Markéta Trnečková

Jazyk C

Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci

Tento text slouží jako učební materiál pro kurz Jazyk C, který je vyučován v bakalářském studiu na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci. Text je určen pro studenty, u kterých se předpokládá, že znají základy programování v libovolném programovacím jazyce.

Poslední změna proběhla 13. září 2022. Pokud v textu naleznete chyby, prosím, kontaktujte mne prostřednictvím e-mailové adresy marketa.trneckova@upol.cz.

Děkuji Martinu Trnečkovi a Petru Osičkovi za jejich připomínky a cenné podněty, které přispěly ke kvalitě textu.

Markéta Trnečková

Obsah

Úvod do jazyka C

Tato kapitola je věnována představení jazyka C, jeho historii a také způsobu zpracování kódu. Dále zde budou zmíněny základní pojmy, které budeme v dalším textu používat.

Jazyk C

Jazyk C je nízkoúrovňový programovací jazyk, který je primárně určen pro systémové programování. Je to strukturovaný *imperativní* (procedurální) jazyk. To znamená, že výpočet je popsán posloupností příkazů, které mění stav programu (mají tzv. *vedlejší efekt*)[1] a určuje přesný postup (algoritmus), jak danou úlohu řešit. *Strukturovaný* znamená, že skoky typu goto nahrazuje pomocí podmíněných cyklů[2] a jiných strukturovaných příkazů, které je možné do sebe vnořovat.
Jazyk C je slabě[3] staticky typovaný[4] jazyk s lexikálním rozsahem platnosti proměnných.
Jazyk C významně ovlivnil syntax některých moderních jazyků, jako je Java, JavaScript, C#, PHP a jiné. Pro mnoho úloh je efektivnější a rychlejší, než jiné jazyky.
Jazyk C byl navržen a implementován pod operačním systémem UNIX a téměř celý UNIX je psán v jazyce C. Přesto se jazyk C na OS UNIX nijak neváže a neváže se ani na jiný OS nebo počítač. Programy psané v jazyce C jsou snadno přenositelné mezi počítači a operačními systémy.

Průvodce studiem
Jazyk C je nízkoúrovňový, strukturovaný, imperativní, slabě staticky typovaný jazyk.

[1] Typickým příkladem je příkaz přiřazení.
[2] Opakuj dokud platí podmínka.

[3] Slabý = dovoluje přiřadit hodnoty jednoho typu do jiného (dovoluje implicitně měnit typ hodnot).

[4] Staticky typovaný = ověřuje typy výrazů, zda odpovídají svým kontextům. Typ se ověřuje a je znám při kompilaci.

Historie

První kniha o jazyce C, *The C Programming Language*,[5] vyšla v roce 1978. Verze jazyka popsána v této knize byla považována za jeho první standard. V literatuře se můžeme setkat s označením *K&R standard*.
V roce 1988 vyšel nový standard *ANSI C*, který z K&R standardu vycházel. Jeho součástí je i přesná specifikace knihoven, kterou každá implementace ANSI C musí obsahovat. Naprostá většina dnešních překladačů této normě vyhovuje. Proto se této normě budeme v tomto

[5] Kernighan, B. W. and Ritchie, D. M. The C Programming Language. Prentice-Hall, Inc. (1978).

kurzu věnovat.[6] Výhodou používání této normy je, že program napsaný v tomto standardu s použitím pouze standardních knihoven je téměř 100% přenositelný na libovolný počítač.[7]
V současné době existuje rozšiřující standard *C99*.[8] Jeho některé prvky[9] nemusí všechny překladače podporovat. Na tyto prvky budeme dále upozorňovat. Novinky jsou spíše kosmetického rázu.
Některá vylepšení představená v dalších normách, například *C11*, *C1X*,[10] se neujala a nejsou plně podporována.

[6] Konkrétně verzi ISO/ANSI C x3.159-189

[7] Pod libovolný operační systém.

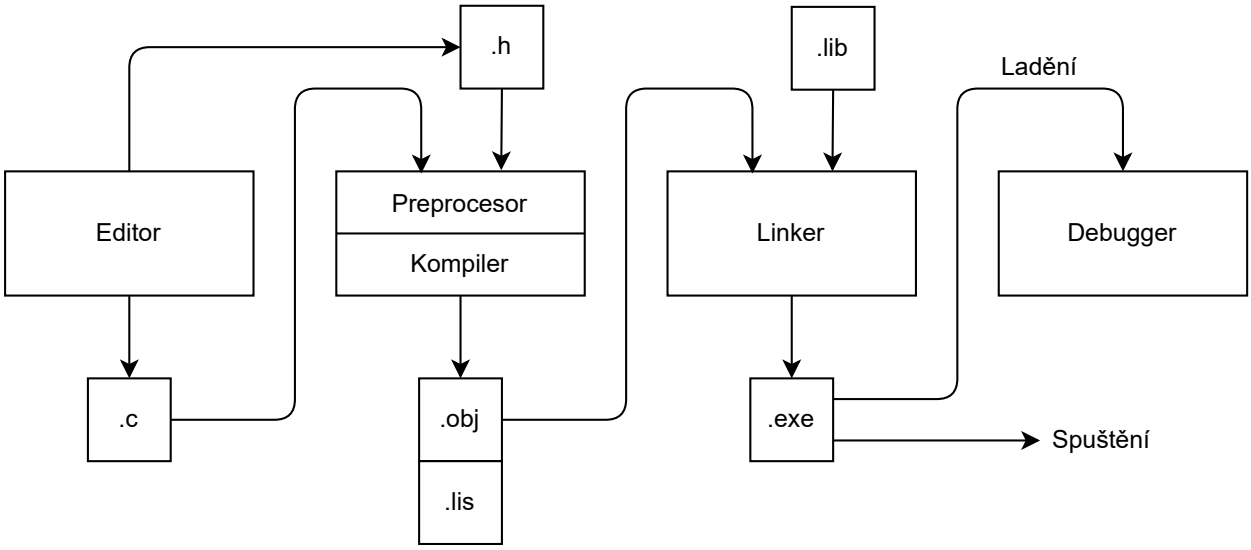
[8] Formálně ISO/IEC 9899:1999

[9] Například inline funkce, možnost definovat proměnné kdekoliv, nové datové typy, dynamická pole aj.

[10] ISO/IEC 9899:2011

Způsob zpracování programu

Způsob zpracování programu zapsaného v jazyce C je schématicky zobrazen na obrázku níže.



Nejprve je pomocí *editoru* vytvořen zdrojový kód, který se ukládá s příponou `.c`. Ten je poté zpracován pomocí *preprocesoru*, který je součástí *kompileru* (kompilátoru, překladače). Preprocesor, jak už název napovídá, předzpracovává zdrojový kód[11] tak, aby měl překladač jednodušší práci. Výsledkem je textový soubor, který je přímo předáván kompileru. Kompiler provádí překlad zdrojového souboru předzpracovaného preprocesorem do relativního (objektového) kódu počítače (vzniká soubor s příponou `.obj`). *Relativní kód*, nebo také jazyk relativních adres, je téměř hotový program. Adresy proměnných a funkcí nejsou známé,[12] jsou v souboru zapsány relativně. Vedlejším produktem je soubor s příponou `.lis`, ve kterém je uložena informace o chybách nalezených kompilerem.[13] Tento soubor se dále nepoužívá a generuje se jen na přání uživatele. *Linker* (sestavovací program) přidělí relativním adresám absolutní adresy a provede všechny odkazy na dosud neznáme identifikátory.[14] Výsledkem je spustitelný soubor s příponou `.exe` (v textu před-

[11] Např. vynechává komentáře, zajišťuje správné vložení hlavičkových souborů (souborů s příponou `.h`), stará se o rozvoj maker a jiné.

[12] Například jsou uloženy v knihovně.

[13] Například syntaktické chyby.

[14] Například na volané knihovní funkce uložené v souborech s příponou `.lib`.

pokládáme pod operačním systémem Windows). *Debugger* (česky „odvšivovač“) je program sloužící k hledání chyb (ladění), které nastávají za běhu programu.[15]

[15] Více se hledání chyb budeme věnovat v kapitole Ladění.

První program

Klasický program, který vypíše na obrazovku „ahoj svete“, zapsaný v jazyce C vypadá následovně.

```
#include <stdio.h>

int main(){
    /*program vypise ahoj svete*/
    printf("ahoj svete");
    return 0;
}
```

Průvodce studiem
Každý program psaný v jazyce C musí obsahovat právě jednu funkci `main()`.

Program obsahuje funkci `main()`. Obecně můžeme pojmenovávat funkce téměř libovolně,[16] ale všechny programy musí obsahovat právě jednu funkci `main()`.[17] Vykonávání programu začíná na začátku funkce `main()`. Ve funkci `main()` jsou většinou volány další funkce, ať už uživatelsky definované nebo dostupné ve formě knihoven.
O výpis textu na obrazovku se stará funkce `printf()`.[18] Kód neobsahuje její definici, abychom jí mohli použít, je potřeba informaci o této funkci do kódu vložit. `printf()` je součástí standardní knihovny pro vstup a výstup `stdio` (Standard Input/Output)[19] a o její vložení do kódu se stará příkaz

[16] Více se pojmenovávání proměnných a funkcí budeme věnovat v části Štábní kultura.

[17] Jednu funkci `main()` musí obsahovat i v případě, kdy se program skládá z více souborů (modulů).

[18] Více si o této funkci řekneme později.

[19] Striktně řečeno `stdio` není knihovna, ale součást standardní knihovny.

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

Jak již víme z dřívějšího studia, funkcím je možné předávat argumenty. V jazyce C se argumenty píší do kulaté závorky hned za název funkce. V našem případě je funkce `main()` definována tak, že neočekává žádné vstupní argumenty.[20] `int` před názvem funkce udává typ návratové hodnoty.[21]
Příkazy funkce jsou uzavřeny v `{` a `}` (příkazům uzavřeným ve složených závorkách říkáme *blok*). Jednotlivé příkazy jsou ukončeny `;` (středníkem).
Funkce `main()` mimo jiné obsahuje příkaz

[20] Není to však zcela pravda, argumenty by se jí předaly, jen by se nenavázaly na žádné lokální proměnné. Pro naše účely nyní ale toto zjednodušení stačí.

[21] Funkcím a datovým typům se budeme věnovat později.

```
printf("ahoj svete");
```

Průvodce studiem
Funkce `printf()` se používá pro formátovaný standardní výstup.

Příkazem je volána funkce `printf()`, které předáváme argument ''`Ahoj svete`''. Funkce je ukončena příkazem `return`, o jehož významu si řekneme v následující kapitole.

Kód je možné doplnit o *komentáře*. Komentáře se píší do bloku ohraničeného `/*` a `*/`. Tyto komentáře mohou být víceřádkové a mohou se objevit všude, kde se v kódu vyskytuje bílý znak.[22] V komentáři se mohou vyskytovat libovolné znaky. ANSI C standard nepovoluje vnořené komentáře.[23]
Komentáře jsou důležitou součástí programu, slouží k jeho zpřehlednění, je tedy dobré je používat před každou logicky ucelenou částí. Měly by obsahovat jen důležité informace týkající se kódu.

[22] Bílým znakem rozumíme mezery, tabulátory, nové řádky a podobně.

[23] `/* /* vnoreny */ komentar */`

Nový standard ISO
Nový standard umožňuje *jednořádkové komentáře*. Komentář začíná `//` a končí s koncem řádku.

Štábní kultura

Než si vyzkoušíme napsat a přeložit první program, je dobré si říci něco o pojmenovávací konvenci, kterou budeme v textu používat.
Při pojmenovávání souboru obsahujícího kód v jazyce C se snažíme vyhnout nejednoznačným názvům. Název by měl odpovídat tomu, co daný program dělá. Pokud se program skládá z více souborů (modulů),[24] použijeme několik prvních znaků názvu pro identifikaci projektu a zbytek pro název modulu. Například `pr1_vstup.c`, `pr1_vystup.c`, ...
Každá proměnná a funkce má svůj název, kterému říkáme *identifikátor*. Jazyk C je *case sensitive*, což znamená, že rozlišuje malá a velká písmena. `id`, `Id`, `ID` označují tři různé identifikátory. Identifikátor je v jazyce C kombinace malých a velkých písmen, číslic a znaku `_` (podtržítko). Identifikátor nemůže začínat číslicí a jako identifikátor není možné použít *klíčová slova* jazyka.[25] Názvy začínající podtržítkem se používají pro systémové identifikátory. Pro přehlednost se nedoporučuje podtržítko používat na konci názvu. Délka identifikátoru není omezená, ale ANSI C rozeznává pouze prvních 31 znaků (další bere jako nevýznamné).
Obecně se doporučuje používat v názvech identifikátorů malá písmena a využívat podtržítka (této konvenci se říká *snake konvence*). Znak `_` je použit pro oddělení slov (pokud jich je více). Například `nazev_promenne` nebo `ukazatel_na_cislo`. Nedoporučuje se používat podobné názvy[26] nebo používat stejné identifikátory lišící se pouze velikostí písma.[27] Názvy by měly naznačovat, co funkce dělá, případně jakou informaci proměnná obsahuje.[28] Běžně se v jazyce C používají následující významové identifikátory:

- `i`, `j`, `k` – indexy
- `c`, `ch` – znaky
- `m`, `n` – čítače
- `f`, `r` – reálná čísla
- `p_` – začátek ukazatele (pointeru)
- `s` – řetězce

[24] Rozdělení programu do více souborů se budeme věnovat později.

[25] Slova, která mají v jazyce speciální význam, například `while`, `if`, `for`, ... Klíčová slova jsou psána malými písmeny, jinak jsou brána jako identifikátory. `While` je identifikátor, zatímco `while` je klíčové slovo.

[26] Například `pr1` a `prl`.

[27] Například `Id` a `id`.

[28] Z názvu `plat` je zřejmé, co identifikátor uchovává za informaci, na rozdíl od nic neříkajícího názvu `promenna_1`.

Shrnutí

Stručně jsme si představili jazyk C, jeho historii a také, jakým způsobem se kód psaný v něm zpracovává. Dále jsme si napsali a přeložili první program psaný v jazyce C.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co znamená, že je jazyk imperativní?
2. Co znamená, že je jazyk slabě staticky typovaný?
3. Jakým způsobem je zdrojový kód jazyce C zpracováván?
4. Jakou funkci musí každý kód psaný v jazyce C obsahovat?

Cvičení

Úkol 1

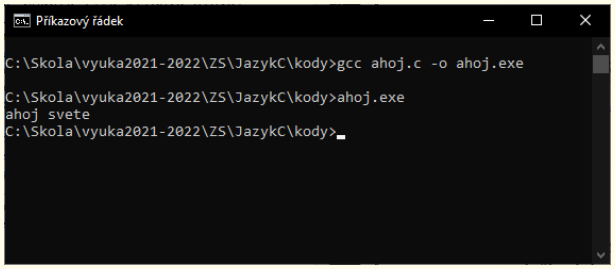
1. Nainstalujte si překladač jazyka C. V ukázkách textu bude použit *gcc* překladač.
2. Pomocí libovolného textového editoru vytvořte soubor `ahoj.c`. Soubor bude obsahovat následující kód:

```c
#include <stdio.h>

int main(){
    /*program vypise ahoj svete*/
    printf("ahoj svete");
    return 0;
}
```

3. Otevřete příkazový řádek (případně terminál), přesuňte se do složky, kde máte uložený soubor `ahoj.c` a pomocí následujícího příkazu jej přeložte:

```
gcc ahoj.c -o ahoj
```

Pomocí přepínače `-o` specifikujeme, jak se má přeložený soubor jmenovat.[29] Ve složce vznikl soubor ahoj.exe, který je možné spustit. Po spuštění se do příkazového řádku vypíše text „ahoj svete“. Viz následující obrázek.



[29] Pomocí příkazu `gcc --help` zobrazíte nápovědu k příkazu `gcc`.

Ve většině systémů je možné tyto dva kroky sloučit do jednoho příkazu s použitím `&&`

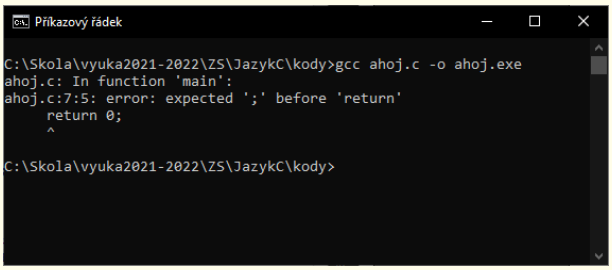
```
gcc ahoj.c -o ahoj && ahoj
```

4. Použitím přepínače `-E` při překladu zdrojového kódu se provede jen předzpracování pomocí preprocesoru. Vyzkoušejte následující příkaz a podívejte se na vygenerovaný soubor `ahoj.txt`.

```
gcc ahoj.c -E -o ahoj.txt
```

5. Vyzkoušejte i jiné přepínače, například `-S`, nebo `-c`.

Úkol 2
Vyzkoušejte smazat středník za příkazem `printf()` a kód přeložte. V příkazovém řádku by se měla objevit chyba při překladu, viz následující obrázek.



Při použití složeného příkazu pro přeložení a spuštění

```
gcc ahoj.c -o ahoj && ahoj
```

se v tomto případě program nespustí, pouze se vypíše seznam chyb.

Úkol 3
Nainstalujte si libovolné vývojové prostředí[30] a vyzkoušejte předchozí kód zkompilovat pomocí něj. [31]

Úkol 4
Upravte předchozí kód tak, aby vypsal jiný text.

[30] Vývojové prostředí, tzv. *IDE* (Integrated Development Environment) obsahuje editor zdrojového kódu, kompilátor (případně interpret) a většinou také debugger.

[31] V textu budeme používat CodeBlock `http://www.codeblocks.org/`. Dále je možné využívat například Visual Studio (v tom případě je třeba vytvářet prázdný projekt).

Začátky s jazykem C

V této kapitole se budeme zabývat základními stavebními prvky programů, kterými jsou *hodnota*, *výraz*, *příkaz*, datovým typům a operátorům.

Základní datové typy

V jazyce C existuje několik základních datových typů, ze kterých lze dále vytvářet typy složené.[32] Všechny základní typy jsou číselné. Každý typ je dán svým klíčovým slovem a také svým rozsahem hodnot, kterých může proměnná daného typu nabývat (rozsah je dán velikostí paměti, která je proměnné přiřazena).
Mezi *celočíselné* datové typy patří `int`, `short int` (zkráceně `short`) a `long int` (zkráceně `long`). Velikost `int` není pevně daná, obvykle odráží velikost celých čísel na počítači (většinou 32 bitů). Dle standardu vždy platí
`sizeof(short int)` $\leq$ `sizeof(int)` $\leq$ `sizeof(long int)`.[33]
Pro reprezentaci znaků abecedy, číslic a jiných znaků se používá datový typ `char` (`char` je také celočíselný datový typ). Má vždy velikost 1 byte. Čísla základních znaků udává ASCII tabulka, která je zobrazena v tabulce 1.
Pro čísla s pohyblivou desetinnou čárkou existují datové typy `float` (jednoduchá přesnost), `double` (dvojitá přesnost) a `long double` (čtyřnásobná přesnost). Opět dle standardu vždy platí
`sizeof (float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)`.
Datový typ `float` odpovídá většinou číslům s 6–9 číslicemi, `double` s 15–17. Čísla s pohyblivou desetinnou čárkou se definují s desetinnou tečkou nebo v exponenciálním zápise.
Všechny celočíselné datové typy mohou být buď typu `signed` (znaménkové) nebo `unsigned` (neznaménkové).[34] Vždy platí
`sizeof(unsigned int)` $=$ `sizeof(signed int)`[35]
Rozdíl mezi `signed` a `unsigned` je v rozsahu čísla. Proměnné typu `unsigned` nemohou zobrazit záporná čísla a mají rozsah od 0 do $2^n - 1$, kde n představuje počet bitů alokovaných danému typu. `signed` mají rozsah od -2^{n-1} do $2^{n-1} - 1$. Například pro typ `char`

[32] Složeným datovým typům se budeme věnovat později.

Nový standard ISO
Pro hodně velká celá čísla byl zaveden typ `long long int`. Některé překladače sice tento typ znají, ale nepracují s ním správně.

[33] Operátor `sizeof()` vrací velikost typu v bytech.

Nový standard ISO
Nový standard poskytuje celočíselné typy s přesně daným počtem bitů. V hlavičkovém souboru `stdint.h` jsou informace o typech ve tvaru `intN_t` (případně pro neznaménkové `uintN_t`), kde `N` je číslo vyjadřující počet bitů, například `int8_t`, `int16_t`, `int32_t`, `uint8_t`, ...

[34] Implicitně jsou `short int`, `int` a `long int` typu `signed`. Pro typ `char` to záleží na konkrétní implementaci.

[35] Stejně tak pro ostatní datové typy.

Průvodce studiem
Všechny základní datové typy v jazyce C jsou číselné.

číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak	číslo	znak
0	(nul)	16	(dle)	32	␣	48	0	64	@	80	P	96	‘	112	p
1	(soh)	17	(dc1)	33	!	49	1	65	A	81	Q	97	a	113	q
2	(stx)	18	(dc2)	34	"	50	2	66	B	82	R	98	b	114	r
3	(etx)	19	(dc3)	35	#	51	3	67	C	83	S	99	c	115	s
4	(eot)	20	(dc4)	36	$	52	4	68	D	84	T	100	d	116	t
5	(enq)	21	(nak)	37	%	53	5	69	E	85	U	101	e	117	u
6	(ack)	22	(syn)	38	&	54	6	70	F	86	V	102	f	118	v
7	(bel)	23	(etb)	39	'	55	7	71	G	87	W	103	g	119	w
8	(bs)	24	(can)	40	(	56	8	72	H	88	X	104	h	120	x
9	(tab)	25	(em)	41	)	57	9	73	I	89	Y	105	i	121	y
10	(lf)	26	(eof)	42	*	58	:	74	J	90	Z	106	j	122	z
11	(vt)	27	(esc)	43	+	59	;	75	K	91	[	107	k	123	{
12	(np)	28	(fs)	44	,	60	<	76	L	92	\	108	l	124	|
13	(cr)	29	(gs)	45	-	61	=	77	M	93	]	109	m	125	}
14	(so)	30	(rs)	46	.	62	>	78	N	94	^	110	n	126	~
15	(si)	31	(us)	47	/	63	?	79	O	95	_	111	o	127	(del)

Tabulka 1: Základní ASCII tabulka. Jednotlivé znaky jsou reprezentovány pomocí 7-bitových čísel. Znaky v závorkách jsou speciální řídící znaky.

to znamená, že `unsigned char` může mít hodnoty 0 až 255 a `signed char` v rozmezí −128 a 127.
Jazyk C neposkytuje přímo datový typ pro logické hodnoty (boolean). Logické hodnoty jsou reprezentovány pomocí celočíselných datových typů, kde nulová hodnota představuje nepravdu (false) a nenulová pravdu (true).

Nový standard ISO
V hlavičkovém souboru `stdbool.h` je definovaný datový typ `_Bool` a makra `true` a `false`.

Uživatelské datové typy

S využitím operátoru `typedef` je možné vytvořit nové jméno datového typu. Syntaxe je následující:

Průvodce studiem
Pomocí operátoru `typedef` můžeme vytvořit nový datový typ.

```
typedef definice_typu nazev_noveho;
```

`nazev_noveho` označuje název nového datového typu, který definuje
`definice_typu`. Konkrétní příklad následuje.

```
typedef int cele_cislo;
```

`cele_cislo` je v tomto případě synonymem pro typ `int`.[36] Použití je stejné jako u základních datových typů.

[36] Takto se operátor `typedef` používá málokdy, spíše se bude hodit později, až se budeme bavit o strukturovaných datových typech.

```
cele_cislo cislo; /* ekvivalentni int cislo;*/
```

Výčtový typ

Výčtový typ (enumerate type) se v programech používá celkem často, umožňuje totiž zpřehlednit kód. Pomocí výčtového typu lze snadno definovat seznam symbolických (pojmenovaných) konstant, které mají spolu nějakou souvislost. Výčtový typ se definuje následovně.

Průvodce studiem
Výčtový typ nám umožní vytvořit proměnnou, která může obsahovat pouze hodnoty konstant určených při deklaraci výčtového typu.

```
enum nazev {
    HODNOTA1, HODNOTA2, ..., HODNOTAN};
```

Tímto příkazem definujeme datový typ `enum nazev`. Proměnné tohoto datového typu mohou nabývat hodnot `HODNOTA1`, ..., `HODNOTAN` (tyto hodnoty nazýváme také *výčtové konstanty*). Názvy hodnot, jelikož je bereme jako konstanty, zpravidla píšeme velkými písmeny. Pokud jim explicitně nepřiřadíme nějakou hodnotu,[37] tak nabývají hodnot postupně `0`, `1`, `2` atd. Je možné přiřadit více výčtovým konstantám stejné hodnoty.
Pomocí `typedef` je možné datový typ pojmenovat. Nejčastěji se tedy setkáte s následující definicí výčtového typu:

[37] Explicitní inicializace vypadá následovně:

```
enum nazev {
    HODNOTA1 = hodnota1,
    HODNOTA2 = hodnota2,
    ...,
    HODNOTAN = hodnotan};
```

`HODNOTA1` má přiřazenu hodnotu `hodnota1` a tak dále. Není potřeba inicializovat všechny hodnoty (pak se jim přiřadí hodnota automaticky), ale velmi se to nedoporučuje.

```
typedef enum {
    HODNOTA1, HODNOTA2, ..., HODNOTAN
    } NAZEV;
```

Vytvořili jsme anonymní (bezejmenný) výčtový typ a pomocí `typedef` jsme mu přiřadili název `NAZEV` (i zde se zpravidla používají velká písmena).
Typickým příkladem výčtového typu je typ BOOLEAN.

```
typedef enum {
    FALSE, TRUE
} BOOLEAN;
```

Jména položek výčtového typu není možné přímo vytisknout, pouze jejich hodnoty.

Literály

Celočíselné hodnoty můžeme zapisovat v různých soustavách. V *desítkové soustavě*, kde je číslo tvořeno posloupností číslic, z nichž první nesmí být `0` (výjimkou je číslo `0`). Příkladem jsou čísla `1`, `42`, `123`. V *osmičkové soustavě*, kde každé číslo začíná číslicí `0` a pak následuje posloupnost osmičkových číslic (`1-7`). Například čísla `01`, `052`, `0173`. Čísla v *hexadecimální soustavě* začíná číslicí `0` následuje znak `x` (případně `X`) a pak posloupnost hexadecimálních číslic (`0-9`, `A-F`,

a-f). Například čísla 0x1, 0x2a, 0X7B. Celočíselné hodnoty jsou typu int. Pro hodnoty typu long int použijeme příponu L (případně l), např. 123L. Další příponou je U (u). Ta určuje, že hodnota je typu unsigned. Příkladem jsou čísla 123u nebo 123LU. Záporné hodnoty jsou uvozené znakem -.
Reálné hodnoty jsou implicitně typu double a jsou to čísla obsahující desetinnou tečku (ta může být i na začátku i na konci). Například .123, 123., 1.23. Reálná čísla také můžeme zapisovat v exponenciálním tvaru 1e23 nebo 12E3. Pokud chceme reálnou hodnotu typu float použijeme příponu F (f), pro long double použijeme L (l).
Znakové hodnoty (znaky) se zapisují do apostrofů, například 'a', '*', '2'. Odpovídající číselná hodnota je odvozena z ASCII tabulky (viz tabulka 1). Některé znaky, například znak nového řádku, tabulátor, apostrof a jiné, není možné přímo do apostrofu zapsat. Tyto znaky zapisujeme pomocí tzv. *escape sekvence*, což je sekvence několika znaků uvozená znakem \. Mezi často používané escape sekvence patří '\n' představující nový řádek, '\t' tabulátor, '\'' apostrof, '\\' zpětné lomítko nebo '\0' nulový znak.[38]
Řetězcové hodnoty (řetězce) jsou uzavřené do uvozovek. Mezi ně je možné psát znaky v libovolném zápisu (tedy i pomocí escape sekvencí). Příkladem je řetězec ''Ja jsem \'' retezcova \'' hodnota \n''
ANSI C automaticky zřetězí dlouhé řetězce oddělené mezerami, novými řádky nebo tabulátory, což zlepšuje přehlednost kódu.

[38] Případně se můžeme setkat se zápisem ve tvaru '\ddd', kde ddd představuje kombinaci tří osmičkových číslic, nebo '\xhh', kde hh jsou hexadecimální číslice. Čísla reprezentují ASCII hodnotu znaku.

Proměnné

Každá proměnná má své jméno (identifikátor), jehož konvenci jsme zavedli v předchozí kapitole. Jak bylo již řečeno, jazyk C je staticky typovaný jazyk. Staticky typované jazyky vždy spojují typ se jménem proměnné. V těchto jazycích je při vytváření proměnné nutno určit i její typ, případně se při překladu její typ odvodí. Proměnná pak může obsahovat pouze data tohoto typu. Před prvním použitím proměnné, je nutné jí předem definovat nebo deklarovat následovně.[39]

[39] Definicí rozumíme, že vznikla proměnná a překladač pro ní alokuje místo v paměti. Deklarací říkáme překladači, že taková proměnná existuje (například je definovaná v jiném souboru), překladač nepotřebuje takové proměnné alokovat místo. Pokud pracujeme s nedefinovanou/nedeklarovanou proměnnou, dojde k chybě.

```
typ jmeno;
```

typ představuje datový typ proměnné s identifikátorem jmeno. Konkrétním příkladem je definice proměnné cislo typu int.

```
int cislo;
```

Na jednom řádku je možné definovat více proměnných stejného datového typu.[40] Například:

[40] To se obecně používá jen u pomocných proměnných.

```
int cislo_1, cislo_2, cislo_3;
```

Deklarace/definice proměnných se mohou vyskytnout vně funkce (globální proměnné) nebo uvnitř funkce (lokální proměnné),[41] například:

[41] Více se platnosti proměnných budeme věnovat později.

```
int i;
int main(){
    int j;
    return 0;
}
```

Proměnná `i` je globální zatímco `j` je lokální proměnná.
V ANSI C se musí proměnné definovat vždy na začátku bloku (pro připomenutí, bloky jsou části kódu omezené `{` a `}`) a až poté mohou následovat další příkazy. Platnost proměnné je v rámci těchto bloků. Definice proměnných nesmí být v hlavičkových souborech.[42] Lokální proměnné nejsou automaticky inicializovány (jejich hodnota je náhodná), globální proměnné jsou implicitně inicializovány na 0.[43]

Nový standard ISO
Ve standardu C99 je možné definovat proměnné kdekoliv v bloku, tedy ne nutně na jeho začátku. Platnost proměnné je od její definice do konce bloku.

[42] O hlavičkových souborech a tzv. *extern* deklaraci budeme mluvit později.

[43] `int` proměnné mají hodnotu `0`, `float` mají hodnotu `0.0`, `char` mají hodnotu `'\0'`, atd.

Operátory

Operátor představuje operaci, pomocí které je možné kombinovat výrazy a tím vytvářet výrazy složitější. Každý operátor aplikujeme na *operandy* (aplikací operátoru vznikne hodnota, můžeme tedy říct, že vrací hodnotu). Ke každému operátoru je třeba vědět, na kolik operandů jej můžeme aplikovat. Tomuto číslu se říká *arita*.[44]
Kromě arity u operátorů zavádíme také pojem *priority*. A to z důvodu, abychom věděli, jak vyhodnocovat výrazy, které obsahují více operátorů. Priority operátorů jsou vypsány v tabulce 2.[45] Čím je číslo priority nižší, tím dříve se tento operátor vyhodnocuje (má vyšší prioritu).
Například

[44] Pokud lze operátor aplikovat jen na jeden operand, pak mluvíme o *unárním operátoru*, na dva *binárním operátoru*, na tři *ternárním operátoru*. Operátory s vyšší aritou v jazyce C nejsou.

[45] Podrobněji se konkrétním operátorům budeme věnovat později.

```
a = 2;
b = 4;
c = a++ - b / 2;
```

ve výrazu `a++ - b / 2` se nejprve vypočítá `a++`, protože operátor `++` má nejvyšší prioritu (`a` se nově rovná 3, ale vrácena je hodnota 2), následně se vypočítá výraz `b / 2` (výsledkem je číslo 2) a nakonec se vyhodnotí `2 - 2`, protože operátor `-` má z operátorů nejnižší prioritu.

priorita	operátor	význam	asociativita

1	`()`
	`[]`
	`->`
	`.`
		volání funkce
		index do pole
		přístup k prvku struktury
		přístup k prvku struktury
			zleva doprava

2	`!`
	`~`
	`++`
	`--`
	`+`
	`-`
	`(typ)`
	`*`
	`&`
	`sizeof`
		negace logického výrazu
		jedničkový doplněk
		inkrementace
		dekrementace
		unární plus (+1)
		unární mínus (-1)
		přetypování
		dereference
		reference (adresový operátor)
		velikost typu
			zprava doleva

3	`*`
	`/`
	`%`
		součin
		dělení
		dělení modulo
			zleva doprava

4	`+` `-`	součet rozdíl	zleva doprava
5	`<<` `>>`	bitový posun doleva bitový posun doprava	zleva doprava

6	`<`
	`<=`
	`>`
	`>=`
		menší než
		menší nebo rovno než
		větší než
		větší nebo rovno než
			zleva doprava

7	`==` `!=`	rovnost nerovnost	zleva doprava
8	`&`	bitový součin	zleva doprava
9	`^`	exkluzivní bitový součin	zleva doprava
10	`|`	bitový součet	zleva doprava
11	`&&`	logický součin	zleva doprava
12	`||`	logický součet	zleva doprava
13	`? :`	ternární (podmínkový) operátor	zprava doleva

14	`=`
	`>>=`
	`<<=`
	`+= -=`
	`*= /=`
	`%= &=`
	`|= ^ =`
		přiřazovací operátory	zprava doleva

15	`,`	operátor čárky	zleva doprava

Tabulka 2: Tabulka priorit a asociativita operátorů. Nižší číslo značí vyšší prioritu.

Pokud se ve výrazu vyskytnou operátory se stejnou prioritou, o pořadí jejich vyhodnocení rozhoduje jejich *asociativita* (asociativita udává z jakého směru se operátory vyhodnocují), která je ve čtvrtém sloupci tabulky.

```
a = 2;
b = 4;
c = a + 2 - b;
```

Ve výrazu `a + 2 - b` se objevují operátory + a -, které mají stejnou prioritu a vyhodnocují se zleva doprava. Postupně se tedy vyhodnotí `a + 2` (jedná se o výraz `2 + 2` a ten se vyhodnotí vyhodnotí na `4`), poté `4 - 4`, celý výraz se tedy vyhodnotí na `0`.
Pro základní operace s číselnými hodnotami slouží *aritmetické operátory*. Dalším speciálním operátorem, o kterém si v této kapitole řekneme je *operátor přiřazení*.

Operátor přiřazení

K přiřazení hodnoty proměnné se používá *(jednoduchý) operátor přiřazení* =.

```
l_value = r_value;
```

`l_value` představuje „adresu“ (odkazuje na nějaké fyzické místo v paměti), kam se ukládá hodnota výrazu `r_value`. `r_value` má hodnotu, ale nemusí nutně existovat v paměti, může to být konstanta,[46] výsledek operace a podobně. `l_value` může být proměnná, ale nemůže to být výraz ani konstanta.[47] Po přiřazení má výraz hodnotu levého operandu.
Přiřazovat hodnotu proměnné je možné už při její definici (je možné proměnnou inicializovat). Různé typy přiřazovacích příkazů jsou uvedeny v následujícím příkladu.

[46] Ta v paměti existuje, ale není možné jí měnit hodnotu.

[47] Pozor na terminologii.
`i + 2` je *výraz*
`i = 2` je *přiřazení*
`i = 2;` je *příkaz*

```
int a = 5, b = 6, c, d;

c = 7;
c = b;
b = 3;
c = c - b;
```

Úkol 5
Jaké jsou hodnoty *b* a *c* v jednotlivých krocích?

Protože přiřazení je výraz, je možné několikanásobné přiřazení.

```
a = b = c = 2;
```

Tento příkaz se vyhodnocuje zprava doleva, tedy proměnné `c` je přiřazena hodnota `2`. Výraz `c = 2` je roven hodnotě `2` a tak dále.

Průvodce studiem
Nejen k ovlivnění pořadí vykonávání operátorů, ale také pro lepší čitelnost kódu, se používají kulaté závorky.

```
a = (b = (c = 2));
```

U operátoru přiřazení je výsledek vždy toho typu, jako je `l_value`. Pokud přiřadíme hodnotu typu, který je větší než typ `l_value`, dojde k zaokrouhlení nebo jiné ztrátové úpravě této hodnoty (tzv. implicitní typová konverze).[48]

[48] Pojmu typová konverze se budeme věnovat později.

Aritmetické operátory

Aritmetické operátory slouží k základním operacím s číselnými hodnotami, jsou obdobou aritmetických operátorů, které znáte z matematiky.

Unární operátory + a -

Unární plus + a unární mínus - se v jazyce C používají v běžném významu, jak známe z matematiky. Unární + bylo do jazyka přidáno kvůli symetrii k unárnímu - a prakticky se nepoužívá.

Binární operátory +, -, *, /, %

Operátor + značí sčítání, - rozdíl, * součin, / podíl[49] a % zbytek po celočíselném dělení.
Aplikací operátoru vznikne výsledek. Výsledek použití aritmetického operátoru je vždy toho typu, který má z typů všech jeho operandů nejvyšší prioritu. Typy s plovoucí čárkou mají vyšší prioritu, než celočíselné typy. Obecně typy, které mají větší rozsah, mají vyšší prioritu.[50] Z toho plyne, že pokud použijeme operátor / na dvě celá čísla, jedná se o celočíselné dělení. Pokud je alespoň jeden s pohyblivou desetinnou čárkou, pak je dělení reálné.
Podívejme se na následující příklad. Výsledky jsou popsány v komentářích.

[49] Buď celočíselný nebo reálný, v závislosti na operandech.

[50] U celočíselných typů (`char`, `int`, `long int`) je `long int` před `int` a oba mají vyšší prioritu než `char`.

```
int i = 10, j = 4, k;
float f, g = 10.0, h;
/* k i f jsou rovny 2, protoze i a j jsou typu int, podil je
 typu int. Vysledek dostaneme odstranenim  desetinne casti
     */
k = i / j;
f = i / j;
/* f bude mit hodnotu 2.5, protoze g je float,
 podil je take typu float. */
f = g / j;
/* k bude rovno 2, ztrati se desetinna cast. */
k = f;
/* h bude rovno 0.5. Proc? */
h = f - k;
```

Operátory ++ a --

Operátory inkrementace a dekrementace. Operátor ++ zvýší hodnotu o 1, operátor -- sníží hodnotu o 1. Tyto operátory lze psát před, nebo za operand,[51] obojí má ale různý význam.
Pokud operátor napíšeme za operand, pak se operand o jedničku zvětší/zmenší, ale výsledkem operace (hodnota vrácena provedením operace) bude původní hodnota operandu. Pokud operátor napíšeme před operand, hodnota operandu se zvětší/zmenší a vrácená hodnota bude rovna nové hodnotě operandu. Podívejme se na následující příklad:[52]

[51] Operand musí být `l_value` (např. proměnná). Není možné použít `42++` nebo `++(i + j)`.

[52] Pro připomenutí, funkce `printf()` slouží k výpisu na obrazovku. Více o této funkci si řekneme v následující části.

```c
#include <stdio.h>

int main(){
    int a = 0;
    int b = 10;
    a = b++;
    printf("hodnota a je %d a hodnota b je %d \n", a, b);
    a = ++b;
    printf("hodnota a je %d a hodnota b je %d \n", a, b);
    return 0;
}
```

Na začátku jsou proměnné `a` a `b` inicializovány na hodnotu `10`. Výraz `a = b++;` zvýší hodnotu proměnné `b` o jedna (`b` je rovno `11`), výsledkem `b++` je původní hodnota proměnné `b`, která je uložena do `a` (`a` je rovno `10`). Funkce `printf()` vypíše `''hodnota a je 10 a hodnota b je 11''`. Výraz `a = ++b;` zvýší hodnotu proměnné `b` o jedna (`b` je rovno `12`), výsledkem `++b` je nová hodnota proměnné `b`, která je uložena do `a` (`a` je rovno `12`). Tedy funkce `printf()` vypíše `''hodnota a je 12 a hodnota b je 12''`.

Přiřazovací operátory

Obdobně jako se používá samotný operátor přiřazení, se používají i další operátory přiřazení, které mohou transformovat i přiřazovat hodnotu v jediné operaci. Například

- `+=` přiřazení sčítání, zkratka pro

  `l_value = l_value + r_value`

- `-=` přiřazení odečítání, zkratka pro

  `l_value = l_value - r_value`

- `*=` přiřazení násobení, zkratka pro

  `l_value = l_value * r_value`

- `/=` přiřazení dělení, zkratka pro

  `l_value = l_value / r_value`

Operátor lze kombinovat i s dalšími aritmetickými a logickými operátory.

Operátor čárky

Operátor čárky je binární operátor, který se zapisuje následovně:

```
vyraz1, vyraz2
```

Tento výraz se zpracovává tak, že se vyhodnotí `vyraz1` a výsledek tohoto vyhodnocení je zapomenut, pak se vyhodnotí `vyraz2` a jeho výsledek je výsledkem celého výrazu.

```
int i = 2, j = 3; /* Toto neni operator carky */

j = (i++, i - j);
```

Po vyhodnocení výrazu z předchozího příkladu bude `i` rovno `3` a `j` bude `0`. Závorka je v příkladu nutná, protože operátor čárky má nižší prioritu než operátor přiřazení.

Typová konverze

Pojmem typová konverze (zkráceně *přetypování*) se označuje převod hodnoty určitého datového typu na jiný. Například převod celočíselného `int` na desetinný `float`. Některé konverze se provádějí automaticky (například již zmíněné změny při provádění aritmetických operací a přiřazování). Tyto automatické konverze nazýváme *implicitní*. Mimo to lze i přetypování vynutit pomocí tzv. *explicitní* typové konverze.

Průvodce studiem
Některé převody hodnot z jednoho typu na druhý jsou bezpečné. Tím myslíme, že je možné hodnoty tohoto typu převést, aniž bychom ztratili nějakou informaci. Mezi bezpečnou konverzi patří například přetypování `char` hodnoty na `int`, protože `char` vždy v paměti zabírá méně místa než `int`. U některých konverzí, ale může dojít ke ztrátě nějaké informace. Například, když převádíme číslo typu `float` na celé číslo `int`. Zde dojde ke ztrátě informace o desetinné části čísla.

Implicitní typová konverze

Kromě datové konverze při aplikaci aritmetických operátorů a také při operaci přiřazení, dochází i k následující konverzi. Před vykonáním operace se operandy základních datových typů konvertují na `int` v případě, že se jedná o datové typy `char` a `short`, na `double` v případě typu `float`. Pokud `int` může reprezentovat hodnotu `unsigned char` a `unsigned short` (hodnota není větší, než maximální hodnota, kterou lze pomocí `int` vyjádřit), konvertují se na `int`, jinak na `unsigned int`.

Explicitní datová konverze

Explicitní přetypování se nejčastěji používá při práci s ukazateli (pointery), kterým se budeme věnovat později, nebo při provádění aritmetických operací (zejména při dělení). Přetypování má následující formu

```
(typ) vyraz
```

a znamená, že `vyraz` (výraz nebo proměnná) je v čase překladu konvertován na typ `typ`.

```
int i = 10, j = 4, k;
float f;
/* k i f = 2, protoze i a j jsou typu int, podil je typu int. Vysledek je roven celociselnemu
    deleni. */
k = i / j;
f = i / j;
/* f bude mit hodnotu 2.5, protoze citatel je float, podil je take typu float. k bude rovno 2,
   i pres to, ze vysledek prave strany je 2.5 . k je typu int, dochazi zde k implicitni
   konverzi. Desetinna cast cisla se odstrani. */
k = (float)i / j;
f = (float)i / j;
```

Vstup a výstup programu

Součástí jazyka C není žádná vstupně/výstupní operace. Vstupy a výstupy jsou řešeny pomocí standardních knihoven. Důvodem je to, že právě vstupní a výstupní operace jsou nejvíce závislé na stroji,[53] který program vykonává.
Funkce sloužící k výstupu a vstupu z konzole (terminálu) jsou obsaženy v knihovně `stdio.h`.

[53] Strojem máme na mysli konkrétní HW a OS.

Výstupní funkce

Funkce `putchar()`

Pro výpis jednoho znaku se používá funkce `putchar()`. Tato funkce pracuje s argumentem typu `int` a vypíše na obrazovku znak odpovídající ASCII hodnotě argumentu. Důvodem, proč není argument typu `char`, je to, že funkce vrací hodnotu tištěného znaku za předpokladu, že se výstup podařil, jinak vrací symbolickou konstantu `EOF` (end of file), která bývá definovaná jako záporné číslo (často -1).

Funkce `printf()`

Funkce `printf()` slouží k formátovanému výpisu do konzole. Obecná syntaxe této funkce je následující:

```
printf(format, h1, ..., hn);
```

`format` je řetězec, který určuje text k vypsání. Tento řetězec může obsahovat speciální formátovací instrukce, ty označují místa, na která se postupně vypisují další předané argumenty. *Formátovací instrukce* začíná znakem `%`, za kterým následuje jeden nebo více znaků, určující jakým způsobem se má následující hodnota vypsat. Uveďme si příklad.

```
printf("Promenna a ma hodnotu %d", a);
```

`%d` v řetězci je formátovací instrukce. Formátovací instrukce mají obecně tvar
`%[příznak][šířka][.přesnost][modifikátor]konverze`.
Jediná povinná položka je *konverze*.[54] Jedná se o jeden znak, který musí být vždy uveden. Nejčastěji používané jsou následující.

- `%c` znak
- `%i` číslo `int`
- `%d` číslo `int` v desítkové soustavě (znaménkově)
- `%u` číslo `int` v desítkové soustavě (neznaménkově)
- `%x` číslo `int` v šestnáctkové soustavě (malými písmeny, např.: `1a2b`)
- `%X` číslo `int` v šestnáctkové soustavě (velkými písmeny, např.: `1A2B`)
- `%o` číslo `int` v osmičkové soustavě[55]
- `%f` desetinné číslo typu `float`[56]
- `%e` desetinné číslo v exponenciálním tvaru
- `%s` řetězec

Více informací o ostatních položkách naleznete v příručce jazyka C. Funkci je nutné předat tolik nepovinných argumentů, jako je počet formátovacích instrukcí v řetězci `format`. Pokud se počet neshoduje, překladač nehlásí chybu ani varování, ale funkce nepracuje správně.
Kromě formátovacích instrukcí může řetězec vypisovaný funkcí `printf()` obsahovat jakékoliv znakové konstanty, včetně escape sekvencí. Uveďme si několik příkladů použití funkce `printf()`.

[54] Všechny části v hranatých závorkách jsou nepovinné.

[55] Přidáním `l` před znaky `d`, `u`, `x`, `X`, `o` pracujeme s číslem typu `long`

[56] Přidáme-li před `f` znak `l` (`%lf`), vypisujeme desetinné číslo typu `double`. Přidáme-li `L` (`%Lf`), vypisujeme desetinné číslo typu `long double`.

```c
printf("Soucet je %d\n", i + j);
/* Vypise: Soucet je 11 (v zavislosti na hodnotach i a j) a
    odradkuje */

printf("Soucet je %d\tSoucin je %d\n", i + j, i * j);
/* Vypise: Soucet je 11   Soucin je 28 a odradkuje */

printf("Potrebuji %s jeden priklad.", "vymyslet");
/* Vypise: Potrebuji vymyslet jeden priklad. */
```

Úkol 6
Vytvořte proměnnou `znak` typu `char`, přiřaďte jí nějakou hodnotu. Pomocí funkce `printf()` vypište tuto proměnnou a ASCII hodnotu příslušnou této hodnotě. Například pro proměnnou `znak = 'A'` program vypíše: `Znak 'A' ma ASCII hodnotu 65.`

Vstupní funkce

Stejně jako pro výstup i pro vstup existuje v knihovně `stdio` funkce pro práci s jedním znakem i pro formátovaný vstup.

Funkce `getchar()`

Tato funkce pracuje s proměnnými typu `int`. Použití funkce je následující.

```c
#include <stdio.h>

int main()
{
    int znak;
    znak = getchar();
    /* Vypis nacteneho znaku.*/
    putchar(znak);
    return 0;
}
```

Funkce `getchar()` čte z klávesnice jeden znak a vrátí jeho ASCII hodnotu jako typ `int`. Zapsáním znaku z klávesnice, se ale funkce nevykoná. Znaky se postupně ukládají do tzv. bufferu[57] a až po zmáčknutí klávesy *enter* se první znak z bufferu uloží do proměnné `znak` a poté je funkcí `putchar()` vypsán.
Pokud je v bufferu více znaků, při dalším volání funkce `getchar()` program nečeká na vstup od uživatele, ale načte následující znak uložený v bufferu.

[57] Podrobněji o práci s bufferem budeme mluvit později.

Funkce `scanf()`

Tato funkce pracuje obdobně jako funkce `printf()` i její syntaxe je podobná.

```
scanf(format, p1, ..., pn);
```

Na rozdíl od funkce `printf()` argumenty za formátovacím řetězcem nejsou proměnné, ale ukazatele do paměti, kam se má načtený vstup uložit. Podrobněji se tomu budeme věnovat později. V tuto chvíli se spokojíme s faktem, že pokud chceme hodnotu uložit do proměnné, musíme před její název přidat znak `&`. Konkrétní příklad následuje.

```c
#include <stdio.h>

int main()
{
    int i;
    /* Nacteni celeho cisla a ulozeni do promenne i. */
    scanf("%d", &i);
    return 0;
}
```

Nejprve vytvoříme proměnnou `i`. Ve funkci `scanf()` z formátovacího řetězce vidíme, že očekáváme jako vstup celé číslo. Při vykonávání programu se zde program zastaví a čeká, dokud uživatel nezadá vstup, který potvrdí klávesou `enter`. Pak je tato hodnota načtena jako číslo a uložena do proměnné `i`.

Shrnutí

Řekli jsme si, jaké jsou základní datové typy používané v jazyce C a jakým způsobem je možné vytvářet datové typy vlastní. Také jsme si představili základní operátory, nejen aritmetické, ale zejména operátor přiřazení. Představili jsme funkce pro standardní vstup a výstup z programu.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jaké jsou základní datové typy v jazyce C?
2. Čím je dáno pořadí vyhodnocování operátorů ve výrazu?
3. Jaký je rozdíl mezi `a++` a `++a`?
4. Co je to implicitní a explicitní datová konverze?
5. Jaké jsou základní funkce pro výstup z programu?

Cvičení

Úkol 7
Napište program, který načte stranu čtverce a vypíše jeho obvod a obsah.

Úkol 8
Napište program, který vypočte hodnotu matematického výrazu

$$\frac{3}{2} + 12 + \frac{5 * 5 - 2 * 2}{6}$$

a výsledek vypíše na obrazovku. Správnost výsledku ověřte výpočtem.

Úkol 9
Pro celá čísla (`int`) `a = 2, b = 2, c = 1, d = 1, e = 4` odhadněte výsledné hodnoty výrazů a zkontrolujte tím, že napíšete program, který výsledky spočítá.

- `a++ / b-- * d++`
- `b-- * c++ + e`
- `-b + --c`
- `++a + a--`
- `c / a * d++ / --c`
- `a %= b = d = 1 + e / 2`

Úkol 10
Napište program, který načte velké písmeno, a vypíše jej jako malé. Využijte znalosti ASCII tabulky.

Úkol 11
Načtěte ze vstupu trojciferné číslo a poté vypište jeho první a poslední číslici.

Úkol 12
Načtěte ze vstupu reálné číslo a vypište jeho celou část.

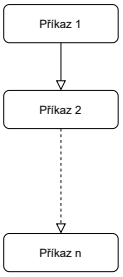
Úkol 13
Napište program, který na obrazovku vypíše přesně text:

```
Uspesnost predmetu "Diskretni struktury 1", ktery ma
zkratku KMI/DISK1, je 50%.
```

Řízení běhu programu

Programy, které jsme doposud vytvářeli, byly vykonávány *sekvenčně*, tedy postupně se prováděly příkazy ve funkci `main()`. Průběh programu je znázorněn na obrázku 1.
Pořadí, ve kterém se příkazy v programu vykonávají, se nazývá *tok programu* (program flow). Kromě sekvenčního vykonávání (*sekvence*), je možné tok programu měnit pomocí větvení (*selekce*) a cyklů (*iterace*). Právě těmto konstrukcím se tuto kapitolu budeme věnovat.

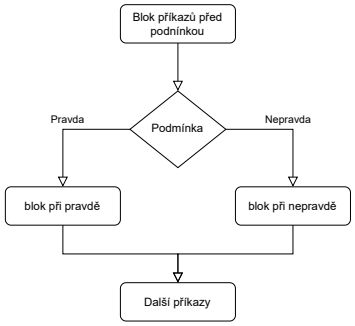
Obrázek 1: Grafické znázornění průběhu programu.



Větvení

Větvení programu umožňuje rozhodnout na základě *podmínky*, jaká část programu se bude vykonávat. Větvení se skládá z logického výrazu reprezentujícího podmínku a bloků příkazů, které se vykonávají v závislosti na tom, zda je nebo není podmínka splněna. Průběh programu je znázorněn na obrázku 2.

Obrázek 2: Grafické znázornění průběhu programu obsahujícího větvení.



Větvení pomocí `if` a `else`

Pro zápis větvení programu se v jazyce C používají příkazy `if` a `else`. Zápis vypadá následovně.

```
if (podminka)
    blok_pri_pravde
else
    blok_pri_nepravde
```

Přičemž `else` část je možné vynechat.
`podminka` je logický výraz. Pokud je podmínka splněna (je pravdivá), provede se `blok_pri_pravde`. Pokud není, vykoná se `blok_pri_nepravde`. Jak už víme, jazyk C neobsahuje logický datový typ. Hodnota `0` představuje *nepravdu* (ať je jakéhokoliv typu) a ostatní hodnoty se vyhodnocují jako *pravda*.
Většina podmínek se skládá z logických binárních operátorů:

- $<$, $<=$ menší, menší rovno

Průvodce studiem
Jednotlivým blokům následujících za `if` a `else` se také říká větev programu.

- >, >= větší, větší rovno
- == rovnost (pozor nezaměňovat s operátorem přiřazení =)
- ! = nerovnost

Podmínky je možné spojovat do složitějších podmínek pomocí logických operací

- || nebo
- && a zároveň
- ! negace

Logické operátory || a && se vyhodnocují *líně*. Pokud je levý argument || pravdivý, druhý argument se už nevyhodnocuje (není potřeba ho vyhodnocovat, výsledek je vždy pravda). Obdobně u && pokud je první argument nepravda, není potřeba vyhodnocovat druhý (výsledek je vždy nepravda). Příklad vhodného použití je následující.

```
if ((y != 0) && ((x / y) < z))
```

Výraz je zcela správný a nikdy nedojde k dělení nulou. Pokud by bylo y rovno 0, pak první část výrazu `y != 0` ukončí jeho vyhodnocování.
V následujícím kódu je příklad programu, který pro dvě zadaná čísla vypíše větší z nich.

```
if (a >= b){
    printf("%d", a);
}
else{
    printf("%d", b);
}
```

Podmínky je vhodnější kvůli přehlednosti psát v kladném znění tvrzení a ne v negaci, obzvláště ve složených výrazech. Například následující výraz není úplně vhodný.

```
/* nevhodne */
if(!(c == '\0' || c == ' ' || c == '1'))
```

Než si ukážeme řešení, jak by stejná podmínka měla vypadat v kladném znění, zkuste se nad tím zamyslet sami. Zkuste výraz upravit tak, aby neobsahoval negaci na začátku.
Lepší zápis stejné podmínky je následující.

Úkol 14
Co vypíše následující část kódu?

```
int a = 0;

if (a = 0){
  printf("je rovno 0");
}
else{
  printf("neni rovno 0");
}
```

Vyzkoušejte, zda máte pravdu.

```
/* vhodnejsi */
if (c != '\0' && c != ' ' && c != '1')
```

Vnoření větvení

V kódu je možné do sebe jednotlivá větvení vnořovat. Bloky `blok_pri_pravde` i `blok_pri_nepravde` mohou obsahovat další `if` konstrukce. Pokud blok obsahuje jen jediný příkaz,[58] je možné vynechat ohraničující závorky `{ }`, ale nedoporučuje se to, obzvláště u složitějších vnořených větvení, abychom dali najevo, které `if` a `else` patří k sobě. Bez použití závorek `else` vždy patří k nejbližšímu `if`. Porovnejme výsledky následujících dvou kódů.

[58] Často se závorky vynechávají i v případě, kdy chceme máme více podmínek, podle kterých chceme program větvit. Jinak řečeno, když se za `else` objevuje další `if`. Viz následující kód.

```
if(...) { ... }
else if(...) { ... }
else if(...) { ... }
...
else { ... }
```

Tomuto uspořádání se také říká *žebřík*.

```
/* Prvni kod */
int a = 5, b = 1, c = 3, foo = 10;

if (a > b){
    if (b > c)
        foo = b;
}
else foo = c;
```

```
/* Druhy kod */
int a = 5, b = 1, c = 3, foo = 10;

if (a > b)
    if (b > c)
        foo = b;
    else foo = c;
```

V prvním případě je výsledná hodnota proměnné `foo` rovna `10`. Podmínka `a > b` je splněna, ale `b > c` není, druhá podmínka však neobsahuje `else` větev, proto hodnota proměnné `foo` zůstane nezměněna.
V druhém případě `else` větev patří k vnořenému větvení a proměnná `foo` je tedy nastavena na hodnotu proměnné `c`.

Větvení pomocí `switch`

V některých případech je použití větvení pomocí `if` a `else` nepřehledné, například v případech, kdy je jich potřeba použít mnoho. Jazyk C nabízí ještě jiné konstrukce umožňující větvení. Jednou z nich je `switch`, která má následující syntaxi.

```
switch (vyraz){
    case konstanta1:
        blok1
    case konstanta2:
        blok2
    ...
    default:
        blok
}
```

Tato konstrukce se používá pokud v jednotlivých podmínkách kontrolujeme rovnost výrazu s celočíselnou konstantou.
Vyhodnocení tohoto kódu probíhá následovně. Odshora hledáme shodu hodnoty `vyraz` s konstantami `konstanta1`, `konstanta2` a tak dále. S hodnotou `default` se shoduje vždy.[59] Při shodě se začnou provádět příkazy, které následují po shodné konstantě až do té doby, než se narazí na konec `switch`, nebo první příkaz `break`. Ten se většinou píše na konec každého bloku. Konkrétní příklad následuje.

[59] Větev `default` nemusí být uvedena na konci, ale z konvence je vždy jako poslední. Příkazy za `default` se prování vždy až tehdy, když není nalezena žádná vyhovující větev, ať je tato větev uvedena v přepínači kdekoliv.

```
if (a == 1)
    printf("a je jedna");
else if (a == 2)
    printf("a je dva");
else if (a == 3)
    printf("a je tri");
else
    printf("a neni 1, 2 ani 3");
```

Tento kus kódu lze přepsat pomocí konstrukce `switch` následovně.

```
switch (a){
    case 1:
        printf("a je jedna");
        break;
    case 2:
        printf("a je dva");
        break;
    case 3:
        printf("a je tri");
        break;
    default:
        printf("a neni 1, 2 ani 3");
}
```

Pokud má být pro více hodnot proveden stejný kus kódu (blok), je možný následující zápis.

```
switch (vyraz){
    case konstanta1:
    case konstanta2:
    case konstanta3:
        blok1;
    case konstanta4;
    ...
}
```

Podmínkový operátor

Pro zápis jednoduchého větvení lze použít podmínkový operátor `? :`. Obecný zápis jednoduché podmínky pomocí tohoto operátoru vypadá následovně.

Průvodce studiem
Pro podmínkový operátor se v mnoha jazycích používá název *ternární operátor*, protože je to jediný operátor s aritou 3.

```
podminka ? vyraz1 : vyraz2;
```

Pokud je splněna `podminka`, vyhodnotí se `vyraz1`, pokud ne, vyhodnotí se `vyraz2`. Následující kód

```
if (a < b)
    x = a;
else
    x = b;
```

lze pomocí podmínkového operátoru zapsat následovně.

```
x = (a < b) ? a : b;
```

Cykly

Cykly se používají k opakování části kódu (těla cyklu) vícekrát v závislosti na ukončovací podmínce (kód se vykonává dokud je podmínka splněna) nebo předem daného počtu opakování.
V jazyce C existují tři typy cyklů:

- cyklus `while`,
- cyklus `for`,
- cyklus `do while`.

Každé vykonání části kódu, která se opakuje, se označuje jako *iterace cyklu*.

Cyklus `while`

Postup vykonávání kódu při použití cyklu `while` je zachycen na obrázku 3. Tělo cyklu (blok příkazů) je opakováno, dokud je splněna podmínka.
Zápis tohoto cyklu vypadá následovně.

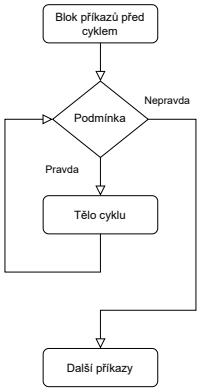
```
while (podminka)
    telo_cyklu
```

`podminka` představuje logický výraz. Pokud je tento výraz vyhodnocen na pravdu, dojde k vykonání bloku kódu `telo_cyklu`.[60] Po jeho vykonání je opět vyhodnocena podmínka a situace se opakuje. Vyhodnocením podmínky na nepravdu program pokračuje kódem za tělem cyklu (říkáme, že *cyklus končí*). Následující část kódu slouží k vypsání čísel od 0 do 9.

```
int j;

j = 0;
while(j < 10){
    printf("%d ",j);
    j = j + 1;
}
```

Nejprve je proměnná `j` inicializovaná na hodnotu `0`. Podmínka `j < 10` je pravdivá, vykoná se tělo cyklu, tedy nejprve se vypíše číslo `j` a následně je hodnota proměnné `j` inkrementována (zvětšena o `1`). Poté se ověří, zda je podmínka splněna. `j` je rovno `1`, takže podmínka je opět pravdivá. Cyklus se opakuje do chvíle, kdy je `j` rovna `10`. Podmínka není splněna, cyklus končí.

Obrázek 3: Grafické znázornění průběhu cyklu `while`.



[60] Tělo cyklu může být prázdné. Možným použitím je například vynechání všech mezer na vstupu.

```
while (getchar() == ' ')
    ;
```

Pro přehlednost je vhodné psát středník na nový řádek.

Úkol 15
Jak vytvoříte cyklus, který nikdy neskončí?

Cyklus `for`

Druhým typem cyklu je cyklus `for`. Nejčastěji se používá pro cykly s předem daným počtem opakování (iterací). Pro počítání iterací se používá tzv. *krokovací proměnná*. Krokovací proměnné je na začátku cyklu přiřazena počáteční hodnota, která se v každé iteraci zvyšuje (iteruje) a porovnává se s koncovou podmínkou. Cyklus `for` se zapisuje následovně.

```
for (init; podminka; iter)
    telo_cyklu
```

`init` je blok příkazů oddělených čárkou,[61] které se provedou před samotným cyklem. `podminka` má stejný význam, jako u cyklu `while`.

[61] Jedná se o operátor čárky. `init` je tedy výraz.

`iter` je seznam příkazů, které jsou odděleny čárkou. Tyto příkazy se vykonají na konci každého vykonání těla cyklu. Průběh vykonávání cyklu je znázorněn na obrázku 4.
Následující kód slouží k výpisu čísel od 0 do 9.

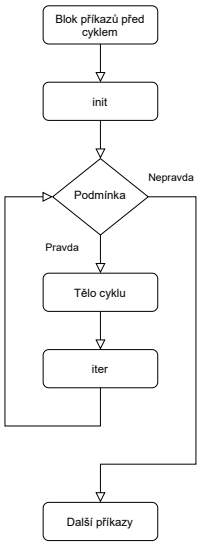
```
int j;

for(j = 0; j < 10; j = j + 1){
    printf("%d ",j);
}
```

`init`, `podminka` i `iter` jsou nepovinné, je možné je vynechat (nechat prázdné). Při vynechání `podminka` je podmínka vždy považována za pravdivou.
V následujícím příkladu je uveden stejný cyklus zapsaný různými způsoby.

```
int i = 0;

/* doporucene pouziti cyklu for */
for(i = 0; i < 10; i++){
    printf("%d ",i);
}

/* vyuziti inicializace i pri definici. */
for(; i < 10; i++){
    printf("%d ",i);
}

/* ridici promena se meni v tele cyklu */
for(i = 0; i < 10; ){
    printf("%d ",i++);
}

/* vyuziti operatoru carky */
for(i = 0; i < 10; printf("%d ",i), i++)
    ;
```

Z příkladu je patrné, že poslední tři způsoby zápisu nejsou tak přehledné, jako první.

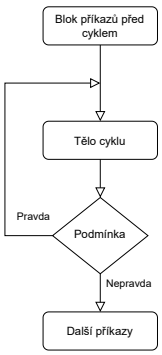
Obrázek 4: Grafické znázornění průběhu cyklu `for`.



Cyklus `do while`

Posledním typem cyklu je cyklus `do while`, jehož průběh vykonávání je znázorněn na obrázku 5. Konstrukce je analogická cyklu `while`, pouze s tím rozdílem, že podmínku testujeme poté, co je vykonáno tělo cyklu. To znamená, že u `do while` vždy proběhne tělo cyklu alespoň jednou. Tento cyklus se zapisuje následovně.

Obrázek 5: Grafické znázornění průběhu cyklu `do while`.



```
do
   telo_cyklu
while (podminka);
```

Průvodce studiem
Všechny způsoby zápisu cyklu jsou ekvivalentní. Každý cyklus lze přepsat pomocí dalších dvou konstrukcí.[62]

[62] Při přepisování však musíme být opatrní. Například pokud bychom přepisovali `for` cyklus obsahující příkaz `continue` pomocí `while` cyklu, musíme si dát pozor na inkrementaci (tj. nezapomenout dát před `continue` inkrementaci).

Příkazy `break` a `continue`

Při práci s cykly, zejména pokud potřebujeme vystoupit z těla cyklu dříve, než sám skončí, se nám mohou hodit speciální příkazy a to příkazy `break` a `continue`.
Příkaz `break` způsobí okamžité ukončení vykonávání cyklu a program pokračuje až za ním.
Příkaz `continue` způsobí okamžité ukončení vykonávání těla, ale nedojde k ukončení celého cyklu. Pokračuje se další iterací. To znamená, že je opět zkontrolována podmínka a při její platnosti se tělo cyklu vykonává znovu. V případě cyklu `for` je i vyhodnocen výraz `iter`.
Pro představu, jak tyto příkazy pracují porovnejte následující kódy.

```
int j;
for(j = 1; j < 10; j = j + 1){
    if((j % 3) == 0)
        continue;
    printf("%d ", j);
}
```

Kód vypíše čísla `1`, `2`, `4`, `5`, `7`, `8`. Použití příkazu `continue` způsobí to, že pro čísla dělitelní `3` se neprovede příkaz `printf`.

```
int j;
for(j = 1; j < 10; j = j + 1){
    if((j % 3) == 0)
        break;
    printf("%d ", j);
}
```

Výstupem tohoto kódu budou čísla `1`, `2`. V iteraci, ve které je `j` rovno `3`, dojde k vykonání příkazu `break`, jenž způsobí úplné ukončení cyklu.

Příkaz `goto`

Málo používaný příkaz měnící tok programu je příkaz `goto`. Ten má následující syntaxi.

```
goto navesti;

...

navesti:

...
```

Pokud program narazí při běhu na tento příkaz, pokračuje se vykonáváním programu za *návěštím* `navesti`. Návěští může být téměř kdekoliv. Není možné však „skákat“ mezi funkcemi.
V dobře napsaných programech se tento příkaz téměř nevyskytuje,[63] protože v jazyce C se mu lze vždy vyhnout. Jedno z mála odůvodnitelných (možná jediné) použití tohoto příkazu je opuštění vnořených cyklů (příkazem `break` opustíme jen aktuální cyklus). Viz následující příklad.

[63] Příkladem, kdy se v praxi používá, je pro úklid paměti po chybách programu.

```
for (i = 0; i < 10; i++){
    for (j = 0; j < 10; j++){
        for (k = 0; k < 10; k++){
            /* x je pole a x[k] vraci jeho k-ty prvek */
            if (x[k] == 0)
                goto chyba;
            printf("%d", (x[i] + x[j]) / x[k]);
        }
    }
}
...
chyba:
printf("Nelze delit nulou\n");
```

Úkol 16
Přepište příklad bez použití příkazu `goto`.

Za žádných okolností by neměl být používán příkaz `goto` pro skok dovnitř bloků větví příkazu `if` z vnějšku, skok z kladné větve `if` do negativní, skok dovnitř těla `switch` z vnějšku, skok dovnitř složeného příkazu (bloku) z vnějšku.

Příkaz `return`

Posledním příkazem, který si v této kapitole představíme je příkaz `return`. Ten způsobí ukončení právě vykonávané funkce (v případě funkce `main()` to znamená, že se ukončí celý program).[64] Pomocí `return` se z funkce vrací nějaká hodnota. Ta by měla odpovídat návratové hodnotě funkce. O tom ale budeme podrobněji mluvit později. Pokud by příklad uvedený v sekci `goto` neobsahoval jiný kód než cyklus, bylo by vhodnější nahradit `goto` příkazem `return`.[65]

[64] Ve standardní knihovně `stdlib` je definována funkce `exit()`, která vyvolá ukončení programu ať je zavolána z jakékoliv funkce, ne jen `main()`.

[65] V příkladech jsme doposud používali ve funkci `main()` příkaz:

```
return 0;
```

Značí to, že z funkce `main()` vracíme hodnotu `0`. To znamená, že program skončil úspěšně. Při vrácení jiné hodnoty dáváme systému informaci, že při vykonávání kódu nastal nějaký problém.
K výpisu toho, s jakou hodnotou program skončil, můžeme použít v terminálu (konzoli) příkaz
`echo $?`
v případě, že pracujeme na UNIX systému.
`echo %ERRORLEVEL%`
v případě, že pracujeme na Windows.

```
for (i = 0; i < 10; i++){
    for (j = 0; j < 10; j++){
        for (k = 0; k < 10; k++){
            /* x je pole a x[k] vraci jeho k-ty prvek */
            if (x[k] == 0){
                printf("Nelze delit nulou\n");
                return 1;
            }
            printf("%d", (x[i] + x[j]) / x[k]);
        }
    }
}
```

Shrnutí

V této kapitole jsme si ukázali, jakým způsobem můžeme měnit tok programu pomocí větvení a cyklů.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jaké typy větvení jazyk C nabízí?
2. Jakým způsobem se vyhodnocují podmínky složené z několika logických výrazů?
3. Jaká je syntaxe cyklu `for`?
4. Jaký je rozdíl mezi příkazy `break` a `continue`?

Cvičení

Úkol 17
Za použití podmínkového operátoru napište program, který pro zadané číslo vypíše jeho absolutní hodnotu.

Úkol 18
Napište program, který načte celá čísla a a b a pak

1. vypíše prvních a násobků čísla b
2. spočítá a-tou mocninu čísla b
3. určí kolik číslic má číslo a
4. vypočítá a-té Fibonacciho číslo
5. sečte všechna čísla větší než a a menší než b

Úkol 19
Napište program, který rozhodne, zda zadaný rok je přestupný. (Definici přestupného roku naleznete například na wikipedii `https://cs.wikipedia.org/wiki/P%C5%99estupn%C3%BD_rok`).

Úkol 20
Napište program, který rozhodne, zda je zadané písmeno malé nebo velké.

Úkol 21
Pro zadané n vykreslete do konzole následující obrázky

1. pro n=3

```
      *
    * * *
  * * * * *
```

   pro n = 4

```
        *
      * * *
    * * * * *
  * * * * * * *
```

2. pro n=2

```
      *
    * * *
      *
```

   pro n = 3

```
      *
    * * *
  * * * * *
    * * *
      *
```

3. šachovnici o straně n pro n = 4

```
. * . *
* . * .
. * . *
* . * .
```

Úkol 22
Napište program, který pro zadané číslo vrátí číslo zapsané pozpátku. (Pro 1234 vrátí číslo 4321)

Úkol 23
Přiřaďte výstupy k blokům kódu po dosazení bloku do následujícího kódu.

```
#include <stdio.h>

int main(){
    int x = 0;
    int y = 0;
    while (x < 5){
        /* sem
           vlozte
           blok
           kodu */
        printf("%d%d ", x, y);
        x = x + 1;
    }
    return 0;
}
```

Bloky kódu:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5){
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```

Možné výstupy:[66]

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

Řešení ověřte spuštěním kódu.

[66] Ne všechny výstupy patří k nějakému bloku kódu.

Úkol 24

Pro zadané číslo od 1 do 10, vypište sestupnou posloupnost čísel od zadaného čísla po 1. (pro 4 bude výstup 4 3 2 1). K řešení zkuste vhodně použít konstrukci `switch`.[67]

[67] Vhodnější je použít cyklus. Tento úkol je spíše k zamyšlení.

Paměť

V této kapitole se zaměříme na to, jak se v jazyce C pracuje s pamětí a na témata s pamětí spojená.
Každé proměnné musí být přidělen (*alokován*) prostor v paměti, který velikostně odpovídá datovému typu proměnné. Paměť můžeme rozdělit na několik logických částí, viz obrázek 6.
Zásobník (stack) je část paměti, kam se ukládají *lokální proměnné*.
Halda (heap) je dynamická část paměti. Je určena pro proměnné, které vznikají za běhu programu.
Globální proměnné je část paměti, kde jsou ukládány všechny globální proměnné. Globální proměnné vznikají při spuštění programu a je možné jim měnit hodnoty.
Konstanty je část paměti, která je „read-only“. Jsou zde uloženy všechny konstanty, které se používají v kódu. Jsou to hodnoty, které se v programu používají, ale které nechceme, aby se měnily.
Kód je část paměti, která je také „read-only“. Je v ní uložen sestavený kód programu.
Při *deklaraci* (proměnné, funkce, ...) překladač dostane informaci o identifikátoru (jméně) a jeho typu, ale zatím mu nevyhrazuje žádné místo v paměti. K alokaci paměti dochází až při *definici*. Překladač generuje požadavek na přidělení části paměti, do které se dá uložit hodnota identifikátoru (dle jeho typu). Části programu, ve které je identifikátor definován, nebo deklarován (kde je viditelný) se říká *rozsah platnosti identifikátoru*.

Obrázek 6: Logické rozdělení paměti.

Průvodce studiem
Zásobník je abstraktní datová struktura typu LIFO (poslední dovnitř první ven).

Alokace paměti

Existují dva základní typy alokace:

- *Statická alokace paměti* – místo v paměti se alokuje již během překladu (před spuštěním programu).
- *Dynamická alokace paměti* – místo v paměti se alokuje za běhu programu.

Opačný pojem k pojmu alokace je pojem *dealokace* (uvolnění). Uvolňovat nepotřebnou paměť je důležité, protože počítač nemá neomezené zdroje.

Statická alokace

Statická alokace se používá zejména pokud umíme překladači předem říci, jaké bude mít program paměťové nároky. Například budeme-li v programu sčítat dvě celá čísla, víme, že potřebujeme dvě proměnné typu `int`. Při překladu překladač určí paměťové nároky a zavaděč (systémový program, který spustí náš program) alokuje tuto paměť v čase začátku programu. Při běhu programu se neprovádí žádné přidělování paměti. Po skončení programu operační systém takto přidělenou paměť uvolní.
Statická alokace proměnných vymezuje místo v takzvané části paměti, kterou jsme označili Kód v obrázku 6.[68]

[68] Z operačních systému znáte spíše pojem *datová oblast* nebo segment.

Dynamická alokace paměti

Existují dva druhy dynamické alokace:

- Na zásobník – alokace a dealokace se provádí automaticky.
- Na haldu – o alokaci a dealokaci se stará programátor.

Dynamická alokace na zásobníku

Paměť alokujeme pomocí definice lokální proměnné. Existence lokálních proměnných začíná při vstupu do funkce a končí při výstupu z této funkce (paměť může být po ukončení funkce opět použita).
Konkrétně, v okamžiku zavolání funkce se alokuje paměť pro všechny lokální proměnné (proměnné definované v těle funkce) a pro parametry funkce. Do této paměti jsou překopírovány hodnoty předané funkci (říkáme, že argumenty funkce *předáváme hodnotou*). Všechna paměť je uvolněna po skončení funkce.
Pokud například funkce `main()` volá funkci `A()` a v ní se postupně volají funkce `B()` a `C()`, pak se nejprve alokuje paměť pro funkci `A()`, pak se alokuje pro `B()`. Po jejím skončení se paměť pro `B()` dealokuje a alokuje se pro `C()`. Po skončení se dealokuje paměť pro `C()` a následně dojde k dealokaci paměti pro `A()`. Jelikož je zásobník tzv. LIFO struktura, je zřejmé, že globální proměnné se alokují jako první a dealokují jako poslední.

Dynamická alokace na haldě

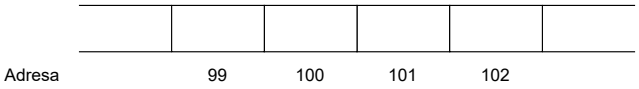
Alokace se provádí pomocí speciálních příkazů, kdy teprve za běhu programu volíme velikost požadované paměti (není zde omezení

uvedené v předchozí části). Tato paměť není pojmenovaná (nemá symbolické jméno) a přistupuje se k ní pomocí ukazatele (viz dále).

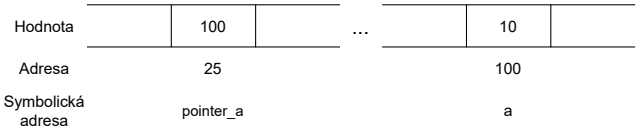
Ukazatele

Ukazatel (pointer) je jedním z nejdůležitějších pojmů v jazyce C. Ve skutečnosti je pointer proměnná, jen její hodnota má jiný význam, než proměnné, které známe.
Paměť si můžeme představit jako posloupnost paměťových buněk (o velikosti 1 byte), které jsou naskládány za sebou a očíslovány.
Pořadovému číslu paměťové buňky říkáme *(absolutní) adresa*.

Průvodce studiem
Pointery používá většina imperativních programovacích jazyků, jako např. jazyk C a Pascal.



Pointer (jeho hodnota) představuje *adresu v paměti*. Názvy proměnných představují tzv. *symbolickou adresu*.



Proměnná `pointer_a` je ukazatel. Leží na symbolické adrese `pointer_a`. Hodnota tohoto ukazatele je `100` (hodnota uložená na symbolické adrese `pointer_a` je `100`). Jelikož tato proměnná představuje ukazatel, tak hodnota `100` představuje absolutní adresu v paměti. Na této adrese je v paměti uložena hodnota `10`. `pointer_a` ukazuje na hodnotu `10`, ale sám má hodnotu `100`.
Každý pointer je svázán s nějakým datovým typem (ukazuje na objekt daného typu, který se na dané adrese nachází). Vícebytové objekty jsou uložené v paměti hned za sebou a ukazatel ukazuje na první z bytů. Abychom mohli hodnotu, na kterou ukazatel ukazuje, správně interpretovat, musíme vědět, kolik bytů k hodnotě přísluší. Správně by se tedy mělo používat *pointer na typ*. Konkrétně, máme pointer na typ, víme tedy, že musíme přečíst `sizeof(typ)` bytů a interpretovat je jako `typ`. Deklarace pointeru na typ vypadá následovně.

```
typ *jmeno;
```

V následujícím příkladu definujeme pointer na `int`, který pojmenujeme `ptr`.

```
int *ptr;
```

Hvězdičku pro přehlednost píšeme k názvu ukazatele, ne k datovému typu. Na jednom řádku je možné definovat více ukazatelů a proměnných daného typu, viz následující příklad.

```
int a = 3, b, *ptr1, *ptr2;
```

V příkladu proměnné `a` a `b` představují proměnné typu `int` (`a` je navíc inicializovaná číslem 3). Proměnné `ptr1` a `ptr2` jsou ukazatele na `int`.
Pro ukazatel, který nikam neukazuje (*nulový ukazatel*) používáme symbolickou konstantu `NULL`.
Pro výpis hodnoty ukazatele používáme ve funkci `printf()` řídící sekvenci `%p`.[69]

[69] Adresa je vypsána jako hexadecimální číslo.

```
printf("%p", ptr);
```

Pro základní práci s ukazateli používáme operátory `&` a `*`. *Operátor adresy* `&` používáme k získání adresy proměnné.[70] Tento operátor není možné použít na jiné výrazy než na proměnné (na `l_value`).[71]
Pro přístup k hodnotě na adrese používáme *operátor dereference* `*`.

[70] U dříve zmíněné funkce `scanf()` jsme si říkali, že před název proměnné musíme dát znak `&`, nyní už víme proč.

[71] Následující dva příkazy nejsou správné.

```
/* konstanta neni l_value
    */
ptr = &15;

/* vyraz neni l_value */
ptr = &(a + 2);
```

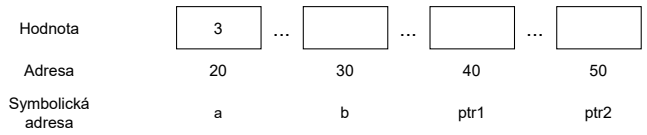
Příklad

V následujícím příkladu si ukážeme základní práci s ukazateli.

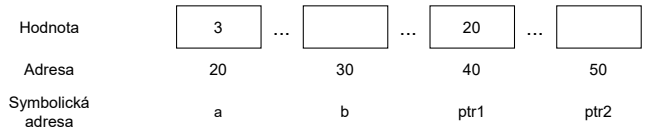
```
int a = 3, b, *ptr1, *ptr2;

/* ukazatel ptr1 ukazuje na promennou a  */
ptr1 = &a;

/* hodnota b je 5 (*ptr1 je rovna 3) */
b = *ptr1 + 2;

/* ptr1 a ptr2 ukazuji na stejne misto */
ptr2 = ptr1;

/* zmenime hodnotu na miste, kam ukazuje ptr2 */
*ptr2 = 5;

/* hodnota b bude 8 */
b = a + 3;
```

Paměť by mohla po vykonání definice (na prvním řádku) tohoto příkladu vypadat následovně.[72]

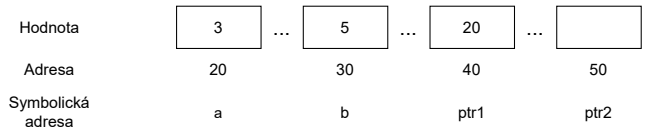
[72] Ve skutečnosti jsou nějaké hodnoty uloženy i v ostatních paměťových buňkách, ale nevíme jaké.



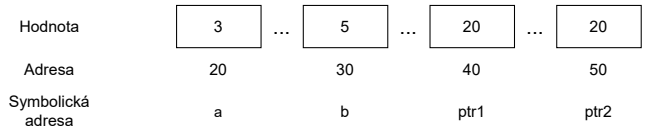
Příkaz `ptr1 = &a;` uloží do ukazatele `ptr1` adresu proměnné `a`. Jak již totiž víme, operátor `&` vrací adresu proměnné.



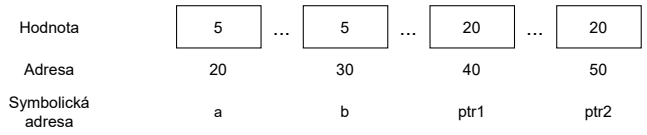
Po vykonání `b = *ptr1 + 2;` bude v proměnné `b` uložena hodnota `5`. `*ptr1` vrátí hodnotu uloženou na adrese `20` (ta odpovídá proměnné `a`), což je hodnota `3`. Zde je nutné připomenout, že operátor `*` má vyšší prioritu než operátor `+`.



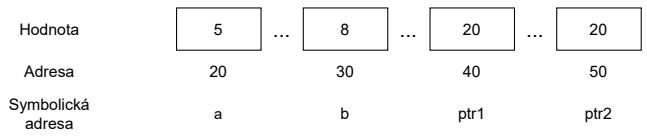
`ptr2 = ptr1;` přiřadí proměnné `ptr2` stejnou hodnotu, jakou má `ptr1` (oba ukazatele ukazují na stejné místo).



V příkazu `*ptr2 = 5;` změníme hodnotu v paměťové buňce, na kterou ukazuje `ptr2`, na hodnotu `5`. `*ptr2` vrátí objekt uložený na místě, kam ukazuje `ptr2` (tedy objekt označený symbolicky `a`).



Jelikož máme v proměnné `a` nyní uloženou hodnotu `5`, bude po vykonání příkazu `b = a + 3;` do proměnné `b` uložena hodnota `8`.



Pole

S ukazateli úzce souvisí pojem pole. Pole představuje datovou strukturu, která obsahuje několik prvků stejného datového typu. Pole, nebo nějaké jejich obdoby, se vyskytují ve všech programovacích jazycích. Ty se velmi liší v tom, jak s polem pracují. V nižších kompilovaných jazycích, jakým je i jazyk C, se musí specifikovat pevná velikost pole ve zdrojovém kódu. Velikost pole není možné za běhu programu měnit. Samozřejmě existuje možnost, jak tuto limitaci obejít, můžeme vytvořit tzv. *dynamicky alokované pole*. O něm budeme mluvit později.

Statické pole

Pole deklarujeme jako běžnou proměnnou, pouze za její název do hranatých závorek uvedeme počet prvků pole.

```
typ jmeno[velikost];
```

`typ` určuje, jakého typu jsou prvky uložené v poli, `jmeno` představuje název pole (identifikátor) a `velikost` je celočíselná kladná konstanta.
`velikost` nemůže být proměnná ani jiný výraz. Poli je vyčleněn kus paměti pro `velikost` hodnot typu `typ`. Tato část paměti mohla být předtím využívaná jinou aplikací, není tedy jisté, co je v poli uloženo za hodnoty. Pole je možné při deklaraci i inicializovat a to tak, že se jednotlivé prvky napíší do složených závorek. Inicializace vypadá následovně.

Nový standard ISO
Ve standardu C99 bylo zavedeno tzv. *pole proměnné délky* (nebo také runtime-sized pole). V něm je možné při deklaraci jako velikost pole použít i proměnnou.

```
typ jmeno[velikost] = {p1, p2, ..., pn};
```

Prvních `n` prvků bude postupně nabývat hodnot `p1`, `p2`, ... Ve výčtu může být méně prvků, než je velikost pole. Pokud bychom chtěli, aby bylo vytvořeno pole s velikostí rovnou počtu prvků ve výčtu, pak je možné při definici vynechat `velikost`.

Průvodce studiem
Mezi další jazyky, které povolují pole proměnlivé délky, patří například Ada, Algol, C# nebo Object Pascal.

```
typ jmeno[] = {p1, p2, ..., pn};
```

Vytvoří se pole velikosti `n`, ve kterém jsou uloženy po řadě hodnoty `p1`, `p2`, ..., `pn`. Konkrétní příklad následuje.

```
/* Neinicializovane pole velikosti 6 */
int pole1[6];

/* Pole o velikosti 10, se 3 inicializovanymi prvky */
int pole2[10] = {1, 2, 3};
```

```
/* Pole o velikosti 4 se vsemi inicializovanymi prvky */
int pole3[] = {1, 2, 3, 4};
```

Ukládat hodnoty do pole pomocí výčtu je možné jen při deklaraci. Dále je potřeba hodnoty ukládat do pole na jednotlivé indexy zvlášť. K jednotlivým prvkům pole přistupujeme přes jejich *indexy* pomocí hranatých závorek (indexační operátor). Prvky pole v jazyce C indexujeme od 0.[73] První prvek má index 0, druhý má index 1, atd.

[73] Je to dáno hlavně kvůli efektivitě přístupu k jednotlivým prvkům a kvůli těsné návaznosti na ukazatele. `pole` je ukazatel na `typ`, který obsahuje adresu prvního prvku v poli.

...	1. prvek	2. prvek	3. prvek	4. prvek	...
Index:	0	1	2	3	

Přístup k prvku tedy vypadá následovně.

```
jmeno[index];
```

Tento výraz vrátí hodnotu prvku pole `jmeno` na indexu `index`. `index` je celočíselná hodnota. `jmeno[index]` můžeme používat jako proměnné, získávat jejich hodnotu, přiřazovat jim jinou hodnotu atd.
Jazyk C nekontroluje meze polí[74] (je možné jako indexy použít i záporná čísla, nebo čísla větší, než je počet prvků pole), což je opět důsledkem toho, že jsou pole úzce spjata s pointery.
Plnit pole ručně prvek po prvku by bylo příliš zdlouhavé a pracné. Běžně se využívá cyklů pro průchod pole. Obzvláště vhodné je použít cyklus `for`, kde jeho řídící proměnná vystupuje v roli indexu, viz následující kód.

[74] Kompilátor dokonce ani neposkytuje varovná hlášení.

Úkol 25
Zkuste odhadnout, které z následujících výrazů jsou špatně a proč. Řešení naleznete v kapitole *Řešení vybraných úkolů*.

```
int pole[5] = {1, 2, 3,
    4, 5};

pole[2] = 10;
pole[3] = pole[1] + pole
    [2];
pole[1] = pole[5];
pole[7] = 10;
```

```
int pole[20];
int i;

for(i = 0; i < 20; i++){
    pole[i] = 2*i;
}
```

Jak bylo už zmíněno dříve, název identifikátoru pole je vlastně ukazatel na první prvek tohoto pole. Tento ukazatel je však *konstantní* a nelze mu přiřadit jinou hodnotu. Jeho hodnotu však lze přiřadit jinému pointeru. V takovém případě ale ztrácíme informaci o velikosti pole, viz následující kód.

```
int pole[] = {1, 2, 3};
int *p_pole = pole;
```

```
/* pole i p_pole ukazuji na stejny kus pameti */
printf("%p %p\n", *pole, *p_pole);

/* ztratili jsme informaci o velikosti */
printf("%d %d\n", sizeof(pole), sizeof(p_pole));
```

Operátor `sizeof()` v případě `pole` vrátí počet prvků krát velikost datového typu, v případě `p_pole` vrátí velikost ukazatele na datový typ.[75]

[75] Na 32-bitovém operačním systému má pointer velikost 4 byty, na 64-bitovém 8 bytů.

Řetězce

Speciálním případem polí jsou řetězce. Jsou to pole typu `char`, která jsou ukončena znakem `'\0'`.[76]

[76] Tento znak má hodnotu 0.

'a'	'h'	'o'	'j'	' '	's'	'v'	'e'	't'	'e'	'\0'

Kvůli ukončujícímu znaku potřebujeme pro pole vyčlenit o jeden prvek více, než je délka řetězce. Práce s řetězcem se neliší od práce s polem jiného typu. Definice řetězce ''`ahoj svete`'' je následující.

```
char retezec[11] = {'a', 'h', 'o', 'j', ' ', 's', 'v', 'e', 't', 'e', '\0'};
```

I zde je možné vynechat velikost řetězce v hranatých závorkách. Navíc je možné řetězec inicializovat pomocí řetězcové konstanty následovně.

```
char retezec[] = "ahoj svete";
```

Pozor ale na následující definici.

```
char *retezec = "ahoj svete";
```

Tato definice nevytvoří pole typu char, pouze vytvoří ukazatel na konstantní řetězec. Pokud `retezec` vytvoříme tímto způsobem, není ho možné měnit. Jakmile se v kódu vyskytne řetězcová konstanta (v tomto případě ''`ahoj svete`''), vytvoří se pro ní místo v paměti přiřazené konstantám. Tato část paměti je „read-only“ a není tedy možné prvky řetězce měnit. Při vytvoření pole a jeho inicializaci řetězcovou konstantou se v paměti na zásobníku alokuje dostatečně velký prostor (délka řetězce + 1) a řetězec se do této paměti zkopíruje.
Ani následující kód není správný. Poli není možné přiřadit konstantní řetězec (konstantní pointer).

```
char retezec[5];
retezec = "duha";
```

Pokud chceme do řetězce vložit data, musíme to udělat buď při deklaraci, nebo prvek po prvku.[77]
Pro výpis řetězce pomocí funkce `printf()` použijeme řídící sekvenci `%s`.

[77] Případně použít nějakou knihovní funkci pro práci s řetězci.

```
printf("%s", retezec);
```

Pozor na načítání řetězce pomocí funkce `scanf()`. Nejprve je potřeba vytvořit si dostatečně velké pole (o 1 větší, než je délka řetězce, který chceme načíst).
Jako řídící sekvence se ve funkce `scanf()` použije `%Xs`, kde `X` je celé kladné číslo menší než je velikost pole, do kterého budeme řetězec načítat.[78] Funkce načte `X` znaků z konzole (případně méně, pokud narazí dříve na bílý znak[79]) a uloží je do pole, jehož jméno je dalším argumentem ve funkci. Zde je nutné upozornit, že oproti dříve zde není před názvem pole znak `&`, identifikátor pole je totiž ukazatel na první prvek v poli. Uveďme si konkrétní příklad.

[78] Pokud bychom nespecifikovali, kolik znaků chceme načíst, funkce by načítala znaky dokud narazí na nějaký bílý znak. Jelikož v jazyce C nedochází ke kontrole „přetečení" pole, mohlo by se stát, že se znaky budou ukládat i do paměti, která poli nepatří.

[79] Pokud bychom chtěli přečíst řetězec obsahující bílé znaky, museli bychom použít jinou funkci. Například `fgets()`, o které budeme mluvit později.

```
char text[11];
scanf("%10s", text);
```

Funkce pro práci s řetězci jsou součástí standardní knihovny `string.h`. Mezi funkcemi zde implementovanými najdete například funkci pro zřetězení dvou řetězců `strcat()`, funkci pro nalezení podřetězce v řetězci `strstr()` nebo kopírování řetězce `strcpy()` a jiné.

Pointerová aritmetika

S pointery lze provádět některé aritmetické operace. Konkrétně je s nimi možné provádět operace:

- součet pointeru a celého čísla,
- rozdíl pointeru a celého čísla,
- porovnání dvou pointerů (na stejný datový typ),
- rozdíl dvou pointerů (na stejný datový typ).

Ostatní aritmetické operace sice lze s pointery také provést, ale nemají žádný smysl.
V následujících částech si jednotlivé operace vysvětlíme. Ve všech částech předpokládáme, že máme ukazatel `ptr` a celé číslo `n`.

Součet/rozdíl pointeru a celého čísla

Výraz `ptr + n` znamená, že se odkazujeme na `n`-tý prvek za prvkem, na který ukazuje pointer `ptr`. Výsledkem přičtení čísla `n` k pointeru `ptr` je adresa, kterou získáme tak, že k adrese `ptr` přičteme `n` krát velikost datového typu `ptr`, tedy `ptr + n * sizeof(typ)`. Uvažme následující příklad.

```
char *ptr_c = 10; /* sizeof(char) = 1 */
int *ptr_i = 10; /* sizeof(int) = 2 */
float *ptr_f = 10; /* sizeof(float) = 4 */
```

Všechny ukazatele ukazují na adresu 10. ptr_c + 1 ukazuje na adresu 11, ptr_i + 1 na adresu 12 a ptr_f + 1 na adresu 14.
Analogicky funguje i odečítání celého čísla od pointeru.
Následující příklad ukazuje, jakým způsobem je možné použít pointerovou aritmetiku pro průchod pole.

```
int pole[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *ptr;
int i;

ptr = pole;

for (i = 0; i < 10; i++){
    /* hodnota prvku pole */
    printf("Hodnota *ptr = %d\n", *(ptr + i));
    /* adresa prvku pole */
    printf("Adresa ptr = %p\n\n", (ptr + i));
}
```

Případně můžeme použít operátor ++. Výsledky obou kódů budou stejné.

```
int pole[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *ptr;
int i;

ptr = pole;

for (i = 0; i < 10; i++){
    /* hodnota prvku pole */
    printf("Hodnota *ptr = %d\n", *ptr);
    /* adresa prvku pole */
    printf("Adresa ptr = %p\n\n", ptr);
    ptr++;
}
```

Zamyšlení: Proč pole indexujeme od 0?

Jak bylo řečeno, identifikátor pole je ukazatel na první prvek pole. Je tedy možné hodnotu prvního prvku pole přečíst pomocí hranatých závorek, nebo za pomoci operátoru dereference následovně.

```
int pole[] = {1, 2, 3};

printf("1. prvek je %d \n", pole[0]);
printf("1. prvek je %d \n", *pole);
```

Jelikož víme, že přičtením jedničky k ukazateli dostaneme adresu následujícího prvku pole, můžeme 3. prvek získat následujícími způsoby.

```
int pole[] = {1,2,3};

printf("3. prvek je %d \n", pole[2]);
printf("3. prvek je %d \n", *(pole+2));
```

Obecně tedy platí, že `pole[index]` a `(pole + index)` jsou ekvivalentní výrazy. Index je jen číslo, které je přičteno k ukazateli pro nalezení prvku.[80]

[80] V jazyce C je překvapivě v pořádku i tento zápis.

```
int pole[] = {1, 2, 3};

printf("%d", 2[pole]);
```

Protože platí `pole[2] == (pole + 2) == (2 + pole) == 2[pole]`.

Porovnávání dvou pointerů

K porovnávání dvou pointerů lze použít operátory <, >, <=, >=, == a !=.
Výrazy typu `ptr1 < ptr2` mají smysl pouze v případě, kdy jsou stejného typu a ukazují na stejný úsek paměti (například jedno pole). Konkrétní příklad může být následující.

```
int pole[] = {1,2,3,4,5,6,7,8,9,10};
int *ptr;

for (ptr = pole; ptr < pole + 10; ptr++){
    /* hodnota prvku pole */
    printf("Hodnota *ptr = %d\n", *ptr);
    /* adresa prvku pole */
    printf("Adresa ptr = %p\n\n", ptr);
}
```

Odečítání dvou pointerů

Jedná se o výrazy `ptr1 - ptr2`. Ty mají opět význam pouze, pokud jsou pointery stejného typu a ukazují-li na stejný úsek paměti.

Výsledkem rozdílu je počet prvků (datového typu, na který jsou oba pointery definovány), které se mezi dvě adresy vejdou.

Funkce pro práci s pamětí

Funkce pro přidělování a uvolňování paměti jsou definovány v knihovně `stdlib`. Je tedy nutné tuto knihovnu do kódu přidat. V následující části si okrajově popíšeme ty nejčastěji používané funkce.

Přidělování paměti

Nejčastěji používanou funkcí je funkce `malloc()`. Ta bere jako argument celé číslo, které udává kolik bytů chceme alokovat. Tato funkce vrací ukazatel na `void`, který představuje adresu prvního přiděleného prvku. Tento pointer je vhodné přetypovat na pointer na typ, pro který paměť alokujeme. `void` totiž nenese žádnou informaci o tom, jak s přidělenou pamětí pracovat (nelze provádět aritmetické operace, operace dereference a jiné).
Pokud v paměti není dostatek místa, pak má vrácený pointer hodnotu `NULL`.
Přetypování je možné provést dvěma způsoby:

- implicitně: přiřazením `novy_typ *p = malloc(...);`
- explicitně: `(novy_typ) p;`

Nejčastější použití funkce `maloc()` vypadá následovně.

```
int *ptr_i;
ptr_i = malloc(sizeof(int));
```

Uvolnění paměti

Veškerou paměť, kterou programátor alokuje, by také měl dealokovat.[81] K tomu slouží funkce `free()`. Jako argument se funkci předává ukazatel na alokovanou paměť.[82] Obecně je vhodné paměť dealokovat hned v okamžiku, kdy už není potřeba, tedy ne až na konci programu. Po uvolnění paměti přestává dynamická proměnná fakticky existovat, i přes to, že ukazatel stále existuje. Ukazatel ukazuje na místo v paměti, kde už nic není.[83] Dobrým zvykem je tomuto ukazateli po uvolnění paměti přiřadit hodnotu `NULL`.

Další funkce

Další funkcí na alokování paměti je funkce `calloc()`, která se využívá v případech, kdy je nutné alokovat paměť pro `n` prvků stejné

Průvodce studiem
Garbage collection je způsob automatické správy paměti. Funguje tak, že speciální algoritmus (garbage collector) vyhledává a uvolňuje úseky paměti, které již program nepoužívá.

[81] Jazyk C neobsahuje *garbage collector*, který by automaticky spravoval paměť. Operační systém paměť uvolní až po skončení procesu.

[82] Je třeba předat přesně ten ukazatel, který vrátila alokační funkce. Pokud tuto proměnnou modifikujeme, například pomocí pointerové aritmetiky, nedokáže funkce `free()` uvolnit paměť správně.

[83] Přesněji řečeno, něco v paměti být může, ale může to být to část paměti, ke které už nemáme přístup.

velikosti. Funkce přijímá dva argumenty – počet prvků a jejich velikost.

```
ptr = calloc(n, sizeof(int));
ptr = (int *) malloc(n * sizeof(int));
```

Oba příkazy uvedené v příkladu alokují místo pro `n` prvků typu `int`. Funkce `calloc()` navíc tyto prvky vynuluje.
Pro *změnu velikosti alokované paměti* používáme funkci `realloc()`, která jako první argument bere pointer na již dříve alokovanou paměť a jako druhý novou požadovanou velikost. Hlavička této funkce je následující.

```
void *realloc(void *adresa, size_t velikost);
```

Obsah paměti se nezmění.
Kromě knihovny `stdlib` i knihovna `string` obsahuje užitečné funkce pro práci s pamětí (např. `memcpy()`, `memset()`, `memmove()`).

Vícerozměrná pole

Doposud jsme pracovali s poli, jejichž prvky byly základní datové typy. Takovým polím se říká *jednorozměrná*. Pole, jehož prvky jsou jednorozměrná pole, je pole *dvourozměrné*. Taková pole si můžeme představit jako tabulku. Samozřejmě mohou být pole i vícerozměrná.
K prvkům dvourozměrných polí přistupujeme pomocí dvou indexů, jeden určuje, ve kterém jednorozměrném poli se prvek vyskytuje, druhý index je indexem prvku v tomto poli. Pokud si představíme dvourozměrné pole jako tabulku, tak jeden z indexů by představoval index řádku a druhý sloupce.
Definice vícerozměrného pole je obdobná, jako definice pole jednorozměrného. V hranatých závorkách je potřeba specifikovat velikosti všech dimenzí daného pole. Každá dimenze se píše zvlášť do hranatých závorek.

```
typ jmeno[v1]...[vn]
```

Příklad použití následuje.

```
/* definice dvourozmerneho pole */
int pole[2][3];

/* definice ctyrrozmerneho pole */
int pole2[2][3][4][5];
```

Při definici pole je možné pole rovnou inicializovat obdobně, jako u jednorozměrných polí. I zde se hodnoty píší do složených závorek. Každý prvek je inicializací pole o jeden rozměr menší, než právě inicializované pole.

```
/* jednorozmerne pole */
int pole1[4] = {1, 2, 3, 4};

/* dvourozmerne */
int pole2[2][3] = {{1, 2, 3}, {1, 2, 3}};

/* trojrozmerne */
int pole3[2][3][4] = {{{1, 2, 3, 4}, {1, 2, 3, 4}, {1, 2, 3, 4}},
                      {{5, 6, 7, 8}, {5, 6, 7, 8}, {5, 6, 7, 8}}};
```

Přístup k jednotlivým prvkům vícerozměrného pole pomocí indexů je naprosto stejný, jako přístup do jednorozměrného pole, jen je potřeba specifikovat potřebný počet indexů.

```
/* prvek jednorozmerneho pole pole1 */
pole1[2] = 3;

/* prvek dvourozmerneho pole pole2 */
pole2[1][1] = 2;

/* prvek trojrozmerneho pole pole3 */
pole3[0][1][2] = 3;
```

Dvourozměrné pole je uloženo v paměti po řádcích. Tj. pro pole

```
int x[2][3];
```

se v paměti alokuje `2*3*sizeof(int)` bytů.
Předpokládejme, že `sizeof(int) = 2`, pak by uložení v paměti mohlo vypadat následovně.

adresa	100	102	104	106	108	110	112
prvek	`x[0][0]`	`x[0][1]`	`x[0][2]`	`x[1][0]`	`x[1][1]`	`x[1][2]`	volno

Pro efektivní práci s vícerozměrnými poli je dobré si uvědomit, co reprezentují proměnné `x`, `x[0]` a `x[0][0]`. Všechny tyto výrazy přísluší stejnému místu v paměti. Dvourozměrné pole je jednorozměrné pole, jehož prvky jsou pointery. Tedy prvek `x[0]` je ukazatel na první „řádek“ dvourozměrného pole `x`. Jinými slovy, typ jednorozměrného pole `x` je tříprvkové pole prvků typu `int`.

adresa	100	102	104	106	108	110	112
prvek	`x[0][0]`	`x[0][1]`	`x[0][2]`	`x[1][0]`	`x[1][1]`	`x[1][2]`	volno
	`x[0]`			`x[1]`			
	`x`						

`x` a `x[0]` sice představují tutéž adresu, ale jsou to pointery jiných typů. Použijeme-li pointerovou aritmetiku, tak `x + 1` reprezentuje adresu 106 a `x[0] + 1` reprezentuje adresu 102.

Alokace vícerozměrného pole

Jak už víme, alokace a dealokace na zásobníku se děje automaticky. O alokaci a dealokaci na haldě se stará programátor. Pří alokaci postupujeme analogicky, jako při alokaci jednorozměrných polí. Postup při alokaci dvourozměrného pole o rozměrech `m`, `n` je následující.

1. Alokujeme pole pointerů o `m` prvcích.
2. Alokujeme `m` jednorozměrných polí o `n` prvcích, přičemž pointery uložíme do pole alokovaného v prvním kroku.

Konkrétní příklad je v následujícím kódu.

```
/* 1. krok */
int **pole2d = malloc(m * sizeof(int *));

/* 2. krok */
for (i = 0; i < m; i++)
    pole2d[i] = malloc(n * sizeof(int));
```

Pointer může ukazovat na další pointer. Například `pole2d` ukazuje na pointer, který ukazuje na prvek typu `int`. Při deklaraci takového pointeru se používají dvě `*`. U vícenásobných pointerů se používá více `*`. Při dealokaci je potřeba postup obrátit.

1. Dealokujeme `m` jednorozměrných polí.
2. Dealokujeme pole pointerů.

Konkrétní příklad následuje.

```c
/* 1. krok */
for (i = 0; i < m; i++){
    free(pole2d[i]);
    pole2d[i] = NULL;
}


/* 2. krok */
free(pole2d);
pole2d = NULL;
```

Struktury, uniony

Doposud jsme pracovali jen s proměnnými, které obsahovali pouze jednu hodnotu, nebo poli, která obsahovala více hodnot, ale všechna musela být stejného typu. Nyní si ukážeme, jak vytvářet proměnné, které obsahují více hodnot, na které neklademe podmínku, že musí být stejného typu. K tomu budeme využívat takzvané *struktury*. Také si ukážeme méně používanou konstrukci tzv. *union* pro vytvoření datového typu, který může nabývat hodnot různých datových typů.

Struktury

Struktura je datový typ, který může být složen z datových prvků různých datových typů a který navenek vystupuje jako jediný objekt. Jinak řečeno, struktury nám umožňují spojit více hodnot, které mohou být různých typů, dohromady pod jedno jméno.
Definicí struktury v podstatě zavedeme nový datový typ `struct jmeno`, se kterým pak můžeme pracovat stejným způsobem, jako s datovými typy jazyka. Struktury se definují následujícím způsobem.

Průvodce studiem
Struktury jsou velmi podobné třídám v objektově orientovaných jazycích.

```c
struct jmeno{
 polozka_1;
 polozka_2;
 ...
 polozka_n;
};
```

Klíčové slovo `struct` představuje deklaraci struktury. Ta je seznamem deklarací uzavřených ve složených závorkách. `jmeno` představuje pojmenování struktury. `polozka_1`, ..., `polozka_n` se nazývají *složky* struktury. Každá složka je dána svým typem a jménem.

Jak už bylo řečeno, `struct` definuje datový typ. Za složenými závorkami může následovat seznam proměnných, které chceme deklarovat, stejně jako u jiných datových typů (analogie `int promenna_1, promenna_2;`).

```
struct jmeno{ ... } promenna_1, promenna_2, ..., promenna_n;
```

Znamená to, že se vytvoří proměnné `promenna_1, promenna_2, ..., promenna_n`, které jsou datového typu `struct jmeno`.
Jméno struktury `jmeno` v tomto případě není nutné.

```
struct { ... } promenna_1, promenna_2, ..., promenna_n;
```

Proměnné nemusíme vytvářet hned při deklaraci struktury, ale později. V takovém případě je nutné jméno struktury uvést. Proměnné se deklarují následovně.

```
struct jmeno{ ... };

struct jmeno promenna_1, promenna_2, ..., promenna_n;
```

Při deklaraci proměnné jí můžeme rovnou inicializovat. Inicializace je analogická jako u polí. Jednotlivé hodnoty se napíší do složených závorek.

```
struct jmeno promenna = {h1, ..., hn};
```

Ve složených závorkách postupně zadáváme jednotlivé hodnoty, které jsou přiřazeny postupně položkám struktury `polozka_1, ..., polozka_n`.
Přiřazením jedné proměnné (typu `struct jmeno`) do druhé dojde k nakopírování jejího obsahu na místo alokované pro druhou proměnnou.
Přístup k jednotlivým položkám struktury se děje pomocí tečkového operátoru následovné.

```
promenna.polozka1
```

Struktury můžeme do sebe vnořovat, můžeme struktury používat jako položky jiných struktur. Příkladem může být struktura představující úsečku. Úsečka je dána dvěma body, takže by její deklarace mohla být následující.

Nový standard ISO
Inicializaci lze provést i tak, že inicializujeme jen některé položky (tomuto způsobu se říká *designated* inicializace). Ve složených závorkách je vždy potřeba specifikovat název položky (před tento název se dá `.`, která představuje tečkový operátor) a její hodnotu.

```
struct jmeno promenna =
    {.polozka_i = hi,
     .polozka_j = hj,
         ...};
```

Ostatní položky jsou ve struktuře vynulované. Následující kód se často používá pro vytvoření vynulované struktury.

```
struct jmeno promenna =
    {};
```

```c
struct bod{
 float x;
 float y;
 float z;
};

struct usecka{
 struct bod A;
 struct bod B;
};
```

K jednotlivým složkám opět přistupujeme pomocí tečkových operátorů. `usecka1.A.x`[84] vrátí `x` ovou souřadnici bodu `A` náležící úsečce `usecka`.
Jednotlivé položky struktury jsou v paměti uloženy za sebou (mezi jednotlivými položkami však můžou být „vycpávky“). Například položky úsečky mohou být v paměti uloženy následovně.

[84] Není potřeba používat závorky, tento výraz se vyhodnocuje zleva.

A.x	A.y	A.z	B.x	B.y	B.z

Velikost této struktury (kolik paměti je potřeba přiřadit proměnné tohoto typu) nemusí vždy odpovídat velikosti součtu velikostí jednotlivých položek struktury.

Úkol 26
Je možné, aby struktura obsahovala jako položku samu sebe? Odpověď se dozvíme později.

Úkol 27
Vyzkoušejte si následující kód:

```c
#include <stdio.h>
struct struktura{
    float a;
    char b;
    int c;
};

int main(){
   printf("Velikost struktury: %i\n", sizeof(struct struktura));
   printf("Velikost polozek: %i\n", sizeof(float) + sizeof(int) + sizeof(char));
   return 0;
}
```

Také u struktur můžeme vytvořený datový typ pojmenovat pomocí `typedef` a vyhnout se nutnosti psát klíčové slovo struct u námi definovaného typu.

```
typedef struct{
 polozka_1;
 polozka_2;
 ...
 polozka_n;
} jmeno;
```

Jak již bylo řečeno, prvky struktury jsou v paměti ukládány za sebou (obdobně jako u polí), proto při deklaraci potřebujeme vědět, jak velká část paměti je potřeba pro jednotlivé proměnné. To je důvod, proč není možné, aby prvkem definované struktury byl prvek stejného typu, jako je právě definovaná struktura.

```
struct data{
    struct data d;  /* chyba */
    ...
}
```

Pokud potřebujeme, aby struktura obsahovala položku stejného typu, jako je právě definovaná struktura, lze to provést tak, že do struktury zahrneme pointer typu právě definované struktury (velikost pointeru je známá a nezávisí na velikosti struktury). Pointer na strukturu se definuje stejným způsobem, jako pointer na jiný typ.

```
typedef struct{
    polozka1
    ...
    polozkan
} jmeno;

jmeno s, *p_s;
```

`s` je struktura typu `jmeno` a `p_s` je pointer na strukturu typu `jmeno`, která nemá samozřejmě zatím přidělenou paměť.
Paměť pro strukturu lze dynamicky přidělit pomocí funkce `malloc()`.

```
p_s = malloc(sizeof(jmeno));
```

Pointer lze samozřejmě použít i pro odkaz na již existující strukturu, tedy například následovně.

```
p_s = &s;
```

Definujme strukturu následujícím způsobem.

```
jmeno s, *p_s = &s;
```

K prvkům struktury s nyní můžeme přistupovat různými způsoby, viz následující příklad.

```
/* pomoci jmena struktury */
s.polozka1;

/* pomoci pointeru p_s komplikovane */
(*p_s).polozka1;

/* pomoci pointeru p_s jednoduseji */
p_s->polozka1;

/* tento zapis je chybny, protoze operator . ma prednost
    pred operatorem dereference *  */
*p_s.polozka1;
```

Uniony

V dynamicky typovaných jazycích je možné proměnné přiřadit jakoukoliv hodnotu. To v jazyce C není možné, protože je nutné každé proměnné přiřadit její datový typ. Union nám umožňuje vytvořit datový typ, který obsahuje v jednom okamžiku jednu položku, ale je možné, aby tato položka byla různých datových typů. Union si můžeme zjednodušeně představit jako obyčejnou strukturu s tím rozdílem, že všechny její položky sdílejí jedno paměťové místo, tedy se překrývají (jsou na stejné adrese). V unionu se alokuje místo pouze pro největší položku, oproti strukturám, kde položky byly v paměti uloženy za sebou.
Definice unionů je prakticky shodná s definicí struktur. Nejčastěji používaná varianta je s pojmenováním pomocí `typedef`.

```
typedef union{
    polozka_1;
    polozka_2;
    ...
    polozka_n;
} nazev_union;
```

Jednotlivé položky jsou definovány svým typem a názvem. Jelikož jednotlivé prvky uniony sdílí stejnou paměť, je zřejmé, že přiřazením hodnoty některé z položek, můžeme pracovat pouze s touto položkou až do doby, než dojde k novému přiřazení. V opačném případě bychom pracovali s poškozenými daty.

Nový standard ISO
`union` můžeme inicializovat obdobně jako u struktur pomocí tzv. designated inicializace.

```
nazev_union u = {.
    polozka_i =
    hodnota_i};
```

Případně ve standardu `C89` byla zavedena inicializace první položky pouze hodnotou ve složených závorkách.

```
nazev_union u = {
    hodnota_1};
```

Přiřazení se nejčastěji provádí pomocí tečkového operátoru, obdobně, jako u struktur.
Kompletní příklad použití můžeme vidět v následujícím kódu.

```
#include <stdio.h>
typedef union{
    short pocet;
    float vaha;
    float objem;
} mnozstvi;

typedef struct{
    char nazev[20];
    mnozstvi mnozstvi;
} polozka;

int main(){
    polozka kosik[3];

    strcpy(kosik[0].nazev,"mrkev");
    kosik[0].mnozstvi.vaha = 0.5;

    strcpy(kosik[1].nazev,"mleko");
    kosik[1].mnozstvi.objem = 1.0;

    strcpy(kosik[2].nazev,"okurka");
    kosik[2].mnozstvi.pocet = 2;

    printf("%s %f\n", kosik[0].nazev, kosik[0].mnozstvi.vaha);
    printf("%s %f\n", kosik[1].nazev, kosik[1].mnozstvi.objem);
    printf("%s %d\n", kosik[2].nazev, kosik[2].mnozstvi.pocet);
   return 0;
}
```

V unionu není možné uchovávat informaci o tom, jaká z položek je právě využívána. Tento nedostatek se dá obejít pomocí struktury, která jako jednu ze svých položek bude mít `union` a další položka bude sloužit k identifikaci právě používané položky. Například následujícím způsobem.[85]

[85] Pro položku `info` by se hodilo použít výčtový typ `enum`.

```
typedef struct{
    int info;
    nazev_union x;  /* union */
} nazev_struktury;
```

Shrnutí

V této kapitole jsme se věnovali práci s pamětí. Představili si jeden z nejzákladnějších pojmů v jazyce C a to pointer. Představili jsme si pointerovou aritmetiku. A také, jsme si představili, jakým způsobem vytvářet proměnné, které uchovávají více hodnot, konkrétně pole (jak jednorozměrná, tak vícerozměrná), struktury a uniony.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jaký je rozdíl mezi alokací na zásobníku a na haldě?
2. V které části paměti se ukládají lokální proměnné?
3. Co je to ukazatel?
4. Jakou informaci získáme odečtením dvou pointerů?
5. Jaký je rozdíl mezi polem, strukturou a unionem?

Cvičení

Úkol 28
Napište program, který zjistí, zda jsou dvě pole stejná (tj. mají na všech indexech stejné prvky). Můžeme předpokládat, že pole mají stejný počet prvků.

Úkol 29
Upravte předchozí program tak, aby zjistil, zda pole obsahují stejné hodnoty (mohou být na různých indexech). Pozor, pole mohou obsahovat hodnotu i víckrát!

Úkol 30
Zkuste odhadnout, jaký bude výstup následujícího kódu. Na kterém místě bude znak `'B'`? Vaši domněnku ověřte.

```c
#include <stdio.h>

int main(){
    char retezec[] = "ABC";
    char znak = retezec[2];
    retezec[2] = retezec[1];
    retezec[1] = retezec[0];
    retezec[0] = retezec[2];
    retezec[2] = retezec[1];
    retezec[1] = znak;
    printf("%s", retezec);
    return 0;
}
```

Úkol 31
Napište program, který určí, zda je zadaný řetězec palindromem (čte se stejně od začátku i od konce).

Úkol 32
Napište program, který v zadaném řetězci spočítá počet mezer, souhlásek a samohlásek.

Úkol 33
Zkuste odhadnout, jaký bude výstup následujícího kódu. Vaši domněnku ověřte.

```c
#include <stdio.h>

int main(){
    int pole[] = {1, 2, 3};
    int *prvek = pole;
    pole[0] = 2;
    pole[1] = pole[2];
    pole[2] = *prvek;
    printf("%d", pole[2]);
    return 0;
}
```

Úkol 34
Předpokládejme, že máme pole `int pole[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};`

1. Jak zjistíte adresu 4. prvku pole?
2. Máme ukazatel, který ukazuje na některý prvek pole pole. Jak zjistíte index tohoto prvku v poli?
3. Máme dva ukazatele do stejného pole. Jak zjistíme, který z ukazatelů ukazuje na prvek, který je v poli dříve?

Úkol 35
Bez použití operátoru `sizeof()` zjistěte, jak velkou část paměti zabírají typy `char`, `int` a `double`.

Úkol 36
Napište program, který pro celočíselné argumenty m a n (zadané od uživatele) alokuje dvourozměrné pole o velikosti $m \times n$. Prvky pole budou reprezentovat násobky. Tedy prvek pole na indexu i, j bude roven $i \cdot j$. Toto pole vypište.

Úkol 37
Napište program, který převrátí pořadí prvků pole. Úkol vyřešte pomocí přístupu k prvkům pole přes hranaté závorky a pomocí pointerové aritmetiky. Srovnejte dobu běhu programu obou dvou přístupů.[86]

[86] Například za použití funkce `time()` následovně.

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

int main(){
    time_t zacatek = time(
        NULL);
    /* kod programu */
    time_t konec = time(
        NULL);
    printf("Cas behu je %d
        sekund", (konec -
        zacatek));
    return 0;
}
```

Úkol 38
Definujte strukturu pro reprezentaci zlomku. Pro dva zadané zlomky vypište jejich součet, rozdíl a součin.

Úkol 39
Napište program, který spojí dva textové řetězce (zadané přímo ve funkci `main()`). Ve funkci vytvořte nový řetězec a ten pak vypište.

Úkol 40
Napište program, který porovná dva textové řetězce a vypíše, který z nich je menší,[87] případně, že jsou shodné. Při práci s textovými řetězci používejte výhradně ukazatele, operátor dereference a pointerovou aritmetiku.

[87] Menší ve smyslu lexikografického uspořádání.

Úkol 41
Máme sejf definovaný následujícím zdrojovým kódem.

```c
#include <stdio.h>

typedef struct{
    char* popis;
    float hodnota;
} lup;

typedef struct{
    lup *lup;
    char *sekvence;
} kombinace;

typedef struct{
    kombinace cislo;
    char *vyrobce;
} sejf;

int main(){
    lup zlato = {"ZLATO!", 1000000.0};
    kombinace cislo = {&zlato, "1234"};
    sejf s  = {cislo,"SEJF2021"};

    return 0;
}
```

Jakou kombinaci následujících částí je potřeba zadat, abyste získali slovo ''ZLATO''?[88]

kombinace	.	s	+	lup	.	hodnota
s	->	cislo	.	popis	-	lup
cislo	:	lup	->	hodnota	->	popis
zlato	-	zlato	-	sekvence	+	zlato

[88] Z každého sloupce vyberte jednu z možností.

Funkce

Jak již víme, každý program v jazyce C se skládá z definice alespoň jedné funkce a to funkce `main()`. Vykonávání programu začíná voláním této funkce a končí opuštěním této funkce. Funkce `main()` je popsána sekvencí příkazů, které se postupně provádí. Ve funkci `main()` jsme doposud volali různé *předem definované* funkce (funkce obsažené v některé z knihoven). Nyní se budeme věnovat vytváření vlastních (*uživatelsky definovaných*) funkcí.

Definice funkce

Funkce můžeme chápat jako pojmenované části kódu, kterým předáváme *vstupní data* (argumenty funkce) a jejím vykonáním dostaneme data *výstupní*. Hodnotě, kterou dostaneme zavoláním této funkce, říkáme *návratová hodnota*.
Z předchozích kapitol již známe funkci `printf()`, které se jako vstupní data předává formátovací řetězec a případně další nepovinné argumenty. Při zavolání této funkce, se tato funkce vykoná (vypíše se řetězec do konzole).
Funkce se zejména používají pro zpřehlednění kódu, jeho lepší strukturování a také v případě, kdy chceme nějakou část kódu vykonávat opakovaně s jinými hodnotami.[89]
V jazyce C každá funkce začíná svojí hlavičkou, která vypadá následovně.

[89] Programátorská abstrakce.

```
typ jmeno(typ1 a1, typ2 a2, ..., typn an)
```

`typ` označuje datový typ návratové hodnoty.[90] Je možné vytvořit funkci, která žádnou návratovou hodnotu nemá (tj. nic nevrací), pak jako `typ` použijeme klíčové slovo `void` (takovým funkcím říkáme *procedury*). `jmeno` označuje jméno funkce (identifikátor funkce). Pro jména funkcí platí obdobná pravidla, jako pro názvy proměnných. V kulatých závorkách hned za jménem pak specifikujeme takzvané *parametry funkce*. Každý parametr je zadán dvojicí `typ` a `jmeno`. Jednotlivé parametry jsou v kulatých závorkách odděleny čárkou. S těmito parametry ve funkci pracujeme, jako s proměn-

[90] Ve starších překladačích mohl být datový typ návratové hodnoty vynechán, V tom případě byl implicitně návratový typ `int`. Uvádíme to zde jen pro úplnost, používat by se to nemělo.

nými. Funkce nemusí mít žádné parametry.[91] Za hlavičkou následuje blok příkazů (*tělo funkce*), které se mají po zavolání funkce vykonat. Jak již víme, pro návrat hodnoty z funkce použijeme příkaz `return hodnota;`. `hodnota` může být libovolný výraz, jehož výsledkem je hodnota.[92] Pokud má funkce v hlavičce jiný návratový typ, než je `void`, pak musí funkce nějakou hodnotu vracet!
Funkce vždy definujeme vně všech funkcí.[93]
Konkrétním příkladem definice funkce může být následující kód.

```
/* Vypocet mocniny s prirozenym exponentem */
double mocnina(double zaklad, int exponent){
    double vysledek = 1.0;
    int i;

    for(i = 0; i<exponent; i++){
        vysledek *= zaklad;
    }
    return vysledek;
}
```

Napíšeme-li tento kód do zdrojového kódu nějakého programu, nic se nestane, dokud tuto funkci nezavoláme. Funkci zavoláme tak, že napíšeme její jméno a do kulatých závorek seznam předaných hodnot argumentů. Jejich pořadí a počet musí odpovídat definici funkce. Před provedením těla funkce se tyto hodnoty předané jako argumenty naváží na parametry specifikované v hlavičce funkce.
Následující kód zavolá funkci `mocnina` s parametry `2` a `3`.

```
double y = mocnina(2,3);
```

Konkrétně, na proměnnou `zaklad` se naváže hodnota `2` (`2.0`), na proměnnou `exponent` hodnota `3`. Vykoná se tělo funkce a vrátí se hodnota uložená v proměnné `vysledek`. Tato hodnota se ve funkci, odkud byla funkce `mocnina` volána, naváže na proměnnou `y`.
Abychom mohli funkci zavolat, musí program vědět, že tato funkce existuje. Tedy musí být buď před jejím zavoláním definovaná nebo alespoň *deklarovaná*. Jak definovat funkci jsme si ukázali výše. Problém by mohl nastat, kdybychom měli více funkcí, které by se volaly navzájem.
Například máme funkce `funkceA()`, `funkceB()` a `funkceC()`. V těle `funkceA()` voláme funkci `funkceB()`, ve které voláme funkci `funkceC()` a v té voláme funkci `funkceA()`. Zdrojový kód následuje.

[91] Můžeme se setkat s tím, že se do kulatých závorek píše klíčové slovo `void`.

```
typ jmeno(void)
```

[92] Pokud není typ návratového výrazu shodný s návratovým typem funkce, provede se implicitní typová konverze.

[93] V jazyce C není možné funkce vnořovat.

```c
void funkceA(){
    funkceB();
}

void funkceB(){
    funkceC();
}

void funkceC(){
    funkceA();
}
```

Abychom mohli definovat funkci `funkceA()`, musíme znát funkci `funkceB()`, ale pro její definici potřebujeme znát funkci `funkceC()`, kterou ovšem definujeme pomocí funkce `funkceA()`.
Řešením je deklarace funkce a až pak její definice.[94] Funkci deklarujeme zapsáním její hlavičky ukončené středníkem, přičemž jména argumentů lze vynechat (tomuto zápisu se říká *funkční prototyp*), stačí pouze jejich typy. Deklarací funkce sdělujeme, že funkce s danou hlavičkou existuje. Někde za deklarací funkce však v kódu musí existovat i její definice.[95] Deklarace funkce `mocnina()` vypadá následovně.

```c
double mocnina(double zaklad, int exponent);
/* pripadne */
double mocnina(double, int);
```

Rekurzivní volání funkce

V těle funkce je možné volat jiné funkce, ale také sebe sama (z dřívějšího studia známe pojem *rekurzivní funkce*). Rekurzivní funkce v jazyce C vypadají úplně stejně jako klasické funkce. Příkladem je funkce pro výpočet faktoriálu níže.

```c
#include <stdio.h>
int fakt(int n){
    if(n <= 0)
        return 1;
    return n * fakt(n - 1);
}

int main(){
    int n = 10;
    printf("Faktorial cisla %d je roven %d", n, fakt(n));
    return 0;
}
```

[94] Tvrzení, že musí být funkce předem deklarovaná není úplně přesné. Pokud jí předem nedeklarujeme, pak se implicitně bere, že má funkce návratovou hodnotu a všechny její argumenty typu `int`. To však může a mnohdy vede k chybám, proto na implicitní deklaraci nespoléhejme.

[95] Ve skutečnosti existuje jeden způsob, který pochází z verze jazyka K&R, ale ten se běžně nepoužívá. Deklarace volané funkce se provede ve funkci volající, nebo kdekoliv na globální úrovni.

```c
int main(int b){
    double mocnina();

    mocnina(3,2);
}

double mocnina(double
    zaklad, int exponent){
    /* Telo funkce */
}
```

Do deklarace se v tomto případě nepíší typy vstupních argumentů.

U rekurzivních funkcí je potřeba si dát pozor na to, aby obsahovala tzv. *koncovou podmínku*, která určuje, zda má dojít k opětovnému (rekurzivnímu) voláni či nikoliv.

Předávání argumentů funkci

Když se bavíme o funkcích, je důležité také zmínit, jakým způsobem jsou parametry funkcím předávány. Máme-li takovýto kód, jaký bude jeho výstup?

```
#include <stdio.h>

void zmen_cislo_na_5(int cislo){
    cislo = 5;
}

int main(){
    int cislo = 4;
    zmen_cislo_na_5(cislo);
    printf("%d ", cislo);
    return 0;
}
```

Správná odpověď je 4. Jazyk C předává parametry tzv. *hodnotou*. To znamená, že při volání funkce se argumenty uloží na jiné místo v paměti[96] a případná změna proběhne na tomto místě. Po opuštění funkce je tato změna ztracena. Výhoda tohoto volání je, že je možné, aby argumentem předávaným funkci byl libovolný výraz. Nevýhodou je již výše zmíněná nemožnost měnit parametry uvnitř funkce.

[96] Na zásobníku se při zavolání funkce alokuje blok paměti, na něm se vytvoří místo pro všechny lokální proměnné, včetně proměnných definovaných jako vstupní argumenty funkce.

Pokud chceme, aby funkce mohla měnit hodnoty nějaké předané proměnné, předáme jí adresu této proměnné (říkáme, že argument *předáváme odkazem*). Adresa je nakopírována do lokální proměnné (adresa samotná je předána hodnotou), ale ukazuje stále na stejné místo.

Pro zopakování, adresu proměnné získáme pomocí operátoru `&` a ve funkci pak k hodnotě přistupujeme pomocí operátoru dereference `*`.

Příkladem může být následující funkce, která vrací celočíselný podíl dvou čísel `a`, `b` a zbytek uloží do předané proměnné `r`.

```
int deleni(int a, int b, int *r){
    /* r - zbytek po deleni */
    *r = a % b;
    return a / b;
}
```

```c
int main(){
    int x, y;
    /* x = 2, y = 3*/
    x = deleni(13, 5, &y);
    return 0;
}
```

Pole jako argument funkce

Pokud předáváme pole jako argument funkci, je nutné kromě pole samotného předat funkci i informaci o jeho velikosti. Jak víme, pole můžeme chápat jako ukazatel na jeho první prvek. Tím, že tento ukazatel předáváme hodnotou, ztrácíme vazbu na to, že ukazatel je polem. Konkrétní příklad následuje.

```c
#include <stdio.h>

void vynuluj_pole(int pole[], int velikost){
    int i;
    for(i = 0; i < velikost; i++){
        pole[i] = 0;
    }
}

int main(){
    int pole[5] = {1, 2, 3, 4, 5};
    vynuluj_pole(pole, 5); /* Vsechny prvky rovny 0 */
    return 0;
}
```

Všimněme si, jakým způsobem je definovaná hlavička funkce. To, že očekáváme pole naznačíme hranatými závorkami za názvem parametru.

Rozsah platnosti proměnných

V této části se budeme věnovat oblastem platnosti proměnných. Jinak řečeno, řekneme si, v jaké části kódu jsou jednotlivé proměnné viditelné (je možné s nimi pracovat).
Pro připomenutí, proměnné jsou platné od své definice až do konce bloku, ve kterém byly definovány. Pokud v aktuálním bloku proměnná nebyla definovaná, hledáme její definici v bloku nadřazeném (bloku obsahující aktuální blok). Pokud jí nenajdeme ani zde, hledáme v dalších nadřazených blocích. Pokud nenajdeme definici ani v nejvyšším bloku (globální rozsah),[97] nastane chyba při překladu. Proměnným, které jsou definovány v nejvyšším bloku

[97] Pokud se program skládá z více souborů, proměnné jsou platné jen do konce souboru ve kterém byly definovány.

říkáme *globální proměnné*. Ostatní proměnné nazýváme *lokální proměnné*.
Důsledkem je to, že v kódu může existovat více proměnných stejného jména. Například v následujícím kódu proměnné `vysledek` spolu vůbec nesouvisí.

```
int funkceA(){
    int vysledek = 1;
    /* Zbytek funkce*/
}

int funkceB(){
    int vysledek = 10;
    /* Zbytek funkce*/
}
```

Obě proměnné jsou platné jen ve funkcích, kde jsou definované. Ani jeden z bloků není nadřazený druhému. V následujícím příkladu je situace složitější.

Úkol 42
Jaký myslíte, že bude výsledek tohoto programu?

```
int foo = 5; /* Globalni promenna*/

void funkceA(){
    int foo = 6;
    printf("%d ", foo);
}

void funkceB(){
    foo = 7;
    printf("%d ", foo);
}

int main(){
    printf("%d ", foo);
    funkceA();
    funkceB();
    printf("%d ", foo);
    return 0;
}
```

První výpis proměnné `foo` bude 5. Protože v bloku funkce `main()` proměnná `foo` neexistuje, je definovaná v nadřazeném bloku (je to globální proměnná), kde má hodnotu 5. Pak je zavolána `funkceA()`. Zde je proměnná `foo` definovaná a inicializována hodnotou 6, která

se zde vypíše. Ve funkci `funkceB()` není definice proměnné `foo`, proto tato funkce hledá definici v nadřazeném bloku, takže pracuje s globální proměnnou.[98] Funkce změní její hodnotu na 7 a proto následující dva výpisy budou 7.
Globální proměnné bychom měli v kódu používat jen výjimečně. Globální proměnné zapříčiňuji, že jsou jednotlivé části kódu vzájemně provázané. To snižuje přehlednost kódu.[99]
To, že se program může skládat z více souborů problematiku platnosti proměnných ještě komplikuje. Někdy je potřeba sdílet proměnné i v rámci různých souborů. Platnost proměnných můžeme upravit pomocí tzv. *paměťových tříd*.

[98] Striktně řečeno, pracuje s vazbou v globálním prostředí.

[99] Můžeme se setkat s pojmem *špagety kód*, který označuje kód, jehož jednotlivé části jsou velmi těsně provázány.

Paměťové třídy

Proměnným je kromě typu vždy přiřazena tzv. paměťová třída (i když jsme o tom doteď nemluvili). Paměťové třídy určují, kde (v jaké části paměti) budou překladačem proměnné vytvořeny. Mimo to také určují, kde všude je proměnná viditelná (tím se rozšiřují možnosti rozdělení proměnných na lokální a globální).
V jazyce C existují následující paměťové třídy:

- `auto`
- `extern`
- `static`
- `register`

Třída `auto`

Proměnné v této třídě nazýváme automatické. Tato třída je implicitní pro lokální proměnné. Je-li proměnná definována uvnitř funkce a není uvedena žádná paměťová třída, pak je právě třídy `auto` a je uložena na zásobníku. Proměnná vzniká při vstupu do funkce a zaniká při výstupu z funkce. Tyto proměnné nejsou implicitně inicializované.

```
void func(){
    int j;
    ...
}
/* ma stejny vyznam jako */
void func(){
    auto int j;
    ...
}
```

Třída `extern`

Proměnné této třídy jsou uvozené klíčovým slovem `extern`. Tyto proměnné jsou uloženy v datové oblasti (Globální proměnné v obrázku 6). Tato třída je implicitní paměťovou třídou pro globální proměnné. Třída se používá zejména pokud máme program rozdělen do více souborů a potřebujeme proměnnou, kterou budou soubory sdílet. V jednom souboru tuto proměnnou definujeme bez klíčového slova `extern` a ve všech ostatních musí být deklarována s použitím `extern`.

```
/* Soubor 1 */
typ promenna; /* definice/deklarace */

/* Soubor 2 */
extern typ promenna; /* deklarace */
```

Třída `static`

Proměnné definované v této třídě jsou uloženy také v datové oblasti. Třída je uvozena klíčovým slovem `static`. Statická třída se používá pro globální proměnné nebo funkce, což má ten význam, že jsou viditelné pouze v tom souboru, ve kterém jsou definovány. Také se používá pro lokální proměnné, které si uchovávají svojí hodnotu mezi jednotlivými voláními funkce. Statická lokální proměnná existuje od prvního volání příslušné funkce až do doby ukončení programu, ale jako lokální proměnná není přístupná z vnějšku funkce.

```
void f(){
    int x = 0;
    static int y = 0;

    y++;
    x++;
}
```

Proměnná `x` je lokální (automatická) proměnná a `y` je lokální statická proměnná. Pokaždé, když voláme funkci `f()`, může být pro `x` alokováno jiné místo části zásobníku a vždy je inicializováno na hodnotu `0`. Proměnná `y` je inicializována na `0` při prvním vstupu do funkce `f()` a svou hodnotu si zachovává mezi jednotlivými voláními. Inicializace na `0` už se víckrát neprovede.

Třída `register`

Jazyk C je nízkoúrovňový jazyk a je možno, aby programátor vytvořil proměnnou, která nebude ukládána do paměti, ale do registru. Přístup do registru je rychlejší než přístup do paměti. Klíčovým slovem `register` navážeme proměnnou na určitý registr počítače (to na který je věcí překladače).

```
register typ promenna;
```

Toto znamená, že proměnná `promenna` může být uložena do registru,[100] pokud je nějaký volný (při označení více proměnných třídou `register` nemusí být všechny uloženy v registru). U těchto proměnných se neprovádí žádná implicitní inicializace a používají se jako lokální proměnné (nelze mít globální proměnnou typu `register`).

[100] Nelze získat adresu proměnných typu `register`.

Typové modifikátory

Proměnné (libovolného typu i paměťové třídy) mohou být ještě modifikovány *typovým modifikátorem*. Jazyk C rozeznává dva modifikátory:

- `const`
- `volatile`

Zápis je následující.

```
pametova_trida typovy_modifikator typ nazev_promenne;
```

Modifikátor `const`

Použitím modifikátoru `const` říkáme, že definované proměnné nesmí být po její inicializaci měněna hodnota. Na rozdíl od *symbolických konstant*, takto definované konstanty mají datový typ.
Pozor na vytváření konstantních pointerů. Hvězdička určující, že se jedná o ukazatel se nepíše k názvu proměnné (jak by bylo intuitivní), ale před klíčové slovo `const`.[101] Typické použití klíčového slova `const` je v definici formálních parametrů funkce.[102]

[101] `typ_ukazatele *const nazev`

[102] Zde se nejedná o konstantní ukazatel, ale ukazatel na konstantu.

```
void funkce(const char *str);
```

Tím ovšem neříkáme, že funkci `funkce()` musíme volat s konstantním řetězcem (můžeme, ale nemusíme), ale to, že tento parametr bude zpracováván pouze jako vstupní (bude jen pro čtení) i přes to, že se jedná o ukazatel a bylo by možné parametr `str` modifikovat.

Modifikátor `volatile`

Modifikátor `volatile` se používá v konkurentních (paralelních) programech. Tento modifikátor upozorňuje překladač, že takto definovaná proměnná může být modifikována nějakou událostí mimo náš program (např. pomocí zasílání signálů).

Funkce s proměnným počtem argumentů

Jak víme, při každém volání funkce jsou hodnoty předaných parametrů kopírovány na zásobník (a jsou uloženy přímo za sebou). Pokud známe adresu a typ (velikost paměti potřebné k jejímu uložení), můžeme přistoupit také k následujícímu parametru. Toho můžeme využít při vytváření funkcí s proměnlivým počtem argumentů. Příkladem takové funkce, kterou už známe, je funkce `printf()`. Funkce s proměnným počtem parametrů (nazýváme je *variadické* funkce) musí mít alespoň jeden pevný parametr a musí být určeno, kolik parametrů v paměti následuje (je znám počet, nebo je použita nějaká zarážka) a jaké jsou typy těchto parametrů. Tyto informace mohou být zadány následujícími způsoby:

- počet argumentů a jejich typy jsou předány formátovacím řetězcem (např. jako u `printf()`),
- předpokládáme typ parametrů a funkci je předán jejich počet,
- předpokládáme typ parametrů a máme určenou zarážku (např. práce s řetězci).

V deklaraci se používá výpustka `...` (tři tečky), která následuje po povinných argumentech.

```
typ nazev(typ povinny, ...)
```

Příklad použití následuje.

```
double prumer(int pocet, ...){
 /* Telo funkce */
}

prum = prumer(5, 1.2, 3.4, 5.6, 7.8, 9.0);
```

Díky knihovně `stdarg.h` nemusíme znát, jak je implementován zásobník volání. V této knihovně jsou implementovány:

- typ `va_list`, který se používá k uložení parametrů v zásobníku,
- makro[103] `va_start()`,

Průvodce studiem
Variadické funkce mohou v některých programovacích jazycích představovat bezpečnostní riziko. V Jazyce C jsou například útoky pomocí funkce `printf()`, známé jako útoky na formátovací řetězce. Jelikož se s předanými argumenty pracuje pomocí ukazatelů, je možné, aby se funkce pokusila číst více argumentů ze zásobníku , než tam bylo umístěno. CERT Koordinační centrum považuje funkce s proměnlivým počtem argumentů v C za bezpečnostní riziko.

[103] Makro je definice pravidla, jak bude vstupní posloupnost transformována na výstupní posloupnost (znaků, akcí, výpočtů a podobně). Makrům se budeme věnovat více později.

- makro `va_arg()`,
- makro `va_end()`.

V těle funkce je nutné deklarovat alespoň jednu proměnnou typu `va_list` a tu nastavit na první z nepovinných argumentů pomocí makra `va_start()`. `va_start()` bere jako argument identifikátor nastavované proměnné typu `va_list` a identifikátor posledního povinného argumentu.

```
va_list parametry;
va_start(parametry, posledni_povinny);
```

Jednotlivé parametry poté získáváme prostřednictvím makra `va_arg()`, jehož argumenty jsou ukazatele na zásobník a očekávaný datový typ dalšího parametru. Toto makro pracuje pouze s datovými typy `int` (případně `unsigned int`) a `double`. Všechny celočíselné argumenty jsou přetypovány na `int` a `float` je přetypován na `double`. Vyhodnocení `va_arg()` má za následek i posunutí ukazatele na zásobník.

```
cislo = va_arg(parametry, double);
```

Pokud jsme získali všechny nepovinné argumenty, je nutné ukončit práci s ukazatelem na zásobník pomocí makra `va_end()`.

```
ve_end(parametry);
```

Celý kód funkce na výpočet aritmetického průměru by mohl vypadat následovně.

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

double prumer(int pocet, ...){
    double soucet = 0;
    int i;
    va_list parametry;
    va_start(parametry, pocet);
    for(i = 0; i < pocet; i++){
        soucet += va_arg(parametry, double);
    }
    va_end(parametry);
    return soucet / pocet;
}
int main(){
    printf("%f \n", prumer(5, 1.2, 3.4, 5.6, 7.8, 9.0));
    return 0;
}
```

Parametry příkazové řádky

Funkce `main()` je funkce, která je spuštěna při startu programu. Této funkci mohou být předány argumenty, s nimiž je funkce volána. Jsou to dva argumenty:

- `argc` (argument count) – počet argumentů, se kterými je příkaz spuštěn,
- `argv` (argument vector) – ukazatel na pole znakových řetězců obsahujících argumenty. Vždy jeden parametr v jednom řetězci.

Průvodce studiem
Funkci `main()` je možné volat i explicitně, nicméně to není zcela standardní chování.

Hlavička funkce `main()` vypadá takto.

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

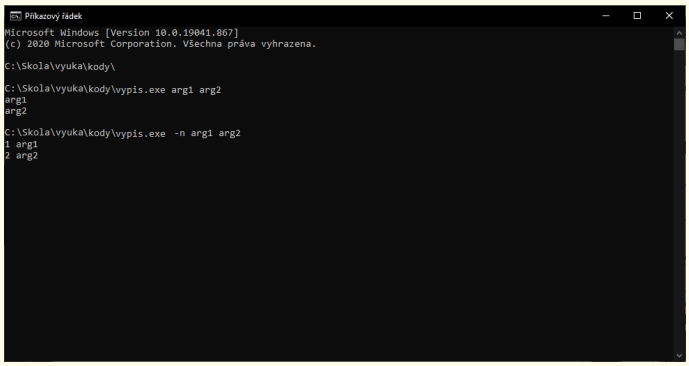
Jelikož je funkce `main()` spuštěna automaticky, o naplnění jejích argumentů se stará zavaděč programu (parametry jsou naplněny už při spouštění programu). Motivací k využití parametrů funkce `main()` je, že ne vždy je vhodná interakce s uživatelem při běhu programu a možnost spouštění programu s parametry nám poskytuje nástroj k tomu, jak spouštět úlohy dávkově.
Příkladem by mohl být program na sčítání dvou čísel. Nám doposud známý způsob je takový, že v kódu `main()` použijeme funkci `scanf()`, díky které získáme od uživatele dvě čísla, která následně sečteme a vypíšeme výsledek.
Způsob, o kterém mluvíme nyní, je takový, že zkompilujeme program (tím vytvoříme např. soubor `soucet.exe`), ten poté z příkazové řádky spustíme se dvěma argumenty (`soucet.exe 2 3`). Tyto dva argumenty se ve funkci sečtou a vypíše se výsledek.
Při tomto spuštění (`soucet.exe 2 3`) bude v proměnné `argc` uložena hodnota 3 a první tři položky `argv` budou obsahovat následující řetězce:
`argv[0] = ''soucet.exe''`
`argv[1] = ''2''`
`argv[2] = ''3''`
Jednotlivé parametry jsou odděleny mezerami. Pokud bychom potřebovali, aby argument obsahoval mezeru, napíšeme jej do uvozovek. Spustíme-li tedy program následujícím způsobem:
`soucet.exe 2 ''3 4''`, budou jednotlivé prvky pole `argv` následující:
`argv[0] = ''soucet.exe''`
`argv[1] = ''2''`
`argv[2] = ''3 4''`
Práci s předáváním argumentů funkci `main()` si ukážeme na jednoduchém příkladu.

```c
#include <stdio.h>

int main(int argc, char* argv[]){
    int i;

    for(i = 0; i < argc; i++){
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Takto zapsaný program po spuštění vypíše celé pole `argv`, tedy název programu a všechny argumenty, se kterými byl volán. Každý z nich bude na novém řádku.

Úkol 43
Upravte kód tak, aby vypisoval pouze předané argumenty.

Úkol 44
Běžnou konvencí programů v jazyce C (obzvláště pod systémem UNIX) je, že argument, který začíná znaménkem -, uvozuje volitelný parametr.
Upravte předchozí kód tak, že pokud bude prvním argumentem -n, tak se jednotlivé argumenty očíslují (tento argument se nevypíše).[104] Výpisy by měly vypadat následovně.



[104] Existuje knihovna (není to ale standardní knihovna jazyka C) `unistd`, která obsahuje funkci `getopt()`, která po každém volání vrátí následující možnost (parametr začínající -). Úlohu řešte bez této funkce.

Ukazatele na funkce

V jazyce C nejsou funkce proměnné, ale i přes to na ně lze ukazovat a tyto ukazatele je možné přiřazovat, ukládat do polí i vracet jako výsledek funkce.
Při definici funkce, je tato funkce uložena v paměti,[105] existuje tedy adresa, která ukazuje na začátek této funkce (při vytvoření funkce se vytvoří konstanta shodná s názvem funkce a ta uchovává adresu této funkce). Pomocí ukazatele na funkci lze funkci zavolat (a to i s příslušnými argumenty). Díky tomu je možné napsat funkci,

[105] V části, kterou jsme v obrázku 6 nazvali Kód

která bude mít jako argument ukazatel na jinou funkci, kterou je pak možné v kódu zavolat.
Uveďme si jako příklad problém třídění, který je možné rozdělit do tří částí:

- porovnávání – určení vzájemné pozice pro dva prvky,
- výměna – obrácení pořadí dvou prvků,
- třídící algoritmus, který provádí porovnání a výměny.

Průvodce studiem
Řadicí nebo třídicí algoritmus je algoritmus zajišťující uspořádání prvků (například pole) do požadovaného pořadí.

Třídící algoritmus je na operacích porovnání a výměny nezávislý. Změnou porovnávací funkce (případně i funkce na výměnu) dostaneme algoritmus třídící podle jiných kritérií.
Při deklaraci proměnné typu ukazatel na funkci je nutné vždy specifikovat také počet argumentů, jejich datové typy a návratovou hodnotu.

```
typ (*nazev)(typ1, typ2, ..., typn);
```

Proměnná `nazev` je ukazatel na funkci s n parametry (typu po řadě `typ1`, `typ2`, ..., `typn`) a návratovou hodnotou `typ`.
Závorky kolem jména funkce a hvězdičky jsou nutné. Pokud by se vynechaly, byla by to deklarace funkce, která má jako návratový typ `typ *`.
Jako konkrétní příklad si ukážeme definici funkce `polynomA` a vytvoření ukazatele na tuto funkci.

```
double polynomA(double x){
    return 3*x*x + 2*x + 1;
}

double (*ptr_polynom)(double) = polynomA;
```

Identifikátor přiřazované funkce (v tomto případě `polynomA`) se musí uvádět bez závorek se seznamem parametrů. I přes to, že chceme pointeru přiřadit adresu funkce, tak se zde nepoužívá `&` pro získání adresy. Je to proto, že kompilátor dokáže rozeznat, že chceme přiřadit ukazatel na funkci a ne funkci samotnou. Přiřadit proměnné přímo funkci totiž v C není možné.
Pro usnadnění práce s ukazateli na funkce můžeme využít `typedef` pro pojmenování datového typu ukazatele na funkci.

```
typedef double (*PTR_FUN)(double);

PTR_FUN ptr_polynom = polynomA;
```

Práce s ukazateli na funkce

Přiřazení adresy ukazateli provedeme následujícím způsobem.

```
ptr_polynom = polynomA;
```

Funkci pomocí ukazatele zavoláme následujícím příkazem.

```
v = ptr_polynom(-1);

/* pripadne */
v = (*ptr_polynom)(-1);
```

Tyto možnosti se neliší. Je možné používat obě. První je ale jednodušší, proto jí budeme v textu používat. Výsledkem vyhodnocení tohoto výrazu bude `double` hodnota 2, stejně, jako bychom zavolali funkci `polynomA(-1)`, na kterou ukazatel ukazuje.

Ukazatel na funkci jako parametr funkce

Funkcím, které jako argument berou jiné funkce, říkáme *funkce vyšších řádů*. Výpočet těchto funkcí je možné za běhu programu modifikovat pomocí funkcí, které jim jsou při jejich volání předány jako parametr. Příkladem je následující funkce.

```
int *map(int (*fce)(int), int *vstup, int pocet){
    /* telo funkce */
}
```

Tato funkce bere jako argument funkci, která má být aplikovaná na pole vstupních hodnot typu `int`, vstupní hodnoty a jejich počet. Funkci zavoláme následovně.

```
/* Funkce vracejici treti mocninu prvku x */
int na3(int x){
    return x * x * x;
}

pole_vysledku = map(na3, pole_vstupni, velikost_pole);
```

Pole ukazatelů na funkce

Pokud máme funkce stejných typů (se stejnou návratovou hodnotou), které berou stejné vstupní argumenty, můžeme tyto funkce uchovávat v poli.

```
/* Deklarace */
int (*pole_fci[5])(int);

/* Deklarace + inicializace */
int(*pole_fci[5])(int) = {na0, na1, na2, na3, na4};

/* Volani */
vysledek = pole_fci[1](-1);
```

Pole se definuje a inicializuje stejně jako pole jiného typu. K jednotlivým prvkům pole se přistupuje přes hranaté závorky (stejně jako u jiných polí).

Shrnutí

V této kapitole jsme si ukázali, jakým způsobem se vytváří uživatelské funkce (procedury, funkce s proměnlivým počtem argumentů ale i funkce vyšších řádů). Také jsme mluvili o rozsahu platnosti proměnných a o modifikátorech paměťových tříd.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jak vytvořit funkci, která nic nevrací?
2. Co znamená pojem variadická funkce?
3. Jakým způsobem můžeme funkci předávat argumenty?

Cvičení

Úkol 45
Napište program, který nalezne všechna čtyřciferná čísla, která jsou dělitelná číslem, které dostaneme jako sumu čísla tvořeného první a druhou číslicí a čísla tvořeného třetí a čtvrtou číslicí. Například číslo `3417` je dělitelné číslem `34 + 17`. Vhodné části algoritmu realizujte pomocí funkcí.

Úkol 46
Napište program, který pro dané přirozené číslo spočítá jeho rozdíl od nejbližšího většího prvočísla.

Úkol 47
Napište program, který pro dané přirozené n spočítá sumu:

$$1! + (1+2)! + (1+2+3)! + ... + (1+2+...+n)!$$

Úkol 48
Naprogramujte funkci, která bere jako argument dvě proměnné a provede výměnu hodnot těchto dvou proměnných.

Úkol 49
Naprogramujte funkci, která jako argument bere pole čísel a vrátí (nové) pole jejich druhých mocnin (tedy prvek na indexu `i` vráceného pole bude mít hodnotu druhé mocniny prvku na indexu `i` vstupního pole).

Úkol 50
Napište funkci `int porovnej`, která porovná dva předané textové řetězce a vrátí `-1`, pokud je první řetězec menší než druhý, `0`, pokud jsou řetězce shodné, nebo `1`, pokud je druhý řetězec menší než první. Při práci s textovými řetězci používejte výhradně ukazatele, operátor dereference a pointerovou aritmetiku.

Úkol 51
Napište funkci, která bere dva argumenty (`text` a `podretezec`). Funkce v daném textovém řetězci `text` vyhledá první výskyt zadaného podřetězce `podretezec`. Funkce vrací ukazatel na první znak nalezeného podřetězce nebo konstantu `NULL`, pokud podřetězec `podretezec` nebyl nalezen.

Úkol 52
Napište funkci, která vrací nejmenší z předaných celočíselných parametrů. Funkce bere jako první argument počet předaných celých čísel.

Úkol 53
Napište funkci `my_printf()`, která se bude chovat obdobně jako funkce `printf()`. Bude mít jeden povinný argument – řetězec, který může obsahovat formátovací sekvence, a libovolný počet nepovinných argumentů.
Formátovací sekvence:

- `*i` nahradí celým číslem, který byl předán jako nepovinný argument,

- `*c` nahradí znakem,
- `*f` nahradí desetinným číslem.

Pozor! funkce `va_arg()`, pracuje jen se základními datovými typy. Znak (`char`), je potřeba načíst jako `int` a přetypovat na char
`(char)va_arg(parametry, int)`
a desetinná čísla je potřeba načíst jako typ `double`
`va_arg(parametry, double)`
Uvnitř funkce je možné použít funkci `printf()`.

Úkol 54
Napište program `soucet`, který sečte celá čísla zadaná v příkazové řádce (terminálu). Každé číslo je samostatný argument. Například: `soucet 2 3 4` vrátí 9. Pozor! Jednotlivé argumenty jsou textové řetězce. Ty je potřeba převést na čísla.[106]

[106] Můžete použít funkci `atoi()`, která je součástí standardní knihovny `stdlib`.

Úkol 55
Napište program `nejdelsi`, který vrátí nejdelší (obsahující nejvíce znaků) ze svých argumentů předaných přes příkazovou řádku (terminál). Například: `nejdelsi ahoj jak se mas`
vypíše `ahoj`

Úkol 56
Naprogramujte funkce `na0`, `na1`, `na2`, `na3` a `na4`, vytvořte pole, do kterého tyto funkce uložíte, a vyzkoušejte si práci s tímto polem.

Úkol 57
Naprogramujte obecnou funkci pro třídění s tím, že jí je možné předat jako parametry funkci porovnání a funkci výměny.[107]

[107] Součástí standardní knihovny je funkce `qsort()`, která třídí prvky podle kritéria (funkce porovnání), které je jí předáno jako argument.

Úkol 58
Dopište následující program tak, že proměnné `i` a `j` definujete nejprve jako globální `int` a pak lokální `register int`. Porovnejte rychlost obou programů.

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

Práce se soubory

Doposud jsme pracovali tak, že vstup zadával uživatel z klávesnice (v konzoli), buď za běhu programu, nebo při jeho spuštění. Tento přístup se však nehodí pro programy, kde bychom chtěli zadávat velké množství dat. Z tohoto důvodu si ukážeme, jak můžeme v jazyce C pracovat se soubory.

Soubor

Soubor je z implementačního hlediska posloupnost bytů uložených na datovém médiu (např. disku) v několika blocích. Bloky mají stejnou velikost, ale nemusí nutně ležet přímo za sebou. Jak se s nimi pracuje je věcí operačního systému a nás to teď nemusí zajímat.[108] Z uživatelského hlediska je soubor posloupnost po sobě jdoucích bytů od začátku do konce souboru (často se používá výraz *proud dat*, nebo *stream*). Operační systém se stará o to, abychom dostali tyto bloky ve správném pořadí. Existují dva druhy souborů: textové a binární.

[108] Ani knihovny nepracují přímo se soubory. Jen žádají operační systém, aby danou operaci provedl.

Průvodce studiem
Proud je zdrojem nebo cílem dat a může být spojen s diskem, nebo jiným periferním zařízením. S proudy se pracuje například také v jazyce PHP.

Textové soubory

Textové soubory mají hlavní výhodu v tom, že jsou člověkem čitelné. Obsah souborů je možné prohlížet, vytvářet nebo opravovat běžným textovým editorem. Textové soubory jsou organizovány po řádcích (každý řádek končí znakem `'\n'`).[109]

[109] Ve Windows tomuto znaku předchází znak `'\r'`. Při přenosu textových souborů mezi systémy je na to potřeba pamatovat.

Binární soubory

Když otevřeme binární soubor v libovolném textovém editoru, tak nevidíme žádnou smysluplnou informaci, jak tomu bylo u textových souborů, ale jen změť dat. Binární soubor může být vnitřně libovolným způsobem organizován a při zběžném pohledu nevíme, co máme číst. Tyto soubory nejen že nejdou běžným editorem číst,[110] ale ani opravovat, nebo vytvářet.
Přes tyto nevýhody se binární soubory používají v praxi často. Například pro ukládání velkých dat. Pro uchování stejně velkého

[110] Přesněji, číst je lze, ale jsou v něm nesmysly.

množství informace je v textových souborech potřeba mnohem více prostoru. Příkladem mohou být například čísla. Číslo `65535` zabere v textovém souboru prostor `5` bytů, zatímco v binárním stačí byty `2`.
Další výhodou binárních souborů je rychlejší práce s nimi. Důvodem je jednak velikost souborů a také nutnost konverze čísla na text v případě textových řetězců, což je pomalá operace. Tato konverze v binárních souborech odpadá, protože se do nich zapisuje přímo obsah paměti po bytech.
Je zřejmé, že pro ukládání textů (znaků) nemá využití binárních souborů takovou výhodu, protože znaky v paměti zabírají jeden byte, stejně jako v textovém souboru.
Binární soubory nejsou stoprocentně přenositelné mezi různými operačními systémy. Záleží na délce jednotlivých základních typů (kolik místa zabírá například `int` v paměti) a způsobu uložení (endianita), což může být pro různé systémy různé.

Práce se soubory

Průvodce studiem
Jazyk C umožňuje se soubory pracovat buď tzv. *přímým voláním*, kde se volají služby jádra systému, proto je tento přístup systémově závislý a není součástí ANSI C. Dalším způsobem je pomocí *proudů*.

Se soubory se pracuje po jednotlivých blocích. Do paměti (*bufferu*) se načte celý blok a pracuje se s tímto blokem. Pokud například chceme přečíst dva znaky, načteme jeden blok do bufferu a z bufferu přečteme nejprve jeden znak a pak druhý (v tuto chvíli již nepřistupujeme k disku).
Obdobně je tomu i u výstupních operací. Nejprve ukládáme data do bufferu a, když je plný, zapíše se automaticky celý jeho obsah na disk do souboru jako jeden blok dat.[111]
Základní datový typ pro práci se souborem v jazyce C je ukazatel na objekt typu `FILE`.[112]

```
FILE *f;
```

Proměnná `f` se dá použít jak pro čtení, tak pro zápis do souboru. Tentýž identifikátor by ale neměl být v jednom programu používán pro obojí. Je vhodné používat identifikátor `*fr` pro soubor, který čteme, a `*fw` pro soubor, do kterého chceme zapisovat.
Chceme-li se souborem pracovat, je potřeba ho otevřít pomocí funkce `fopen()`, která bere jako argumenty název souboru a jakým stylem budeme se souborem pracovat.[113]

```
f = fopen(soubor, rezim);
```

`soubor` představuje cestu k souboru, který chceme otevřít. Ta může být relativní, nebo absolutní. Při práci s absolutní adresou si musíme dávat pozor na použití znaku \. V adrese je potřeba ho zadat

[111] Některé knihovny nabízí i nebufferované vstupní a výstupní funkce, ale ANSI C je přímo nepodporuje. Například pomocí funkce `setvbuf()` lze nastavit, jaké bufferování se má použít. Více se tomu budeme věnovat dále.

[112] Pole proměnných typu `FILE` je již definováno systémem a jeho deklarace je v knihovně `stdio`. Při otevření souboru se nealokuje další prvek pole, ale přiřadí se některý volný. Norma jazyka C nestanovuje, jak by měl být datový typ `FILE` definován. Programátor nesmí s proměnnou tohoto datového typu (s jejími položkami) pracovat přímo. Vedlo by to k nepřenositelnosti programu.

[113] Je možné mít současně otevřeno více souborů. Informace o maximálním počtu otevřených souborů, je uložena v symbolické konstantě `FOPEN_MAX` (často má hodnotu 20), která je definovaná v knihovně `stdio`.

zdvojeně, v jazyce C je totiž \ escape sekvence. Zadávání absolutních cest k souboru může způsobit problémy při přenosu programu mezi operačními systémy (UNIX a Windows).
`rezim` je textový řetězec, který udává, jakým způsobem budeme se souborem pracovat.
Jazyk C nerozlišuje mezi binárními a textovými soubory. Rozlišuje ale *binární a textový režim otevření* souboru.[114] V jazyce C je možné otevřít textový soubor jako binární a binární jako textový. Dokonce to má někdy i opodstatnění. Například otevření textového souboru jako binárního se využívá při komprimování.
Existuje 12 režimů, jak soubor otevírat. 6 režimů pro binární otevření (přidáme na druhou pozici řetězce `b`) a 6 pro textové (přidáme na druhou pozici řetězce `t`).[115] V následující tabulce je uvedeno 6 režimů a jejich vlastnosti.

[114] Toto rozlišená nemá v prostředí UNIX opodstatnění, tam není z hlediska zpracování souborů rozdíl.

[115] Pokud typ souboru nespecifikujeme, pak se s ním pracuje, jako s textovým.

Požadavek	Režim otevření					
	`"r"`	`"w"`	`"a"`	`"r+"`	`"w+"`	`"a+"`
soubor musí existovat	×			×		
existující soubor bude vymazán		×			×	
existující soubor bude rozšiřován			×			×
neexistující soubor bude založen		×	×		×	×
data lze odkudkoliv číst	×		×	×	×	×
data lze kamkoliv zapisovat		×		×	×	
data lze zapisovat pouze na konec			×			×

U režimu otevření souboru, ve kterém z něj lze číst i do něj zapisovat,[116] není možné, aby hned po zápisu následovala operace čtení. Mohl by nastat problém s pamětí (jak víme, zapisuje se do bufferu). Před čtením je nutné vynutit zápis celého bufferu do paměti, například pomocí funkce `fflush()`, kterou zmíníme později.
U souborů otevřených v binárním režimu je stanoveno, že knihovní funkce neprovádí žádnou dodatečnou akci s přečtenými, nebo zapisovanými byty (funkce nic k bytu nepřidávají ani neubírají). V textovém režimu je potřeba správně zacházet s různými konci řádků v různých systémech (viz výše). V textovém režimu čtecí funkce znaky `'\r'` odstraní (bohužel i když za ním není znak `'\n'`) a zapisovací funkce tento znak přidají před každý znak `'\n'`.
S daty v souboru lze pracovat dvěma způsoby. Prvním způsobem je *formátovaně*. Při tomto zpracování čtecí (případně zapisovací) funkce ovlivňují data (jsou jinak reprezentovaná v souboru, než v paměti[117]). Typickými funkcemi pro formátovou práci se soubory jsou funkce `fprintf()` a `fscanf()`. Formátované vstupy a výstupy jsou většinou využívány pro práci s textovými soubory otevřenými v texto-

[116] Režimy mající specifikátor + (update).

[117] Např. načítáme celé číslo, i když je v souboru uloženo jako řetězec.

vém režimu. V binárním případě nemají smysl (i když se použít dají).
Druhým způsobem je *neformátovaně*. Zde jsou data reprezentována stejně jako v paměti počítače. Vstupy a výstupy se dále rozlišují podle toho, kolik dat najednou zpracovávají.
Najednou můžeme číst/zapisovat buď jeden znak = byte (toho se využívá jak v textových souborech, tak binárních), jeden řádek (smysl má jen v textových souborech) nebo jeden blok dat (má smysl v binárních souborech).
Ke zpracování jednoho znaku slouží makra `getc()`, `putc()` a funkce `fgetc()`, `fputc()`.[118] Při otevření binárního souboru v textovém režimu zde můžeme narazit na problémy s některými hodnotami bytů. Jedním z příkladů takových bytů je byte s hodnotou 255. Tento byte může být za nevhodných okolností považován za konec souboru (pro `signed char` je 255 rovno -1 a tudíž rovno konstantě `EOF`[119]).
Ke zpracování jednoho řádku slouží funkce `fgets()` a `fputs()`.
Blok si můžeme představit jako pole o určitém počtu prvků určité velikosti. Nejjednodušším případem jsou prvky typu `char`. Za blokem jednoho typu může následovat blok jiné délky složený z prvků jiného typu. Tento způsob se však často nepoužívá. Nejčastěji jsou prvky v souboru jednoho typu. Pro čtení a zápis se využívají funkce `fread()` a `fwrite()`.
Když ukončíme práci se souborem (už z něj nepotřebujeme více číst, nebo do něj už nechceme zapisovat), musíme to systému oznámit. Říkáme, že soubor chceme zavřít, a systému to dáme vědět příkazem `fclose()`.

[118] Funkce i makra pracují stejně, pouze s tím rozdílem, že makra by měla být rychlejší.

[119] `EOF` (end of file) je symbolická konstanta, která se používá pro detekci konce souboru.

```
fclose(f);
```

Sice se v mnoha systémech soubory po ukončení programu automaticky zavřou, ale není dobré na to spoléhat. Navíc počet současně otevřených souborů je omezený. Díky tomu, že se zapisuje do souboru přes buffer (jak bylo řečeno dříve), tak by se mohlo stát, že po havárii programu by nemuselo být do souboru zapsáno vše (zůstalo to v bufferu).
Je dobré testovat, zda se podařilo soubor otevřít/zavřít. Nejčastější příčinou toho, že se nepodařilo soubor otevřít, je zadání špatné adresy/názvu souboru, nebo absence oprávnění. Další možnou příčinou je to, že je možné mít otevřeno jen omezené množství souborů najednou. Při uzavírání souborů může nastat problém s tím, že tak velký soubor není možné na disk uložit.
Obě funkce `fopen()` i `fclose()` vrací hodnotu, která se používá právě k této kontrole. Při nesprávném otevření `fopen()` vrací `NULL`. Při nesprávném uzavření `fclose()` vrací konstantu `EOF`.

```c
#include <stdio.h>

int main(){
    FILE *fr;

    if((fr = fopen("soubor.txt", "r"))==NULL)
        printf("soubor.txt se nepodarilo otevrit \n");

     if(fclose(fr)==EOF)
        printf("soubor.txt se nepodarilo uzavrit \n");

    return 0;
}
```

Čtecí a zapisovací funkce

Formátovaný vstup a výstup

Funkce `fscanf()` má obdobnou syntaxi jako funkce `scanf()` a používá se k formátovanému čtení vstupu ze souboru. Prvním argumentem je proměnná typu `FILE*`, druhým formátovací řetězec a dále ukazatele na proměnné, kam se mají načtená data uložit. Funkce vrací počet úspěšně přečtených položek (při čtení konce souboru vrací `EOF`[120]).
Symbolická konstanta `EOF` má hodnotu `-1`, je tedy vhodné znaky načítat jako `int`.
Jiným způsobem, jak otestovat, zda jsme nenarazili na konec souboru, je využití makra `feof()`. Toho se využívá hlavně v případě, že pracujeme s binárními soubory. Toto makro vrací nenulovou hodnotu (`true`), pokud poslední čtený znak byl za koncem souboru, nulovou hodnotu (`false`), pokud ještě nejsme na konci souboru.
Funkce `fprintf()` je obdobou funkce `printf()`. Jediný rozdíl je v tom, že se jako první argument předá proměnná typu `FILE*` určující proud, do kterého budeme zapisovat.

[120] `EOF` vrací i v případě, že nastala chyba.

```c
/* formatovane cteni ze souboru */
fscanf(f, "format", argumety);

/* formatovany zapis do souboru */
fprintf(f, "format", argumenty);
```

Čtení a zápis jednoho znaku

Jak již bylo řečeno, pro čtení a zápis jednoho znaku se používají makra `getc()`, `putc()` a funkce `fgetc()`, `fputc()`, které mají stejnou syntaxi i funkcionalitu. Příklad použití následuje.

```
/* cteni znaku ze souboru */
c = getc(f);
c = fgetc(f);

/* zapis znaku c do souboru */
putc(c,f);
fputc(c,f);
```

`f` v kódu představuje proud dat a `c` označuje proměnnou, z/do které budeme načítat znak.

Čtení a zápis jednoho řádku

Při práci s textovými soubory můžeme pracovat s jednotlivými řádky souboru. Pro přečtení jednoho řádku slouží funkce `fgets()`, která uloží přečtený řádek do řetězce. Hlavička funkce vypadá následovně.

```
char *fgets(char *str, int max, FILE *fr);
```

Funkce čte řetězec ze souboru `fr` až do konce řádku, maximálně však `max` znaků a to včetně znaků `'\n'` a `'\0'`. Funkce vrací ukazatel na `str` nebo při dosažení konce souboru `NULL`. V případě, že je řádek delší, než stanovené `max` bude do řetězce načteno `max-1` znaků a na konec bude přidán znak `'\0'`. Opětovné volání `fgets()` bude pokračovat načítáním stejného řádku z pozice, kde skončilo předchozí čtení. Tj. pokud předchozí volání `fgets()` nepřečetlo celý řádek, pokračuje následující `fgets()` čtením téhož řádku.
Pro zápis celé řádky (celý řetězec) do souboru použijeme funkci `fputs()`.

```
char *fputs(char *str, FILE *fw);
```

Funkce po zapsání řetězce do souboru nepřidává znak dalšího řádku, ani nezapisuje ukončovací znak. Funkce vrací nezápornou hodnotu, pokud byl zápis úspěšný, `EOF` pokud ne.
Dalším způsobem, jak přečíst řádek souboru, je číst jednotlivé znaky (například pomocí `getc()`), v tomto případě je třeba detekovat konec řádku. Jak ale již víme, v jazyce C znak `'\n'` označuje nový řádek. Tedy pokud bychom chtěli číst znaky do konce řádku, tak budeme číst znaky dokud nenarazíme na tento znak (případně konec souboru).

Čtení a zápis jednoho bloku

Pro čtení a zápis bloků se využívají funkce `fread()` a `fwrite()`.

```
/* cteni bloku */
int fread(void *kam, size_t rozmer, size_t pocet, FILE *fr);

/* zapis */
int fwrite(void *odkud, size_t rozmer, size_t pocet, FILE *
    fw);
```

Prvním parametrem `fread()` je ukazatel `kam`, který ukazuje na začátek místa v paměti, kam se budou data ze souboru načítat. `rozmer`, je velikost položky v bytech, kterou načítáme (může to být např. `sizeof(typ)`). Třetím parametrem je `pocet` představující počet položek, které čteme. Celkem tedy přečteme `pocet * rozmer` bytů. Posledním argumentem je datový proud `fr`, ze kterého data čteme. Obdobně funkce `fwrite()` zapíše `pocet` prvků velikosti `rozmer` do souboru `fw` z paměti kam ukazuje pointer `odkud`.
Obě funkce vrací počet skutečně přečtených, nebo zapsaných položek. Ten se liší od parametru `pocet`, pokud jsme při čtení došli na konec souboru nebo pokud došlo k chybě. To lze zjistit pomocí funkcí `feof()` a `ferror()`.
Funkce `ferror()` testuje, zda nedošlo k chybě (obdobně jako `feof()` testuje konec souboru). Pokud nedošlo k chybě, vrací `0`, jinak kladnou hodnotu.

Pozice čtení a zápisu

Při otevření souboru pro čtení, ukazuje *kurzor* (souborový indikátor pozice v souboru) na začátek souboru. Pokud použijete libovolnou funkci pro čtení jednoho bytu, posune se kurzor na další byte. Tímto způsobem se čte soubor byte za bytem od začátku do konce. Pokud bychom chtěli číst soubor zase od začátku, je možné soubor zavřít a znovu ho otevřít. Tento postup je pomalý a nepraktický. V případě, že bychom chtěli znovu číst jen některé poslední znaky je to neefektivní (museli bychom projít celý soubor).
Při otevření souboru pro zápis zase ukazuje kurzor na konec souboru a problém je, jak řešit situaci, kdybychom chtěli zapisovat doprostřed souboru.
Jazyk C má nástroje na modifikaci kurzoru na požadovanou pozici v souboru. Jednou z možností je použití funkce `fseek()`. Tato funkce pracuje se souborovým streamem `f` a posouvá kurzor o `posun` bytů od místa `odkud`. `posun` může být i záporná hodnota. Funkce vrací v případě úspěchu nulu.

```
int fseek(FILE *f, long posun, int odkud);
```

Speciální konstanty, které můžeme použít jako parametr `odkud` jsou:

- `SEEK_SET` – označuje začátek souboru,[121]

[121] Pomocí něj zadáváme relativní vzdálenost od začátku souboru

- `SEEK_CUR` – označuje aktuální pozici
- `SEEK_END` – označuje konec souboru

Další používané funkce jsou `ftell()` a `rewind()`. Funkce `ftell()` vrací aktuální pozici kurzoru v souboru `f` (jeho posun od začátku souboru v bytech). Funkce `rewind()` posune kurzor na začátek souboru `f`. Je to totéž, jako `fseek(f, 0, SEEK_SET)`.

Úkol 59
Praktickým příkladem, kdy použít funkci `ftell()`, je pro zjištění velikosti souboru. To je možné udělat tak, že kurzor posuneme na konec souboru a zjistíme jeho vzdálenost od začátku. Vyzkoušejte.

Vrácení přečteného znaku zpět do bufferu

V mnoha případech při čtení znaků zjišťujeme, že máme přestat číst až po přečtení znaku navíc. Tento znak není možné vždy zahodit (může být součástí další informace). Pro vrácení tohoto znaku zpět do bufferu používáme funkci `ungetc()` s následující syntaxí.[122]

[122] Do vstupního bufferu lze vrátit i jiný znak než znak naposledy přečtený.

```
ungetc(znak, proud);
```

Povede-li se vrácení znaku, vrátí tento znak, při neúspěchu funkce vrací `EOF`.

Standardní vstup a výstup

Práce se soubory se příliš neliší od práce s obrazovkou a klávesnicí. Ve skutečnosti C pracuje s klávesnicí a obrazovkou jako se souborem (pomocí proudů). V knihovně `stdio` jsou definované dva konstantní pointery (typu `FILE*`), které představují dva proudy, otevřené operačním systémem po spuštění programu. Je to vstupní proud `stdin` (většinou vstup z klávesnice) a výstupní `stdout` (většinou výpis na obrazovku).[123] Za normálních okolností tyto proudy směřují do konzole, ze které byl program spuštěn. Tyto proudy je možné použít ve funkcích zmíněných v této kapitole. Například následující zápisy jsou ekvivalentní.

[123] Ve `stdio` je definován ještě proud `stderr`, který se používá pro výpis chybových zpráv.

```
/* cteni jednoho znaku z klavesnice */
getc(stdin);
getchar();

/* vypis jednoho znaku na obrazovku */
putc(c, stdout);
putchar(c);
```

Cíl/zdroj dat proudu lze přesměrovat pomocí prostředků operačního systému. Spustíme-li program následujícím způsobem

```
C:\Umisteni_souboru\program.exe > output.txt
```

přesměruje se výstupní proud `stdout` do souboru `output.txt`.[124] Při spuštění programu příkazem

[124] Použijeme-li `2>` následujícím způsobem
`C:\Umisteni_souboru\program.exe 2> error.txt`
přesměruje se proud obsahující chyby `stderr` do souboru `error.txt`.

```
C:\Umisteni_souboru\program.exe < input.txt
```

je proud `stdin` (vstup z klávesnice) nahrazen obsahem souboru `input.txt`.
Programy je možné zřetězit pomocí znaku `|`, tak, že výstup jednoho programu (`stdout`) bude vstupem druhého programu (`stdin`). Například následující příkaz

```
C:\Umisteni_souboru\program1.exe | program2.exe
```

spustí programy `program1.exe` a `program2.exe`. Veškeré výstupy `stdout` programu `program1` budou předány do proudu `stdin` programu `program2`.

Bufferování

Již dříve jsme zmínili, že operace pro čtení a zápis pracují tzv. bufferovaně. Buffer je část operační paměti spravované systémem. Zde jsou data dočasně uložena, než jsou zaslána na konečné místo určení (v případě čtení je to nejčastěji datová oblast, v případě zápisu místo na disku). Buffer má nejčastěji velikost `512` nebo `1024` bytů.[125]

[125] Tato velikost je závislá na velikosti nejmenšího úseku dat zapsatelného na disk.

Bufferování je výhodné zejména z hlediska rychlosti práce se soubory. Přístup k disku je pomalá operace. Čtení jednoho byte a `512` byte najednou trvá téměř stejnou dobu. Z operační paměti se pak čte mnohonásobně rychleji. Stejně tomu tak je u zápisu.
Bufferování má ale i své nevýhody. Největší z nich je, že může dojít k nesouladu dat v bufferu a na disku (při čtení dostáváme jiná data, než očekáváme). Další nevýhodou je, že může dojít k selhání zápisu bufferu na disk. Zápis bufferu na disk bez ohledu na to, zda je či není buffer naplněn, můžeme vynutit funkcí `fflush()`.
Operační systém přednastavuje pro různé vstupní a výstupní funkce různé druhy bufferování (toto nastavení lze většinou změnit, ale nedoporučuje se to).
ANSI C poskytuje následující možnosti bufferování.

- *Řádkové bufferování* – Do bufferu se načítá nebo zapisuje, dokud není plný nebo se v něm neobjeví znak nového řádku. Typicky je takto nastaven proud `stdin`, kde se načítá vstup z klávesnice dokud není stisknuta klávesa `ENTER`.
- *Blokové bufferování* – Do bufferu se načítá celá jeho velikost a nové načtení probíhá až když je celý jeho obsah zpracován. Do bufferu se zapisuje, dokud není plný a až pak se jeho obsah zapíše na disk. Typicky se tak pracuje se soubory bez ohledu na režim otevření.
- *Žádné bufferování* – Znak se zapisuje nebo čte okamžitě po volání příslušné funkce. Typicky se tak pracuje se `stdout` a `stderr`.

Funkce `setbuf()`

Funkci `setbuf()` můžeme použít pro vypnutí bufferování, nastavíme-li hodnotu `buffer` na `NULL`. Dále ji můžeme použít pro přesměrování bufferu do paměti, ke které můžeme přistupovat z našeho programu. Tuto můžeme použít na otevřený soubor před tím, než z něho bylo poprvé čteno nebo do něj zapsáno. Funkce `setbuf()` má následující funkční prototyp.

```
void setbuf(FILE *f, char *buffer);
```

Funkce `setvbuf()`

Funkce s funkčním prototypem

```
int setvbuf(FILE *f, char *buffer, int rezim, int velikost);
```

je rozšířením funkce `setbuf()`. Umožňuje díky dalším parametrům nastavit režim bufferování[126] či velikost bufferu (v bytech).

[126] Režimy bufferování jsou následující:

- `_IOFBF` – blokové (F – full)
- `_IOLBF` – řádkové (L – line)
- `_IONBF` – žádné (N – none)

Úkol 60
Zkuste experimentovat s velikostí bufferu a jak se změní rychlost vstupních a výstupních operací.

Další užitečné funkce z knihovny `stdio`

V této části jen ve stručnosti uvedeme další užitečné funkce z knihovny `stdio`.

Funkce `freopen()`

Funkce `freopen()` s funkčním prototypem

```
FILE *freopen(const char *jmeno, char *rezim, FILE *stream);
```

slouží k přesměrování proudu. Funkce otevře soubor s názvem `jmeno` v režimu `rezim` (stejný jako u `fopen()`) a přesměruje do něho již otevřený proud `stream` a zavře ho. Funkce při neúspěchu vrací `NULL`, při úspěchu ukazatel na nově otevřený soubor.

Funkce `rename()`

Funkční prototyp funkce `rename()` vypadá následovně.

```
int rename(const char *stary, const char *novy);
```

Funkce změní staré jméno souboru na nové. V případě úspěchu vrací nulu, v opačném případě nenulovou hodnotu. Pro správné fungování by měl být přejmenovávaný soubor zavřený.

Funkce `remove()`

Funkce `remove()` vymaže soubor (ten by neměl být otevřený) a v případě úspěchu vrátí nulu, nenulovou hodnotu v opačném případě. Funkční prototyp vypadá následovně.

```
int remove(const char *jmeno);
```

Funkce `tmpfile()`

Při práci se soubory občas potřebujeme mezivýsledky ukládat do pomocného souboru, který se na konci smaže. Funkce `tmpfile()` vytvoří dočasný pomocný soubor (v režimu ''`wb+`''), který po zavolání funkce `fclose()` smaže. Funkce vrací `NULL` v případě, že se nepodařilo soubor vytvořit a otevřít, jinak vrací ukazatel na tento soubor. Funkce má následující hlavičku.

```
FILE *tmpfile();
```

Shrnutí

V této kapitole jsme si ukázali, jakým způsobem můžeme v jazyce C pracovat se soubory, jak textovými, tak binárními.

Kontrolní otázky
Odpovězte na následující otázky:

1. V jakých režimech můžeme otevírat soubory?
2. K čemu se používá bufferování?
3. Co je to proud dat?
4. Jak se jmenuje standardní proud pro výstup a kam defaultně vede?

Cvičení

Úkol 61
Napište program, který bude číst ze souboru desetinná čísla (libovolný počet) a vrátí jejich průměr.

Úkol 62
Napište program tak, aby zkusil číst neexistující soubor. Zajistěte, aby program na tuto situaci vhodně reagoval.

Úkol 63
Napište program, který spočítá celkový počet znaků v souboru.

Úkol 64
Napište program, který uloží čísla od 0 do 9 do binárního i textového souboru. Zjistěte velikosti těchto souborů (stejně jako v Úloze 69). Otevře si tyto soubory v textovém editoru.

Úkol 65
Napište program, který uloží číslo 1234567 do binárního i textového souboru. Zjistěte velikosti těchto souborů (stejně jako v Úloze 69). Proč je v minulém případě velikost binárního souboru větší a v tomto ne?

Úkol 66
Napište program, který do binárního souboru uloží pole desetinných čísel (záleží na vás, jak velké).

Úkol 67
Napište program, který z desetinných čísel uložených v binárním souboru (například vytvořeného v předchozí úloze) spočítá průměr. Měl by být napsán obecně tak, aby fungoval i když nebudeme vědět, kolik je v souboru čísel.

Úkol 68
Napište program, který čte znaky ze vstupního souboru a opisuje je buď na obrazovku, nebo do souboru ''`novy.txt`''. To, kam se má provést výstup, zvolí uživatel.

Úkol 69
Napište program, který čte čísla ze vstupního souboru a ukládá je do jiného souboru. Pokud je čtené číslo větší než `100`, číslo se neuloží do souboru, ale na obrazovku se vypíše chybová hláška `Špatně zadané číslo CISLO` (`CISLO` je konkrétní číslo). Použijte vhodně proud `stderr`.

Úkol 70
Předpokládejme, že existuje soubor zprava.txt, obsahující následující text (bez uvozovek, včetně mezer):
''jakuzyp k sec stje v cajooecl''
Máme následující program `tajemstvi`:

```
#include <stdio.h>

int main(){
    char c;
    int i=0;
    while((c=getchar()) != EOF){
        if((i % 4) > 1)
            putc(c,stdout);
        else
            putc(c,stderr);
        i++;
    }
    return 0;
}
```

Co bude obsahem souborů `zprava1.txt` a `zprava2.txt`, pokud program spustíme následujícím způsobem?
`tajemstvi.exe < zprava.txt > zprava1.txt 2> zprava2.txt`
Svou domněnku ověřte.

Práce s preprocesorem

V první kapitole jsme si řekli, jakým způsobem je kód napsaný v jazyce C zpracováván do podoby spustitelného souboru. Zdrojový kód je nejprve předzpracován pomocí preprocesoru, následně je takto upravený kód předán překladači a poté je zpracován linkerem.
V této kapitole se budeme věnovat preprocesoru, který jsme doposud využívali nevědomky (například jsme používali direktivu `#include`). Preprocesor má mnoho možností, které dávají jazyku C další sílu.
Jak již bylo řečeno, preprocesor zpracovává zdrojový kód před překladačem, ale nekontroluje jeho syntaktickou správnost. Pouze provádí záměnu textů, odstraňuje z kódu komentáře a připravuje podmíněný překlad.
Řádky zpracovávané preprocesorem začínají # (přičemž před ani za # nesmí být mezera).

Průvodce studiem
V mnoha implementacích překladačů jazyka je preprocesor samostatný program spouštěný překladačem v rámci první fáze překladu.

Makra

Makro je definice pravidla, podle kterého vstupní posloupnost transformujeme na výstupní posloupnost. Tuto transformaci označujeme jako *substituci* nebo *expanzi* makra.

Makra bez parametru (konstanty)

Makra bez parametrů nazýváme *symbolické konstanty*. Symbolické konstanty zbavují kód tzv. magických nepojmenovaných čísel.[127]
Například v následujícím vzorečku na výpočet obvodu kružnice je jím číslo 3.14.

```
o = 2 * 3.14 * r;
```

Makra se obvykle definují na začátku programu. Preprocesor při úpravě kódu nahradí tyto konstanty skutečnou hodnotou. Konvencí je, jak již bylo řečeno, že se název píše velkými písmeny. Syntaxe je následující.

[127] Takto definované konstanty oproti proměnným definovaným modifikátorem `const` zabírají méně místa, ale mohou způsobovat problémy při ladění (překladač nezná jejich názvy, debugger je nevidí).

```
#define JMENO hodnota
```

Na konec řádku se nepíše znak ;. Následuje několik příkladů.

```
#define PI 3.14
#define AND &&
#define ERROR printf("Chyba programu");
```

Při zpracování je každý výskyt jména konstanty v následujícím textu zdrojového kódu nahrazen hodnotou této konstanty.

```
#define PI 3.14

int main(){
    float r = 3;
    float o = 2 * PI * r;  /* o = 2 * 3.14 * r; */
    return 0;
}
```

Výjimkou jsou výskyty jména konstanty uzavřené v uvozovkách (části textového řetězce), viz následující příklad.

Úkol 71
Jakým způsobem bychom vypsali jméno?

```
#define JMENO "Marketa"

int main(){
    /* Vypise: Moje jmeno je JMENO */
    printf("Moje jmeno je JMENO");
    return 0;
}
```

Rozsah platnosti konstanty je od definice až do konce souboru. Pokud je nutné změnit hodnotu konstanty, je nutné jí zrušit pomocí `#undef JMENO` a znovu definovat. Viz následující příklad.

```
#define JMENO "Marketa"

#undef JMENO

#define JMENO "TRNECKOVA"
```

Pokud je potřeba definici makra napsat na více řádků, přidáme na konec každého řádku znak \

```
#define DLOUHA_KONSTANTA 1.23456798\
910111213
```

Preprocesor tento znak vynechá a pokračuje ve zpracování hodnoty na následujícím řádku. Je-li potřeba použít znak \, je nutné ho zdvojit.
V ANSI C jsou předdefinovaná následující makra. Využívají se zejména při ladění.

- `__LINE__` – číslo právě zpracovávaného řádku programu (desítkové číslo),
- `__FILE__` – jméno právě zpracovávaného programu (řetězec),
- `__DATE__` – aktuální datum (řetězec ve tvaru mmm dd yyyy),
- `__TIME__` – čas překladu (řetězec ve tvaru hh:mm:ss),
- `__STDC__` – má hodnotu 1 pokud se jedná o ANSI C .

Makra s parametry

Pokud v programu používáme funkci, která je tvořena velmi malým počtem příkazů, bývá výpočet značně neefektivní, protože režie spojená s voláním funkce (předání parametrů funkci, skok do funkce, úschova návratové adresy, ...) je výpočetně náročnější, než provedení příkazů v těle funkce.[128] Místo klasické funkce lze použít makro s parametry, které nevytváří žádnou režii za běhu programu. Nevýhodou je vznik delšího (většího) programu a nemožnost použít rekurzi (viz dále).
Dle konvence se názvy takovýchto maker píšou malým písmem, jako klasické funkce. Syntaxe je následující.

[128] V rámci optimalizace někdy kompilátor krátké funkce „inlinuje“, tedy vloží tělo funkce do funkce, ze které je volána.

```
#define jmeno(arg_1, arg_2, ..., arg_n) telo_makra
```

Mezi jménem a závorkou nesmí být mezera (bylo by to bráno jako konstanta). `arg_1, ..., arg_n` se chovají podobně jako argumenty funkcí.[129]
Vzhledem k tomu, že při expanzi maker dochází pouze k nahrazení jednoho textu jiným, je doporučováno uzavřít celé tělo makra do závorek a stejně tak každý výskyt argumentů v makru, viz následující příklad.

[129] Samozřejmě je možné vytvořit makro s nulovým počtem argumentů.

```
#define na2(x) ((x) * (x))
#define na2a(x) x * x

na2(f + g);  /* ((f + g) * (f + g)) */
na2a(f + g); /* f + g * f + g */
```

Měli bychom se vyvarovat použití argumentů s vedlejším efektem.[130]

[130] Například argumenty obsahující operátory ++ nebo --.

Úkol 72
Podívejte na následující kód. Než ho zkusíte přeložit, zamyslete se, co bude výsledkem.

```
#define na2(x) ((x)*(x))

int i = 2;
na2(i++);
```

Makra, která se objeví ve svém vlastním těle se v ANSI C znovu nerozvíjejí. Pokud se název makra objeví v jeho vlastním těle, přepíše se tento název do výsledného zdrojového kódu.[131]

```
#define m(x) ((x) < 0 ? m(-x) : m(x))
```

Avšak je v pořádku použít v těle jednoho makra jiné makro. Viz následující příklad.

```
#define add(x,y) (x) + (y)
#define plus(x,y) add(x,y)
```

Úkol 73
Pro porovnání různého chování maker a funkcí naprogramujte funkci `fna2`, která bude vracet druhou mocninu. Co bude jejím výsledkem pro argument i++?

[131] Starší preprocesory se mohou zacyklit.

Variadická makra

Stejně jako funkce, i makra můžeme definovat tak, že mohou přijímat proměnlivý počet argumentů. Syntaxe definice takového makra je obdobná definici variadické funkce. Opět se používají tři tečky jako výpustka. Nepovinné argumenty (argumenty uvedené za všemi pojmenovanými argumenty) v těle makra nahrazují `__VA_ARGS__`.[132] Příklad následuje.

```
#define vypis(format, ...) printf(format, __VA_ARGS__)
```

Příklad použití a rozvoj tohoto makra následuje.

```
vypis("%i %i", 1, 2); /* printf("%i %i", 1, 2); */
```

[132] Výpustku je možné v makru pojmenovat a toto jméno pak v těle makra používat místo `__VA_ARGS__`. Jméno se píše bezprostředně před výpustku. Příklad následuje.

```
#define vypis(format,
    argumenty...) printf(
    format, argumenty)
```

Operátory # a

Tyto operátory se používají v makrech obsahujících parametry. Operátor # se píše před název parametru. V makru je tento výraz nahrazen řetězcem, který je stejný, jako argument makra. Operátor ## se používá ke spojení dvou argumentů v jeden řetězec. Příklad následuje.

```
#define plus(X,Y) printf("%s + %s = %d", #X, #Y, (X) + (Y))
/* plus(3,2);
   printf("%s + %s = %d", "3", "2", (3) + (2)); */

#define var(X) promenna ## X
/* var(10)
   promenna10 */
```

Podmíněný překlad

Podmíněný překlad používáme pro dočasné vynechání části kódu při kompilaci. Typicky se používá pro vynechání ladících částí programu, překlad platformově závislých částí zdrojového kódu či dočasné odstranění (zakomentování) větší části zdrojového kódu.
Syntaxe vypadá následovně.

```
#if podminka
    kod
#endif
```

Preprocesor vyhodnocuje podmínku, která následuje za `#if`. Pokud je tato podmínka vyhodnocena jako `false`, pak vše mezi `#if` a `#endif` je ze zdrojového kódu vypuštěno.
Za podmínkou `#if` může být další podmínka `#elif` (může jich být více). Ta se vyhodnotí v případě, že předchozí podmínka nebyla splněna. Pokud nejsou splněny žádné podmínky, vyhodnotí se část kódu za `#else` (není povinný). Podmíněný překlad se ukončuje `#endif`. Syntaxe je v takovém případě následující.

```
#if podminka1
    cast_1
#elif podminka2
    cast_2

    ...

#else
    cast_n
#endif
```

V podmínce musí být výrazy, které může vyhodnotit preprocesor (například v nich nelze používat hodnoty proměnných).
Podmíněný překlad můžeme řídit konstantním výrazem. Syntaxe je následující.

```
#if konstantni_vyraz
    cast_1
#else
    cast_2
#endif
```

Případně s použitím else-if větve.

```
#if konstantni_vyraz
    cast_1
#elif konstantni_vyraz2
    cast_2
#else
    cast_3
#endif
```

Pokud je hodnota `konstantni_vyraz` nenulová, překládá se `cast_1`, jinak se překládá `cast_2`.[133] Konkrétní příklad následuje.

[133] Ve druhém případě se zjišťuje, zda je `konstantni_vyraz2` nenulový a `cast_2` se překládá jen v tom případě, jinak se překládá `cast_3`.

```
#define ENG 1

#if ENG
    #define ERROR "error"
#else
    #define ERROR "chyba"
#endif
```

Úkol 74
Upravte kód, tak aby se rozeznávaly čtyři různé jazyky pro chybovou hlášku.

K dispozici jsou i direktivy `#ifdef` a `#ifndef`, které slouží k otestování, zda byla nějaká symbolická konstanta definovaná. `#ifdef JMENO` znamená, že kód následující za touto podmínkou se vykoná jen pokud je makro `JMENO` definováno (obdobně `#ifndef JMENO`, pokud `JMENO` není definováno).

```
#define ENG 1

#ifdef ENG
    #define ERROR "error"
#else
    #define ERROR "chyba"
#endif
```

`#ifdef` a `#ifndef` testují existenci jednoho makra. Pro složitější podmínky je potřeba použít operátor `defined` a logické spojky. Viz následující modifikace příkladu.

```
#define ENG 1

#if defined(ENG) && ENG
    #define ERROR "error"
#else
    #define ERROR "chyba"
#endif
```

Navíc na rozdíl od `#ifdef` a `#ifndef`, je možné použít složitější větvení pomocí `#elif`.

```
#define ENG 1

#if defined(ENG) && ENG
    #define ERROR "error"
#elif !defined(ENG)
    #define ENG 0
    #define ERROR "chyba"
#else
    #define ERROR "chyba"
#endif
```

Vkládání souborů

Bez předchozího vysvětlení jsme používali příkaz `#include <nazev>`, abychom mohli používat funkce z knihovny `nazev`. Preprocesor tento příkaz nahradí obsahem souboru `nazev` (tento soubor je do kódu inkludován).
`#include` lze použít dvěma způsoby následovně.
`#include <nazev>` (jak jsme používali doposud) a nebo `#include ''nazev''`. Tyto dva příkazy se liší tím, kde se soubor `nazev` má hledat.
`#include ''nazev''` hledá soubor ve stejném adresáři, ve kterém je uložen „volající“ program. Používá se zejména pro vkládání souborů, které jsme vytvářeli sami.
`#include <nazev>` hledá v systémovém adresáři. Používá se pro vkládání standardních knihoven.

Další direktivy preprocesoru

Kromě výše zmíněných direktiv existují ještě další, které se používají k řízení překladu. Jsou to direktivy `#line`, `#error`, `#warning` a `#pragma`.
Direktiva `#pragma` se používá k uvození implementačně závislých direktiv. Pokud překladač narazí na tuto direktivu a nerozumí jí, pak ji ignoruje. Každý překladač rozumí různým direktivám. Vývojová prostředí je například používají k uložení různých informací, které nemají se zdrojovým kódem nic společného.
Direktiva `#line` se používá k nastavení hodnot maker `__LINE__` a `__FILE__`.
`#error` a `#warning` vypisují při překladu varování (`#error` překlad ukončí). Nejčastěji se používají s podmíněným překladem. Za direktivou následuje text chybového hlášení, který se má vypsat.

Shrnutí

V této kapitole jsme si připomenuli, jakým způsobem je zdrojový kód psaný v jazyce C převeden do instrukcí, kterým rozumí počítač. Obzvláště jsme se zaměřili na preprocesor a jakým způsobem ho můžeme při programování využít.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jakým způsobem vytváříme symbolické konstanty?
2. K čemu využíváme podmíněný překlad?
3. Jaký je rozdíl mezi funkcí a makrem s parametry?

Cvičení

Úkol 75
Definujete makro `cti_int(i)`, které čte z klávesnice celé číslo. Makro musí jít použít i ve výrazu
`if((j=cti_int(k)) == 0)`
Nápověda: možná budete potřebovat operátor čárky.

Úkol 76
Definujte makro `je_velike(c)`, které vrátí `0` není-li znak velké písmeno a `1`, pokud je.

Úkol 77
Napište makro `na_3(x)`, které bude počítat třetí mocninu. Proměnné `i` a `j` předem definujte. Vyzkoušejte ho na následujících výrazech.

```
na_3(3)
na_3(i)
na_3(2 + 3)
na_3(i * j + 2)
```

Úkol 78
Následující kód upravte tak, aby se ve funkci `funkce()` hodnoty proměnných vypisovaly jen v případě, že je makro `VERBOSE` nastaveno na 1.

```
#include <stdio.h>

int funkce(int a, int b){
    int i, j = 0;

    for(i = 0; i < a; i = i + b){
        printf("i = %d \n", i);
        if(i > j){
            j = j + i;
        }
        printf("j = %d \n", j);
    }
    return i + j;
}

int main(){
    printf("Vysledek: %d ", funkce(50,5));
    return 0;
}
```

Úkol 79
Přeložte předchozí programy s použitím přepínače `-E` (jak už víme, při překladu zdrojového kódu se provede jen předzpracování pomocí preprocesoru). Podívejte se na výsledek.

```
gcc program.c -E -o program.txt
```

Dělení do souborů

V této kapitole se budeme věnovat dělení programu do více souborů a také oddělenému překladu. Rozdělíme-li program do více souborů, které pak pomocí `#include` vložíme do jednoho souboru, při překladu vznikne pouze jeden `.OBJ` soubor. *Odděleným překladem* souborů rozumíme přeložení každého souboru zvlášť (vznikne více `.OBJ` souborů) a jejich spojení pomoci linkeru do jednoho programu.

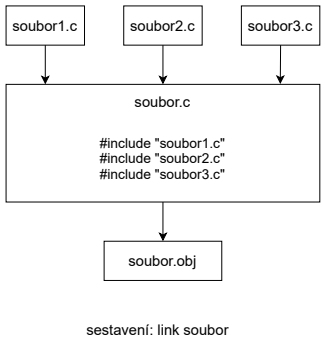
Představme si, že máme tři soubory `soubor1.c`, `soubor2.c` a `soubor3.c`. Funkce `main()` je v prvním uvedeném. Všechny soubory „inkludujeme“ do jednoho souboru `soubor.c`, který obsahuje tři příkazy `#include` a nic víc. Tento soubor se přeloží a vznikne jeden `.OBJ` soubor a ten se sestaví viz obrázek 7.

Nevýhodou tohoto přístupu je, že při změně v jednom ze souborů je potřeba přeložit i zbylé dva. Dále vzniká problém s globálními proměnnými, které mají stejný název, nebo které nechceme, aby byly viditelné v ostatních souborech.

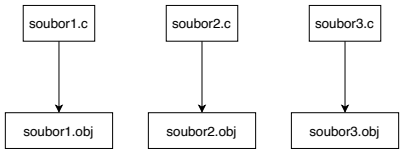
Při odděleném překladu se každý soubor přeloží zvlášť, vzniknou tedy tři `.OBJ` soubory viz obrázek 8. Výhodou je, že při změně jednoho souboru, není potřeba znovu překládat všechny ostatní.

Prvním krokem při dělení kódu do více souborů je vždy rozmyslet dělení problému na menší a co nejméně závislé části. Dále je potřeba si rozmyslet, jakým způsobem budou jednotlivé části navzájem komunikovat (je nutné stanovit tzv. *interface*). To je možné buď skrze sdílené globální proměnné, nebo lépe skrze volání funkcí z jednotlivých modulů.

Obrázek 7: Vkládání souborů.



Obrázek 8: Oddělený překlad.



Rozšíření platnosti globálních proměnných

Při použití komunikace přes globální proměnné, je potřeba rozšířit jejich platnost ze souboru, kde byly definovány, do všech souborů pomocí klíčového slova `extern`.

Podívejme se na následující příklad. Vytvoříme dva zdrojové kódy. `pomocny.c` vypadá následovně.

Průvodce studiem
Linker je program, který propojuje objektové kódy s knihovními moduly. V průběhu sestavování spustitelného souboru jsou nalezeny doposud neznámé adresy proměnných a funkcí a odpovídající knihovní moduly jsou připojeny k výslednému programu.

```c
int a;
extern int b;

int funkce1(){
    return a + b;
}
```

a `hlavni.c` obsahuje následující zdrojový kód.

```c
#include <stdio.h>

extern int a;
extern int funkce1(); /* zde je nutne uvest cely funkcni
    prototyp */
int b;

int main(){
    a = 3;
    b = 3;
    printf("%d\n", funkce1());
    return 1;
}
```

Rozsah platnosti proměnných `a` a `b` je díky `extern` v obou souborech. Pro sestavení programu použijeme následující příkaz.[134]

```
gcc -o program hlavni.c pomocny.c
```

Po spuštění programu `program` se na výstup vypíše číslo 6.
Většina vývojových prostředí provádí oddělený překlad automaticky.[135]

[134] Protože jsou externí deklarace zpracovávány linkerem, je možné, že by mohl vzniknout problém při použití dlouhých identifikátorů. Je zvykem používat identifikátory sdílených proměnných a funkcí s maximální délkou 8 znaků.

[135] Oddělený překlad se provádí tak, že se vytvoří projekt, ve kterém se vytvoří jednotlivé soubory, a překládá se celý projekt.

Statické globální funkce a proměnné

Pokud nechceme, aby z jiných souborů byly viditelné některé globální proměnné či funkce, použijeme pro ně třídu `static`. Tyto proměnné jsou viditelné (platné) jen v souboru, ve kterém byly definovány, a to i když je použito v ostatních souborech klíčové slovo `extern`. Použití `static` globálních proměnných i funkcí je výhodné zejména, pokud na projektu pracuje více lidí, nemusí se pak obávat použití stejného identifikátoru.
Podívejme se opět na příklad. Máme následující dva soubory.
`pomocny.c` vypadá následovně.

```
static int a;
extern int b;

static int funkce1(int x){
    return x;
}

int funkce2(){
    return funkce1(b) + a;
}
```

`hlavni.c` obsahuje následující zdrojový kód.

```
#include <stdio.h>

extern int a;
extern int funkce2();
int b;

int funkce1(){
    printf("%d\n", funkce2());
}

int main(){
    a = 3;
    b = 3;
    funkce1();
    return 1;
}
```

Překlad proběhne úspěšně, při sestavování se ale objeví chyba, že není definovaná proměnná `a`. V souboru `pomocny.c` byla tato proměnná definovaná jako `static`, tudíž, ani když použijeme v `hlavni.c` `extern int a;`, tuto proměnnou v souboru nevidíme. Smažeme-li klíčové slovo `static`, překlad i sestavení proběhne bez problémů. Nevadí ani, že v obou souborech používáme stejný identifikátor pro funkci `funkce1()`, protože funkce je v souboru `pomocny.c` definovaná jako `static`, nemá tudíž s funkcí definovanou v `hlavni.c` nic společného.

Dělení programu na menší části

V této části uvedeme několik obecně platných pravidel, které je dobré dodržovat při vytváření programů, jež se skládají z více částí (modulů), a to zejména, pokud se na projektu podílí více lidí. Ke každému zdrojovému kódu (soubor s příponou `.c`) vytváříme tzv. *hlavičkový soubor* (soubor s příponou `.h`).

Definice funkcí a definice proměnných, vytváříme v souborech `.c` (například `program1.c`). V tomto souboru striktně rozlišujeme, které proměnné a funkce budeme dávat k dispozici vně souboru. Snažíme se však, aby jich bylo málo. Spíše vytváříme speciální funkce, které zajišťují manipulaci s nimi.[136] Všechny ostatní funkce a globální proměnné označíme, jako `static`. Funkční prototypy všech funkcí a definice globálních proměnných zkopírujeme do hlavičkového souboru, který nese stejné jméno, jako zdrojový soubor (`program1.h`), a označíme je zde `extern`. Pokud z modulu poskytujeme nějaké symbolické konstanty, definují se v hlavičkovém souboru. Na začátku souboru `.c` vložíme hlavičkový soubor (`#include ''program1.h''`). Tím jsme zaručili, že ve zdrojovém kódu známe funkční prototypy všech funkcí i všechny symbolické konstanty. Deklarace globálních proměnných diky klíčovému slovu `extern` také nezpůsobí žádný problém. Do dalších modulů (například `program2.c`) vkládáme pouze hlavičkové soubory ostatních modulů (`#include ''program1.h''`).

[136] Tomuto přístupu říkáme *autorizovaný přístup*.

Doporučený obsah `.c` souboru

Soubor `.c` by měl obsahovat dokumentační část (jméno souboru a verze, stručný popis modulu, jméno autora a případně datum). Hlavička souboru by mohla vypadat následovně:

```
/*
 * obd_main.c   verze 1.0
 *
 * Program pro pocitani s obdelniky
 * ================================
 *
 * M. Trneckova, zari 2021
 *
 */
```

Poté by mělo následovat vložení všech potřebných hlavičkových souborů. Nejprve se vkládají systémové hlavičkové soubory příkazem `#include < >` a pak vlastní `#include '' ''`. Následují deklarace externích proměnných a funkcí (pokud nejsou v hlavičkovém souboru). Dále se definují globální proměnné, které se budou sdílet s ostatními moduly (nesmíme zapomenout přidat jejich deklaraci označenou `extern` do hlavičkového souboru). Následuje definice symbolických konstant a maker s parametry, které se používají pouze v tomto modulu, definice lokálních typů (`typedef`), definice statických globálních proměnných, funkční prototypy lokálních funkcí (nejlépe úplné). Poté se definuje funkce `main()`, pokud má být součástí modulu, následovaná definicí ostatních funkcí. Nejprve se uvádí globální funkce, tedy funkce, které mohou být volány

z ostatních modulů, a poté lokální, které mají modifikátor `static`.

Doporučený obsah hlavičkových souborů `.h`

Pro hlavičkové soubory platí podobná pravidla, jako pro soubory `.c`. Ke každému souboru `.c`, s výjimkou souboru, ve kterém je uvedena funkce `main()`,[137] by měl být vytvořen hlavičkový soubor. Hlavičkové soubory nesmí obsahovat žádné definice a inicializace proměnných. Neměly by obsahovat žádné příkazy `#include` kromě vkládání standardních knihoven. Dále by měly zajišťovat ochranu proti opakovanému vkládání souborů (pomocí podmíněného překladu).
Soubory by měly obsahovat dokumentační část (stejně jako u odpovídajícího `.c` souboru), podmíněný překlad proti opakovanému vkládání, definice symbolických konstant a maker s parametry (o kterých se předpokládá, že se budou používat i v jiných modulech), definice globálních typů pomocí `typedef`. Dále obsahuje deklarace globálních proměnných příslušného modulu a úplné funkční prototypy globálních funkcí (obojí označené jako `extern`).

[137] Tento soubor je pro všechny soubory nadřazený. Nemusí tedy ostatním modulům předávat žádné informace.

Konkrétní příklad

V této části s uvedeme konkrétní (ale ne kompletní) příklad, jak by mohlo vypadat členění programu do více souborů. Program bude počítat obvod a obsah obdélníku. Bude se skládat ze dvou modulů, modul `obd_funkce` (pro modul vytvoříme soubory `.h` a `.c`) a hlavní modul `obd_main`.
Začneme modulem obsahující funkce pro výpočet obvodu a obsahu obdélníku. Zdrojový soubor (`obd_funkce.c`) může vypadat následovně.

```
/*
 * obd_funkce.c   verze 1.0
 *
 * Funkce pro vypocet obsahu a obvodu
 * ==================================
 *
 * M. Trneckova, zari 2021
 *
 */

 /* systemove hlavickove soubory */
 /* zadne nejsou */

 /* vlastni hlavickove soubory */
 #include "obd_funkce.h" /* hlavickovy soubor vlastniho
     modulu */
```

```
 /* hlavickove soubory ostatnich modulu - zadne nejsou */

 /* def. globalnich promennych */
 /* zadne nejsou */

 /* lokalni def. symbolickych konstant a maker */
 /* zadne nejsou */

 /* lokalni def. novych typu */
 /* zadne nejsou */

 /* def. statickych globalnich promennych */
 /* zadne nejsou */

 /* uplne funkcni prototypy lokalnich funkci */
 /* zadne nejsou */

 /* funkce main */
 /* v tomto modulu neni */

 /* def. lokalnich funkci */
 /* zadne nejsou */

 /* def. globalnich funkci */
 double obvod(double a, double b){
    return (2 * (a + b));
 }

 double obsah(double a, double b){
    return (a * b);
 }
```

Příslušný hlavičkový soubor obsahuje následující kód.

```
/*
 * obd_funkce.h   verze 1.0
 *
 * Hlavickovy soubor pro modul obd_funkce.c
 * ========================================
 *
 * M. Trneckova, zari 2021
 *
 */

 /* podmineny preklad proti nasobnemu vkladani */
 #ifndef OBD_FUN
 #define OBD_FUN

 /* def. symbolickych konstant pouzitych v jinych modulech
```

```
     */
 /* zadne nejsou */

 /* def. maker s parametry */
 /* zadne nejsou */

 /* def. globalnich typu */
 /* zadne nejsou */

 /* def. globalnich promennych modulu obd_funkce.c */
 /* zadne nejsou */

 /* uplne funkcni prototypy globalnich funkci modulu
     obd_funkce.c */
 extern double obvod(double a, double b);
 extern double obsah(double a, double b);

 #endif /* OBD_FUN */
```

Hlavní zdrojový kód je následující.

```
/*
 * obd_main.c   verze 1.0
 *
 * Program pro praci s obdelniky
 * =============================
 *
 * M. Trneckova, zari 2021
 *
 */

 /* systemove hlavickove soubory */
 #include <stdio.h>

 /* vlastni hlavickove soubory */
 /* hlavickovy soubor vlastniho modulu - neni */
 #include "obd_funkce.h" /* hlavickove soubory ostatnich
     modulu */

 /* def. globalnich promennych */
 /* zadne nejsou */

 /* lokalni def. symbolickych konstant a maker */
 #define CHYBA -1.0
 #define kontrola(delka) (((delka) >= 0.0) ? (delka) : CHYBA
     )

 /* lokalni def. novych typu */
 /* zadne nejsou */
```

```c
/* def. statickych globalnich promennych */
static double a;
static double b;

/* uplne funkcni prototypy lokalnich funkci */
static double nacti();
static void vypis(double o, double s);

/* funkce main */
int main(){
   double o, s;
   a = nacti();
   b = nacti();

   if(a == CHYBA || b == CHYBA){
       /* Vypis chybove hlasky */
       return 1;
   }

   o = obvod(a, b);
   s = obsah(a, b);

   vypis(o, s);
   return 0;
}

/* def. lokalnich funkci */
static double nacti(){
   double vstup;
   /* nacteni strany od uzivatele */
   scanf("%lf", &vstup);
   return kontrola(vstup);
}

static void vypis(double o, double s){
   printf("\nObvod = %f, obsah = %f", o, s);
}

/* def. globalnich funkci */
/* zadne nejsou */
```

Kontrola vstupních dat by správně měla být už ve funkci, která data čte.

Shrnutí

V této kapitole jsme se zabývali problematikou dělení zdrojového kódu do více souborů. V kapitole je popsána doporučená struktura jednotlivých souborů.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co znamená pojem oddělený překlad?
2. Co by měl obsahovat hlavičkový soubor?

Cvičení

Úkol 80
Vytvořte dva soubory `soubor1.c` a `soubor2.c`. Oba budou sdílet proměnnou `a`. Proměnná je definovaná v `soubor1.c` a v `soubor2.c` se vypisuje.

Úkol 81
Změňte předchozí program tak, aby proměnná `a` byla definovaná jako globální `static`. Vyzkoušejte chování programu.

Úkol 82
Vytvořte program `funkce.c`, který bude obsahovat jen funkce s funkčními prototypy `float plus(float a, float b);` a `float minus(float a, float b);` počítající součet (rozdíl) dvou čísel. Dále vytvořte program `main.c`, který bude obsahovat pouze funkci `main()`, ve které budou volány funkce `plus()` a `minus()`.
Pomocí `#include` vložte soubor `funkce.c` do souboru `main.c` a přeložte.

Úkol 83
Vyzkoušejte funkci programu z předchozího cvičení, pokud nebude `funkce.c` do souboru `main.c` vkládaná, ale budou překládány samostatně.

1. Soubory odděleně překládejte a v souboru `main.c` uveďte úplné funkční prototypy funkcí `plus()` a `minus()`.
2. Vytvořte soubor `funkce.h`, do něhož vložte úplný funkční prototypy funkcí `plus()` a `minus()`. Pomocí `#include` zajistěte spojení souborů `funkce.c` a souboru `main.c`.

Bitové operace a bitová pole

Jazyk C je nízkoúrovňový programovací jazyk, který nabízí programátorům na rozdíl od vysokoúrovňových programovacích jazyků i operátory, pomocí kterých mohou nepřímo pracovat i s jednotlivými bity.

Bitové operace

Práce s jednotlivými bity většinou vyžaduje hlubší znalost operačního systému, například jakým stylem se ukládají čísla. Operace s jednotlivými bity však dokáže výrazně zrychlit program a některé operace ani pomocí dříve zmíněných konstrukcí není možné provést, jelikož pro ně byl nejmenší jednotkou informace byte.
V jazyce C máme k dispozici operátory *bitový součin*, odpovídající logické operaci AND, *bitový součet*, který je ekvivalentem operace OR, *bitový exkluzivní součet* (XOR), *bitový posun doleva*, *bitový posun doprava* a *jedničkový doplněk* (NOT). Pro všechny platí, že jejich argumenty mohou být pouze celá čísla (jak `signed` tak `unsigned`). Priorita bitových operátorů je nízká, ve složitějších výrazech je potřeba používat závorky.

Průvodce studiem
Logické operace jsou operace s výroky, jejichž výsledkem je opět výrok, jehož pravdivostní hodnota závisí na pravdivosti výroků a druhu operace.

Bitový součin (and bit po bitu)

Bitový součin má následující syntaxi.

```
x & y
```

`i`-tý bit výsledku bitového součinu `x & y` bude roven `1` pokud `i`-tý bit `x` i `y` budou rovny `1`, jinak bude výsledný bit roven `0`.
Výsledkem operace `100 & 50` bude `32`, jelikož platí následující vztah.

```
    100   0 1 1 0 0 1 0 0
     50   0 0 1 1 0 0 1 0
100 & 50  0 0 1 0 0 0 0 0
```

Bitový součin se často používá k výběru jen určitých bitů (nastavení ostatních bitů na 0). Například, pokud chceme zjistit, zda je číslo liché, stačí nám znát hodnotu nejméně významného bitu (0. bitu).[138]

[138] V textu předpokládáme, že nejméně významný bit je vpravo.

```
#define liche(x) (1 & (x))
```

Je třeba si uvědomit rozdíl mezi bitovým a logickým součinem.

```
int i = 1, j = 2, k, l;

k = i && j;
l = i & j;
```

Výsledky se budou lišit. Zatímco `k` je rovno `1`, protože obě `i` i `j` jsou nenulová, tak `l` bude rovno `0`. A to proto, že čísla mají následující binární vyjádření.

```
1  =  0 0 0 0 0 0 0 1
2  =  0 0 0 0 0 0 1 0
```

Bitový součet (or bit po bitu)

Bitový součet má následující syntaxi.

```
x | y
```

`i`-tý bit výsledku bitového součtu `x | y` bude roven `1` pokud `i`-tý bit `x` nebo `y` bude roven `1`. Budou-li oba nulové výsledný bit bude roven `0`. Často se používá k nastavení určitých bitů na `1`, s tím, že ostatní bity si ponechají původní hodnotu.
Následující příklad ukazuje použití bitového součtu pro převod velkého písmena na malé.

```
#define na_mala(c) (c | 0x20)
```

`0x20` odpovídá binárně `0 0 1 0 0 0 0 0`, což odpovídá hodnotě 32. Ze znalosti ASCII tabulky víme, že právě malá a velká písmena se v této hodnotě liší.

Bitový exkluzivní součet (exkluzivní or bit po bitu)

Bitový exkluzivní součet se zapisuje následujícím způsobem.

```
x ^ y
```

`i`-tý bit výsledku bitového exkluzivního součtu `x ^ y` bude roven `1`, pokud se `i`-tý bit `x` nerovná `i`-tému bitu `y`. Budou-li oba shodné (oba rovny `0`, nebo `1`) výsledný bit bude roven `0`.
Následující kód slouží k porovnání dvou čísel.

```
if(x ^ y)
    /* cisla jsou rozdilna */
```

Pokud jsou `x` a `y` rozdílná, pak alespoň jeden bit bude roven `1` a tedy výsledek bude nenulový (pravdivý).

Bitový posun doleva

Bitový posun doleva má následující zápis.

```
x << n
```

Posune bity v `x` o `n` pozic doleva. Při tomto posunu se bity zleva ztrácí a vpravo jsou doplňovány `0`. Typickým použitím je použití k násobení mocninou 2. `x = y << 1` vynásobí číslo 2.

```
1       =  00000001
1 << 1  =  00000010
```

`x = y << 3` vynásobí číslo `8` (2^3).
Takovéto násobení je rychlejší než běžné násobení. Než násobit číslo číslem `80`, je lepší ho vynásobit `64` (posun o 6 bitů) a `16` (posun o 4 bity) a výsledky sečíst. Viz následující příklad.

```
i = j * 80; /* pomalejsi */

i = (j << 6) + (j << 4); /* rychlejsi */
```

Bitový posun doprava

Bitový posun doprava má následující syntaxi.

```
x >> n
```

Posune bity v `x` o `n` pozic doprava. Při tomto posunu se bity zprava ztrácí a vlevo jsou doplňovány `0`.[139] Má opačný význam než bitový posun doleva. Často se používá k celočíselnému dělení mocninou 2. `x = y >> 1` vydělí číslo číslem 2, `x = y >> 3` vydělí číslo `8`. Stejně jako u bitového posunu doleva, takovéto dělení mocninou 2 je rychlejší, než klasické dělení.

[139] Toto tvrzení platí pro `unsigned` typy. Pro `signed` je výsledek implementačně závislý.

Jedničkový doplněk (negace bit po bitu)

Jedničkový doplněk se zapisuje následujícím způsobem.

```
~x
```

Nulové bity nahradí `1` a jedničkové `0`. Nulovány jsou i počáteční nuly, takže je výsledek závislý na typu čísla (kolik bitů obsahuje). Jiný výsledek bude pro `char` (8 bitů) a `int` (32 bitů).

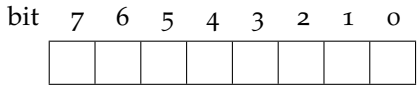
```
int main(){
    int i = 10, i2;
    unsigned char j = 10, j2;

    i2 = ~i;
    j2 = ~j;

    printf("%d %d \n", sizeof(int), sizeof(char));
    printf("%u %d", i2, j2);
}
```

Pro `char` dostaneme jako výsledek jedničkového doplňku čísla 10 číslo 245 a pro `int` číslo 4294967285.

Práce se skupinou bitů

Bitové operace se často používají pro práci se skupinou bitů (těm také říkáme *vlajky* nebo `flagy`), která přestavuje například stavové slovo definované následovně.

```
unsigned int status;
```



Definujeme si konstanty, které určí pozice příznakových bitů ve stavovém slově. Například použijeme `3.`, `4.` a `5.` bit pro příznaky číst, psát, vymazat.

```
#define READ 0x8
#define WRITE 0x10
#define DELETE 0x20
```

Pro nastavení všech příznaků na 1 použijeme následující příkaz.

```
status |= READ | WRITE | DELETE;
```

[140] Operátor | představuje bitový součet. Operátor |= je operátor přiřazení s bitovým součtem. Obdoba už známých aritmetických přiřazovacích operátorů +=, -= a pod.

Pokud chceme na 1 nastavit jen příznaky pro čtení a zápis, použijeme obdobný výraz, konkrétně následující výraz.

```
status |= READ | WRITE;
```

Pro nastavení všech příznaků na 0 použijeme bitový součin následujícím způsobem.

```
status &= ~(READ | WRITE | DELETE);
```

Chceme-li nastavit na 0 jen příznak pro čtení, použijeme výraz

```
status &= ~READ;
```

Pro testování, zda je nějaký příznak nastaven na 0 použijeme operátor bitového součinu. Konkrétně, pokud chceme zjistit, zda je příznak čtení roven 0, použijeme následující podmínku.

```
if (! (status & READ ))
```

Bitové pole

Speciálním případem struktur je *bitové pole*. Velikost této struktury je omezená velikostí typu `int`. Nejmenší délka jedné položky v této struktuře je jeden bit. Definuje se obdobně, jako se definují nám již známé struktury, pouze položky struktury nejsou určeny libovolným typem a jménem, ale jménem a délkou v bitech. Typy jednotlivých položek mohou být jen `unsigned` (neznaménkové) nebo `signed` (znaménkové) a za každým členem uvádíme za dvojtečkou počet bitů, které budou pro tento člen vyhrazeny.
Předchozí příklad řešený pomocí bitového pole je následující.[141]

[141] Položka `zacatek` je zde jen kvůli přeskočení prvních 3 bitů.

```
typedef struct{
    unsigned zacatek : 3; // bity od 0 do 2
    unsigned read : 1; // bit 3
    unsigned write : 1; // bit 4
    unsigned delete : 1; // bit 5
} STAV;


STAV status;
```

S jednotlivými bity se pracuje pomocí tečkové notace, jako je tomu u struktur. Konkrétně pro nastavení příznaku pro čtení na `1` použijeme příkaz `status.read = 1;`, pro nastavení příznaku pro zápis na `0` příkaz `status.write = 0;` a podobně. Pro test, zda je příznak pro čtení nastaven na `1` použijeme podmínku ve tvaru `if(status.read)`.

Shrnutí

V této kapitole jsme si ukázali, jak můžeme v jazyce C pracovat s jednotlivými bity pomocí bitových operací. Také jsme si zavedli pojem bitové pole.

Kontrolní otázky
Odpovězte na následující otázky:

1. K čemu můžeme využít bitový posun?
2. Jak se bitové pole liší od klasického pole?

Cvičení

Úkol 84
Z příkladu v sekci o bitovém součtu víme, že se malé písmeno a odpovídající velké liší v `5.` bitu. Napište makro, které převede malé písmeno na velké.

Úkol 85
Napište funkci, která pro zadané číslo `cislo` a číslo `n` zjistí hodnotu `n`-tého bitu čísla `cislo`. Například hodnota `1.` bitu čísla `3` je `1`, `2.` bit má hodnotu `0`. Bude se vám hodit operace bitového posunu.

Úkol 86
Vyzkoušejte si práci se stavovým slovem. Vytvořte funkci, která bude brát jako vstup celé číslo `n` a `stav` a vypíše všechna čísla od `0` do `n` a bude vynechávat násobky podle `stav`. `stav` bude uchovávat informaci, zda se mají vypisovat násobky `2`, `3` a `5`. Například ve `stav` je nastaven příznak, že se nemají vypisovat násobky `2`, pak funkce vypíše jen lichá čísla do `n`.

Úkol 87
Upravte předchozí kód tak, aby stav nastavoval uživatel po dotazu v konzoli.

Úkol 88
Upravte předchozí příklad tak, aby `stav` byl definován jako bitové pole.

Úkol 89
Napište funkci, která způsobí rotaci (ne posun) čísla `x` o `n` bitů doprava. Pro číslo `char x = 3` (`00000011`) a `n = 3` dostaneme `96` (`01100000`).

Úkol 90
Napište program, který bude využívat bitového pole pro úschovu času (hodiny, minuty, vteřiny). Vytvořte i funkce na převod času do bitového pole a funkci, která vypíše bitové pole jako čas.

Ladění

Tato kapitola je věnována chybám programu. Ukážeme jaké chyby se v programech mohou vyskytovat, a také, jakým způsobem chyby odhalovat.

Chyby

Než si řekneme, jakým způsobem chyby dělíme, zopakujeme pojem sémantiky a syntaxe jazyka. *Syntaxe jazyka* popisuje pravidla pro zápis programu. *Sémantika* popisuje chování programu, tedy to, co program dělá. Obecně lze chyby rozdělit na *chyby syntaktické* a *sémantické chyby*.

Syntaktické chyby

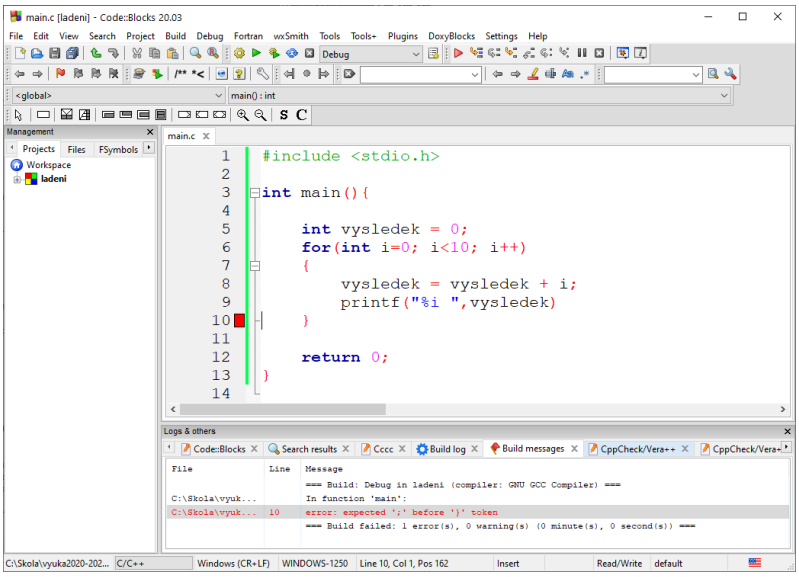
Syntaktické chyby jsou způsobeny nesprávným zápisem programu. Například častou syntaktickou chybou je chybějící středník na konci řádku. Tyto chyby jsou nalezeny překladačem už při překladu a při jejich výskytu překladač není schopný sestavit spustitelný soubor a vypíše seznam chyb. Většinou je popsána chyba a řádek, na kterém tato chyba vznikla. Syntaktické chyby je snadné nalézt a opravit.
Kromě již zmíněného chybějícího středníku jsou to často překlepy v názvech identifikátorů či chybějící nebo přebývající závorka.
Na obrázku 9 vidíme, že se ve vývojovém prostředí při překladu vyskytla chyba. Ta je vypsaná v konzoli ve spodní části obrazovky. Vidíme, že na řádku 10 je očekáván znak `;` před znakem `}`.[142]

Průvodce studiem
Pro velmi detailní analýzu zdrojového kódu existuje v UNIX samostatný program *lint*. Programy typu lint (např. splint a jiné) kontrolují nejen syntaxi, ale částečně i sémantiku.

[142] Na řádku 9 nebyl příkaz `printf()` ukončen `;`.

Sémantické chyby

Záludnější chyby jsou chyby sémantické. Ty je možné odhalit až za běhu programu (někdy se přímo označují jako chyby za běhu programu, nebo run-time chyby).
Sémantické chyby ještě můžeme dále dělit na ty, které jsou způsobeny *nesprávným použitím jazyka* a na ty, co jsou způsobené *špatným zdrojovým kódem*.



Obrázek 9: Syntaktická chyba.

První skupina chyb vzniká například při dělení nulou či špatné indexaci polí (přístup k prvkům mimo rozsah pole) nebo při práci se špatnými typy (například výsledek dělení dvou celých čísel je celé číslo a my očekáváme desetinné). Také často vznikají při špatné práci s pamětí, jako je neuvolňování paměti (program používá více paměti, než potřebuje), přístup ke špatné části paměti a tak dále.
Pokud chyba není způsobena tím, jak je program zapsán, ale přesto program nedělá to, co bychom chtěli, tak se bavíme o druhé skupině sémantických chyb. Na vině je algoritmus, který je špatně zapsán (špatně převeden do zdrojového kódu) nebo je chyba v samotném algoritmu.
Hlavní je v tomto případě identifikovat místo, kde chyba vzniká. Je nutné zjistit, co se za běhu programu děje. K tomu potřebujeme najít poslední příkaz, který se vykonal než nastala chyba (kde se v programu nacházíme), a také hodnoty jednotlivých proměnných. Jedním z přístupů, jak sémantickou chybu nalézt je pomocí *analýzy kódu*. Jinak řečeno, opakovaně čteme kód a přemýšlíme nad tím, jak tento kód pracuje (co se stane při vykonání tohoto příkazu, jak se změní hodnoty proměnných a podobně).
Dalším způsobem je pozorování programu za běhu. Nejjednodušší způsob je přidat do kódu (na vhodná místa) výpis hodnot proměnných, které nás zajímají a zkontrolovat, zda mají tyto proměnné hodnoty, které očekáváme.

```c
#include <stdio.h>

int main(){
    int a, b, r;
    printf("Zadejte cisla a a b: ");
    scanf("%i %i",&a, &b);
    printf("Vypocet gcd pro %i a %i\n", a, b);
    while(b != 0){
        r = a % b;
        a = b;
        b = r;
        printf("a: %i, b: %i\n", a, b);
    }
    printf("Nejvetsi spolecny delitel je %i", a);
    return 0;
}
```

Například v předchozím kódu vypisujeme hodnoty proměnných `a` a `b` v každé iteraci cyklu `while`.
Případně můžeme využít nástrojů IDE. Konkrétně se jedná o tzv. *debugger*.[143]

[143] Debuggery existují i mimo IDE.

Ladění pomocí debuggeru

Debugger dokáže zastavit běh programu na programátorem zvoleném místě (tomuto místu se říká *breakpoint*). V této době (když je program zastaven) je možné si prohlédnout hodnoty jednotlivých proměnných (a také seznam všech neukončených volání funkcí). Dále je možné postupovat po jednotlivých krocích[144] (v každém kroku se provede další příkaz) nebo pokračovat k následujícímu breakpointu.

[144] Krokování programu.

Další nástroje pro ladění

V některých případech není použití debuggerů dostačující. Například při ladění dynamických procesů se chyba nemusí projevit při každém spuštění programu. Proto je vhodné kromě debuggerů přidávat do kódu i další ladící prostředky, jako jsou například ladící výstupy na `stderr`, využití podmíněného překladu, *self-test* moduly nebo funkce,[145] využití knihovny `<assert.h>` a jiné.

[145] Tento pojem vysvětlíme dále.

Ladící výpisy na `stderr`

Již dříve jsme si představili standardní výstup `stderr`, který se používá pro výpis chyb. Do tohoto proudu můžeme zapisovat například pomocí příkazu `fprintf(stderr, text, ...)`. Ve standard-

ním nastavení se vypíše `text` do konzole. Tento proud můžeme zavřít pomocí `fclose(strderr)` a tím potlačit veškeré jeho výpisy.[146] Toto není úplně čisté řešení. Lepší je výpis přesměrovat do ladícího souboru pomocí funkce `freopen()`. Viz následující kód.

[146] Výpisy ponecháme jen v době, kdy ladíme program.

```
fw = freopen("ladeni.log", "w", stderr);
```

Oba tyto přístupy nám umožňují ovlivnit, co vše se bude vypisovat na obrazovku. Výpisy však v kódu zůstávají a zpomalují program.

Využití podmíněného překladu

Podmíněný překlad nám dává možnost zahrnout do kódu části, které můžeme díky preprocesoru vynechat, když je nepotřebujeme. Přebytečné části kódu zůstávají ve zdrojovém souboru, ale nejsou ve výstupu programu.
Nejjednodušším způsobem je použití podmíněného překladu využívajícího existence symbolické konstanty následujícím způsobem.

```
#ifdef DEBUG /* zacatek podmineneho prekladu */

/* ladici cast */

#endif /* konec podmineneho prekladu */
```

Pokud není symbolická konstanta `DEBUG` definovaná,[147] pak je ladící část kódu preprocesorem odstraněna.
Kromě tohoto přístupu, můžeme podmíněný překlad použít i tak, že nás bude zajímat hodnota konstanty `DEBUG` a podle ní můžeme ovlivnit, jak bude výstup vypadat. Viz následující příklad.

[147] Pro připomenutí symbolickou konstantu definujeme následujícím způsobem.
`#define DEBUG`
Na hodnotě této konstanty v tomto případě nezáleží. Jen na tom, zda je, nebo není definovaná.

```
#if defined(DEBUG) && DEBUG == 1 /* zacatek podmineneho
    prekladu */

/* ladici cast 1 */

#elif defined(DEBUG) && DEBUG == 2

/* ladici cast 2 */

...
#endif /* konec podmineneho prekladu */
```

Nevýhodou těchto přístupů je to, že je potřeba změnit kód,[148] pokud chceme ladící část vynechat nebo do kódu zahrnout a zdrojový kód znovu přeložit.

[148] Definovat symbolickou konstantu, nebo její definici odstranit.

Self-testy modulů nebo funkcí

Využití tzv. self-testů (samotestovací mechanismus) využíváme zejména při odděleném překladu. Tyto self-testy obsahují jednotlivé moduly a je tedy možné je otestovat nezávisle na ostatních modulech. Jejich zapínání a vypínání je realizované pomocí podmíněného překladu.
V každém modulu je funkce `main()` s parametry, aby se dala spouštět z příkazové řádky s testovacími hodnotami. V následujícím příkladu máme pomocný modul, ve kterém definujeme funkci s funkčním prototypem `int funkceA(int a)`. Funkce `main()` je do kódu vložena jen v případě, že je definovaná symbolická konstanta `SELF`.

```
/* funkce.c
 * Modul obsahujici definici funkce funkceA
 */

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

#ifdef SELF

int funkceA(int a);

int main(int argc, char *argv[]){
    if(argc == 1){
        fprintf(stderr, "Nepredany zadny argument.");
        return 1;
    }
    printf("Vysledek = %d", funkceA(atoi(argv[1])));
    return 0;
}

#endif /* konec SELF */

int funkceA(int a){

    /* Definice funkce*/

    return vysledek;
}
```

Využití knihovny `assert.h`

Jak již víme, v programu se mohou objevit chyby, které se projeví už při překladu, a také chyby, které se objeví až za běhu programu. Takové chyby se ale nemusí vyskytnout při každém spuštění pro-

gramu, ale objeví se jen někdy. Jejich hledání a odstranění je velmi časově náročné. Dají se hledat za použití podmíněného překladu testovacích částí.
Dalším způsobem je využít principu aserce. Aserce je skupina predikátů o nichž se domníváme, že budou vždy pravdivé. Tyto predikáty můžeme použít k detekci stavů, které by za žádných okolností v rámci programu nastat. Knihovna `assert.h` obsahuje nástroje, které implementují aserci. Zejména se využívá při práci s poli, řetězci a ukazateli, kde vznikají chyby typu špatného přístupu do paměti.[149]
Například příkaz `pole[-2] = 10;` je nejspíš chybný, ale překladači nevadí.[150] Program obsahující tento příkaz může spadnout nebo „jen“ přepsat část paměti mimo pole.
Princip je ten, že na konkrétním místě v kódu vyhodnotí logický výraz. Pokud je pravdivý, nic se neděje a program běží dál. Pokud se výraz vyhodnotí na nepravdu, provede se chybový výpis a ukončí se běh programu. Použití je ukázáno v následujícím příkladu.

[149] U polí jsou to například chyby zápisu do paměti za polem (index o jedna větší). Můžeme se setkat s pojmem *off-by-one-error*.

[150] Viz pointerová aritmetika.

Průvodce studiem
Aserce jsou predikáty, o nichž se domníváme, že budou na daném místě vždy pravdivé. Nejedná se o ošetření očekávaných chybných stavů. Jedná se pouze o detekci stavů, které by za žádných okolností v rámci programu neměly nastat.

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

#define VELIKOST 10

int main(){
    int pole[VELIKOST];
    int cislo;
    int index;

    /* Nejaky kod   */

    assert(index >=0 && index < VELIKOST);
    pole[index] = cislo;

    return 0;

}
```

V příkladu je zřejmé, že v programu může dojít k tomu, že proměnná `index` nabude hodnot mimo meze pole `pole`. Překladač by na tuto chybu neupozornil, ale `assert()` ano.
Není přesně definováno, kdy by se měla konstrukce `assert()` použít, ale obecně platí, že by se měla vložit do míst, kde víme, že některé hodnoty jsou chybné, ale velmi nepravděpodobné a proto není úplně vhodné v těchto částech kódu kvůli jeho přehlednosti používat testování hodnoty pomocí `if`.
Testy `assert()` lze při překladu vypustit pomocí mechanismu podmíněného překladu. K jejich odstranění stačí definovat symbolickou

konstantu `NDEBUG`.[151] Po její definici nebudou do kódu zahrnuty žádné příkazy `assert()`. Zrušením definice konstanty `NDEBUG` je opět do kódu zahrneme.

[151] `#define NDEBUG`

Shrnutí

V této kapitole jsme se zabývali chybami v programu, zejména jejich hledání.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jaký je rozdíl mezi syntaktickými a sémantickými chybami?
2. Co je to ladění?
3. Jak můžeme pro ladění využít podmíněný překlad?

Cvičení

Úkol 91
Určete, zda se v následujících kódech vyskytuje chyba a pokud ano, jakého je typu, případně, jak bychom jí opravili.

1.

```c
#include <stdio.h>

int main(){
    int x = 10;
    int y = 20;

    printf("(%i, %i)", x, y)
    return 0;
}
```

2. Následující kód vypíše 10. Proč?

```c
#include <stdio.h>

int main(){
    int i;

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

3.

```c
#include <stdio.h>

int main(){
    int i;

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

4.

```c
#include <stdio.h>

int main()
{
    int pole[3]={1, 2, 3};

    printf("%i ", pole[3]);
    return 0;

}
```

5.

```c
#include <stdio.h>

int main(){
    int b;

    while(b > 0){
        printf("*");
        b--;
    }
    return 0;
}
```

6.

```c
#include <stdio.h>

int main(){
    int n = 10;
    int m = 0;

    m = n / 0;

    printf("%i ", m);
    return 0;
}
```

7.

```c
#include <stdio.h>

int main()
{
    int a=0;

    if(a=0){
        printf("a je 0");
    }
    else{
        printf("a neni 0");
    }
    return 0;
}
```

8.

```c
#include <stdio.h>

int main(){
    int a = 10, b = 20, c;
    a + b = c;
    return 0;
}
```

9.

```c
#include <stdio.h>
int i;

void vykresli_trouhelnik(int n){
    for(i = 1; i <= n; i++){
        for(int j = 0; j < i; j++){
            printf("*");
        }
        printf("\n");
    }
}

int main(){
    int n = 5;
    for(i = 1; i <= n; i++){
       vykresli_trouhelnik(i);
    }
    return 0;
}
```

10.

```c
#include <stdio.h>

typedef struct{
    int x;
    int y;
    int pole[2];
} struktura;

int main(){
    struktura m;
    m = {10, 2, {10, 2}};
    return 0;
}
```

Úkol 92
Následující kód skončí chybou při překladu. Proč tomu tak je?

```
#include <stdio.h>

int main(){
    char *karty = "JQK";
    char karta = karty[2];

    karty[2] = karty[1];
    karty[1] = karty[0];
    karty[0] = karty[2];
    karty[2] = karty[1];
    karty[1] = karta;

    printf("%s", karty);
    return 0;
}
```

Úkol 93
Určete, které z následujících kódů se podaří zkompilovat. V případě neúspěchu jaké se v kódu objevují chyby? V případě úspěchu určete, jaký bude výstup a zkuste odhadnout, zda se program chová tak, jak by měl.

1.

```
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1)
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    else
        printf("Eso");

    return 0;
}
```

2.

```c
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1){
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    else
        printf("Eso");
    }

    return 0;
}
```

3.

```c
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1){
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    }else
        printf("Eso");

    return 0;
}
```

4.

```c
#include <stdio.h>

int main(){
    int karta = 1;

    if(karta > 1){
        karta = karta - 1;
        if(karta < 7)
            printf("Mala karta");
    else
        printf("Eso");

    return 0;
}
```

Úkol 94
Vyzkoušejte si práci s `assert()`.

Příklady k procvičení

Úkol 95
Napište program, který k zadané ceně připočítá 25% daň a vypíše novou cenu.

Úkol 96
Napište program, který z konzole přečte tři malá písmena a vypíše je jako velká v obráceném pořadí. Pro znaky `'a' 'b' 'c'` vypíše `'C' 'B' 'A'`.

Úkol 97
Napište program, který vypíše maximální číslo, které je možné uložit do `unsigned int` a do `signed int`.[152]

[152] `-1` jako `signed int` je maximální `unsigned int` a maximální `signed int` je polovina maximálního `unsigned int`.

Úkol 98
Jaký bude výstup následujícího kódu? Vyzkoušejte vaší domněnku a zdůvodněte výsledky.

```
i = 5;
printf("%d\n", i == 8);
printf("%d\n", i = 8);
printf("%d\n", i == 8);
```

Úkol 99
Napište funkci, která pro vstupní řetězec, který obsahuje binární číslo vrátí toto číslo v desítkové soustavě.

Úkol 100
Napište funkci, která pro zadané číslo v desítkové soustavě vypíše toto číslo v binární podobě.

Úkol 101
Vytvořte program, který pro zadaný řetězec, který obsahuje matematický výraz obsahující celá čísla a základní aritmetické operace `+`, `-`, `*`, `/`, vypíše výsledek tohoto výrazu.
Například pro takto definovaný program

```c
#include <stdio.h>

int main()
{

    char retezec[] = "10+2*3";


    /* TO DO */

    return 0;
}
```

bude výsledek roven 16.[153]

[153] Pro jednoduchost budeme předpokládat, že výraz je ve správném formátu (neobsahuje nepovolené znaky, mezi dvěma operátory je číslo, výraz začíná a končí číslem...) a není tedy potřeba ověřovat jeho správnost. Také není potřeba ověřovat dělení nulou.

Úkol 102
Napište funkci s hlavičkou `void transformace(int pole[], int delka)`, která změní prvky pole podle následujících pravidel:

- Pokud je prvek dělitelný `4` a zároveň je buď menší než `50` nebo větší než `65`, vynásobte `20` jeho zbytek po dělení číslem `3`.[154]
- Pokud je index prvku v poli dělitelný `2` ale není dělitelný `4`, vynásobte prvek počtem cifer tohoto čísla.
- Pokud je číslo větší než `100` nahraďte ho číslem s číslicemi v opačném pořadí.[155]
- Ostatní prvky nechte beze změny.

Co bude výsledkem pro pole s prvky `{62, 60, 20, 32, 68, 842, 31, 12}`? Vypište prvky jako znaky.

[154] Pro `16` je výsledek `20`.

[155] Pro prvek `123` bude výsledkem `321`.

Úkol 103
Napište program, který pro zadané `n` vypíše čísla od `1` do `n` s tím, že místo čísel dělitelných `3` vypíše `TIK`, místo čísel dělitelných `5` vypíše `TAK`. Pokud je číslo dělitelné jak `3` i `5` vypíše `TIKTAK`.

Úkol 104
Napište funkci, které zadáte počet `1` korun, `2` korun a `5` korun a hodnotu. Funkce vrátí odpověď, zda je možné ze zadaných mincí sestavit určenou hodnotu.[156]

[156] Hlavička funkce
`int platba(int pocet_1, int pocet_2, int pocet_5, int hodnota);`

Úkol 105
Napište funkci, která pro zadaný řetězec vypíše všechna slova z řetězce začínající písmenem 'a'. Slova jsou oddělena mezerou.

Úkol 106
Upravte předchozí funkci tak, že kromě řetězce bere jako vstup i začínající znak a vypíše všechna slova z řetězce začínající zadaným znakem.

Úkol 107
Upravte předchozí funkci tak, že jako vstup bere dva řetězce a vypíše všechna slova z `1`. řetězce začínající `2`. řetězcem.
Př. pro řetězce `''bazen balon bonbon trouba''` a `''ba''` vypíše `''bazen''` a `''balon''`.

Úkol 108
Naprogramujte funkci, která pro zadané `n` vrátí n-tý prvek posloupnosti, která je zadána rekurentním vztahem: $a_1 = 14688$, $a_{n+1} = \frac{1}{2}a_n + 1200$.[157]

[157] 10. člen je roven 2424.

Úkol 109
Naprogramujte funkci `void vypis(int *pole, int zacatek, int krok, int konec)`, která vypíše prvky pole od indexu zacatek po index konec s krokem krok. Například pro `pole = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}` a `zacatek = 0`, `krok = 2`, `konec = 9` vypíše prvky `1`, `3`, `5`, `7`, `9`.

Úkol 110
Naprogramujte funkci, která jako vstup bere 2 celočíselné kladné argumenty `m` a `n` větší rovny 2 a pracuje podle následujícího pseudokódu:

1. Vypiš n-2 mezer, pak řetězec ''`(\\o/)`''
2. Opakuj m krát:
    - Na nový řádek vypiš n teček, velké `X` a n teček.
3. Na nový řádek vypiš 2*n + 1 krát `X`.
4. Opakuj krok 2.

Úkol 111
Napište funkci, která pro 2 zadaná čísla vrátí, zda je možné udělat jejich podíl a pokud ano, vrátí i jejich podíl.

Úkol 112
Napište funkci, která bere 2 argumenty (`text` a `podretezec`). Funkce v daném textovém řetězci `text` vyhledá první výskyt zadaného podřetězce `podretezec`. Funkce vrací ukazatel na první znak nalezeného podřetězce nebo konstantu `NULL`, pokud podřetězec `podretezec` nebyl nalezen.

Úkol 113
Pomocí dvourozměrného pole lze reprezentovat hrací pole při piškvorkách (prázdné políčko = 0, křížek = 1, kolečko = 2). Napište funkci, která prohledá toto dvourozměrné pole a vrátí nejdelší souvislou posloupnost křížků nebo koleček

1. na řádku
2. ve sloupci
3. diagonálně

Naprogramujte hru piškvorky pro dva hráče.[158]

[158] Dokud v poli nebude posloupnost 5 stejných znaků (využijte předchozí funkci) se budou hráči střídat a umisťovat svůj znak do pole (na střídačku budou hráči vyzváni, aby zadali 2 souřadnice, kam chtějí umístit znak).

Úkol 114

V následujícím kódu si vytvoříme pole, do kterého budeme ukládat celá čísla. Pole se bude samo zvětšovat, aby se do něj všechna čísla vešla. Dále si v proměnné `hlava` budeme uchovávat informaci a počtu prvků uložených v poli.
Čísla budeme ukládat od začátku (1. bude na indexu číslo 0). `hlava` bude indexem na 1. prázdné políčko (na tento index budeme přidávat další prvek, po přidání se toto číslo zvětší).
Dále je potřeba si uchovávat informaci o velikosti alokované paměti (`velikost`). Pokud je `hlava == velikost`, musíme velikost pole zvětšit pomocí funkce `realloc()`.
Základem je následující kód:

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

int *data;
int velikost;
int hlava;

void init(int);
void uvolni();
void pridej(int);

int main()
{
    int i=7;
    init(4);
    pridej(i);
    uvolni();
    return 0;
}
```

1. Doprogramujte funkci `init()`, která inicializuje pole `data` na předaný počet prvků (alokuje pro ně paměť), funkci `pridej()` pro přidání prvku do pole a funkci `uvolni()`, která na konci programu uvolní alokovanou paměť.
2. Doprogramujte funkci na vypsání prvků v poli data. Zavolejte tuto funkci po každém přidání prvku do pole.
3. Nahraďte použití globálních proměnných pomocí strukturovaného typu.
4. Doprogramujte funkci na odebrání posledního prvku v poli `data`.
5. Doprogramujte funkci, která zjistí zda se číslo předané jí jako argument nachází v datové struktuře.
6. Doprogramujte funkci, která smaže prvek předaný jí jako argument z datové struktury.
7. Upravte funkci mazání prvku tak, že pokud je počet prvků ve struktuře menší než polovina jeho velikosti, zmenší paměť alokovanou pro pole data na polovinu.

Úkol 115
Vytvořme si lineární seznam obsahující celá čísla, seznam reprezentujeme následují strukturou.

```
typedef struct _node{
    int data;
    struct _node *next;
} node;
```

Každému uzlu v seznamu bude odpovídat jedna struktura `node`, jejíž položka `data` bude obsahovat dané číslo. Celý seznam si budeme v programu pamatovat jako pointer na první uzel seznamu. Funkce pro přidání prvku do seznamu vypadá následovně.

```
node *add(node **list, int data)
{
    node *new = malloc(sizeof(node));
    new->data = data;
    new->next = *list;
    *list = new;
    return new;
}
```

Napište funkci, která:

1. vypíše všechny prvky seznamu,
2. zjistí délku seznamu,
3. přidá prvek na konec seznamu,
4. smaže prvek na začátku seznamu,
5. smaže prvek na konci seznamu,
6. pro zadané `i` vypíše `i`-tý prvek seznamu,
7. pro zadané `i` vypíše `i`-tý prvek seznamu od konce,
8. pro zadané `i` smaže `i`-tý prvek seznamu,
9. vytvoří kopii seznamu. Funkce musí fungovat tak, že pokud změníme kopii seznamu, originální seznam se nezmění.

Úkol 116
Napište funkci `komplexni suma(int pocet, ...)`, která vypočítá součet předaných komplexních čísel. Počet sčítaných čísel je určen pevným parametrem `pocet`, za nímž pak ve volání funkce následují hodnoty, které má funkce sčítat. Pro práci s komplexními čísly je nutné vytvořit strukturovaný datový typ `komplexni`.

Úkol 117
Napište program `vyraz` vyhodnocující výpočty zapsané v obrácené polské notaci a zadané z příkazové řádky, kde každý operand nebo operátor je samostatným argumentem. Například:
`vyraz.exe 2 3 4 + *`
vypočítá výraz $2 \cdot (3 + 4)$

Úkol 118
Napište funkci `double akumulator(double (*fce)(double, double), double cisla[], int pocet)` , která zpracuje pomocí předané funkce `fce` hodnoty z pole `cisla`, jehož velikost je dána parametrem `pocet`. Vytvořenou funkci otestujte ve funkci `main()`. Použitými akumulačními funkcemi mohou být například funkce pro součet nebo součin dvou reálných čísel, které je ovšem pro testování potřeba dodefinovat. Detaily najdete zde: `http://jazykc.inf.upol.cz/ukazatele-na-funkce/akumulator.htm`

Úkol 119
Definujte strukturu se třemi prvky. První prvek bude typu `float`, druhý `char` a třetí `int`. Dále definujte union se stejnými prvky. Zjistěte adresy struktury a unionu a všech jejich položek. Dále zjistěte velikost struktury a unionu.

Úkol 120
Naprogramujte funkci `prevod_cisla`, která provádí převod čísel mezi dvěma obecnými pozičními soustavami. Tedy vstupem je zápis čísla `X` v soustavě o základu `r` a informace o cílovém základu `s`. Například pro vstup `cislo = 25`, `zaklad = 7`, `cil = 3` je výsledek roven `201`. Pro jednoduchost počítejte se základy od `2` do `20`.

Úkol 121
Naprogramujte funkci, která pro zadanou množinu (pole) prvků vypíše všechny jeho permutace.[159]

[159] Nápověda: K řešení tohoto úkolu je možné použít rekurzi. Například všechny permutace `n` prvkové množiny dostaneme tak, že vezmeme jeden z prvků na první místo a pak zavoláme generování všech permutací na zbývajících `n - 1` prvků. Toto opakujeme pro všechny prvky.

Úkol 122
Naprogramujte funkci, která pro zadanou množinu (pole) prvků a `k` vypíše všechny jeho `k` prvkové variace.

Úkol 123
Naprogramujte funkci, která pro zadanou množinu (pole) prvků a `k` vypíše všechny jeho `k` prvkové variace s opakováním.

Úkol 124
Napište funkci, která pro zadané `n` vykreslí `n` vnořených čtverců. Například pro `n = 4` dostaneme následující výpis.

```
* * * * * * * * * * * * * *
*                         *
*   * * * * * * * * * *   *
*   *                 *   *
*   *   * * * * * *   *   *
*   *   *         *   *   *
*   *   *   * *   *   *   *
*   *   *         *   *   *
*   *   * * * * * *   *   *
*   *                 *   *
*   * * * * * * * * * *   *
*                         *
* * * * * * * * * * * * * *
```

Úkol 125
Naprogramujte funkci, která předaný řetězec zašifruje tak, že každou `n`-tici (`n` je také argumentem funkce) zapíše pozpátku. Například pro řetězec „HESLOJEINFORMATIKA“ a `n = 3` dostaneme „SEHJOLNIEROFTAMAKI“.

Úkol 126
Naprogramujte funkci, která předaný řetězec zašifruje tak, že první znak bude prvním znakem ve výsledném řetězci, druhý bude posledním, třetí druhý od začátku, čtvrtý druhý od konce atd. Například pro řetězec „HESLOJEINFORMATIKA“ dostaneme „HSOENOMTKAIARFIJLE“.

Úkol 127
Napište program, který bude využívat bitového pole pro úschovu data (den, měsíc, rok).[160] Vytvořte i funkce na převod data do bitového pole a funkci, která vypíše bitové pole jako datum.

[160] Rok bude představovat rok, ke kterému se přičte konstanta `1980` (tu definujte pomocí makra). Rok `15` tedy znamená rok `1995`.

Úkol 128
Napište funkci `invert` s celočíselnými argumenty `x`, `n` a `p`, která invertuje (zamění `1` za `0` a naopak) `n` bitů čísla `x` od pozice `p` včetně. Ostatní bity zůstanou nezměněny.

Úkol 129
Naprogramujte funkci, která je deklarovaná následovně `clovek* nacti_csv(char *, int *);`. První argument představuje adresu csv souboru, který má funkce načíst do pole strukturovaného datového typu `clovek`. Druhý vstupní argument představuje ukazatel na proměnnou, do které funkce uloží počet načtených prvků ze souboru.
csv je jednoduchý textový formát, kde každý řádek představuje jeden záznam. Jednotlivé položky v záznamu jsou odděleny `;` (na konci řádku středník není). V našem případě každý záznam obsahuje 3 položky - jméno, příjmení a věk.
Je potřeba vytvořit strukturovaný datový typ `clovek`, který bude obsahovat 3 položky (`jmeno`, `prijmeni`, `vek`). Předpokládejme, že maximální délka jména i příjmení je předem známá (například 20). Vytvořte pro tato čísla konstanty.

Řešení vybraných úkolů

Dodejme, že mohou existovat i jiná řešení, ne nutně horší, než ta, která jsou zde uvedena.

Řešení úkolu 8:

```c
#include <stdio.h>

int main(){
    float vysledek;
    vysledek = (3.0 / 2) + 12 + ((5 * 5) - (2 * 2)) / 6.0;
    printf("%f\n", vysledek);
    return 0;
}
```

Řešení úkolu 11:

```c
#include <stdio.h>

int main(){
    int cislo, prvni, treti;

    printf("Zadejte trojciferne cislo: ");

    /* Nacteni cisla */
    scanf("%i", &cislo);

    /* Vypocet prvni cislice */
    prvni = cislo / 100;

    /* Vypocet treti cislice */
    treti = cislo % 10;

    printf("Prvni cislice cisla %i je %i, treti cislice je %i", cislo, prvni, treti);
    return 0;
}
```

Řešení úkolu 14:
Kód vypíše „`neni rovno 0`“, protože v podmínce je místo operátoru porovnání operátor přiřazení. Výsledkem aplikace operátoru přiřazení je hodnota `r_value`, která je v tomto případě rovna 0, což je považováno za nepravdu.

Řešení úkolu 17:

```c
#include <stdio.h>

int main(){
    int cislo = -10;

    printf("|%i| = %i ", cislo, (cislo < 0) ? -cislo : cislo);
    return 0;
}
```

Řešení úkolu 22:

```c
#include <stdio.h>

int main(){
    int cislo = 1234;
    int vysledek = 0;

    while(cislo > 0){
        vysledek = vysledek * 10 + cislo % 10;
        cislo /= 10;
    }
    printf("%i", vysledek);
    return 0;
}
```

Řešení úkolu 24:

```c
#include <stdio.h>

int main(){
    int n = 8;

    switch(n){
        case 10:
            printf("10 ");
        case 9:
            printf("9 ");
        case 8:
            printf("8 ");
        case 7:
            printf("7 ");
        case 6:
            printf("6 ");
        case 5:
            printf("5 ");
        case 4:
            printf("4 ");
        case 3:
            printf("3 ");
        case 2:
            printf("2 ");
        case 1:
            printf("1 ");
    }
    return 0;
}
```

Řešení úkolu 25:

`pole[2] = 10;` - správně

`pole[3] = pole[1] + pole[2];` - správně

`pole[1] = pole[5];` - pole má pouze 5 prvků, tedy neexistuje `pole[5]`

`pole[7] = 10;` - pole má pouze 5 prvků, tedy neexistuje `pole[7]`

Řešení úkolu 34:

```
int pole[] = {1 2 3 4 5 6 7 8 9 10};

/* adresa 4. prvku */
pole + 3;
&pole[3];

/* index prvku na ktery ukazuje ptr */
ptr - pole;

/* porovnani ukazatelu ptr1 a ptr2 */
ptr1 < ptr2 /* prvek na ktery ukazuje ptr1 je driv */
ptr1 == ptr2 /* ukazuji na stejny prvek */
ptr1 > ptr2 /* prvek na ktery ukazuje ptr2 je driv */
```

Řešení úkolu 35:

```
#include <stdio.h>

int main(){
    char *c;
    int *i;
    double *d;

    printf("sizeof(char) = %i\n", ((int)(c + 1) - (int)c));
    printf("sizeof(int) = %i\n", ((int)(i + 1) - (int)i));
    printf("sizeof(double) = %i\n", ((int)(d + 1) - (int)d));

    return 0;
}
```

Řešení úkolu 47:

```
/* funkce pro vypocet faktorialu */
int faktorial(int n){
    if(n == 1)
        return 1;
    return n * faktorial(n - 1);
}

/* funkce, ktera secte cisla od 1 do n */
int sum_n(int n){
    int i, suma = 0;

    for(i = 1; i <= n; i++)
        suma += i;

    return suma;
}

/* funkce pro vypocet
   1! + (1 + 2)! + (1 + 2 + 3)! + ... + (1 + 2 + ... + n)!*/
int suma(int n){
    int i, vysledek = 0;

    for(i = 1; i <= n; i++)
        vysledek += faktorial(sum_n(i));

    return vysledek;
}
```

Řešení úkolu 54:

```
int main(int argc, char* argv[]){
    int i, soucet = 0;

    for(; i < argc; i++){
        soucet += atoi(argv[i]);
    }

    printf("%i\n", soucet);
    return 0;
}
```

Řešení úkolu 57:

```
int vetsi(int a, int b){
    if(a >= b)
        return 1;

    return 0;
}

void vymena(int *a, int *b){
    int pom = *a;
    *a = *b;
    *b = pom;
}

// bubble sort
int* trideni(int (*fce_porovnani)(int,int), void (*fce_vymeny)(int*,int*), int* pole_vstup, int
    pocet){
    int i, j;
    int *pole = malloc(pocet * sizeof(int));
    memcpy(pole, pole_vstup, pocet * sizeof(int));
    for (i = 0; i < pocet - 1; i++)

       // poslednich i prvku uz je setrizeno
       for (j = 0; j < pocet - i - 1; j++)
           if (!fce_porovnani(pole[j], pole[j + 1]))
              fce_vymeny(&pole[j], &pole[j + 1]);

    return pole;
}


/* Priklad volani */
vystup = trideni(vetsi, vymena, pole_hodnot, 10);
```

Řešení úkolu 69:

```c
#include <stdio.h>

int main(){
    char soubor_in[] = "soubor.txt";
    char soubor_out[] = "soubor2.txt";
    int cislo;

    FILE *fr, *fw;
    fr = fopen(soubor_in,"r");
    fw = fopen(soubor_out,"w");
    /* chybi zde kontrola otevreni souboru */

    while(fscanf(fr, "%i", &cislo) != EOF){
        if(cislo <= 100)
            fprintf(fw, "%i ", cislo);
        else
            fprintf(stderr, "Spatne zadane cislo %i\n", cislo);
    }

    fclose(fr);
    fclose(fw);
    /* opet chybi kontrola */

    return 0;
}
```

Řešení úkolu 75:

```c
#define cti_int(x) (scanf("%i", &x), x)
```

Řešení úkolu 85:[161]

```c
/* posun n teho bitu na konec a binarni soucin s 1 */
(cislo >> n) & 1;
```

[161] Toto řešení nekontroluje, zda nechceme zjistit hodnotu bitu, který číslo neobsahuje.

Řešení úkolu 89:

```c
char rotace(char cislo, int n){
    char zacatek = cislo << (8 - n);
    char konec = cislo >> n;
    char vysledek = zacatek | konec;
    return vysledek;
}
```

Řešení úkolu 91:

1. Syntaktická chyba – chybí středník za příkazem `printf()`.
2. Za cyklem `for` je středník, jedná se o prázdný cyklus. Výpis se provádí až po jeho ukončení.
3. Syntaktická chyba – v cyklu `for` jsou použity `,` místo `;`.
4. `pole[3]` indexuje prvek, který není součástí pole.
5. Proměnná `b` není inicializovaná.
6. Dělíme `0`.
7. V podmínce používáme operátor přiřazení (=) místo operátoru porovnání (==).[162]
8. Levý operand na druhém řádku ve funkci `main()` není `l-value`.
9. Kvůli použití globální proměnné `i` nepracuje kód, jak by se očekávalo.
10. Do struktury `m` ukládáme hodnoty špatným způsobem. Tímto způsobem můžeme hodnoty přidat pouze při deklaraci (inicializaci). Poté je potřeba hodnoty ukládat zvlášť pomocí tečkového operátoru.

[162] Podrobnější vysvětlení viz řešení úkolu 14.

Řešení úkolu 92:
Proměnná `karty` je ukazatel na konstantní řetězec, není možné mu měnit hodnotu.

Řešení úkolu 97:[163]

```
printf("max unsigned int = %u \n", -1);
printf("max signed int = %i \n", (unsigned int)(-1) >> 1);
```

[163] V knihovně `limits.h` jsou mimo jiné definované symbolické konstanty `UINT_MAX` (unsigned int) a `INT_MAX` (signed int), které uchovávají tyto hodnoty.

Řešení úkolu 108:

```
int posloupnost(int n){
    int i;
    int a = 14688;

    for(i = 1; i < n; i++){
        a = a / 2 + 1200;
    }

    return a;
}
```

Řešení úkolu 109:

```
#include <stdio.h>

void vypis(int *pole, int zacatek, int krok, int konec){
    int i;

    for(i = zacatek; i < konec; i = i + krok){
        printf("%i ", *(pole + i));
    }
}

/* Priklad volani */
vypis(pole, 0, 2, 9);
```

Řešení úkolu 114:

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

int *data;
int velikost;
int hlava;

/*
typedef struct{
    int *data;
    int velikost;
    int hlava;
} POLE;

POLE array;
*/
```

```
/* v pripade pouziti struktury v kodu nahradit
data    za   array.data
velikost     array.velikost
hlava        array.hlava

*/

void init(int v){
  data = malloc(sizeof(int) * v);
  /* chybi zde kontrola, zda se zdarila alokace. */
  hlava = 0;
  velikost = v;
}

void uvolni(){
    free(data);
}

void vypis(){
    for(int i = 0; i < hlava; i++)
        printf("%i \n", data[i]);
}

/* pridani prvku na konec pole */
void pridej(int prvek){
    if(hlava == velikost){
        data = realloc(data, sizeof(int) * (velikost * 2));
        velikost = velikost * 2;
    }
    data[hlava] = prvek;
    hlava = hlava + 1;
   /* vypis(); */
}

/* odebere posledni prvek pole */
void odeber_posledni(){
    hlava = hlava-1;

    /* pocet prvku, nemuze byt zaporny */
    if(hlava < 0)
        hlava = 0;

    /* pripadne zmenseni velikosti pole */
    if(hlava < (velikost / 2)){
        data = realloc(data, sizeof(int) * (velikost / 2));
        velikost = velikost / 2;
    }
}
```

```
/* odebere prvek na indexu index */
void odeber_prvek(int index){
        if (index < hlava){
            for(int i = index; i < hlava - 1; i++){
                data[i] = data[i + 1];
            }
            hlava = hlava - 1;
        }

    /* pripadne zmenseni velikosti pole */
    if(hlava < (velikost / 2)){
        data = realloc(data, sizeof(int) * (velikost / 2));
        velikost = velikost / 2;
    }
}

/* vrati index prvku, nebo -1 pokud neni prvek nalezen */
int najdi(int cislo){
    int i;
    for(i = 0; i < hlava; i++){
        if(data[i] == cislo)
            return i;
    }
    return -1;
}
```

Řešení úkolu 116:

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

typedef struct{
    float realna_cast;
    float imaginarni_cast;
} komplexni;

komplexni suma(int pocet, ...){
    int i;
    va_list parametry;
    va_start(parametry, pocet);
    komplexni pomocne_cislo;

    komplexni vysledek;
    vysledek.imaginarni_cast = 0;
    vysledek.realna_cast = 0;

    for(i=0; i<pocet; i++){
        pomocne_cislo = va_arg(parametry, komplexni);
        vysledek.realna_cast += pomocne_cislo.realna_cast;
        vysledek.imaginarni_cast += pomocne_cislo.imaginarni_cast;
    }

    return vysledek;
}
```