Kapitola 2

Typy, operátory a výrazy

Proměnné a konstanty jsou základní datové objekty, s nimiž program manipuluje. Deklarace vytvářejí seznam proměnných, které budeme používat, uvádějí jejich typ a v některých případech i jejich počáteční hodnoty. Operátory určují, co se má s proměnnými a konstantami provést. Výrazy vytvářejí nové hodnoty kombinováním proměnných a konstant. Typ objektu vymezuje operace, které lze s tímto objektem provádět, a množinu hodnot, kterých může objekt nabývat. To vše probereme v této kapitole.

Standard ANSI přináší mnoho menších doplňků a změn týkajících se základních typů a výrazů. Všechny celočíselné typy mají nyní formy `signed` a `unsigned` a nově jsou kodifikovány i zápisy konstant bez znaménka a zápisy konstant v šestnáctkové (hexadecimální) soustavě. Operace v pohyblivé desetinné čárce lze nyní provádět s jednoduchou přesností; pro ještě větší přesnost nyní existuje typ `long double`. Řetězcové konstanty lze spojovat už během kompilace. Výčty se staly součástí jazyka, čímž byla formalizována již dlouho používaná vlastnost. Objekty je možné deklarovat jako `const`, tato deklarace pak brání jejich změně. Pravidla automatických převodů aritmetických typů byla rozšířena, aby pojala větší škálu možných typů.

2.1 Jména proměnných

I když jsme se o tom v první kapitole nezmínili, pro jména proměnných a symbolických konstant existují jistá omezení. Jména se skládají z písmen a číslic; prvním znakem musí být písmeno. Podtržítko „_“ se počítá mezi písmena; občas se hodí pro zlepšení čitelnosti jmen dlouhých proměnných. Jména proměnných ale nikdy nezačínejte podtržítkem, protože tato jména často používají knihovní funkce. Rozlišuje se mezi velkými a malými písmeny, proto jsou `x` a `X` odlišná jména. Běžnou praxí je používat malá písmena pro názvy proměnných a velká písmena pro symbolické konstanty.

Nejméně prvních 31 znaků interního jména je významných. Pro jména funkcí a proměnných může být toto číslo menší než 31, neboť externí jména mohou používat asemblery a zaváděcí programy, nad nimiž nemá jazyk žádnou kontrolu. Standard zaručuje jednoznačnost externích jmen pouze pro prvních 6 znaků a jednu velikost písma. Klíčová slova, například `if`, `else`, `int`, `float` atd., jsou vyhrazena: nelze je používat jako jména proměnných. Klíčová slova musí být psána malými písmeny.

Je moudré volit taková jména proměnných, která se vztahují k jejich účelu a která nejsou typograficky matoucí. Pro lokální proměnné používáme spíše krátká jména a pro externí proměnné jména delší.

2.2 Datové typy a velikosti

Jazyk C obsahuje pouze několik základních datových typů:

`char`	jeden bajt, schopný uchovávat jeden znak z místní znakové sady
`int`	celé číslo, obvykle odráží přirozenou velikost celých čísel na hostitelském počítači
`float`	číslo s pohyblivou řádovou čárkou s jednoduchou přesností
`double`	číslo s pohyblivou řádovou čárkou s dvojnásobnou přesností

Kromě toho existují kvalifikátory, které je možné aplikovat na tyto základní typy. `short` a `long` lze aplikovat na celá čísla:

```
short int sh;
long int pocitadlo;
```

Slovo `int` lze v takovýchto deklaracích vynechat a obvykle se to tak dělá.

Snahou je, aby `short` a `long` poskytovaly různé délky celých čísel tam, kde se to hodí; obecně `int` bude mít přirozenou velikost pro daný počítač. `short` má často 16 bitů, `long` 32 bitů a `int` buďto 16 nebo 32 bitů. Každý kompilátor si může zvolit odpovídající velikosti pro svůj hardware; jediným omezením je, že `short` a `int` musí mít minimálně 16 bitů, `long` má minimálně 32 bitů a `short` není větší než `int`, který je menší než `long`.**

Kvalifikátory `signed` a `unsigned` lze aplikovat na `char` a jakýkoli celočíselný typ. Čísla s kvalifikátorem `unsigned` jsou vždy kladná nebo nulová a řídí se aritmetikou modulo 2^n, kde n je počet bitů v typu. Takže je-li například `char` osmibitový, nabývají proměnné typu `unsigned char` hodnot od 0 do 255, zatímco `signed char` nabývají hodnot od -128 do 127 (na stroji používajícím dvojkový doplněk). Záleží na konkrétní architektuře, jsou-li proměnné samotného typu `char` kvalifikovány jako `signed` či `unsigned`, nicméně tisknutelné znaky jsou vždy kladné.

Typ `long double` určuje číslo s pohyblivou řádovou čárkou s rozšířenou přesností. Velikost objektů s pohyblivou řádovou čárkou je stejně jako u celých čísel závislá na implementaci; `float`, `double` a `long double` mohou reprezentovat jednu, dvě nebo tři odlišné velikosti.

Standardní hlavičkové soubory `<limits.h>` a `<float.h>` obsahují symbolické konstanty pro všechny tyto velikosti současně s parametry počítače a kompilátoru. Více informací můžete nalézt v dodatku B.

Cvičení 2.1. Napište program, který určí rozsahy hodnot proměnných typů `char`, `short`, `int` a `long`, jak `signed` tak `unsigned`, vypsáním odpovídajících hodnot ze standardních hlavičkových souborů a přímým výpočtem. Těžší je tyto hodnoty vypočítat: určete rozsahy hodnot různých typů s pohyblivou řádovou čárkou.

2.3 Konstanty

Celočíselná konstanta, jako je `1234`, je typu `int`. Konstanta typu `long` se píše s `l` nebo `L` na konci, například `12345789L`; celé číslo, které je příliš velké na to, aby se vešlo do `int`, bude také bráno jako `long`. Konstanty bez znaménka se píší na konci s `u` nebo `U` a přípona `ul` nebo `UL` indikuje typ `unsigned long`.

** Poznámka českého vydavatele: Výklad o „základním typu `int`“ a „kvalifikátorech, které upravují jeho význam“ patří mezi folklór jazyka C; týká se však spíše filozofie návrhu jazyka než praktického programování. Jde o to, že `short`, `int`, `long`, `unsigned short`, `unsigned` a `unsigned long` představuje šest datových typů, které jsou navzájem různé, i když si jsou velice blízké. Pro zpestření mají tyto typy několik různých jmen, takže např. `int`, `signed int` a `signed` označují stejný typ.

Konstanty s pohyblivou řádovou čárkou obsahují desetinnou tečku (`123.4`) nebo exponent (`1e-2`) nebo obojí; nemají-li příponu, je jejich typ `double`. Přípony `f` nebo `F` indikují konstanty typu `float`; `l` nebo `L` určují `long double`.

Hodnotu celého čísla je možné specifikovat také v osmičkové nebo šestnáctkové soustavě. Nula na začátku celočíselné konstanty určuje číslo v osmičkové soustavě; `0x` nebo `0X` pak číslo v šestnáctkové soustavě. Například číslo `31` v desítkové soustavě lze zapsat oktalově (v osmičkové soustavě) jako `037` a hexadecimálně (v šestnáctkové soustavě) jako `0x1f` nebo `0X1F`. I oktalové a hexadecimální konstanty mohou mít příponu `L`, která z nich udělá číslo typu `long`, a `U`, která z nich udělá `unsigned`: `0XFUL` je `unsigned long` konstanta s decimální hodnotou `15`.

Znaková konstanta je celé číslo zapsané jako jeden znak uzavřený mezi apostrofy, například 'x'. Hodnota znakové konstanty je číselnou hodnotou znaku ve znakové sadě počítače. Například ve znakové sadě ASCII má znaková konstanta '0' hodnotu `48`, což nemá žádnou spojitost s číselnou hodnotou `0`. Napíšeme-li '0' namísto hodnoty `48`, která závisí na znakové sadě, bude náš program čitelnější a nezávislý na konkrétní hodnotě. Znakové konstanty fungují v číselných operacích stejně jako jiná celá čísla, i když nejčastěji jsou používány pro porovnávání s jinými znaky.

Určité znaky můžeme reprezentovat ve znakových a řetězcových konstantách řídicími posloupnostmi, jako '\n' (znak nového řádku); tyto sekvence vypadají jako dva znaky, ale reprezentují jen jeden. Navíc můžeme zadat libovolnou hodnotu velikosti jednoho bajtu, pokud napíšeme

```
'\ooo'
```

kde *ooo* reprezentuje jedno až tři čísla v osmičkové soustavě (`1..7`), nebo

```
'\xhh'
```

kde *hh* reprezentuje jedno nebo více hexadecimálních číslic (`0...9`, `a...f`, `A...F`). Proto můžeme napsat

```
#define VTAB   '\013'     /* ASCII: vertikální tabulátor */
#define ZVONEK '\007'     /* ASCII znak zvonku */
```

nebo hexadecimálně

```
#define VTAB '\xb'        /* ASCII vertikální tabulátor */
#define ZVONEK '\x7'      /* ASCII znak zvonku */
```

Kompletní množina řídicích posloupností vypadá takto:

`\a`	upozornění (zvonek)
`\\`	zpětné lomítko
`\b`	krok zpět (backspace)
`\?`	otazník
`\f`	přechod na novou stranu
`\'`	apostrof
`\n`	přechod na nový řádek
`\"`	uvozovky
`\r`	návrat vozíku (návrat a začátek řádku)
`\ooo`	číslo v osmičkové soustavě
`\t`	horizontální tabulátor
`\xhh`	číslo v šestnáctkové soustavě

`\v` vertikální tabulátor

Znaková konstanta `'\0'` reprezentuje znak s nulovou hodnotou, prázdný znak. Pro zvýraznění znakového charakteru některých výrazů se často píše `'\0'` namísto `0`, ale číselná hodnota je vždy rovna `0`.

Konstantní výraz je výraz obsahující pouze konstanty. Takové výrazy je možné vyčíslit během kompilace a díky tomu je možné je použít kdekoli, kde se může objevit konstanta, například

```
#define MAXRADEK 1000
char radek[MAXRADEK+1];
```

nebo

```
#define PRESTUPNY 1 /* v přestupných rocích */
int days[31+28+PRESTUPNY+31+30+31+30+31+31+30+31+30+31];
```

Řetězcová konstanta je posloupnost znaků uzavřených do uvozovek; může být i prázdná (nemusí obsahovat žádný znak), například

```
"Já jsem řetězec"
```

nebo

```
"" /* prázdný řetězec */
```

Uvozovky nejsou součástí řetězce, slouží pouze jako oddělovače. V řetězcích platí stejné řídicí posloupnosti jako ve znakových konstantách; `\"` reprezentuje znak uvozovek. Řetězcové konstanty lze při kompilaci spojovat:

```
"ahoj, " "světe"
```

je ekvivalentní

```
"ahoj, světe"
```

To je užitečné při rozdělování dlouhých řetězců do více řádků zdrojového textu.

Technicky vzato je řetězcová konstanta polem znaků. Ve vnitřní reprezentaci řetězce je na konci umístěn znak `'\0'`, takže požadovaná fyzická paměť je o jeden bajt větší než je počet znaků zapsaných mezi uvozovkami. Z této reprezentace vyplývá, že neexistuje žádné omezení délky řetězce, ale na druhou stranu programy musí projít celý řetězec, aby určily jeho délku.

Funkce `strlen(r)` ze standardní knihovny vrací délku znakového řetězce `s` bez koncového znaku `'\0'`. Zde je naše verze:

```
/* strlen: vrací délku r */
int strlen(char r[])
{
   int i;

   i = 0;
   while(r[i] != '\0')
      ++i;
   return i;
}
```

`strlen` a ostatní funkce pracující s řetězci jsou deklarovány ve standardním hlavičkovém souboru `<string.h>`.

Pečlivě rozlišujte mezi znakovou konstantou a řetězcem obsahujícím jediný znak: `'x'` není totéž co `"x"`. První je celé číslo používané pro získání číselné hodnoty písmene `x` ve znakové sadě počítače. Druhé je pole znaků obsahující jeden znak (písmeno `x`) a `'\0'`.

Existuje ještě jiný druh konstanty, *výčtová konstanta*. Výčet je seznam konstantních celočíselných hodnot, například

```
enum boolean { NE, ANO };
```

První jméno v tomto výčtu má hodnotu `0`, další má hodnotu `1` a tak dále, pokud nejsou hodnoty uvedeny explicitně. Nejsou-li uvedeny všechny hodnoty, pak nespecifikované hodnoty pokračují v řadě od poslední určené hodnoty, jak je vidět ve druhém z následujících příkladů:

```
enum escapes { ZVONEK = '\a', BACKSPACE = '\b', TABULATOR = '\t',
     NOVYRADEK = '\n', VTAB = '\v', ENTER = '\r' };
enum mesice { LED = 1, UNO, BRE, DUB, KVE, CER, CEC, SRP, ZAR, RIJ, LIS,
     PRO }; /* UNO je 2, BRE je 3 atd. */
```

Jména v různých výčtech se musí lišit. Hodnoty ve stejném výčtu se lišit nemusí.

Výčty představují šikovný způsob, jak sdružit hodnoty se jmény. Jsou alternativou k `#define` a mají tu výhodu, že program za nás může sám generovat hodnoty. I když je možné deklarovat proměnné typu `enum`, kompilátory nemusí kontrolovat, zda do takové proměnné ukládáte hodnotu, která je platná v daném výčtu. Nicméně výčtové proměnné nabízejí šanci pro kontrolu a jsou často lepším řešením než `#define`. Navíc je možné, že ladicí program bude schopen vypisovat hodnoty výčtových proměnných v symbolické formě.

2.4 Deklarace

Všechny proměnné je nutné před použitím deklarovat, i když některé deklarace mohou vyplývat z kontextu. Deklarace se skládá z typu a jedné nebo několika proměnných tohoto typu, například

```
int dolni, horni, krok;
char z, radek[1000];
```

Proměnné můžeme rozdělit mezi deklarace libovolným způsobem; předchozí seznam bychom mohli stejně dobře napsat

```
int dolni;
int horni;
int krok;
char z;
char radek[1000];
```

Druhá forma zápisu zabírá více místa, ale je vhodnější pro přidávání komentářů k jednotlivým deklaracím a pro pozdější úpravy.

Proměnnou můžeme v deklaraci také inicializovat. Jestliže za jménem následuje znaménko rovnosti a výraz, potom výraz slouží jako inicializátor, například

```
char esc = '\\';
int i = 0;
int limit = MAXRADEK + 1;
float eps = 1.0e-5;
```

Nejde-li o automatickou proměnnou, pak inicializace proběhne pouze jednou, před spuštěním samotného programu; inicializátor musí být konstantním výrazem. Explicitně inicializovaná automatická proměnná je inicializována při každém vstupu do funkce či bloku; jako inicializátor lze použít libovolný výraz. Externí a statické proměnné jsou implicitně inicializovány nulou. Automatické proměnné bez explicitních inicializátorů mají nedefinované hodnoty.

Na deklaraci libovolné proměnné můžeme aplikovat kvalifikátor `const` a tak zajistit, že se její hodnota nebude měnit. Kvalifikátor `const` aplikovaný na pole říká, že jeho prvky nebudou měněny.

```
const double e = 2.71828182845905;
const char msg[] = "varování: ";
```

Deklaraci `const` lze také použít na argumenty typu pole a tím dát najevo, že funkce toto pole nemění:

```
int strlen(const char[]);
```

Výsledek pokusu o změnu proměnné s kvalifikátorem `const` závisí na implementaci.

2.5 Aritmetické operátory

Binární aritmetické operátory jsou +, -, *, / a operátor zbytku po dělení %. Celočíselné dělení ořezává zlomkovou část výsledku. Výraz

```
x % y
```

produkuje zbytek po dělení proměnné `x` proměnnou `y` a je nulový, jestliže `y` dělí `x` přesně. Například rok je přestupný, jestliže je dělitelný 4, ale ne 100, kromě let, která jsou dělitelná 400. Tedy

```
if ((rok % 4 == 0 && rok % 100 != 0) || rok % 400 == 0)
   printf("%d je přestupný rok \n, rok);
else
   printf("%d není přestupný rok \n, rok);
```

Operátor % není možné aplikovat na typy `float` a `double`. Pro záporné operandy jsou směr zaokrouhlení u operátoru / a znaménko výsledku u operátoru % strojově závislé, stejně jako v případě akce vykonané po přetečení či podtečení.

Binární operátory + a - mají stejnou prioritu, která je menší než priorita operátorů *, / a %, a ta je menší než priorita unárních operátorů + a -. Aritmetické operátory se sdružují zleva doprava.

Tabulka 2.1 na konci této kapitoly shrnuje prioritu a asociativitu všech operátorů.

2.6 Relační a logické operátory

Relační operátory jsou

```
>   >=   <   <=
```

Všechny mají stejnou prioritu. Nižší prioritu mají operátory rovnosti

```
==   !=
```

Relační operátory mají menší prioritu než aritmetické operátory, proto výrazy typu `i < lim-1` jsou chápány jako `i < (lim-1)`, což od nich očekáváme.

Zajímavější jsou logické operátory `&&` a `||`. Výrazy spojené pomocí `&&` nebo `||` jsou vyhodnocovány zleva doprava; vyhodnocování skončí, jakmile je znám výsledek. Většina programátorů v jazyce C na tuto vlastnost spoléhá. Například zde je cyklus pro vstupní funkci `nactiradek`, kterou jsme vytvořili v první kapitole:

```
for(i=0; i<lim-1 && (z=getchar()) != '\n' && z != EOF; ++i)
   r[i]=z;
```

Před načtením nového znaku je nutné zkontrolovat, máme-li dost místa pro jeho umístění do pole `r`, proto je *nutné* provést test `i < lim-1` jako první. Navíc, pokud tento test selže, nesmíme pokračovat v načítání dalšího znaku.

Podobně by bylo nešťastné, kdybychom `z` porovnávali s `EOF` před zavoláním `getchar`; proto se volání a přiřazení musí objevit před testováním znaku `z`.

Priorita `&&` je větší než priorita `||` a priority obou jsou menší než priority relačních operátorů a operátorů rovnosti, proto výrazy typu

```
i<lim-1 && (z = getchar()) != '\n' && z != EOF
```

nepotřebují žádné další závorky. Priorita `!=` je však větší než priorita přiřazení, a proto je nutné v

```
(z = getchar()) != '\n'
```

závorky uvést, čímž zajistíme, že nejprve proběhne požadované přiřazení do `z` a teprve pak porovnání s `'\n'`.

Podle definice je číselná hodnota relačního nebo logického výrazu rovna `1`, je-li výraz pravdivý, a `0`, je-li výraz nepravdivý.

Unární operátor negace `!` převádí nenulové operandy na `0` a nulové operandy na `1`. `!` se běžně využívá v konstrukcích typu

```
if (!plati)
```

se kterými se setkáme častěji než s

```
if (plati == 0)
```

Je obtížné říci, která forma je obecně lepší. Konstrukce stylu `!plati` se snadno čtou („neplatí“), ale jsou-li komplikovanější, obtížněji se chápou.

Cvičení 2.2. Napište cyklus ekvivalentní uvedenému cyklu `for` bez použití `&&` a `||`.

2.7 Konverze typů

Má-li operátor operandy různých typů, převedou se na společný typ na základě malé množiny pravidel. Obecně lze říci, že jediné automatické konverze jsou ty, které převádějí „užší“ typ na „širší“ bez ztráty informace, například převedení celého čísla na číslo s pohyblivou desetinnou čárkou ve výrazu typu `f + i`. Výrazy nedávající smysl, například použití `float` v roli indexu pole, nejsou povoleny. Výrazy, u nichž by mohlo dojít ke ztrátě informace, například přiřazení většího celočíselného typu menšímu, nebo čísla s pohyblivou řádovou čárkou celému číslu, mohou vyvolat varovná hlášení, ale nejsou zakázány.

Typ `char` je pouze malým celým číslem, proto můžeme hodnoty typu `char` v aritmetických výrazech používat naprosto volně. To dává značnou flexibilitu v určitých druzích znakových transformací. Jednu z nich si předvedeme naivní implementací funkce `atoi`, jež převádí řetězec číslic na jeho číselný ekvivalent.

```
/* atoi: převádí r na celé číslo */
int atoi(char r[])
{
   int i, n;

   n = 0;
   for (i=0; r[i] >= '0' && r[i] <= '9'; ++i)
      n = 10 * n + (r[i] - '0');
   return n;
}
```

Jak jsme si vysvětlili v první kapitole, výraz

```
r[i] - '0'
```

dává číselnou hodnotu znaku uloženého v `r[i]`, protože hodnoty `'0'`, `'1'` atd. tvoří souvislou rostoucí řadu.

Jiným příkladem konverze typu `char` na typ `int` je funkce `zmensi`, která převede jeden znak na malé písmeno *ve znakové sadě ANSI*. Není-li znakem velké písmeno, funkce ho vrátí v nezměněném tvaru.

```
/* zmensi: převede z na malé písmeno; pouze ASCII */
int zmensi(int z)
{
   if (z >= 'A' && z <= 'Z')
      return z + 'a' - 'A';
   else
      return z;
}
```

Tento program funguje v kódování ASCII, protože číselné hodnoty odpovídajících velkých a malých písmen mají od sebe pevnou vzdálenost a každá z těchto abeced je souvislá – mezi `A` a `Z` jsou pouze písmena anglické abecedy ve správném pořadí. To ovšem neplatí pro znakovou sadu EBCDIC, v ní by náš program převáděl i jiné znaky než jen písmena.

Standardní hlavičkový soubor `<ctype.h>`, popsaný v příloze B, definuje skupinu funkcí pro testy a převody, jež jsou nezávislé na znakové sadě. Například funkce `tolower(c)` vrací hodnotu odpovídajícího malého písmene, obsahuje-li `c` velké písmeno. To znamená, že `tolower` je přenositelnou náhradou za naši funkci `zmensi`. Podobně test

```
z >=  '0' && z <= '9'
```

lze nahradit

```
isdigit(z)
```

Od tohoto okamžiku už budeme používat jen funkce z `<ctype.h>`.

Konverze znaků na čísla se týká jeden menší problém. Jazyk neudává, zda jsou proměnné typu `char` hodnotami se znaménkem nebo bez znaménka. Je možné, aby výsledkem bylo záporné číslo, je-li `char` zkonvertován na `int`? Odpověď se liší podle počítače, podle typu architektury. Na některých počítačích bude `char`, jehož nejméně významný bit je roven `1`,

převeden na záporné celé číslo („znaménkové rozšíření“). Na jiných bude z typu `char` vytvořen `int` přidáním nul k levému konci a výsledek bude proto vždy kladný.

Definice jazyka C zajišťuje, že žádný znak ze standardní znakové sady počítače nebude mít zápornou hodnotu, proto i ve výrazech budou znaky nabývat kladných hodnot. Ale určité bitové vzory uložené ve znakových proměnných mohou být na jistých počítačích vyhodnoceny jako záporná čísla, a to i přesto, že jiné počítače je vždy vyhodnotí jako kladné. Chcete-li v proměnných typu `char` ukládat jiná data než znaky, pak z důvodu přenositelnosti konkretizujte typ proměnné kvalifikátorem `signed` nebo `unsigned`.

Relační výrazy typu `i > j` a logické výrazy spojené pomocí `&&` nebo `||` mají podle definice hodnotu `1`, jsou-li pravdivé, a `0`, jsou-li nepravdivé. Proto přiřazení

```
d = z >= '0' && z <= '9'
```

nastaví `d` na `1`, je-li `c` číslicí, a na `0`, není-li. Nicméně funkce jako `isdigit` mohou jako pravdivý výsledek vrátit libovolnou nenulovou hodnotu. V podmínce příkazů `if`, `while`, `for` apod. je „pravdivý“ ekvivalentní významu „nenulový“, takže zde nevzniká žádný problém.

Implicitní aritmetické konverze fungují v principu tak, jak bychom očekávali. Obecně, mají -li operátory jako + nebo * se dvěma operandy (binární operátory) operandy různých typů, před proběhnutím operace je „nižší“ typ je *rozšířen* na „vyšší“. Výsledek má „vyšší“ typ. Přesná pravidla pro převody lze nalézt v šestém oddílu přílohy A. Nicméně nepočítáme-li s operandy typu `unsigned`, pak stačí následující neformální sada pravidel:

Je-li některý z operandů typu `long double`, převede se druhý na `long double`.

Jinak, je-li některý z operandů `double`, převede se druhý na `double`.

Jinak, je-li některý z operandů `float`, převede se druhý na `float`.

Jinak se převede `char` a `short` na `int`.

Potom, je-li některý z operandů `long`, převede se druhý na `long`.

Všimněte si, že hodnoty typu `float` nejsou ve výrazech automaticky převáděny na typ `double`; to je změna oproti původní definici jazyka. Obecně matematické funkce, například ty v `<math.h>`, používají dvojitou přesnost. Hlavním důvodem pro používání typu `float` je šetření místem ve velkých polích, nebo méně často šetření časem na počítačích, kde je aritmetika s dvojnásobnou přesností příliš nákladná.

Konverzní pravidla jsou komplikovanější při použití operandů typů `unsigned`. Problém spočívá v tom, že porovnání mezi hodnotami `signed` a `unsigned` jsou strojově závislá, jelikož závisí na velikostech různých celočíselných typů. Předpokládejme například, že `int` má 16 bitů a `long` 32 bitů. Potom `-1L < 1U`, protože `1U` (typu `int`) bude převedeno na `signed long`. Ale `-1L > 1UL`, protože `-1L` bude převedeno na `unsigned long` a vznikne z něj velké kladné číslo.

Konverze se odehrávají i v přiřazeních; hodnota pravé strany bude převedena na typ levé strany, který je typem výsledku.

Znak je převeden na celé číslo, se znaménkovým rozšířením či bez něj.

Delší celá čísla jsou převedena na kratší zahozením přebytečných bitů vyšších řádů. Tedy v

```
int i;
char c;

i = c;
c = i;
```

nedojde ke změně hodnoty `c`. To platí, ať dojde ke znaménkovému rozšíření nebo ne. Avšak přehodíme-li pořadí přiřazení, pak může dojít ke ztrátě informace.

Je-li `x` typu `float` a `i` typu `int`, pak jak `x = i`, tak `i = x` si vynutí konverzi; při konverzi `float` na `int` dojde k oříznutí zlomkové části. Závisí na konkrétní architektuře, zda dojde při převodu `double` na `float` k zaokrouhlení či oříznutí.

Protože argument volání funkce je výraz, dochází ke konverzím i při předávání argumentů funkcím. Není-li k dispozici prototyp funkce, `char` a `short` se převedou na `int` a `float` se převede na `double`. Proto jsme deklarovali argumenty funkcí jako `int` a `double` i přesto, že jsme funkci volali s `char` a `float`.

Konečně, v jakémkoli výrazu můžeme explicitní konverzi vynutit pomocí unárního operátoru *přetypování*. V konstrukci

(*jméno typu*) *výraz*

je *výraz* převeden na jmenovaný typ podle uvedených pravidel. Sémantika přetypování odpovídá případu, kdy je *výraz* přiřazen proměnné určeného typu, která je následně použita na místě celé konstrukce. Například knihovní funkce `sqrt` očekává argument typu `double` a dá nesmyslný výsledek, pokud jí nechtěně předáme něco jiného. (`sqrt` je deklarována v `<math.h>`.) Proto, je-li `n` celé číslo, můžeme použít

```
sqrt((double) n)
```

pro převod hodnoty `n` na `double` předtím, než je předána funkci `sqrt`. Poznamenejme, že přetypování poskytne *hodnotu* `n` ve správném typu; `n` samotné se nezmění. Operátor přetypování má stejnou prioritu jako jiné unární operátory. Vše je shrnuto v tabulce na konci kapitoly.

Jsou-li argumenty deklarovány prototypem funkce, jak by správně měly být, pak deklarace zajistí při zavolání funkce automatický převod typů všech argumentů. Proto, máme-li prototyp funkce `sqrt`,

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

pak volání

```
koren2 = sqrt(2);
```

převede celé číslo `2` na hodnotu `2.0` typu `double`, aniž bychom potřebovali explicitní přetypování.

Standardní knihovna poskytuje přenositelnou implementaci generátoru pseudonáhodných čísel a funkci pro inicializaci generátoru; v první z nich můžeme vidět přetypování v praxi:

```
unsigned long int next = 1;

/* rand: vrací pseudonáhodné celé číslo od 0 do 32767 */
int rand(void)
{
   next = next * 1103515245 + 12345;
   return (unsigned int)(next/65536) % 32768;
}

/* srand: inicializuje generátor */
void srand(unsigned int seed)
{
   next = seed;
}
```

Cvičení 2.3. Napište funkci `htoi(s)`, která převede řetězec hexadecimálních číslic (včetně nepovinného `0x` či `0X`) na ekvivalentní celočíselnou hodnotu. Povolené číslice jsou `0` až `9`, `a` až `f` a `A` až `F`.

2.8 Operátory inkrementace a dekrementace

C poskytuje dva neobvyklé operátory pro zvyšování a snižování hodnot proměnných. Operátor inkrementace `++` přičte ke svému operandu `1`, zatímco operátor dekrementace `--` odečte `1`. `++` jsme často používali pro zvyšování hodnot proměnných, například v

```
if (z == '\n')
   ++pr;
```

Neobvyklé je, že `++` a `--` můžeme použít jak v prefixové formě (před proměnnou, `++n`), tak v postfixové (za proměnnou, `n++`). V obou případech dojde ke zvýšení hodnoty `n` o jedna. Ale výraz `++n` zvyšuje `n` *před* použitím její hodnoty, zatímco `n++` zvyšuje `n` *po* použití její hodnoty. To znamená, že `++n` a `n++` se liší v kontextu použití, ne v samotném zvýšení hodnoty `n`. Je-li `n` rovno `5`, pak

```
x = n++;
```

nastaví hodnotu `x` na `5`, ale

```
x = ++n;
```

nastaví hodnotu `x` na `6`. V obou případech je po provedení příkazu `n` rovno `6`. Operátory inkrementace a dekrementace lze aplikovat pouze na proměnné; výraz typu `(i+j)++` je neplatný.

V kontextu, v němž není vyžadována žádná hodnota, pouze efekt přičtení či odečtení jedničky, jako je tomu v:

```
if (z == '\n')
   pr++;
```

poslouží stejně dobře prefixová i postfixová verze. Existují ale situace, které si přímo říkají o tu kterou verzi. Například vezměme funkci `odstran(r,c)`, která odstraňuje všechny výskyty znaku `c` z řetězce `r`.

```
/* odstran: odstraní všechny znaky c z r */
void odstran(char r[], int c)
{
   int i, j;

   for (i = j = 0; r[i] != '\0'; i++)
      if (r[i] != c)
         r[j++] = r[i];
   r[j] = '\0';
}
```

Každý znak odlišný od `c` je zkopírován na aktuální pozici `j`, a teprve pak je `j` zvýšeno o `1`, aby bylo možné zpracovat další znak. To je přesně ekvivalentní

```
if (r[i] != c) {
   r[j] = r[i];
   j++;
}
```