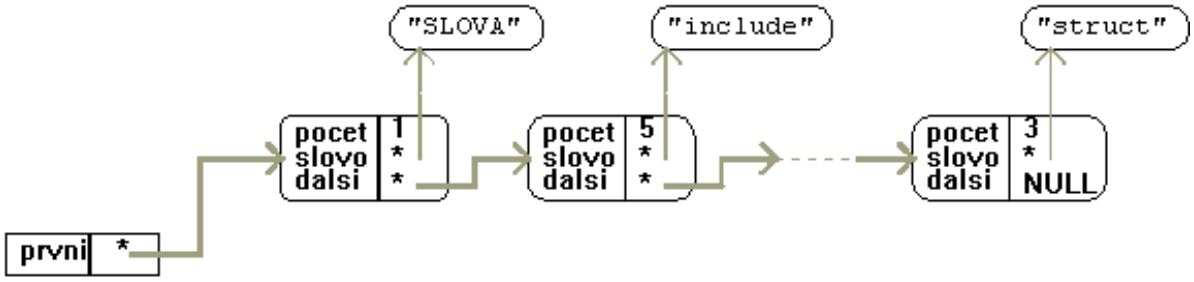
Seznam.

Je dynamická datová struktura, která mezi svými členy obsahuje kromě datové části i vazebný člen, který ukazuje na následující prvek seznamu, nebo `NULL`, je-li posledním prvkem. Datové minimum[2], které pro správu seznamu potřebujeme, je ukazatel na jeho první prvek. Popsaný seznam se rovněž nazývá jednosměrný. Postupně totiž můžeme projít od začátku seznamu až k jeho konci, nikoliv však naopak. Existuje ještě jedna varianta seznamu, mající vazebné členy dva. Jeden ukazuje opět na následníka, druhý na předchůdce.

Ukázku použití seznamu přináší následující úloha (a program). Mějme za úkol spočítat četnost výskytu všech slov ve vstupním textu. Pro jednoduchost budeme rozlišovat malá a velká písmena. Na závěr vytiskneme tabulku, v níž bude uvedena četnost výskytu a řetězec, představující slovo. Za slova považuje program řetězce písmen a číslic. Tuto skutečnost můžeme ovšem měnit úpravou vlastností funkce `odstran_neslova`. Jména

funkcí i proměnných v programu jsou volena s ohledem na jejich maximální popisnost. Vstupem může být soubor, je-li zadán jako 1. argument příkazového řádku. Jinak je vstupem standardní vstup.

Obrázek odpovídá datové struktuře vytvořené v programu. Hodnoty jsou pouze ilustrativní. Jediný nedynamický objekt seznamu je ukazatel `prvni` na začátek seznamu. Ostatní objekty jsou dynamické. V obrázku je tato skutečnost odlišena tvarem rohů obdélníčků.



Jednosměrný seznam

Obrázek může být i návodem, jak dynamické objekty rušit. Tato činnost totiž v programu obsažena není. Rozhodně je zřejmé, že musíme uvolnit jak paměť vyčleněnou pro hodnoty typu `struct_info`, tak i paměť, kterou obsazují řetězce, na něž je ze struktury ukazováno.

```
/*************************************************************/
/* soubor SLOVA.C                                            */
/* provede nacteni vstupniho textu, jeho rozlozeni na slova  */
/* a z techto slov vytvori seznam s jejich cetnosti vyskytu. */
/* Na zaver vytiskne slova a jejich cetnosti.                */
/*************************************************************/
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <ctype.h>
#include <conio.h>
#define DELKA_RADKU 500
#define PAGE        24

typedef struct struct_info {    /* typedef struct info; */
   int pocet;
   char *slovo;
   struct struct_info *dalsi;
  } info;

void odstran_neslova(char *ptr)
/* odstrani mezery, tabelatory a znaky CR, LF ze zacatku retezce */
{char *pom;
 pom = ptr;
 if (strlen(ptr) == 0)
   return;
 while ((*pom != '\0') && ((*pom == 0x20) || (*pom == '\t') ||
         (*pom == '\n') || (*pom == '\r') || (!isalnum(*pom))))
    {pom++;}
 strcpy(ptr, pom);
} /* void odstran_neslova(char *ptr) */

void vrat_slovo(char *r, char *s)
{
 int i = 0;
 while ((r[i] != 0x0) && (isalnum(r[i]) || (r[i] == '_')))
   {i++;} /* while */
 if (i == 0)
   {*s = 0x0;}
 else
   {strncpy(s, r, i);
    s[i] = 0x0;
```

```
    strcpy(r, r + i);
   }
} /* void vrat_slovo(char *r, char *s) */

void pridej_slovo(info **prvni, char *s)
{
 info *prvek;
 prvek = *prvni;
 while (prvek != NULL)
   {
    if (strcmp(s, prvek->slovo) == 0)
      {(prvek->pocet)++;
       return;
      }
    prvek = prvek->dalsi;
   }
 prvek = malloc(sizeof(info));
 prvek->slovo = malloc(strlen(s) + 1);
 strcpy(prvek->slovo, s);
 prvek->pocet = 1;
 prvek->dalsi = *prvni;
 *prvni = prvek;
} /* void pridej_slovo(info **prvni, char *s) */

void tiskni(info *prvni)
{
 int vytisknuto = 0;
 while (prvni != NULL)
   {
    printf("%4d..%s\n", prvni->pocet, prvni->slovo);
    prvni = prvni->dalsi;
    vytisknuto++;
    if ((vytisknuto % PAGE) == 0)
      {
       printf("pro pokracovani stiskni klavesu");
       getch();
       printf("\n");
      }
   }
} /* void tiskni(info *prvni) */

int main(int argc, char **argv)
{
 info *prvni = NULL;
 char radek[DELKA_RADKU + 1], slovo[DELKA_RADKU + 1];
 FILE *f;
 if (argc != 1)
   f = fopen(argv[1], "rt");
 else
   f = stdin;
 if (f == NULL)
   return(1);
 while (fgets(radek, DELKA_RADKU, f) != NULL)
   {
    odstran_neslova(radek);
    while (strlen(radek) != 0)
      {vrat_slovo(radek, slovo);
       odstran_neslova(radek);
       pridej_slovo(&prvni, slovo);
      }
   }
 tiskni(prvni);
 return 0;
} /* int main(int argc, char **argv) */
```

Pole ukazatelů.

Velmi pohodlnou dynamickou datovou strukturou je dynamické pole ukazatelů. Princip je jednoduchý. Požádáme o paměť pro dostatečně veliké pole ukazatelů. Dále již pracujeme stejně, jako by pole bylo statické (až na ten ukazatel na pole navíc). K jednotlivým prvkům můžeme přistupovat pomocí indexu. Nesmíme zapomenout, že prvky jsou ukazatele, a že tedy musíme paměť, na kterou ukazují, také alokovat. Díky funkci `realloc()` máme možnost snadno měnit[3] počet prvků našeho pole ukazatelů. Navíc s tím, že naše dynamická data svou adresu nemění a funkce `realloc()` přenese správné (stále platné) adresy do nového (většího) bloku ukazatelů. My k nim přistupujeme pomocí stejné proměnné[4], indexy všech prvků zůstavají stejné, jen jich přibylo. Skvělé.

Ukázková úloha, kterou řešíme, načítá vstupní řádky textu, ukládá je na hromadu s přístupem pomocí pole ukazatelů a nakonec řádky vytiskne ve stejném pořadí, v jakém byly načteny.

Hodnoty `START` a `PRIRUSTEK` jsou úmyslně voleny malé, aby se ukázala možnost realokace pole i při zadávání vstupu z klávesnice. Pro rozsáhlejší pokusy doporučujeme přesměrovat vstup.

```
/*********************************************/
/* soubor STR_ARRY.C                         */
/* ze standardniho vstupu cte radky, alokuje    */
/* pro ne pamet a ukazatele na tuto kopii       */
/* nacteneho radku uklada do pole ukazatelu     */
/* pole ukazatelu ma pocatecni velikost, ktera  */
/* se ovsem v pripade potreby zmeni - realloc() */
/* po ukonceni vstupu nactene radky vytiskne    */
/*********************************************/
/*******************************/
/* obvykle navratove hodnoty:   */
/*       O.K.      0            */
/*      error      !0 (casto -1)*/
/*******************************/

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

#define LADENI

#define      START            2
#define      PRIRUSTEK        1
#define      DELKA_RADKU      100

typedef char *          retezec;
typedef retezec *       pole_retezcu;


#if defined(LADENI)
 void volno(void)
 {
  printf("\nje volnych %10lu bajtu\n", coreleft()); /* LARGE */
 } /* void volno(void) */

 void uvolni(pole_retezcu *p_r, int pocet)
 {
  int i;
  for (i = 0; i < pocet; i++)
    {
     free((*p_r)[i]);
    }
  free(*p_r);
 } /* void uvolni(pole_retezcu *p_r, int pocet) */
#endif /* defined(LADENI) */

int alokace(pole_retezcu *p_r, int pocet)
{
 *p_r = malloc(pocet * sizeof(retezec));
```

```
 return (*p_r == NULL) ? -1 : 0;
} /* int alokace(pole_retezcu *p_r, int pocet) */

int re_alokace(pole_retezcu *p_r, int novy_pocet)
{
 pole_retezcu pom;
 if (*p_r == NULL)
   if (alokace(p_r, novy_pocet))
     return -1;      /* chyba */
 pom = realloc(*p_r, sizeof(retezec) * novy_pocet);
 if (pom == NULL)
    return -1;
 else
   {
    *p_r = pom;
    return 0;
   }
} /* int re_alokace(pole_retezcu *p_r, int novy_pocet) */

int pridej_radek(pole_retezcu *p_r, retezec s, int index)
{
 int delka = strlen(s) + 1;
 if (((*p_r)[index] = malloc(delka)) == NULL)
   return -1;
 strcpy((*p_r)[index], s);
 return 0;
} /* int pridej_radek(pole_retezcu *p_r, retezec s, int index) */

int cti_a_pridavej(pole_retezcu *p_r, int *pocet, int *alokovano, int prir)
{
 char radek[DELKA_RADKU], *pom;
 puts("Zadavej retezce, posledni CTRL-Z na novem radku");
 do
   {
    if ((pom = gets(radek)) != NULL)
      {
       if (*pocet + 1 > *alokovano)
       {
        if (re_alokace(p_r, *alokovano + prir))
          {
           puts("nedostatek pameti");
           return -1;
          }
        *alokovano += prir;
       }
       if (pridej_radek(p_r, radek, *pocet))
       {
        puts("nedostatek pameti");
        return 1;
       }
       (*pocet)++;
      }
   } while (pom != NULL);
 return 0;
} /*int cti_a_pridavej(pole_retezcu *p_r, int *pocet, int *alokovano, int prir)*/

void zobrazuj(pole_retezcu p_r, int pocet) {
 while (pocet--)
   {
    puts(*p_r++);
   }
} /* void zobrazuj(pole_retezcu p_r, int pocet) */
```

```
int main(void)
{
 int      pocet        = 0,
      alokovano = START,
      prirustek = PRIRUSTEK;
 pole_retezcu p_ret = NULL;

#if defined(LADENI)
 volno();
#endif /* defined(LADENI) */
 if (alokace(&p_ret, alokovano)) {
    puts("nedostatek pameti");
    return 1;
   }
 if (cti_a_pridavej(&p_ret, &pocet, &alokovano, prirustek))
   return 1;
 zobrazuj(p_ret, pocet);

#if defined(LADENI)
 uvolni(&p_ret, pocet);
 volno();
#endif /* defined(LADENI) */

 return 0;
} /* int main(void) */
```

V programu je funkce `volno()` definována v závislosti na skutečnosti, je-li definováno makro `LADENI`. Jde tedy o podmíněný překlad. Proč jsme jej zaváděli je nasnadě. Ve fázi ladění potřebujeme mít jistotu, že paměť, kterou jsme alokovali, později správně vracíme. Jakmile je program odladěn, není tato pomocná funkce potřeba. Nemusíme ale vnášet chyby tím, že ji i její volání bezhlavě smažeme. Výhodnější je, promyslet si takovou situaci již při návrhu programu. Pak ji, stejně jako v příkladu, budeme překládat podmíněně.

Poznamenejme, že obě popsané dynamické datové struktury lze modifikovat přidáním dalších funkcí na zásobník, frontu, ...

Pro podrobnější popis dynamických datových struktur doporučujeme skvělé dílo profesora Niclause Wirtha Algoritmy a struktury údajů.

Odvozené a strukturované typy dat

Dosud jsme se seznámili jen se základními datovými typy. To jest takovými typy, které vyhovují pro jednoduché výpočty či (programové) zpracování textu. Tyto typy máme v C přímo k dispozici (jsou součástí normy jazyka). Také známe preprocesor a víme, že použitím symbolických konstant program zpřehledníme. V této kapitole si ukážeme tvorbu a použití takových strukturovaných datových typů, jaké nám přináší život. Také si ukážeme definici výčtových typů, která umožní hodnoty nejen pojmenovat, ale provádět při překladu i jejich typovou kontrolu. Na úvod kapitoly si necháváme popis definice vlastních datových typů s prakticky libovolnou strukturou.

Uživatelský datový typ.

Vyšší programovací jazyk má k dispozici takové základní datové typy, které pokryjí většinu potřeb. Programátor ovšem musí mít k dispozici mechanismus, kterým si vytvoří datový typ podle svých potřeb. Tento mechanismus se nazývá `typedef` a jeho syntaxe je na první pohled velmi jednoduchá:

`typedef <type definition> <identifier> ;`

Po klíčovém slově `typedef` následuje definice typu `type definition`. Poté je novému typu určen identifikátor `identifier`.

Skalní zastánce nějakého datového typu či jazyka si spojením maker a uživatelských typů může C přetvořit k obrazu svému:

```
/**************/
/* TYPEDEF0.C */
/**************/

#include <stdio.h>

int main()
{
 typedef float real;
 real x = 2.5, y = 2.0;

 printf("%5.1f * %5.1f = %5.1f\n", x, y, x * y);

 return 0;
}
```

Získá tím ovšem jistotu, že jeho programy bude číst pouze on sám.

Složitější typové deklarace

Jestliže se uvedené jednoduché typové konstrukce prakticky nepoužívají, podívejme se naopak na některé konstrukce složitější. Nejprve si shrňme definice, které bychom měli zvládnout poměrně snadno:

deklarace	typ identifikátoru jméno
typ jméno;	typ
typ jméno[];	(otevřené) pole typu
typ jméno[3];	pole (pevné velikosti) tří položek typu (jméno[0], jméno[1],jméno[2])
typ *jméno;	ukazatel na typ
typ *jméno[];	(otevřené) pole ukazatelů na typ
typ *(jméno[]);	(otevřené) pole ukazatelů na typ
typ (*jméno)[];	ukazatel na (otevřené) pole typu
typ jméno();	funkce vracející hodnotu typu
typ *jméno();	funkce vracející ukazatel na hodnotu typu
typ *(jméno());	funkce vracející ukazatel na hodnotu typu
typ (*jméno)();	ukazatel na funkci, vracející typ

Může se stát, že si u některých definic přestáváme být jisti. Uvedeme si proto zásady, podle nichž musíme pro správnou interpretaci definice postupovat. Obecně se držíme postupu zevnitř ven. Podrobněji lze zásady shrnout do čtyř kroků:

1. Začněme u identifikátoru a hledejme vpravo kulaté nebo hranaté zívorky (jsou-li nějaké).
2. Interpretujme tyto závorky a hledejme vlevo hvězdičku.
3. Pokud narazíme na pravou závorku (libovolného stupně vnoření), vraťme se a aplikujme pravidla 1 a 2 pro vše mezi závorkami.
4. Aplikujme specifikaci typu.

Celý postup si ukážeme na příkladu:

```
char *( *( *var) ()) [10];
7 6 4 2 1 3 5
```

Označené kroky nám říkají:

1. Identifikátor `var` je deklarován jako
2. ukazatel na
3. funkci vracející
4. ukazatel na
5. pole 10-ti prvků, které jsou
6. ukazateli na
7. hodnoty typu `char`.

Raději další příklad. Tentokrát již popíšeme jen výsledek. Jednotlivé kroky nebudeme značit:

```
unsigned int *( * const *name[5][10]) (void);
```

Identifikátor `name` je dvourozměrným polem o celkem 50-ti prvcích. Prvky tohoto pole jsou ukazateli na ukazatele, které jsou konstantní. Tyto konstantní ukazatele ukazují na typ funkce, která nemá argumenty a vrací ukazatel na hodnotu typu `unsigned int`.
Následující funkce vrací ukazatel na pole tří hodnot typu `double`:

```
double ( *var (double (*)[3])) [3];
```

Její argument, stejně jako návratová hodnota, je ukazatel na pole tří prvků typu `double`.
Argument předchozí funkce je konstrukce, která se nazývá *abstraktní deklarace*. Obecně se jedná o deklaraci bez identifikátoru. Deklarace obsahuje jeden či více ukazatelů, polí nebo modifikací funkcí. Pro zjednodušení a zpřehlednění abstraktních deklarací se používá konstrukce `typedef`.

Abstraktní deklarace by nás neměly zaskočit ani v případě, kdy `typedef` použito není. Raději si několik abstraktních deklarací uvedeme:

int *	ukazatel na typ int
int *[3]	pole tří ukazatelů na int
int (*)[5]	ukazatel na pole pěti prvků typu int
int *()	funkce bez specifikace argumentů vracející ukazatel na int
int (*) (void)	ukazatel na funkci nemající argumenty vracející int
int (*const []) (unsigned int, ..)	ukazatel na nespecifikovaný počet konstantních ukazatelů na funkce, z nichž každá má první argument unsigned int a nespecifikovaný počet dalších argumentů

Výčtový typ.

Výčtový typ nám umožňuje definovet konstanty výčtového typu. To je výhodné například v okamžiku, kdy přiřadíme hodnotě výčtového typu identifikátor. Pak se ve zdrojovém textu nesetkáme například s hodnotou 13, či dokonce 0x0d, ale například CR, či Enter. Takový text je mnohem čitelnější. Jestliže později zjistíme, že je třeba hodnotu změnit, nemusíme v textu vyhledávat řetězec 13, který se navíc může vyskytovat i jako podřetězec řady jiných řetězců, ale na jediném místě změníme hodnotu výčtové konstanty. Že se jedná o klasické konstanty (či dokonce konstantní makra), které jsme poznali prakticky na začátku textu? Téměř to tak vypadá, ale výčtové konstanty mohou mít navíc pojmenován typ, který reprezentuje všechny jeho výčtové hodnoty. I tím se zvýší přehlednost.

Podívejme se nejprve na příklad. Naším úkolem je zpracovat stisknuté klávesy na standardní 101 tlačítkové klávesnici PC-AT. Pokud bychom do jednotlivých větví umístili pro porovnávání celočíselné konstanty, zřejme bychom sami brzy ztratili přehled. Použijeme-li výčtové konstanty, je situace zcela jiná. Ostatně podívejme:

```
/*********************************************/
/* soubor enum_use.c                         */
/* definice a naznak pouziti vyctovych konstant */
/* pro jednoduchy editor                     */
/*********************************************/

typedef enum {
      Back = 8, Tab = 9, Esc = 27, Enter = 13,
      Down = 0x0150, Left = 0x014b, Right = 0x014d, Up = 0x0148,
      NUL = 0x0103, Shift_Tab = 0x010f,
      Del = 0x0153, End = 0x014f, Home = 0x0147, Ins = 0x0152,
      PgDn = 0x0151, PgUp = 0x0149
} key_t;
 ...
    int znak;
 ...
    else if ((znak == Left) || (znak == Back))
          ...
    else if (znak == Enter)
          ...
    else if (znak == Esc)
          ...
    else if ...
 ...
```

Je zřejmé, že výpis zdrojového textu je krácen. Jde nám o ukázku. Přesto, že je zřejmě průhledná, podívejme se na syntaxi definice výčtového typu:

enum [<type_tag>] {<constant_name> [= <value>], ...} [var_list];

Klíčové slovo **enum** definici uvádí. Nepovinné označení **type_tag** umožňuje pojmenování hodnot výčtového typu bez použití konstrukce **typedef**. Poté následuje seznam výčtových konstant ve složených závorkách. Na závěr definice můžeme (nepovinný parametr) přímo uvést proměnné, které mohou definovaných výčtových hodnot nabývat.

Vraťme se ještě k obsahu bloku. Seznam identifikátorů je důležitý i pořadím jejich definice. Pokud nepoužijeme nepovinnou konstrukci **= <value>,** je první výčtové konstantě přiřazena hodnota nula. Následník pak má hodnotu o jedničku vyšší, než předchůdce. Jak jsme si ovšem ukázali v příkladu, můžeme přiřadit i první konstantě hodnotu jinou, než nulovou, rovněž může mít následník hodnotu nesouvisející s předchůdcem. Tak mohou vzniknout "díry" v číslování, případně i synonyma. Díky této možnosti (příklad nás jistě přesvědčil, že je užitečná), nemůže překladač kontrolovat, zdali nabývá proměnná hodnoty korektní či nikoliv. To je přijatelná cena, kterou platíme za popsané možnosti.

Poznamenejme, že hodnoty výčtových typů nelze posílat na výstup ve tvaru, v jakém jsme je definovali. Můžeme je zobrazit pouze jako odpovídající celočíselné ekvivalenty. Obdobně je můžeme číst ze vstupu. Výčtové konstanty se tedy ve své textové podobě nacházejí pouze ve zdrojovém tvaru programu. Přeložený program pracuje již jen číselnými hodnotami výčtových konstant.

Vrátíme-li se k příkladu, povšimneme si skutečnosti, že nepoužíváme `type_tag`. Tato možnost byla nutná ještě před zavedením konstrukce `typedef`. Dnes je obvyklejší pracovat naznačeným stylem. Přinejmenším pokaždé při deklaraci argumentů ušetříme ono klíčové slovo `enum`.

Typ struktura.

Dosud jsme v C obvykle vystačili se základními datovými typy. Realita, kterou se ve svých programech často neuměle pokoušíme popsat, zřejmě tuto jednoduchost postrádá. Nezřídka se setkáváme se skutečnostmi, k jejichž popisu potřebujeme více souvisejících údajů. Programátor navíc dodá, že různého typu. Užitečnou možností je konstrukce, která takovou konstrukci dovolí a pro její snadné další použití i pojmenuje. Směřujeme k definici struktury. Její korektní syntaktický předpis je následující:

```
struct [<struct type name>] {
   [<type> <variable-name[, variable-name, ...]>] ;
   [<type> <variable-name[, variable-name, ...]>] ;
   ...
 } [<structure variables>] ;
```

Konstrukci uvádí klíčové slovo **struct**. Následuje nepovinné pojmenování **struct type name**, které jako v případě výčtového typu obvykle nepoužíváme. Zůstalo zachováno spíše kvůli starší K&R definici C. Následuje blok definic položek struktury. Po něm opět můžeme definovat proměnné nově definovaného typu. Položky jsou odděleny středníkem. Jsou popsány identifikátorem typu **type**, následovaným jedním, nebo více identifikátory prvků struktury **variable-name**. Ty jsou navzájem odděleny čárkami.

Pro přístup k prvkům struktury používáme selektor struktury (záznamu) **.** (je jím tečka). Tu umístíme mezi identifikátory proměnné typu struktura a identifikátor položky, s níž chceme pracovat. V případě, kdy máme ukazatel na strukturu, použijeme místo hvězdičky a nezbytných závorek raději operátor `->`.

Podívejme se na příklad. Definujeme v něm nové typy `complex` a `vyrobek`. S použitím druhého z nich definujeme další typ `zbozi`. Typ `zbozi` představuje pole mající `POLOZEK_ZBOZI` prvků, každý z nich je typu `vyrobek`. Typ `vyrobek` je struktura, sdružující položky `ev_cislo` typu `int`, `nazev` typu znakové pole délky `ZNAKU_NAZEV+1`. Teprve takové definice nových typů, někdy se jim říká uživatelské, používáme při deklaraci proměnných.

```
typedef
  struct {float re, im;} complex;
typedef
  struct {
          int ev_cislo;
          char nazev[ZNAKU_NAZEV + 1];
          int  na_sklade;
```

```
          float cena;
         } vyrobek;
typedef vyrobek zbozi[POLOZEK_ZBOZI];
```

Syntaxe `struct` sice nabízí snadnější definice proměnných použitých v programu, otázkou zní, jak čitelné by pak bylo například deklarování argumentu nějaké funkce jako ukazatel na typ zboží. Jinak řečeno, Konstrukci struktury pomocí `typedef` oceníme spíše u rozsáhlejších zdrojových textů. U jednoúčelových krátkých programů se obvykle na eleganci příliš nehledí.

Dále se podívejme na přiřazení hodnoty strukturované proměnné při její definici.

```
vyrobek *ppolozky,
        a = {8765, "nazev zbozi na sklade", 100, 123.99};
```

Konstrukce značně připomíná obdobnou inicializaci pole. Zde jsou navíc jednotlivé prvky různých typů.

Následují ukázky přiřazení hodnot prvkům struktury. Nejzajímavější je srovnání přístupu do struktury přes ukazatel. Čitelnost zavedení odlišného operátoru v tomto případě je zřejmá. Můžeme porovnat s druhou variantou uvedenou jako komentář:

```
ppolozky->ev_cislo = 1;
/* (*ppolozky).ev_cislo = 1; */
```

Nyní se podívejme na souvislý zdrojový text.

```
/************************/
/* soubor STRUCT01.C    */
/* ukazka struct        */
/************************/

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

#define ZNAKU_NAZEV     25
#define POLOZEK_ZBOZI   10
#define FORMAT_VYROBEK  "cislo:%5d pocet:%5d cena:%10.2f nazev:%s\n"


typedef
  struct {float re, im;} complex;

typedef
  struct {
          int ev_cislo;
          char nazev[ZNAKU_NAZEV + 1];
          int  na_sklade;
          float cena;
         } vyrobek;
typedef vyrobek zbozi[POLOZEK_ZBOZI];

int main(void)
{
 complex cislo, im_jednotka = {0, 1};
 zbozi polozky;
 vyrobek *ppolozky,
        a = {8765, "nazev zbozi na sklade", 100, 123.99};

 cislo.re = 12.3456;
 cislo.im = -987.654;

 polozky[0].ev_cislo = 0;
 strcpy(polozky[0].nazev, "polozka cislo 0");
 polozky[0].na_sklade = 20;
 polozky[0].cena = 45.15;

 ppolozky = polozky + 1;
 ppolozky->ev_cislo = 1;
/*  (*ppolozky).ev_cislo = 1; */
 strcpy(ppolozky->nazev, "polozka cislo 1");
 ppolozky->na_sklade = 123;
```

```
 ppolozky->cena = 9945.15;

 printf("re = %10.5f im = %10.5f\n", im_jednotka.re, im_jednotka.im);
 printf("re = %10.5f im = %10.5f\n", cislo.re, cislo.im);
 printf(FORMAT_VYROBEK, a.ev_cislo, a.na_sklade, a.cena, a.nazev);
 printf(FORMAT_VYROBEK, polozky[0].ev_cislo, polozky[0].na_sklade,
       polozky[0].cena, polozky[0].nazev);
 printf(FORMAT_VYROBEK, ppolozky->ev_cislo, ppolozky->na_sklade,
       ppolozky->cena, ppolozky->nazev);
 return 0;
}
```

Tento výstup získáme spuštěním programu.

```
re =    0.00000 im =    1.00000
re =   12.34560 im = -987.65399
cislo: 8765 pocet:  100 cena:    123.99 nazev:nazev zbozi na sklade
cislo:    0 pocet:   20 cena:     45.15 nazev:polozka cislo 0
cislo:    1 pocet:  123 cena:   9945.15 nazev:polozka cislo 1
```

O užitečnosti struktur nás dále přesvědčí detailní pohled na typ, který jsme dosud používali, aniž bychom si jej blíže popsali. Je to typ `FILE`. Jeho definice v hlavičkovém souboru `STDIO.H` je:

```
typedef struct {
  short          level;
  unsigned       flags;
  char           fd;
  unsigned char  hold;
  short          bsize;
  unsigned char *buffer, *curp;
  unsigned       istemp;
  short          token;
} FILE;
```

Pokud se rozpomeneme na vše, co jsme se dosud o proudech dozvěděli, naznačí nám některé identifikátory, k jakému účelu jsou nezbytné. Výhoda definice `FILE` spočívá mimo jiné i v tom, že jsme tento datový typ běžně používali, aniž bychom měli ponětí o jeho definici. O implementaci souvisejících funkcí, majících `FILE *` jako jeden ze svých argumentů či jako návratový typ, ani nemluvě.

Pro základní použití struktur již máme dostatečné informace. Intuitivně jsme schopni odhadnout, jak přistupovat k prvku struktury, který je rovněž strukturou (prostě umístíme mezi identifikátory prvků další tečku).

Problém nastane v okamžiku, kdy potřebujeme definovat dvě struktury, které spolu navzájem souvisí. Přesněji řečeno, jedna obsahuje prvek typu té druhé. A naopak. Pravdou sice, je, že se nejedná o častou situaci, nicméně se můžeme podívat na použití *neúplné deklarace*. Nebudeme si vymýšlet nějaké příliš smysluplné struktury. Princip je následující:

```
struct A;                       /* incomplete */
struct B {struct A *pa};
struct A {struct B *pb};
```

Vidíme, že u neúplné deklarace určíme identifikátoru `A` třídu `struct`. V těle struktury `B` se ovšem může vyskytovat pouze ukazatel na takto neúplně deklarovanou strukturu `A`. Její velikost totiž ještě není známa[7].

Typ union.

Syntakticky vypadá konstrukce `union` následovně:

```
union [<union type name>] {
  <type> <variable names> ;
  ...
} [<union variables>] ;
```

Již na první pohled je velmi podobná strukturám. S jedním podstatným rozdílem, který není zřejmý ze syntaxe, ale je dán sémantikou. Z položek unie lze používat v jednom okamžiku pouze jednu. Ostatní mají nedefinovanou hodnotu. Realizace je jednoduchá. Paměťové místo, vyhrazené pro unii je tak veliké, aby obsáhlo jedinou (paměťově největší) položku. Tím je zajištěno splnění vlastnosí unie. Překladač C ponechává na programátorovi, pracuje-li s prvkem unie, který je určen správně či nikoliv. Ostatně, v okamžiku překladu nejsou potřebné údaje stejně k dispozici.

Každý z prvků unie začíná na jejím začátku. Můžeme si představit, že paměťově delší prvky překrývají ty kratší. Této skutečnosti můžeme někdy využít. Nevíme-li, jakého typu bude návratový argument, definujeme unii, mající položky všech požadovaných typů. Dalším argumentem předáme informaci o skutečném typu hodnoty. Pak podle ní provedeme přístup k správnému členu unie.

Bitová pole.

K současným trendům programování patří i oprostění se od nutnosti šetřit každým bajtem, v extrémních případech až bitem, paměti. Plýtvání zdroji (pamětí, diskovou kapacitou, komunikací) je stále častěji skutečností. Přesto jsou i dnes oblasti, v nichž je úsporné uložení dat ne-li nezbytné, tedy alespoň vhodné. Ano, jedná se mimo jiné o operační systémy. Jednou z možností, jak úsporně využít paměť jsou právě bitová pole.

Bitová pole je celé číslo, umístěné na určeném počtu bitů. Tyto bity tvoří souvislou oblast paměti. Bitové pole může obsahovat více celočíselných položek. Můžeme vytvořit bitové pole tří tříd:

1. prosté bitové pole
2. bitové pole se znaménkem
3. bitové pole bez znaménka

Bitová pole můžeme deklarovat pouze jako členy *struktury* či *unie*. Výraz, který napíšeme za identifikátorem položky a dvoutečkou, představuje velikost pole v bitech. Nemůžeme definovat přenositelné bitové pole, které je rozsáhlejší, než typ `int`.

Způsob umístění jednotlivých položek deklarace do celočíselného typu je implementačně závislý.
Podívejme se nyní na příklad bitového pole. Jak můžeme z předchozího textu usuzovat, je spjat s konkrétním operačním systémem. V hlavičkovém souboru `IO.H` překladače BC3.1 je definována struktura `ftime`, která popisuje datum a čas vzniku (poslední modifikace) souboru v OS MS-DOS (řekněme včetně verze 6):

```
struct ftime {
   unsigned ft_tsec  : 5;  /* Two seconds */
   unsigned ft_min   : 6;  /* Minutes */
   unsigned ft_hour  : 5;  /* Hours */
   unsigned ft_day   : 5;  /* Days */
   unsigned ft_month : 4;  /* Months */
   unsigned ft_year  : 7;  /* Year - 1980 */
 };
```

Prvky struktury jsou bitová pole. Výsledkem je výborné využití 32 bitů. Jediným omezením je skutečnost, že sekundy jsou uloženy v pěti bitech a pro rok zůstává bitů sedm. Jinak řečeno, pro sekundy můžeme použít 32 hodnot. Proto jsou uloženy zaokrouhleny na násobek dvou. Rok je uložen jako hodnota, kterou musíme přičíst k počátku, roku 1980. Umístění jednotlivých položek v bitovém poli ukazuje tabulka (z důvodů umístění na stránce je rozdělena do dvou částí):

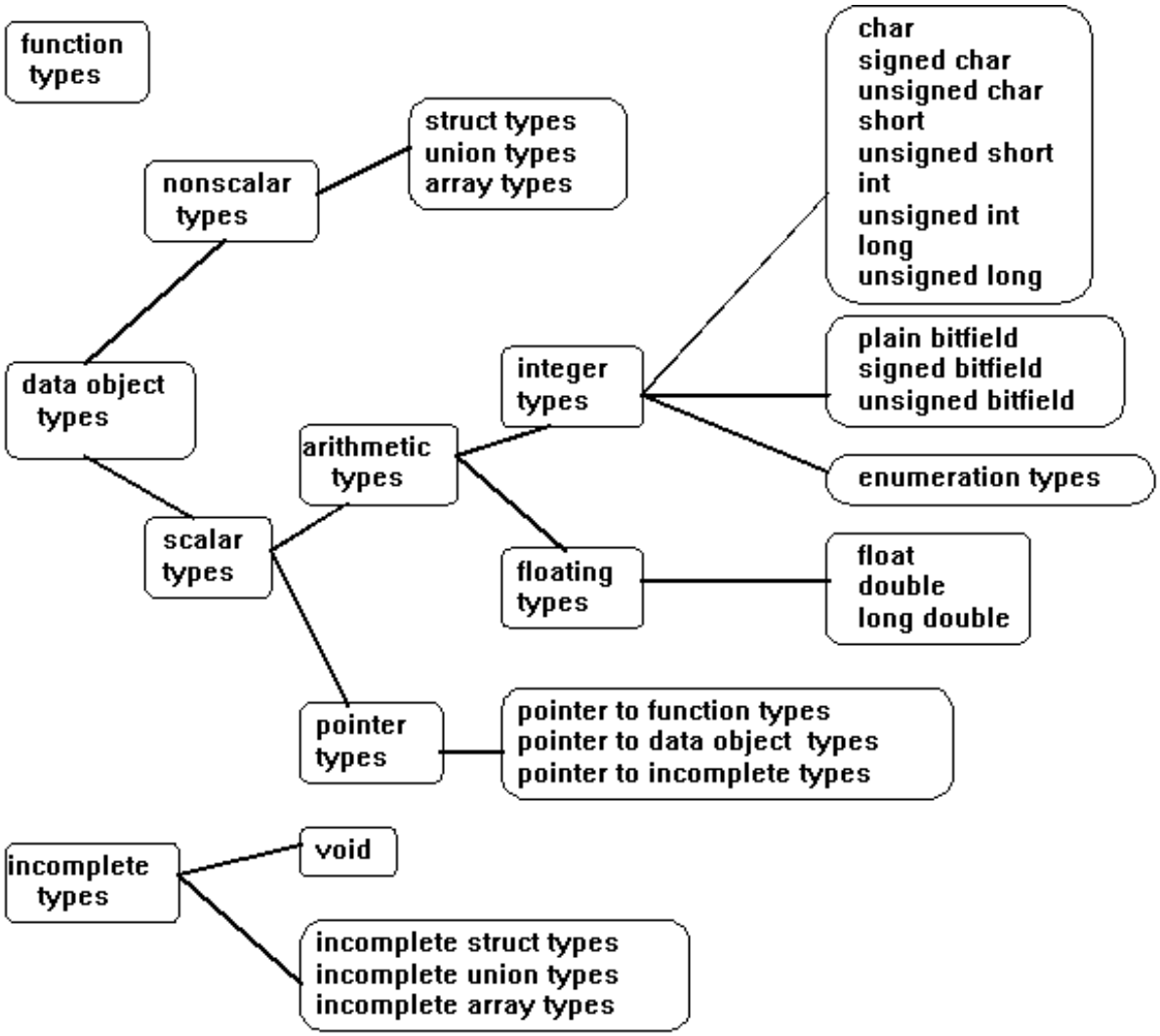
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ft_hour					ft_min						ft_sec				
hodiny					minuty						sekundy/2				

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ft_year							ft_month				ft_day				
rok - 1980							měsíc				den				

Jelikož se snažíme nepoužívat systémově závislé funkce, nebudeme si uvádět příklad použití bitových polí. Výše uvedená ukázka nám postačí.

Klasifikace typů v C

V této kapitole jsme dokončili výklad typů, které máme v C k dispozici. Proto jsme zahrnuli obrázek, který typy v C klasifikuje. Pro větší výrazové schopnosti angličtiny, umožňující popsat na nenší ploše potřebné skutečnosti, jsme v tomto obrázku nepoužili české termíny.



Obrázek: klasifikace typů v jazyce C

OBSAH