Obsah

Předmluva

Otevíráte další díl kursu programování. V této knize se seznámíte se základy práce s daty v programovacích jazycích Turbo Pascal a Borland C++. To znamená, že se naučíte deklarovat a používat proměnné a konstanty různých datových typů, používat vstupní a výstupní operace apod. Pokročilejší práci s daty, tedy např. dynamické alokaci paměti a podobným tématům, věnujeme další díl.

Tato kniha vznikla přepracováním a doplněním třetí části úspěšného seriálu *Cesta k profesionalitě*, který vycházel v letech 1992 – 1994 v časopisu ComputerWorld.

Náš výklad je založen především na překladačích Borland C++ 3.1 a Turbo Pascal 7.0, které mohou běžet na velké většině počítačů, běžně dostupných nejširší čtenářské obci.

Předpoklady

Od čtenářů očekáváme, že jejich znalosti zhruba odpovídají obsahu předchozího dílu. To znamená, že

- ✧ umějí používat běžné programové konstrukce,
- ✧ umějí rozložit úlohu na dílčí algoritmy,
- ✧ umějí zacházet s procedurami a funkcemi bez parametrů,
- ✧ umějí zacházet s vývojovým prostředím pro jazyky Pascal a C.

Terminologie

Čtenáři, kteří sledovali časopiseckou verzi tohoto kursu, zjistí, že jsme poněkud změnili terminologii. Především jsme opustili označení *fiktivní funkce*, používané v jazyce C++ pro funkce s modifikátorem **inline**, a nahradili jsme je termínem *vložená funkce*.

Pro funkce a operátory se stejným jménem, které se liší počtem a typem parametrů, používáme vedle termínu *funkční homonyma*, známého z časopisecké verze kursu, také označení *přetížené funkce* resp. *operátory*. Jde o doslovný (a často používaný) překlad původních termínů *overloaded function* resp. *overloaded operator*.

Pro jazyk C budeme občas používat označení „Céčko“, neboť se s ním lépe zachází než se samotným písmenem. Podobně budeme používat přídavná jména „pascalský“, „céčkovský“, „borlandský“, „pascalista“, „céčkař“ apod., přesto, že proti nim mají někteří jazykoví korektoři výhrady.

Typografické konvence

V textu této knihy používáme následující konvence:

switch	Tučně píšeme klíčová slova.
proud	Tučně píšeme nově zaváděné termíny a také pasáže, které chceme z jakýchkoli důvodů zdůraznit.
readkey	Kurzivou píšeme identifikátory, tj. jména proměnných, funkcí, typů apod. Přitom nerozlišujeme, zda jde o jména standardních součástí jazyka (např. knihovních funkcí) nebo o jména, definovaná programátorem.
stream	Kurzivou také píšeme anglické názvy.
ALT+F4	Kapitálky používáme pro vyznačení kláves a klávesových kombinací.
`cout << x;`	Neproporcionální písmo používáme v ukázkách programů a v popisu výstupu programů.

Části výkladu, které se týkají pouze Pascalu, jsou po straně označeny jednoduchou svislou čarou.

Části výkladu, které se týkají pouze C++, jsou po straně označeny dvojitou svislou čarou.

K této knize lze zakoupit doplňkovou disketu, na níž najdete úplné zdrojové texty všech příkladů, uvedených v knize, a některých dalších, na které v knize jen odkazujeme.

1. Úvod do práce s daty

V knize *Základy algoritmizace* jsme postupně pronikali do tajů programování prostřednictvím robota Karla. Naučili jsme se používat základní algoritmické konstrukce, s jejichž pomocí jsme, někdy možná i dost krkolomným způsobem, docilovali toho, že náš robotek postavil někam značku či si naopak pro značku někam došel.

Asi jste si všimli jedné nedokonalosti, která nám při programování zapříčiňovala nemálo problémů, jež bylo nutno obcházet používáním rekurzivních procedur nebo jiných fíglů a triků. Náš robot Karel byl totiž sklerotik: neuměl si zapamatovat, kam položil značku, ba dokonce ani kde před chvilkou stál. V programu jsme si tedy nemohli žádným rozumným způsobem zapamatovat a později využít informaci o poloze robota ani o počtu značek pod ním.

Nepočítáme-li políčka Karlova dvorku, nepoužívali jsme v našich programech žádné objekty, do nichž by bylo možno tyto informace uschovávat. Při zběžném pohledu bychom tedy mohli říci, že naše programy nepracovaly s daty. Je tomu však skutečně tak? Opravdu šlo při definici nových procedur a funkcí pouze o sestavování algoritmu, který nevyužíval žádná data?

Oborová encyklopedie výpočetní a řídicí techniky (Praha, SNTL 1982) definuje data jako „**obecné označení jakýchkoliv údajů zpracovávaných programem**“. Naše programy, které jsme v předchozí části vytvářeli, pohybovaly robotem Karlem po dvorku a nechávaly jej pokládat a zvedat značky – manipulovaly tedy s daty, kterými byly robot Karel (nebo přesněji jeho pozice), značky rozestavované po dvorku i dvorek sám – přesněji řečeno jejich vnitřní reprezentace v počítači.

Nemůžeme tedy říci, že by naše dosavadní programy nepracovaly s daty. Prostředí robota Karla nám však umožnilo psát programy tak, abychom se nemuseli zabývat **vnitřní reprezentací dat** v počítači.

Umění definice vhodné vnitřní reprezentace zpracovávané skutečnosti patří k velice důležitým programátorským dovednostem. Niklaus Wirth, klasik moderního programování a autor jazyků Pascal a Modula 2, vydal na počátku osmdesátých let knížku s názvem *Algorithms + Data Structures = Programs* (její slovenský překlad *Algoritmy a štruktúry údajov*, který vyšel v nakladatelství Alfa, bohužel tento název nedodržel). Tímto názvem se Wirth snažil zdůraznit skutečnost, že umění návrhu vhodných datových struktur (tj. právě oné vnitřní reprezentace) a umění práce s nimi je při programování přinejmenším stejně důležité, jako umění návrhu vhodných algoritmů.

Postupem doby nabývá tato složka programátorských schopností stále více na důležitosti a s příchodem objektově orientovaného programování se už dokonce může leckomu zdát, že vedle návrhu optimální hierarchie datových struktur je návrh algoritmů jednoduchý úkol vhodný tak pro programátorské začátečníky.

V této knize tedy spolu uděláme prvé krůčky do světa dat a jejich vnitřní reprezentace v počítači. Nejprve si povíme obecně něco o tzv. literálech a zároveň se seznámíme se zá-

kladními příkazy pro výstup údajů na obrazovku. Dále se budeme zabývat konstantami, proměnnými a datovými typy a řekneme si o základních možnostech jejich deklarace.

Pak si vysvětlíme operaci přiřazení hodnoty proměnné a funkci základních aritmetických operátorů. V této kapitole si také napíšeme své první „smysluplné“ programy využívající explicitně práce s daty.

2. Konstanty a proměnné

2.1 Literály a výstupní operace

Každý program musí umět svému uživateli nějakým způsobem předat výsledky, ke kterým došel. V knize *Základy algoritmizace* byly těmito výsledky nové stavy Karlova dvorku a o jejich předání (tj. vykreslení nového stavu dvorku) se staral modul KAREL.

V této kapitole se seznámíme se základními možnostmi tisku údajů. Aby byl výklad co nejjednodušší, omezíme množinu tištěných dat na **literály**, což jsou konstanty, které nemají vlastní identifikátor a uvádějí se v programu přímo hodnotou. (*Literal* znamená anglicky mj. *doslovný*. Literály jsou tedy konstanty, na které se neodvoláváme jménem, ale které v programu vypisujeme doslovně.)

Všechny tisky, o nichž budeme v této kapitole hovořit, budou směřovat do standardního výstupu. To znamená, že směřují standardně na obrazovku, ale pomocí operátorů „>“ a „>>“ v příkazovém řádku programu si je můžeme přesměrovat do souboru nebo na jiné zařízení – např. tiskárnu.[1]

Než přistoupíme k vlastnímu výkladu literálů, osvěžíme si znalosti číselných soustav. Jak víte, programátoři se neomezují pouze na desítkovou soustavu, ale používají i další číselné soustavy, především dvojkovou, osmičkovou a šestnáctkovou. Název číselné soustavy odpovídá počtu stavů, kterých mohou nabývat jednotlivé řády čísel a tedy i počtu číslic, s nimiž se při zápisu čísel v dané číselné soustavě vystačí.

Dvojková soustava vystačí s číslicemi 0 a 1, osmičková s číslicemi 0 až 7. Při zápisu čísel v šestnáctkové soustavě znázorňujeme nultý až devátý stav také číslicemi a při znázornění desátého až patnáctého stavu si vypomáháme písmeny A až F resp. a až f.

Ze školy si určitě vzpomenete, že napíšeme-li v desítkové soustavě číslo 1234, znamená to, že máme 4 jednotky (to znamená 4×10^0), 3 desítky (tedy 3×10^1), 2 stovky (tedy 2×10^2) a jednu tisícovku (tedy 1×10^3).

Podobně fungují i ostatní poziční číselné soustavy, tedy také dvojková, osmičková nebo šestnáctková. Místo mocnin deseti ovšem používají mocnin svého základu, tedy dvou, osmi nebo šestnácti.

Napíšeme-li ve dvojkové soustavě 11101, myslíme tím číslo

$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 = 1 + 4 + 8 + 16 = 29.$

Podobně 1A3 v šestnáctkové soustavě znamená

$3 \times 16^0 + 10 \times 16^1 + 1 \times 16^2 = 419.$

[1] Přesměrování standardního výstupního souboru je možné pouze tehdy, spustíte-li program přímo z operačního systému. Pokud budete program provozovat pod IDE a budete chtít řídit přesměrování pomocí volby *Run|Parameters* (Pascal) resp. *Run|Arguments* (C++), nepodaří se vám to.

Místo dlouhých výkladů uvádíme tabulku zápisů prvých dvaceti čísel v jednotlivých číselných soustavách:

Zápis čísla v soustavě			
dvojkové	**osmičkové**	**desítkové**	**šestnáctkové**
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
10000	20	16	10
10001	21	17	11
10010	22	18	12
10011	103	19	13
10100	104	20	14

Tab. 2.1 Zápis prvních 20 čísel v různých číselných soustavách

Podívejme se nyní na to, jak se zapisují některé základní typy literálů a jak se používají příkazy vstupu a výstupu v každém z probíraných jazyků.

Literály a výstupní operace v Pascalu

Podívejte se nejprve na doprovodný program s různými podobami příkazu tisku a s různými podobami vystupujících literálů.

```
(*  Příklad P2 – 1  *)
{  Demonstrace použití literálů a různých možností výstupů voláním
  procedur write/writeln }
Program Pascal;                 { Hlavní program }
begin
 writeln; {Začni na dalším řádku}
 writeln( 'Tabulka možných způsobů zápisu literálů',
          ' v Pascalu:' );
 writeln( '=======================================',
          '==========' );
 writeln( 'Znak A přímo:        ', 'A', #10,
```

```
    'Znak A kódem dekadicky:   ', #65, #10,
    'Znak A kódem hexadecimálně: ', #$41 );
 writeln( 'Znak apostrofu přímo:    ', '''' );
 writeln;
 writeln( 'Text obsahující apostrof: ->''<-' );
 writeln( 'Text s vloženým ->'#9'<- tabulátorem' );
 writeln( #7'Úvodní, vložený '#7'a závěrečný zvonek'#7 );
 writeln;
 write ( 'Celé číslo dekadicky:   '); writeln( 255 );
 write ( 'Celé číslo hexadecimálně: '); writeln( $FF );
 writeln;
 writeln( 'Reálné číslo přímo:   ', 123.456, #13#10,
    'Reálné číslo "vědecky": ', 1.23456e+2,
    #13#10#13#10,
    'Logická hodnota ANO: ', TRUE, #13#10,
    'Logická hodnota NE: ', false );
 writeln( '========================================' );
end.
```

Doporučujeme, abyste si všimli zejména následující skutečnosti:

- ✧ Základním prostředkem pro výstup údajů jsou procedury *write* a *writeln*. Jediným rozdílem mezi těmito dvěma procedurami je, že *write* vyšle na výstup pouze požadovaný text, kdežto *writeln* přidá na konec vystupujícího textu znak pro přechod na nový řádek.
- ✧ Pomocí těchto procedur můžeme vytisknout i několik položek najednou, přičemž seznam vystupujících položek zapisujeme do závorek za identifikátor *write* nebo *writeln* a jednotlivé položky v tomto seznamu oddělujeme čárkami.
- ✧ Pokud použijeme pro přechod na nový řádek samotný příkaz *writeln*, tj. příkaz s prázdným seznamem vystupujících položek, nepíšeme za identifikátor *writeln* závorky.

Tento kratičký program tiskl hodnoty všech pěti druhů literálů: znaků, textů, celých a reálných (správně bychom měli říkat racionálních) čísel a logických hodnot. Probereme si je nyní postupně.

Znakové literály

- ✧ Běžné znaky zapisujeme tak, že znak, který je hodnotou daného literálu, uzavřeme mezi apostrofy.
- ✧ Řídicí znaky (znaky s kódem od 0 do 31), semigrafické znaky anebo jakékoliv jiné znaky, které bychom chtěli zadat kódem, zapíšeme tak, že daný kód napíšeme za znak # (mříž) – např. #127.
- ✧ Dáváme-li přednost hexadecimálnímu zápisu kódu, předřadíme před vlastní kód ještě znak $ (dolar) – např. #$7F.

- ✧ Znak ' (apostrof) musíme buď zadat kódem, nebo jej musíme zapsat jako dva apostrofy, tj. "".
- ✧ Přechodu na nový řádek lze dosáhnout také vysláním znaku s kódem 10 ($0A). Přesměrujete-li však výstup do souboru, který pak načtete nějakým editorem, můžete zjistit, že daný editor takto zadané přechody na nový řádek neakceptuje. Nápravy dosáhnete tak, že místo samotného znaku #10 budete posílat posloupnost #13#10.

Textové literály

- ✧ Textové literály zapisujeme obdobně jako znakové – řetězec znaků, které tvoří hodnotu literálu, uzavřeme mezi dva apostrofy – např. 'textový literál'.
- ✧ Pokud potřebujeme do textového literálu vložit apostrof, zapíšeme na daném místě dva apostrofy za sebou – např.

```
'vložíme '' apostrof'
```

- ✧ Potřebujeme-li do textového literálu vložit řídicí, semigrafický nebo jiný znak, který bychom chtěli zadat kódem, zařídíme to tak, že část textu před daným znakem ukončíme apostrofem, hned za něj (tj. bez jakýchkoliv vložených mezer) zapíšeme znak # následovaný kódem vkládaného znaku a hned za ním znovu apostrofem uvodíme následující text – např.

```
'vložíme ‘#7' zvonek'
```

- ✧ Zápis textového literálu se nám musí vždy vejít na jeden řádek.
- ✧ Hodnotou textového literálu může být i prázdný řetězec, tj. řetězec, který neobsahuje žádný znak ('').

Celá čísla

- ✧ Celá čísla zapisovaná dekadicky (tj. v desítkové soustavě) zapisujeme naprosto přirozeným způsobem, na nějž jsme z matematiky zvyklí. Nevkládáme však mezeru mezi stovky a tisíce, resp. mezi statisíce a milióny, resp. mezi stamilióny a miliardy – např. 65536, -7238.
- ✧ Pokud chceme zdůraznit kladnou hodnotu daného čísla, můžeme před něj vložit znak plus – např. +12345. Tento znak je však nepovinný.
- ✧ Chceme-li zapsat celé číslo hexadecimálně (tj. v šestnáctkové soustavě), zapíšeme znak **$** (dolar) následovaný hexadecimálním tvarem zapisovaného čísla. Číslo 127 bychom tedy zapsali jako $7F.
- ✧ Je-li číslo uvozeno znakem + (plus) nebo - (minus), vkládáme dolar mezi tento znak a vlastní hodnotu čísla, např. +$7F resp. -$7F.
- ✧ Nezávisle na číselné soustavě, v níž hodnotu celočíselného literálu zapíšeme, se tato hodnota tiskne vždy v desítkové soustavě.

Reálná čísla

- ✧ Reálná čísla zapisujeme dvěma alternativními způsoby: buď v přímém nebo semilogaritmickém (někdy se říká „vědeckém“[2]) tvaru.
- ✧ Čísla v přímém tvaru zapisujeme stejně, jak jsme z matematiky zvyklí, pouze nesmíme zapomenout, že místo desetinné čárky musíme používat **desetinnou tečku** a že, obdobně jako u celých čísel, do nich nevkládáme mezery.
- ✧ Zápis čísla v semilogaritmickém tvaru se skládá z mantisy a exponentu oddělených znakem „**e**“ nebo „**E**“. Mantisa může a nemusí mít desetinnou část a obě složky (tj. mantisa i exponent) mohou být uvozeny znaménkem (plus je nepovinné). Zápis čísla 1234.56E78 je tedy ekvivalentní matematickému zápisu $1234{,}56.10^{78}$.
- ✧ Nezávisle na způsobu zadání se číslo vytiskne vždy v semilogaritmickém tvaru. O jiných možnostech tisku reálných čísel si povíme později.

Logické hodnoty

- ✧ Literálové názvy logických hodnot jsou v Pascalu *TRUE* pro *ANO* a *FALSE* pro *NE*. (Abychom mohli v našich dosavadních programech používat pro logické hodnoty názvy *ANO* a *NE*, bylo nutno názvy *ANO* a *NE* nejprve v modulu KAREL definovat).
- ✧ Nezávisle na velikosti písmen, jimiž hodnotu logického literálu zapíšeme, se tato hodnota tiskne vždy velkými písmeny.

Literály a výstupní operace v C++

V C++ můžeme výstup informací (jako ostatně skoro vše) realizovat několika alternativními způsoby. Každý z nich má své výhody a nevýhody a tím i své aplikace, v nichž je jeho použití optimální.

Pro univerzální formátovaný výstup přibližně ekvivalentní pascalskému *write* (samozřejmě daleko flexibilnější, ale k tomu se ještě dostaneme) se používají dvě možnosti: buď klasicky (tj. stejně jako v jazyku C) pomocí funkce *printf*, nebo objektově pomocí operátoru <<.

Protože objektové řešení je pro nás z řady důvodů výhodnější, budeme je v dalším průběhu kursu preferovat. Chceme-li napsat program alespoň přibližně ekvivalentní uvedenému pascalskému, mohlo by řešení s využitím operátoru << vypadat např. následovně:

```
/*  Příklad C2 – 1  */
// Demonstrace použití literálů a různých
// možností výstupů pomocí operátoru <<

#include <iostream.h>
```

[2] Rádi bychom věděli, co je na tomto zápisu vědeckého. Pokud je nám známo, učí se v osmých třídách základních škol.

```
/**************  Hlavní program  **************/
void /*****/  main  /*****/
()
{
 cout <<   '\n';                //Začni na dalším řádku
 cout <<   "Tabulka možných způsobů zápisu literálů v C++:"
           "\n================================================";
 cout <<   "\nZnak A přímo:        " << 'A';
 cout <<   "\nZnak A kódem oktalově:   " << '\101';
 cout <<   "\nZnak A kódem hexadecimálně: " << '\x41';
 cout <<   "\nZnak apostrofu přímo:    " << '\'';
 cout <<   '\n';
 cout <<   "\nText obsahující uvozovky: ->\"<-"
       "\nText s vloženým ->\x9<- tabelátorem"
       "\n\aÚvodní, vložený \7a záveřečný zvonek\a"
       "\n"                     //Až zde končil posílaný text
   <<  "\nCelé číslo dekadicky:   " << 255
   <<  "\nCelé číslo oktalově:   " << 0377
   <<  "\nCelé číslo hexadecimálně: " << 0xFF
   <<  endl                     //Jiný způsob přechodu na další řádek
   <<  "\nReálné číslo přímo:   "  << 123.456
   <<  "\nReálné číslo \"vědecky\": " << 1.23456e+2
   <<  "\n=========================================\n";
}
```

V uvedeném programu si všimněte následujících věcí:

- ✧ Pokud chceme používat operátor **<<**, musíme jej nejprve zpřístupnit, tj. dovézt. Toho dosáhneme tak, že do zdrojového programu vložíme hlavičkový soubor *iostream.h*.
- ✧ Operátor použijeme tak, že vlevo od něj napíšeme jméno tzv. **výstupního proudu** (časem si vysvětlíme podrobněji, co to je), kterým je v našem případě standardní výstup, jenž se podle konvencí jmenuje *cout* (*C++ output*). Vpravo od operátoru pak napíšeme vystupující hodnotu.
- ✧ Operátor **<<** je definován tak, že jej můžeme (obdobně jako např. + v matematických výrazech) řetězit, tzn., že za vystupující hodnotou napíšeme znovu symbol operátoru a za něj další vystupující hodnotu. Tyto zřetězené operátory se vyhodnocují (a tedy hodnoty tisknou) zleva doprava.

Náš kratičký program tiskl hodnoty čtyř druhů literálů: znaků, textů, celých a reálných (správně bychom měli říkat racionálních) čísel. Logické hodnoty v C++ samostatně nevystupují. Probereme si je nyní postupně.

Znakové literály

- ✧ Běžné znaky zapisujeme tak, že znak, který je hodnotou daného literálu, uzavřeme mezi apostrofy.
- ✧ Řídicí znaky (znaky s kódem od 0 do 31), semigrafické znaky anebo jakékoliv jiné znaky, které bychom chtěli zadat kódem, zapíšeme tak, že daný kód napíšeme v os-

mičkové (oktalové) soustavě za znak \(obrácené lomítko). Tedy např. znak s kódem 127 (oktalově 177) zapíšeme \177. Smíme zapsat nejvýše třímístný kód, to znamená, že můžeme zapsat \001, ale nemůžeme již zapsat \0177 – to by překladač interpretoval jako znak s osmičkovým (oktalovým) kódem 17 (dekadicky 15), za nímž následuje znak '7'.

✧ Dáváme-li přednost hexadecimálnímu zápisu kódu, vložíme mezi obrácené lomítko a vlastní kód ještě znak *x* nebo *X*. Znak s kódem 127 (šestnáctkově 7F) zapíšeme \x7F nebo \x7f nebo \X7F nebo \X7f. Jak vidíte – velikost písmen se při zápisu hexadecimálních čísel nerozlišuje – budeme v kursu preferovat první z uvedených způsobů zápisu.

✧ Obrácené lomítko je znak, který má zvláštní význam: uvozuje tzv. řídicí posloupnosti (*escape sequence*), pomocí nichž zapisujeme některé často používané řídicí a pomocné znaky. Jejich přehled je uveden v tabulce 2.

Řídicí posloupnost	Kód znaku dek	hex	Význam
\a	7	0x07	BEL – Zvukové znamení (*alert*)
\b	8	0x08	BS – Návrat o znak zpět (*backspace*)
\f	12	0x0C	FF – Nová stránka (*form feed*)
\n	10	0x0A	LF – Nový řádek (*line feed*)
\r	13	0x0D	CR – Přesun na počátek řádku (návrat vozíku – *carriage return*)
\t	9	0x09	HT – Horizontální tabulátor
\v	11	0x0B	VT – Vertikální tabulátor
\ooo		0ooo	O = posloupnost až tří oktalových číslic (viz bod 2)
\xhh, \Xhh		0xhh	hh = posloupnost hexadecimálních číslic libovolné délky
\\	92	0x5c	\ (Obrácené lomítko)
\'	39	0x27	' (Apostrof)
\"	34	0x22	" (Uvozovky)

Tab. 2.2 Řídicí posloupnosti

Pokud za zpětným lomítkem nenásleduje žádný ze znaků

```
a b f n r t v 0 1 2 3 4 5 6 7 x X
```

zastupuje následující znak sám sebe. Toho se využívá zejména při zápisu obráceného lomítka, uvozovek a apostrofu.

Textové literály

- ✧ Textové literály zapisujeme mezi uvozovky – např.

  ```
  "textový literál"
  ```

  Jednotlivé znaky přitom můžeme zapisovat jak přímo, tak pomocí řídicích posloupností – např.

  ```
  "první řádek\ndruhý řádek"
  ```

- ✧ Dva a více textových literálů, mezi nimiž jsou ve zdrojovém textu pouze bílé znaky, považuje překladač za jediný literál vzniklý prostým spojením uvedených konstant.
- ✧ Zápis textového literálu by se nám měl vždy vejít na jeden řádek. Pokud chceme zadat literál delší, máme dvě možnosti: buď (horší možnost) zapsat jako poslední znak literálu na daném řádku obrácené lomítko a pokračovat od počátku dalšího řádku, nebo (lepší možnost) využít informace z předchozího bodu a rozdělit literál do několika literálů kratších, z nichž každý se vejde na jeden řádek. Tuto druhou možnost používá i náš program.
- ✧ Jak bylo uvedeno v tabulce 2, vkládáme-li znak definovaný jeho hexadecimálním kódem, očekává překladač za obráceným lomítkem a následným znakem „x“ posloupnost hexadecimálních číslic libovolné délky. Pokud v textu za posloupností následuje znak, který by mohl překladač chápat za hexadecimální číslici, doporučuje se tuto kolizi řešit roztržením řetězce – viz demonstrační program při vkládání kódu zvukového znamení.
- ✧ Hodnotou textového literálu může být i prázdný řetězec, tj. řetězec, který neobsahuje žádný znak ("").

Celá čísla

- ✧ Celá čísla zapisovaná dekadicky (tj. v desítkové soustavě) zapisujeme naprosto přirozeným způsobem, na nějž jsme z matematiky zvyklí. Nevkládáme však mezeru mezi stovky a tisíce resp. mezi statisíce a milióny resp. mezi stamilióny a miliardy – píšeme např. 65536, -7238.
- ✧ Zapíšeme-li číslo tak, že jeho prvním znakem bude nula (např. 013), chápe je překladač jako zápis čísla v oktalové (osmičkové) číselné soustavě!
- ✧ Chceme-li zapsat celé číslo hexadecimálně (tj. v šestnáctkové soustavě), zapíšeme před hexadecimálním tvarem zapisovaného čísla posloupnost 0x resp. 0X. Jak jsme si již řekli u znaků, velikost písmen se při zápisu hexadecimálních čísel nerozlišuje. Číslo 127 bychom tedy mohli zapsat jako 0x7F nebo 0x7f nebo 0X7F anebo 0X7f – v kursu budeme preferovat první z uvedených způsobů zápisu.
- ✧ Nezávisle na číselné soustavě, v níž hodnotu celočíselného literálu zapíšeme, se tato hodnota tiskne vždy v desítkové soustavě. O možnostech tisku hodnoty čísel v jiných číselných soustavách si povíme později.

Reálná čísla

1. Reálná čísla zapisujeme dvěma alternativními způsoby: buď v přímém nebo semilogaritmickém tvaru.
2. Čísla v přímém tvaru zapisujeme stejně, jak jsme z matematiky zvyklí, pouze nesmíme zapomenout na to, že místo desetinné čárky musíme používat **desetinnou tečku** a že, obdobně jako u celých čísel, do nich nevkládáme mezery.
3. Zápis čísla v semilogaritmickém tvaru se skládá z mantisy a exponentu, oddělených znakem „**e**“ nebo „**E**“. Mantisa může a nemusí mít desetinnou část a obě složky (tj. mantisa i exponent) mohou být uvozeny znaménkem (plus je nepovinné). Zápis čísla 1234.56E78 je tedy ekvivalentní matematickému zápisu $1234{,}56.10^{78}$.
4. Nezávisle na způsobu zadání se čísla, jejichž absolutní hodnota je z intervalu $<10^{-4}, 10^{8})$, vytisknou vždy v přímém tvaru a čísla, jejichž absolutní hodnota je mimo tento rozsah, se vytisknou vždy v semilogaritmickém tvaru. O jiných možnostech tisku reálných čísel si povíme později.

2.2 Deklarace konstant, proměnných a datových typů

Jedním z charakteristických znaků všech moderních programovacích jazyků je povinnost deklarovat všechny v programu používané objekty nejpozději při jejich prvním použití. Jak jsme si již řekli u funkcí, prostřednictvím deklarace daného objektu oznamujeme překladači vlastnosti objektu, který se chystáme používat. Na základě těchto informací překladač může kontrolovat, zda některé naše požadavky nejsou v rozporu s deklarovanými vlastnostmi objektu. Kromě toho mu tyto informace umožňují optimalizovat manipulaci s objektem.

Poznámka:
Jak víte, pravidla mívají své výjimky. V jazyku C++ je takovou výjimkou návěští, které smíme v programu použít dříve, než je deklarujeme. Základní vlastnosti všech návěští jsou totiž beztak stejné a vlastní umístění návěští dokáže překladač zapracovat do programu i poté, co jsme je již použili.

Jako důkaz nebezpečnosti jazyků, které netrvají na povinné deklaraci všech použitých objektů, bývá v učebnicích moderního programování uváděn případ jednoho rozsáhlého programu v jazyku FORTRAN, použitého v NASA při kosmickém výzkumu. V jednom místě programu měl být původně příkaz

```
DO 30 I=1, 5
```

který oznamoval, že úsek programu mezi tímto příkazem a příkazem označeným návěštím 30 se má provádět pětkrát, přičemž celočíselná proměnná *I* má v jednotlivých průchodech nabývat hodnoty 1, 2, 3, 4, 5.

V té době se programy ještě zaznamenávaly na děrné štítky; děrovačka, která to dostala na starost, věděla, že ve FORTRANU (alespoň ve verzi, která se v té době používala) není třeba psát mezery, a tak napsala

```
DO30I=1.5
```

Našli jste chybu? (Podotýkáme, že nespočívá ve vynechání mezer.) Předpokládáme, že jste ji přehlédli, stejně jako programátoři z NASA, kteří ji začali hledat až ve chvíli, kdy družice přestala směřovat tam, kam měla.

Jediná tečka zapsaná místo čárky způsobila, že překladač pochopil daný řádek jako příkaz přiřadit proměnné DO30I hodnotu 1.5. Nikde žádné chybové hlášení, nikde žádné varování. Nicméně družice se stále zřetelněji odchylovala z dráhy a vyhrožovala svým poměrně rychlým koncem.

A teď zkuste v nastalém stresu hledat chybu, kterou lze velice těžko najít i ve výpisu z tiskáren určených pro DTP, natož pak z běžné tiskárny ve výpočetním středisku, jehož obsluha ani v USA s výměnou barvicí pásky příliš nepospíchá.

Vraťme se ale k deklaracím. První věcí, která bude překladač u každého objektu zajímat, je informace o tom, zda může daný objekt měnit po dobu běhu programu svoji hodnotu. Podle tohoto hlediska pak budeme datové objekty rozdělovat na **konstanty** a **proměnné**.

Další údaje, po nichž bude překladač toužit, jsou informace o tom, jakých hodnot může daný objekt nabývat a co všechno s ním může dělat. Tyto dvě charakteristiky jsou dány tzv. **typem** dotyčného objektu. Chceme-li tedy v programu deklarovat nějakou konstantu nebo proměnnou, musíme v deklaraci uvést její datový typ.

Podívejme se nyní, jak se v našich dvou jazycích konstanty, proměnné a datové typy deklarují. Začneme Pascalem.

Deklarace v Pascalu

Deklarace a definice se v pascalských programech (přesněji v modulech obsahujících hlavní program) zapisují mezi příkaz **Program** resp. **Uses** a hlavní program.

V jednotkách (tj. modulech neobsahujících hlavní program) platí trochu složitější pravidla. Deklarace vyvážených procedur a funkcí deklarace a definice ostatních vyvážených objektů se zde uvádějí v části **interface** (tj. mezi klíčovými slovy **interface** a **implementation**) a deklarace a definice ostatních objektů spolu s definicemi vyvážených funkcí se uvádějí v části **implementation** (tj. mezi klíčovým slovem **implementation** a inicializační částí modulu).

Kromě toho se mohou deklarace vyskytovat i v definici libovolné procedury a funkce, a to mezi její hlavičkou a tělem, tj. před klíčovým slovem **begin**. Zde se mohou vyskytnout jakékoliv deklarace, tedy i deklarace (a definice) procedur a funkcí, které se pak stanou „soukromým majetkem“ podprogramu, v němž byly definovány. K tomu se ale ještě časem vrátíme.

Objekty, které nejsou deklarovány uvnitř procedur a funkcí, můžete používat všude počínaje místem deklarace, tzn. ve všech následujících podprogramech i v případném hlavním programu nebo v inicializaci. Z hlediska podprogramů jsou tyto objekty **globální**.

Naproti tomu objekty, které jsou deklarovány uvnitř podprogramů, jsou **lokálními** objekty podprogramů, tzn. že o nich okolní podprogramy nic nevědí a nemohou je tedy používat. K této problematice se ještě podrobněji vrátíme v samostatné kapitole.

Návěští, konstanty, proměnné, datové typy, procedury a funkce se v Pascalu deklarují (a definují) v samostatných sekcích. Původní Pascal předepisoval přesné pořadí těchto sekcí (odpovídá pořadí, v němž jsme jednotlivé druhy objektů vyjmenovali), Turbo Pascal však již na tomto pořadí netrvá a dokonce nám povoluje uvádět jednotlivé sekce v deklaracích vícekrát.

Každá sekce s výjimkou sekce deklarací a definic procedur a funkcí je v programu uvedena klíčovým slovem, které určuje druh deklarovaných a definovaných objektů. Sekce deklarací návěští je uvozena klíčovým slovem **label**, sekce deklarací konstant je uvozena klíčovým slovem **const**, sekce deklarací datových typů je uvozena klíčovým slovem **type** a sekce deklarací proměnných je uvozena klíčovým slovem **var**. Jak jsme si již řekli, sekce deklarací a definic procedur a funkcí není uvozena žádným speciálním klíčovým slovem. Není to totiž potřeba, protože jednotlivé deklarace a definice procedur, resp. funkcí jsou uvozeny klíčovým slovem **procedure**, resp. **function**. Procedury a funkce již deklarovat i definovat umíme a při výkladu příkazu **goto** jsme se naučili deklarovat také návěští. Nyní si ukážeme, jak se deklarují (a tím zároveň i definují) konstanty a proměnné.

Deklarace typů

Jak víme, je součástí definice většiny programovacích jazyků i několik předdefinovaných datových typů. Ostatní datové typy musí programátor specifikovat sám.

Základním předdefinovaným typem je v téměř všech programovacích jazycích typ celých čísel. Tento typ se v Pascalu nazývá *integer*.[3]

Druhým základním předdefinovaným datovým typem, s nímž bychom se měli před dalším výkladem alespoň ve stručnosti seznámit, je typ reálných čísel. Tento typ má, stejně jako typ celých čísel, řadu nuancí, o nichž si podrobně povíme v samostatné kapitole. Prozatím postačí, když si budeme pamatovat, že základní „reálný“ datový typ se v Pascalu jmenuje *real*.

V kapitole o literálech jsme se seznámili s typem pro vyjadřování znaků, který se v Pascalu jmenuje *char*, a s typem pro textové řetězce, který se jmenuje *string*. Musíme si však dát pozor na to, že pascalské textové řetězce nesmějí být delší než 255 znaků.

[3] Označení předdefinovaných datových typů nejsou v Pascalu klíčová slova, ale vyhrazené identifikátory.

Posledním základním předdefinovaným typem, s nímž budeme v této kapitole pracovat, je typ logických hodnot, se kterým jsme se setkali již při výkladu funkcí. Z těch dob také víme, že se tento typ v Pascalu nazývá *boolean*.

S dalšími předdefinovanými datovými typy se budeme seznamovat postupně.

Nové datové typy deklarujeme a definujeme v Pascalu tak, že v sekci deklarací datových typů napíšeme identifikátor nově deklarovaného typu, za něj rovnítko a popis (definici) tohoto typu. Jednotlivé deklarace ukončujeme středníkem.

V tuto chvíli se seznámíme pouze se dvěma možnými druhy definic: s přejmenováním datového typu a zúžením oboru hodnot (definicí intervalu).

Při **přejmenování datového typu** vystupuje jako definice typu původní jméno přejmenovávaného datového typu. Přejmenovaný datový typ je s původním typem po všech stránkách kompatibilní a jsou překladačem považovány za identické.

Při **definici intervalu**, tedy při zúžení oboru hodnot, definujeme nový datový typ tak, že napíšeme konstantu označující nejmenší povolenou hodnotu objektů nového typu, dvě tečky a konstantu označující nejvyšší povolenou hodnotu objektů nového typu. Dvě tečky, použité při vymezení rozsahu, jsou považovány za nedělitelný symbol, takže mezi nimi nesmí být bílý znak. Budeme je označovat jako operátor rozsahu.

Jazyk Pascal umožňuje definovat intervaly pouze z pořadových (ordinálních) typů. Z typů, o kterých jsme dosud hovořili, mezi ně patří celá čísla (typ *integer* a další, které poznáme časem), znaky (typ *char*) a logické hodnoty (typ *boolean*).

```
(*   Příklad P2 – 2   *)
Type
  CelE = integer;                    {Počeštění názvu typu}
  int = Cele;                        {Totéž, jako "int = integer"}
      {Typy integer, int a Cele považuje překladač za totožné}
  bool = boolean;                    {Přejmenování typu}
  Bajt = 0 ..255;                    {interval z celých čísel}
```

Deklarace konstant

Při deklaracích konstant v Pascalu se jejich typ neuvádí. Typ konstanty si překladač „domyslí“ podle přiřazované hodnoty. To má své výhody i nevýhody: výhodou je, že se deklarace konstant zjednoduší, nevýhodou naopak je, že si musíme dávat větší pozor na překlepy při zápisu programu.

Konstanty deklarujeme a zároveň definujeme tak, že v sekci deklarací konstant napíšeme identifikátor konstanty, rovnítko a přiřazovanou (inicializační) hodnotu. (Všechny tři části můžeme oddělit libovolným počtem bílých znaků.) Jako inicializační hodnotu můžeme použít i výraz. Tento výraz však musí umět vyhodnotit už překladač, takže v něm smíme používat vlastně jen literály a dříve definované konstanty.

Jednotlivé deklarace ukončujeme středníkem.

Abychom mohli v textu snadno rozlišit konstanty od ostatních objektů jazyka, dohodneme se, že jejich identifikátory budeme psát velkými písmeny.

```
(*   Příklad P2 – 3   *)
const                                    {Sekce deklarace konstant}
  POCET = 10;                            {Celočíselná konstanta}
  PI = 3.1415926;                        {Reálná konstanta}
  PI_2 = PI / 2;
  ESC = #$1B;                            {Znak Escape}
  NL = #13#10;                           {Přechod na novou řádku}
  KURS = 'Cesta k profesionalitě';
  ANO = TRUE;                            {Logická konstanta}
  NE = FALSE;                            {Logická konstanta}
```

Deklarace proměnných

Proměnné deklarujeme (a definujeme) dvěma způsoby, které se liší podle toho, potřebujeme-li danou proměnnou v deklaraci inicializovat (tj. přiřadit jí počáteční hodnotu) nebo ne.

Neinicializované proměnné, tj. proměnné, kterým nepotřebujeme přiřadit počáteční hodnotu hned v deklaraci, deklarujeme v sekci **var**, a to tak, že napíšeme jejich identifikátor následovaný dvojtečkou a specifikací daného datového typu, tedy jeho identifikátorem. (Časem se naučíme další způsoby specifikace.)

Pokud potřebujeme deklarovat více proměnných stejného typu, můžeme před dvojtečku a identifikátor jejich typu napsat seznam identifikátorů deklarovaných proměnných oddělených čárkami (a případnými bílými znaky).

Jednotlivé deklarace ukončujeme středníkem.

```
(*   Příklad P2 – 4   *)
var                    {Deklarace neinicializovaných proměnných}
  Var1: integer;       {Proměnná typu integer}
  Prom2: Cele;         {Proměnná typu Cele}
  i1, i2, i3: int;     {Tři proměnné typu int}
  B: Bajt;             {Proměnná typu Bajt}
```

Inicializované proměnné, tj. proměnné, kterým přímo v deklaraci přiřazujeme počáteční hodnotu, musíme deklarovat v sekci konstant. (Zní to jako strašný nesmysl, ale opravdu jsme střízliví, je tomu tak.)

Inicializované proměnné deklarujeme tak, že za identifikátor deklarované proměnné napíšeme dvojtečku, za ní specifikaci příslušného datového typu, pak rovnítko a za něj výraz, po jehož vyhodnocení obdržíme přiřazovanou počáteční hodnotu. Celou deklaraci ukončíme jako obvykle středníkem.

Na výraz definující přiřazovanou hodnotu jsou kladena stejná omezení, jako na výraz definující hodnotu konstant. To znamená, že v něm mohou vystupovat pouze literály a dosud deklarované konstanty.

Pozor! Nepleťte si konstanty a inicializované proměnné – i když deklarujeme inicializované proměnné v sekci deklarací konstant, nejsou to konstanty.

A ještě jednou: **Pozor! Ve starších překladačích Turbo Pascalu je chyba!** Pokud se spletete a inicializované proměnné se pokusíte přiřadit jako počáteční hodnotu jinou inici-

alizovanou proměnnou (samozřejmě dříve deklarovanou), překladač vás na tuto chybu neupozorní a deklarovanou proměnnou inicializuje nějakým smetím – nejčastěji nulou. Teprve Borland Pascal 7.0 tuto chybu odstranil. Můžete se o tom přesvědčit na následujícím programu:

```
(*   Příklad P2 – 5   *)
Program ChybaVDeklaraciInicializovanychPromennych;
type
  Cele = integer;
  Bajt = 0 .. 255;
const
  NL = #13#10;
  iVar1 : integer = 1;
  iProm2 : Cele = 2;
  iB : Bajt = $FF;
  iVar2 : integer = iVar1;           {Tady je ta chyba}
{Proměnná se ve skutečnosti inicializuje
 nějakým smetím – nejspíše nulou }
begin
     writeln( NL,
                'iVar1 = ', iVar1, NL,
                'iProm2= ', iProm2, NL,
                'iB = ', iB, NL,
                'iVar2 = ', iVar2 );
end.
```

Poznamenejme, že tento program odmítne překladač Borland Pascalu verze 7.0 přeložit a v řádce, obsahující chybnou inicializaci, oznámí *Cannot evaluate this expression* (tento výraz nelze vyhodnotit).

Všechny vysvětlované typy deklarací jsou souhrnně předvedeny v programu, který najdete v souborech *P2–0203A.PAS* (hlavní program) a *P2–0203U.PAS* (jednotka s deklaracemi) na doplňkové disketě.

Deklarace v C++

Modul je v C++ tvořen posloupností deklarací a definic. Na rozdíl od Pascalu se však v C++ deklarace považuje za příkaz, a proto se mohou deklarace vyskytovat v programu všude tam, kde se může vyskytovat příkaz.

Objekty, které nejsou deklarovány uvnitř funkcí, můžete používat od místa deklarace do konce souboru. Z hlediska funkcí jsou tedy tyto objekty **globální**.

Naproti tomu objekty, které jsou deklarovány uvnitř funkcí, jsou **lokální** v tom bloku (složeném příkazu), kde jsou deklarovány. Okolní funkce (a v případě objektů deklarovaných v nějakém složeném příkazu dokonce ani okolí daného složeného příkazu) o nich nic nevědí a nemohou je tedy používat. K této problematice se ještě podrobněji vrátíme v samostatné kapitole.

Deklarace netvoří v C++ žádné sekce; každá deklarace s sebou nese dostatek informací pro to, aby ji překladač správně identifikoval a provedl.

Deklarace typů

Základním předdefinovaným datovým typem je i v C++ typ celých čísel, který je zde označován klíčovým slovem **int**. Tento datový typ je natolik fundamentální, že jej v mnoha případech vůbec nemusíte uvádět a překladač si jej doplní sám.

Kromě typu **int** definuje jazyk ještě řadu dalších celočíselných typů, které si později podrobně probereme ve zvláštní kapitole.

Druhým předdefinovaným typem, který budeme v této kapitole používat, je typ reálných čísel. Také reálných typů je více a podrobně se s nimi seznámíme v samostatné kapitole. Prozatím postačí, když si budeme pamatovat, že základní „reálný“ datový typ se v C++ jmenuje **double**.

V oddílu o literálech jsme se dále seznámili s typem pro vyjadřování znaků; tento typ se v C++ jmenuje **char**. Pro práci s textovými řetězci budeme používat typ, který se jmenuje **char*** (mezi klíčovým slovem **char** a hvězdičkou mohou být bílé znaky). Na rozdíl od Pascalu je v C++ délka textových řetězců omezena pouze dostupnou pamětí. (Na PC to znamená velikostí segmentu, což je v reálném režimu 64 KB.)

Typ **char** je plně kompatibilní s typem **int** (je to v C++ celočíselný typ).

Starší verze C++ (borlandské překladače až po verzi 4.52) nerozlišují celá čísla a logické hodnoty. Nulová hodnota se zároveň chápe jako logické NE a jakákoliv nenulová hodnota jako logické ANO. Má-li logická funkce vrátit hodnotu ANO, bývá dobrým zvykem vracet 1.

Na rozdíl od Pascalu neexistuje v C++ možnost definovat interval, tedy omezit rozsah datového typu. Datové typy však lze přejmenovat.

Jednou z možností, jak lze v C++ definovat nové datové typy, je použít klíčového slova **typedef** následovaného specifikací nového typu (jedinou možností, kterou zatím známe, je uvést jméno známého datového typu) a identifikátorem nově definovaného datového typu. Celá definice končí středníkem.

Datový typ vzniklý přejmenováním považuje překladač za totožný s typem původním.

```
/*   Příklad C2 – 2   */
typedef int Cele;     //Definice datového typu Cele
typedef int Bool;     //C++ nerozlišuje celá čísla a logické hodnoty
```

ANSI C++ zavádí pro logické hodnoty typ **bool** (setkáme se s ním v překladačích Borland C++ 5.0 a pozdějších). Tento typ má dvě hodnoty, vyjádřené klíčovými slovy **true** a **false**. Typ **bool** je plně kompatibilní s celými čísly.

Deklarace konstant

Konstanty v C++ deklarujeme (a zároveň definujeme) tak, že celou deklaraci uvedeme klíčovým slovem **const**, za ním specifikujeme datový typ následovaný identifikátorem konstanty, rovnítkem a výrazem definujícím hodnotu konstanty. Celou definici ukončíme středníkem.

Pokud je konstanta typu **int**, můžeme jeho specifikaci vynechat.

C++ rozlišuje globální a lokální konstanty. Hodnotu globálních konstant musí umět vyhodnotit již překladač. Z toho plyne, že v definičním výrazu mohou vystupovat pouze literály, konstanty, kterým jsme již přiřadili hodnotu, a funkce, které jsme již alespoň deklarovali. (Jejich definice mohou ve zdrojovém textu klidně následovat až za výrazem, v němž je použijeme, nebo mohou být v jiném souboru nebo dokonce i v knihovně.)

V definici lokální konstanty můžeme použít jakýkoli výraz, kompatibilní vzhledem k přiřazení s typem definované konstanty.

Abychom mohli v textu snadno rozlišit konstanty od ostatních objektů jazyka, dohodneme se, že jejich identifikátory budeme psát velkými písmeny.

```
/*   Příklad C2 – 3   */
int Init();                // Deklarace funkce vracející hodnotu typu
  int
int pif = Init();          // Proměnná inicializovaná funkcí
const K = Init() + 2;      // Konstanta typu int
const Cele J = pif * K;    // Konstanta typu Cele
const double M_PI = 3.14159265358979323846;
const double M_PI_2 = M_PI / 2;
const char ESC = 0x1B;     // Kód lze zadat i jako celé číslo
const char NL = '\n';      // Přechod na novou řádku
const char* KURS = "Cesta k profesionalitě"
const ANO = 1;             // Tyto dvě konstanty
const bool NE = 0;         // jsou stejného typu,
// protože přejmenovaný typ je s původním kompatibilní

int Init()                 //Definice funkce Init
{
  return 3 ;               //Vrací hodnotu 3
}
```

Deklarace proměnných

Proměnné v C++ deklarujeme tak, že napíšeme specifikaci typu deklarované proměnné následovanou jejím identifikátorem. Pokud chceme proměnné přiřadit počáteční hodnotu, napíšeme za identifikátor deklarované proměnné rovnítko a za ně výraz, definující inicializační hodnotu. Pro tento inicializační výraz platí stejná omezení, jako pro výraz definující hodnotu konstanty. (Jsou tedy jiná pro globální a jiná pro lokální proměnné.) Celou deklaraci ukončíme středníkem.

Chceme-li zestručnit deklaraci několika proměnných stejného typu, můžeme za specifikací typu uvést seznam identifikátorů deklarovaných proměnných oddělených čárkami.

Pokud chceme některé z proměnných v seznamu inicializovat, zapíšeme do seznamu místo samotného identifikátoru identifikátor následovaný rovnítkem a inicializačním výrazem.

```
/*   Příklad C2 – 4   */
// používáme typy, deklarované v C2 – 3
int i, j=5, k, l=j;
Celé c;
int m = j;
bool Hotovo = NE;
char* Nadpis = "1. lekce";
```

Všechny vysvětlované typy deklarací jsou souhrnně předvedeny v doprovodném programu.

Program pro C++ je jednodušší než program pro Pascal, protože se nám zdálo, že na vás při používání dosud probraných konstrukcí čeká méně záludností. Připravte se však na to, že se zanedlouho situace obrátí.

Poznámka:
Typ **char*** *v jazyku C++ není přesným ekvivalentem pascalského typu* string.[4] *Jeho odlišné vlastnosti jsou v mnoha případech výhodné, v řadě případů by však byly vlastnosti pascalského typu* string *výhodnější. Proto se v C++ definuje jeho ekvivalent* – String.
Aby bylo možno tento ekvivalent používat se stejným komfortem, s jakým můžete používat pascalské textové řetězce, je nutno jej definovat s využitím objektově orientovaných rysů jazyka.
Prozatím si tedy zapamatujte, že ne vše, co je možné dělat s řetězci v Pascalu, je možné i v C++ (a naopak), a používejte proto pouze ty operace a obraty, které pro daný jazyk výslovně uvedeme – ostatní jen na vlastní riziko. (Nemělo by se však stát nic horšího, než to, že budete muset resetovat počítač.)

[4] Proměnná typu *string* v Pascalu opravdu obsahuje textový řetězec – jeden znak vedle druhého. Proměnná typu **char*** však obsahuje pouze odkaz na jiné místo v paměti, ve kterém je řetězec doopravdy uložen. Přesto budeme zatím pro jednoduchost o typu **char*** hovořit jako o typu řetězců.

3. Jednoduché výrazy

3.1 Operace přiřazení

Proměnné dostaly svůj název od toho, že se jejich hodnota v průběhu programu mění. Abychom toho mohli dosáhnout, musíme umět proměnné přiřadit novou hodnotu. K tomu slouží **přiřazovací operátor**, který je v Pascalu identifikován symbolem **:=** (dvojtečka a rovnítko) a v C++ symbolem= (samotné rovnítko).

Teď budeme chvilku mluvit učeně, ale pokuste se to strávit:

Operátor přiřazení je infixový, to znamená, že se zapisuje mezi své operandy.

Na levé straně přiřazovacího operátoru musí být tzv. **l-hodnota**. Tímto názvem označujeme něco, co určuje místo v paměti, na které můžeme přiřazenou hodnotu uložit. Z prvků jazyka, které zatím známe, můžeme prozatím použít jako l-hodnotu pouze identifikátor proměnné. (Název l-hodnota, anglicky *l-value*, vznikl z toho, že smí stát vlevo od přiřazovacího operátoru.)

Na pravé straně operátoru pak musí být výraz (někdy se používá termín r-hodnota), jehož typ je **kompatibilní vzhledem k přiřazení** s typem objektu stojícího vlevo.

O datových typech vzniklých přejmenováním víme, že je překladač považuje za totožné a jsou tedy zákonitě kompatibilní vzhledem k přiřazení. Kompatibilní vzhledem k přiřazení jsou i datové typy vzniklé zúžením rozsahu, avšak musíme dát pozor, aby přiřazovaná hodnota spadala do rozsahu datového typu cílové proměnné.

V Pascalu lze navíc přiřazovat reálným proměnným i celočíselné hodnoty, avšak tím kompatibilita typů vzhledem k přiřazení končí.

Možnosti C++ jsou mnohem širší. Navzájem kompatibilní vůči přiřazení jsou tu s +výjimkou textových řetězců všechny doposud probrané datové typy.

O tom, že C++ pracuje s logickými hodnotami jako s celými čísly, jsme si již řekli. Jak již možná tušíte, přiřadíme-li znakové proměnné celé číslo, přiřazujeme jí kód daného znaku, a naopak, přiřazujeme-li celočíselné proměnné hodnotu znaku, přiřadí se jí hodnota jeho kódu.

Každý z vás asi správně odhadne výslednou hodnotu reálné proměnné, které přiřadíme celé číslo nebo znak. Ne tak jednoznačné je to ve chvíli, kdy celému číslu nebo znaku přiřazujeme hodnotu reálné proměnné. V tomto případě se přiřadí **celá část** přiřazovaného reálného čísla, přičemž kompilátor nehlídá, zda se přiřazovaná hodnota vejde do rozsahu cílové proměnné či nikoliv. To si musí ohlídat programátor.

V Pascalu se operace přiřazení chápe jako příkaz – přesněji jako **přiřazovací příkaz**. Nevýhodou tohoto řešení je to, že přiřazení nemůžeme řetězit a nemůžeme je ani použít na místě, kde bychom potřebovali výraz. Výhodou je naopak to, že programátor nemůže nadměrným využíváním možnosti přiřazení hodnoty v rámci vyhodnocování výrazů znepřehlednit program.

V Pascalu může stát na levé straně přiřazovacího příkazu také identifikátor funkce. Takovéto přiřazení se může vyskytnout jen v těle funkce a definuje hodnotu, kterou funkce

vrací – tedy její výsledek. V následujícím příkladu najdeme jak „obyčejné“ přiřazení tak i přiřazení funkční hodnoty.

```
(*  Příklad P3 – 1  *)
var a:integer;
function (*****) Hodne (*****) : integer;
begin
  Hodne := 32767;
end;
(***** Hodne *****)

procedure (*****) Prirad (*****);
var real c;
begin
  a := Hodne;
  c := Hodne;
  {Nebo nechceme-li, aby se funkce Hodne volala dvakrát }
  a := Hodne;
  c := a;
end;
(***** Prirad *****)
```

V C++ představuje operace přiřazení výraz (přesněji **přiřazovací výraz**), ze kterého vytvoříme příkaz stejně, jako z jakéhokoliv jiného výrazu – připojením závěrečného středníku. Hodnotou přiřazovacího výrazu je přiřazená hodnota.

Toto řešení přináší dvě výhody: za prvé můžeme operace přiřazení řetězit a přiřadit tak danou hodnotu několika proměnným a za druhé může být přiřazení vedlejším efektem vyhodnocování výrazů. (Jinými slovy: přiřazení můžeme zapsat na místě, kde se podle syntaktických pravidel očekává výraz. Možnost zřetězení je pouze speciálním případem takového vedlejšího efektu.)

Zřetězená přiřazení se vyhodnocují zprava doleva. Vezměme např. výraz

```
a = b = c = 17.5
```

ve kterém jsou *a* a *b* proměnné typu **int** a *c* je typu **double**. Zde se nejprve provede přiřazení

```
c = 17.5
```

po něm se hodnota tohoto výrazu (17.5) přiřadí proměnné *b*. Protože přiřazujeme hodnotu typu **double** proměnné typu **int**, „odsekne“ se zlomková část přiřazované hodnoty a do proměnné *b* se uloží 17. Nakonec se hodnota výrazu přiřazujícího hodnotu proměnné *b* (tedy 17) přiřadí proměnné *a*.

V příručkách jazyka C a C++ se většinou místo termínu **pořadí vyhodnocování** používá termín **asociativita** (česky sdružování). Asociativita zprava doleva tedy říká, že překladač operátory sdružuje v pořadí zprava doleva. V našem případě

```
a = (b = (c = 17))
```

V souvislosti s přiřazením bychom se měli ještě zmínit o **přiřazení funkční hodnoty**, jejíž skutečný tvar nám prostředky jazyka C++ umožnily doposud skrývat.

Jak víme, návrat z procedury (nebo chcete-li funkce vracející **void**) způsobíme příkazem **return**. V případě, že proceduru opouštíme řádně za posledním příkazem jejího těla, nemusíme tento příkaz používat, protože v takovém případě si překladač doplní příkaz **return** automaticky sám.

U skutečných funkcí, tj. funkcí, které vracejí nějakou funkční hodnotu, musíme za příkazem **return** ještě uvést výraz, jehož hodnota bude onou hodnotou, kterou funkce vrací volajícímu programu. Z toho také vyplývá, že u skutečných funkcí musíme příkaz **return** použít vždy.

```
/*  Příklad C3 – 1  */
int a;
int /*****/ Hodne /*****/ ()
{
  return 32767;
}
/***** Hodne *****/

int /*****/ Prirad /*****/ ()
{
  double c;
  a = c = Hodne();
  //Funkce Hodne se volá jen jednou
  //Uvedená konstrukce je ekvivalentí konstrukci
  c = Hodne;
  a = c;
}
/***** Prirad *****/
```

Při používání operace přeřazení musíme (nezávisle na použitém programovacím jazyku) dbát na jednu věc: **všechny proměnné, které vystupují ve výrazu na pravé straně přiřazovacího operátoru, musí mít v danou chvíli přiřazenou hodnotu**. Pokud tomu tak není, přiřadí se nějaké blíže nedefinované smetí.

Programátoři v C++ mají život snazší, protože je na tuto chybu překladač sám upozorní (pokud si ovšem sami ve své namyšlenosti toto varování nevypnou). Programátoři v Pascalu mají život těžší, protože si tuto chybu musejí ohlídat sami. Snad proto, že Pascal je jazyk určený pro výuku a u studentů je třeba pěstovat ostražitost.

3.2 Základní aritmetické operace

Abychom mohli začít psát smysluplné programy, musíme umět nejen přiřadit proměnným **nějakou** hodnotu, ale hlavně přiřadit jim **požadovanou** hodnotu. A abychom jim mohli požadovanou hodnotu přiřadit, musíme ji napřed umět vypočítat.

Žádané hodnoty získáváme v programech vyhodnocováním výrazů, což jsou posloupnosti operátorů (např. +) a operandů (výrazy, s nimiž operátory pracují – např. sčítance).

V tomto oddílu se seznámíme se základními aritmetickými operátory, tedy s operátory pro sčítání, odčítání, násobení a dělení.

Na prvých třech toho k vysvětlování moc není – chovají se přesně tak, jak bychom očekávali, včetně toho, že při vyhodnocování výrazu má násobení přednost před sčítáním. Jedinou novinkou pro nás bude asi skutečnost, že operátor násobení v programech neoznačujeme ani tečkou, ani křížkem, ale hvězdičkou (*). A hlavně si musíte pamatovat, že se **nesmí vynechat**, jak jste zvyklí z matematiky.

Pro uvedené tři operátory platí, že pokud jsou oba operandy celočíselného typu, je i výsledek celé číslo. Pokud je jeden z operandů reálného typu, je reálný i výsledek.

S dělením je to však složitější. Operace dělení jsou ve skutečnosti tři: obyčejné dělení, celočíselné dělení a operace modulo (tj. zjišťování zbytku po celočíselném dělení).

Obyčejné **dělení** se v obou jazycích znázorňuje lomítkem (/). Tato operace dělá to, na co jsme z matematiky zvyklí. Jejím výsledkem je reálné číslo (v C++ musí být alespoň jeden z operandů reálný, v Pascalu mohou být oba operandy celočíselné).

Při **celočíselném dělení** musí být oba operandy celočíselné. V Pascalu se celočíselné dělení značí klíčovým slovem **div**. V C++ se značí stejně jako obyčejné dělení symbolem / (lomítko) a od obyčejného dělení je překladač pozná podle toho, že oba operandy jsou celočíselné. Výsledkem celočíselného dělení je celé číslo s největší absolutní hodnotou nepřevyšující absolutní hodnotu výsledku. Podívejme se na příklady v Pascalu:

```
17 div 5 = 3
17 div -5 = -3
-17 div 5 = -3
-17 div -5 = 3
```

Dělení modulo zjišťuje zbytek po dělení. I tento operátor vyžaduje oba operandy celočíselné. V Pascalu se celočíselné dělení značí klíčovým slovem **mod** a v C++ symbolem **%** (procenta). Výsledek dělení modulo je definován tak, že musí platit rovnost

```
i mod j = i – (i div j) * j
```

Abyste si ověřili chování těchto pěti operátorů, vložte do počítače program podobný programům P3 – 1 resp. C3 – 1 a vyzkoušejte si jejich chování ve všech situacích, které vás napadnou. Myslíme si, že to bude poučnější, než kdybychom vám jejich chování sáhodlouze vysvětlovali.

```
(*  Příklad P3 – 2  *)
{  Základní aritmetické operace  }
Program Operace;

const
 NLL   = #13#10#13#10;               {Přechod o dva řádky}

(**************  Hlavní program  **************)
begin
 write(NLL,'Základní aritmetické operace',NLL);
 writeln(NLL,'Operace sčítání: ');
 writeln( '3  + 2  = ', 3+2);         {Typ operandů určuje}
 writeln( '3  + 2.0 = ', 3+2.0);      {typ výsledku}
 writeln( '3.0 + 2  = ', 3.0+2);
 writeln( '3.0 + 2.0 = ', 3.0+2.0);

 writeln(NLL,'Operace odečítání: ');
 writeln( '3  - 2  = ', 3-2);
 writeln( '3  - 2.0 = ', 3-2.0);
 writeln( '3.0 - 2  = ', 3.0-2);
 writeln( '3.0 - 2.0 = ', 3.0-2.0);

 writeln(NLL,'Operace násobení: ');
 writeln( '3  * 2  = ', 3*2);
 writeln( '3  * 2.0 = ', 3*2.0);
 writeln( '3.0 * 2  = ', 3.0*2);
 writeln( '3.0 * 2.0 = ', 3.0*2.0);

 writeln(NLL,'Operace dělení (obyčejné): ');
 writeln( '3  / 2  = ', 3/2);
 writeln( '3  / 2.0 = ', 3/2.0);
 writeln( '3.0 / 2  = ', 3.0/2);
 writeln( '3.0 / 2.0 = ', 3.0/2.0);

 writeln(NLL,'Celočíselné dělení: ');
 writeln( '3  / 2  = ', 3 div 2);

 writeln(NLL,'Dělení modulo (zbytek po dělení): ');
 writeln( '3  / 2  = ', 3 mod 2);
end.
```

```
/*  Příklad C3 – 2  */
// Základní aritmetické operace

#include <iostream.h>

void /*****/  main  /*****/
()
{
  cout << "\n\nZákladní aritmetické operace"
       << "\n\nOperace sčítání: "
       << "\n5  + 2  = " << (5+2)          //Typ operandů
       << "\n5  + 2.7 = " << (5+2.7)  //určuje typ výsledku
       << "\n5.7 + 2  = " << (5.7+2)
       << "\n5.7 + 2.7 = " << (5.7+2.7)

       << "\n\nOperace odečítání: "
       << "\n5  - 2  = " << (5-2)
```

```
      << "\n5  - 2.7 = " << (5-2.7)
      << "\n5.7 - 2  = " << (5.7-2)
      << "\n5.7 - 2.7 = " << (5.7-2.7)

      << "\n\nOperace násobení: "
      << "\n5  * 2  = " << 5*2
      << "\n5  * 2.7 = " << 5*2.7
      << "\n5.7 * 2  = " << 5.7*2
      << "\n5.7 * 2.7 = " << 5.7*2.7

      << "\n\nOperace dělení (obyčejné): "
      << "\n5  / 2  = " << 5/2
      << "\n5  / 2.7 = " << 5/2.7
      << "\n5.7 / 2  = " << 5.7/2
      << "\n5.7 / 2.7 = " << 5.7/2.7

      << "\n\nCeločíselné dělení: "
      << "\n5  / 2  = " << 5/2

   << "\n\nDělení modulo (zbytek po dělení): "
   << "\n5  % 2  = " << 5%2 << endl;

//Operace se zápornými operandy si vyzkoušejte sami
}
```

Nyní si konečně můžeme vyzkoušet to, co jsme se naučili, na několika „praktických“ programech. Protože toho ale zas tak moc ještě neumíme, musí být naše praktické programy dostatečně jednoduché – sáhneme tedy po učebnici matematiky pro 4. třídu základní školy a čteme:

Příklad 1:
Vláďa kupoval cukr. Když platil padesátikorunou, prodavač mu vrátil dvoukorunu. Kolik kilogramů cukru Vláďa kupoval, když jeden kilogram stál 8 Kč?

Příklad 2:
Do závodní jídelny koupili 15 kg ořechů a o 29 kg více švestek. Jablek koupili 6krát více než švestek. Kolik kilogramů jablek koupili?

Příklad 3:
K dopravě 300 brigádníků bylo zapotřebí 8 autobusů. Kolik brigádníků bylo v každém autobusu, když jich bylo všude stejně až na poslední autobus, kde jich bylo méně? Kolik jich bylo v každém z autobusů, jestliže byly autobusy rovnoměrně zatíženy (počet brigádníků se lišil nejvýše o jednoho)?

Zkuste si uvedené příklady sami naprogramovat a pak svá řešení porovnejte s příkladem P3 – 2, resp. C3 – 2.

```
(*  Příklad P3 – 3  *)
Program Priklady;                    { Jednoduché příklady }

const
 NLL   = #13#10#10;                  {Přechod o dva řádky}
```

```
 NL   = #13#10;

{ Deklarace lokálních objektů }
procedure (*****) Priklad_1  (*****);
const
 platil = 50;
 cena  = 8;
var
 kolik_kg: integer;
begin
 write(NLL,'Příklad 1:');
 kolik_kg := (platil-2) div cena;
 write(' Vláďa koupil ',kolik_kg,' kg cukru.');
end;

procedure (*****) Priklad_2  (*****);
const
 orechy = 15;
var
 svestky, jablka: integer;
begin
 write(NLL,'Příklad 2:');
 svestky := orechy+29;
 jablka := svestky*6;
 write(' Do jídelny koupili ',jablka,' kg jablek.');
end;

procedure (*****) Priklad_3  (*****);
const
 brigadniku = 300;
 autobusu  =  8;
var
 radny, posledni: integer;
begin
 write(NLL,'Příklad 3:');
 radny := (brigadniku div autobusu)+1;     {Zaplnění sedmi autobusů}
 posledni := brigadniku mod radny;         {Zaplnění osmého}
 write(' V prvních sedmi autobusech jede ',radny,
    ' a v osmém ',posledni,' brigádníci.');
end;

procedure (*****) Priklad_3a  (*****);
{Jiné řešení: autobusy budou rovnoměrně zatíženy – počet brigádníků
se v jednotlivých autobusech liší nejvýše o jednoho}
const
 brigadniku = 300;
 autobusu  =  8;
var
 pocet, zbytek: integer;
begin
 write(NLL,'Příklad 3 – jiné řešení:',NL);
 pocet := brigadniku div autobusu;         {Obsazení autobusů}
 zbytek := brigadniku mod autobusu;        {Zbylí rozděleni po jednom}
 write('      ');                          {Zarovnání tisku}
 if (zbytek <> 0) then
  write(zbytek,' autobusy vezou ',pocet+1,', ');
 write(8-zbytek,' autobusy vezou ',pocet,' brigádníků.');
end;
```

```
(**************Hlavní program**************)
begin
 Priklad_1;
 Priklad_2;
 Priklad_3;
 Priklad_3a;
end.
```

```
/*  Příklad C3 – 3  */
//  Jednoduché příklady
#include <iostream.h>

const char* NLL   = "\n\n";          //Přechod o dva řádky
const char* NL   = "\n";             //Přechod na následující řádek

void /*****/ Priklad_1  /*****/
()
{
 const platil = 50, cena  = 8;
 int kolik_kg;
 cout << NLL << "Příklad 1:";
 kolik_kg = (platil-2) / cena;
 cout << " Vláďa koupil " << kolik_kg << " kg cukru.";
}

void /*****/ Priklad_2  /*****/
()
{
 const orechy = 15;
 int svestky, jablka;
 cout << NLL << "Příklad 2:";
 svestky = orechy+29;
 jablka = svestky*6;
 cout << " Do jídelny koupili " << jablka << " kg jablek.";
}

void /*****/ Priklad_3  /*****/
()
{
 const brigadniku = 300, autobusu  =  8;
 int radny, posledni;
 cout << NLL << "Příklad 3:";
 radny = (brigadniku / autobusu)+1;   //Zaplněním sedmi autobusů
 posledni = brigadniku % radny;       //Zaplnění osmého
 cout << " V prvních sedmi autobusech jede " << radny
   << " a v osmém " << posledni << " brigádníci.";
}

void /*****/ Priklad_3a  /*****/
()
// Jiné řešení: autobusy budou rovnoměrně zatíženy – počet brigádníků
// se v jednotlivých autobusech liší nejvýše o jednoho
{
 const brigadniku = 300, autobusu = 8;
 int pocet, zbytek;
 cout << endl << endl << "Příklad 3 – jiné řešení:" << endl;
 pocet = brigadniku / autobusu;       //Obsazení autobusů
```

```
 zbytek = brigadniku % autobusu;      //Zbylí rozděleni po jednom
 cout << "      ";                    //Zarovnání tisku
 if (zbytek != 0)
  cout << zbytek << " autobusy vezou " << pocet+1 << ", '";
 cout << 8-zbytek << " autobusy vezou " << pocet << " brigádníků.";
}

void /*****/ main  /*****/
()
{
 Priklad_1();
 Priklad_2();
 Priklad_3();
 Priklad_3a();
}
```

Poznámka:
Konstanty NL *a* NLL*, které používáme k přechodu na nový řádek, vlastně nepotřebujeme. V pascalském programu můžeme místo*

```
write(a, NL);
```

napsat

```
writeln(a);
```

V C++ můžeme do proudu místo konstanty NL *vložit manipulátor* endl *(udělali jsme to v proceduře* Priklad_3a()*). Příkaz*

```
cout << endl;
```

znamená totéž jako

```
cout << NL;
```

3.3 Relační operátory

Nyní již umíme spočítat jednoduché aritmetické výrazy. Zatím jsme si však nic nepověděli o prostředcích, které by nám – obdobně jako Karlovy podmínky – umožňovaly rozhodnout se v klíčových bodech algoritmu o směru dalšího pokračování.

Nejběžnějším programátorským prostředkem, s jehož pomocí získáváme podklady pro svá další rozhodnutí, jsou porovnávací operátory. Tyto operátory můžeme rozdělit do dvou skupin: na operátory testující rovnost dvou objektů a na operátory testující jejich nerovnost.

Operátory testující rovnost můžete aplikovat na konstanty a proměnné všech doposud probraných typů. Musíme však dát pozor, abychom porovnávali objekty, které jsou navzájem kompatibilní vůči přiřazení. Celá čísla tak můžeme v Pascalu porovnávat i s reálnými čísly a v C++ navíc i se znaky.

Symboly operátorů přiřazení se v Pascalu a C++ liší – tato odlišnost je vlastně logickým důsledkem odlišnosti symbolů přiřazení. Zároveň se liší i typ vracené hodnoty: v Pascalu vracejí logickou hodnotu, kdežto v C++ vracejí celé číslo – ale o tom jsme již hovořili. (V ANSI C++ vracejí tyto operátory –podobně jako v Pascalu – logické hodnoty.)

Operátor pro zjišťování rovnosti se v Pascalu znázorňuje symbolem = (rovnítko) a v C++ symbolem == (dvě rovnítka) a vrací logickou hodnotu ANO (v Pascalu *true*, v C++ 1, v ANSI C++ **true**) v případě, že oba porovnávané objekty mají stejnou hodnotu. V opačném případě vrací logické NE (v Pascalu *false*, v C++ je to 0, v ANSI C++ konstanta **false**).

Operátor nerovnosti se v Pascalu znázorňuje symbolem **<>** (menší – větší) a v C++ symbolem **!=** (vykřičník – rovnítko) a vrací hodnotu v případě, že porovnávané objekty nemají stejnou hodnotu, a v opačném případě vrací logické NE.

Poznámka:
V C++ si musíte dát pozor při porovnávání řetězců. Jak jsem již řekl, typ **char*** *není s ekvivalentní s pascalským typem* string*. Prozradím vám, že proměnná či konstanta typu* **char*** *neobsahuje na rozdíl od Pascalu vlastní znaky řetězce, ale pouze adresu místa v paměti, kde je tento řetězec uložen.*
Porovnáváte-li tedy hodnoty dvou objektů typu **char*** *pomocí operátoru =, porovnávají se v C++ adresy jejich uložení v paměti (na rozdíl od Pascalu, který opravdu porovnává hodnoty porovnávaných řetězců).*
Pokud tedy v C++ zjistíme, že dva objekty typu **char*** *jsou totožné, víme jistě, že se jedná o týž znakový řetězec. Pokud zjistíme, že totožné nejsou, nevíme vlastně ještě nic. O tom, jak lze tuto situaci řešit, si povíme o něco později.*

Operátor porovnání jsou čtyři a v obou jazycích se značí stejně. Jejich funkci popisuje následující tabulka:

Operace	Vrací logické ANO když
a < b	*a* je menší než *b*
a > b	*a* je větší než *b*
a <= b	*a* je menší nebo rovno *b*
a >= b	*a* je větší nebo rovno *b*

Tab. 3.1 Význam porovnávacích operátorů

Při porovnávání logických hodnot platí v Pascalu stejně jako v C++

```
ANO > NE,
```

protože Pascal uvnitř také reprezentuje ANO jedničkou a NE nulou.

Porovnávání znaků je v podstatě porovnáváním podle abecedy, kde abeceda je definována kódováním znaků. Musíme vás však upozornit na to, že výsledky porovnávání záleží v C++ na tom, zda pracujeme s kódy znaků v intervalu <0; 255> anebo v intervalu <-128; 127>. Podrobnosti si vysvětlíme později v samostatné kapitole věnované datovým typům

znaků. Do té doby se budeme vyhýbat příkladům, v nichž by bylo potřeba zjišťovat, který znak je „větší“ a který „menší“.

Porovnávání textových řetězců je v Pascalu rovněž v podstatě porovnávání podle abecedy – i když je abeceda definována trochu jinak, než jsme zvyklí.

Jak jste asi po přečtení poznámky o testování řetězců v C++ na rovnost sami odhadli, vyjmenované čtyři operátory nám toho o vzájemných hodnotách textových řetězců mnoho nepovědí. Prozatím si řekneme, že s klasickými textovými řetězci (s jinými jsme se ani nesetkali) se v C++ pracuje pomocí funkcí. Věnujeme tomu samostatnou kapitolu.

Abychom si pořád nepovídali pouze o teorii, zkuste si naprogramovat následující příklad:

Příklad:
O geniálním matematiku Gaussovi se vypráví následující historka: Jednou si chtěl učitel odpočinout od dětí ve třídě, do níž chodil i mladý Gauss, a zároveň chtěl procvičit jejich počtářskou dovednost. Dal jim proto sečíst všechna čísla od jedné do sta. Očekával, že od nich bude mít delší dobu pokoj. Jaké však bylo jeho překvapení, když se za několik málo okamžiků přihlásil mladý Gauss a tvrdil, že je s příkladem hotov. Výsledek je prý 5050. Když se učitel podivoval, jak mohl výsledek tak rychle spočítat, ukázal mu vzoreček, který právě vymyslel:

```
Součet = n * (n + 1) / 2
```

kde *n* je počet čísel, která se mají sečíst.

Co tedy bude vaším úkolem? Napište program, který ověří, že pro čísla od jedné do sta tento vzoreček platí. Dva možné způsoby řešení problému ukazují příklady P3 – 4 a C3 – 4.

```
(*  Příklad P3 – 4  *)
{ Gauss - potvrzení vzorce dvěma způsoby }
Program Gauss;

(** Deklarace lokálních objektů **)
const
  NL   = #13#10;                                    {Přechod na nový
  řádek}
  NLL   = #13#10#13#10;                    {Přechod o dva řádky}
  AzDo: integer = 100;

procedure (*****) Gauss_A  (*****);
var
  Vzorec, Soucet, Cislo, Pocet: integer;
begin
  Pocet := 1;
  while( Pocet <= AzDo )do
  begin
       Vzorec := Pocet * (Pocet+1) div 2;  {Podle vzorce}
       Cislo := 1;
       Soucet := 0;
       while( Cislo <= Pocet )do           {Postupným součtem}
```

```
      begin
          Soucet := Soucet + Cislo;
          Cislo := Cislo + 1;
      end;
      write(NL,'Součet čísel od 1 do ',Pocet,' je ',Vzorec,
           ' - podle Gaussova vzozce je ',Soucet);
      if( Vzorec = Soucet )
      then write(' - souhlasí.')
      else write(' - nesouhlasí!');
      Pocet := Pocet + 1;
  end;
end;

procedure (*****)  Gauss_B  (*****);       {Elegantnější řešení}
var
  Vzorec, Soucet, Pocet: integer;
begin
  Pocet := 1;
  Soucet := 0;
  repeat
      Vzorec := Pocet * (Pocet+1) div 2;  {Podle vzorce}
      Soucet := Soucet+Pocet;             {Kumulovaný součet}
      write(NL,'Součet čísel od 1 do ',Pocet,' je ',Vzorec,
               ' - podle Gaussova vzorce je ',Soucet);
      if( Vzorec = Soucet )
      then write(' - souhlasí.')
      else write(' - nesouhlasí!');
      Pocet := Pocet+1;
  until( Pocet > AzDo );
end;

(************** Hlavní program **************)
begin
  Gauss_A;
  write(NLL,NLL);
  Gauss_B;
end.
```

```
/*  Příklad C3 – 4  */
// Gauss - potvrzení vzorce dvěma způsoby
#include <iostream.h>

/*** Prototypy lokálních funkcí¶ ***/
void Gauss_A();
void Gauss_B();

const AzDo = 100;

/************** Hlavní program **************/

void /*****/  main  /*****/ ()
{
  Gauss_A();
  cout << "\n\nElegantnější řešení\n\n";
  Gauss_B();
}
```

```
/********* Lokální podprogramy *********/
void /*****/  Gauss_A  /*****/ ()
{
  int Vzorec, Soucet, Cislo;
  int Pocet = 0;
  while ( (Pocet = Pocet+1) <= AzDo )
  {
       Vzorec = Pocet * (Pocet+1) / 2;               //Podle vzorce
       Cislo = 1;
       Soucet = 0;
       while( Cislo <= Pocet )             //Postupným součtem
       {
           Soucet = Soucet + Cislo;
           Cislo = Cislo + 1;
       } /* while */
       cout << "\nSoučet čísel od 1 do " << Pocet
        << " je " << Vzorec
            << " - podle Gaussova vzozce: " << Soucet;
       if ( Vzorec == Soucet )
           cout << " - souhlasí.";
       else
           cout << " - nesouhlasí!";
  } /* while */
}
void /*****/  Gauss_B  /*****/ ()          //Elegantnější řešení
{
  int Vzorec, Soucet=0, Pocet=0;
  do
  {
       Vzorec = Pocet * (Pocet+1) / 2;               //Gaussův vzorec
       Soucet = Soucet + Pocet;            //Kumulovaný součet
       cout << "\nSoučet čísel od 1 do " << Pocet
            << " je " << Vzorec
            << " - podle Gaussova vzorce je " << Soucet;
       if ( Vzorec == Soucet )
           cout << " - souhlasí.";
       else
           cout << " - nesouhlasí!";
  }while( (Pocet = Pocet+1) <= AzDo );
}
```

3.4 Vstup dat

Již umíme naprogramovat řešení jednoduchých úloh a poslat výsledek do standardního výstupu. Nyní si ukážeme, jak to zařídit, aby naše programy dokázaly také od svých uživatelů nějaká data převzít.

V Pascalu slouží pro čtení dat ze standardního vstupu procedury *read* a *readln*, které se používají podobně jako procedury *write* a *writeln*. Do závorek za jméno procedury však nepíšeme vysílané hodnoty, ale jména proměnných, do nichž chceme načítané hodnoty uložit.

Obdobně i v C++ je použití vstupního operátoru velice podobné použití operátoru výstupního. Pouze místo **cout**, což je, jak víme, jméno standardního výstupního proudu, napíšeme jméno standardního vstupního proudu **cin** a otočíme směr šipek. Stejně jako v Pascalu nesmíme zapomenout, že načítané hodnoty můžeme ukládat pouze do proměnných (přesněji do modifikovatelných l-hodnot).

Při používání standardního vstupu a výstupu musíme pamatovat na jednu věc: pokud nepoužíváme vstup přesměrovaný z nějakého souboru, ale přímo vstup z klávesnice, nesmíme zapomínat, že standardní vstup je „dávkovaný".

Hodnoty, které zadáváme, nečte totiž rovnou náš program, ale nejprve operační systém. Ten čte znaky posílané z klávesnice jeden za druhým, ukládá je do vyrovnávací paměti (bufferu), kde je můžeme editovat, a ve chvíli, kdy stiskneme klávesu ENTER**,** ukončí toto předzpracování a předá celý řetězec (dávku) vašemu programu. Teprve v této chvíli jej může náš program začít analyzovat.

Protože naše programy prozatím nebudou přímo číst znaky vysílané z klávesnice (to se naučíme až později), ale budou zadávané hodnoty načítat prostřednictvím standardního vstupu, musíme tato jeho omezení respektovat.

Abychom ale výklad dále zbytečně neprodlužovali, zkusíme si hned nějaký příklad.

Napíšeme si jednoduchý program, který bude průběžně monitorovat stav vaší peněženky. Na počátku se nás zeptá na aktuální stav konta a pak se bude průběžně ptát, kolik máme platit. V případě, že je v peněžence dostatek peněz, odečte od jejího obsahu patřičný obnos a sdělí vám nový stav. Pokud v ní dostatečný obnos není, počítač třikrát pípne a upozorní nás na nemožnost zamýšlené platby.

Pokud budeme chtít programu sdělit, že vám do peněženky nějaké nové finance přibyly, zadáme zápornou platbu. Činnost programu ukončíme zadáním nulové platby.

Pokuste se zadanou úlohu nejprve naprogramovat sami a pak si své řešení porovnejte s programy P3 – 5 resp. C3 – 5. Předem bychom vám však chtěli připomenout, abyste nezapomínali, že **při každé komunikaci s operátorem by měl počítač před každým vstupem nejprve vytisknout text, podle nějž operátor pozná, jaká data má vlastně zadat.**

Možná, že vám tato zásada připadá naprosto samozřejmá, ale divili byste se, kolik jsme viděli programů (včetně profesionálních), které nejen neoznamovaly, jaký vstup očekávají, ale dokonce nedaly žádným způsobem najevo, že vůbec nějaký vstup očekávají (např. při spouštění známé hry TETRIS). U takových programů pak není jasné, zda se program někde zacyklil nebo zda čeká na nějaký vstup, a pokud čeká, tak na jaký. A protože člověk (a programátor zvláště) je tvor zapomnětlivý, nedokáže po určitém časovém odstupu s programem efektivně pracovat ani sám autor. Proto tuto zásadu nepodceňujte.

```
(*  Příklad P3 – 5  *)
{ Ukecaná peněženka }
Program Prachy;
(*** Definice lokálních proměnných ***)
const
```

```
  NLL = #13#10#10;
  NL = #13#10;

var
  Hotovost, Platba, Plateb3,
  Plateb5, Minule, SoucetPl: integer;

procedure (*****)  Prijem  (*****);
var
  Prijato, Do32: integer;
begin
  Prijato := -Platba;
  Plateb3 := 0;
  Plateb5 := 0;
  SoucetPl:= 0;
 if(Prijato > 5000 ) then
       writeln(NLL, 'Tak se mi to líbí. Jsi pašák!');
  if(Prijato <= 50 ) then
       writeln(NLL, 'Vydělávat neznamená sbírat po ulici drobné!');
  if(Prijato < Minule ) then
       writeln(NLL, 'Minule to bylo lepší!');
  Minule := Prijato;
  Do32 := 32000 - Hotovost;
  Hotovost := Hotovost + Prijato;
  if( Prijato > Do32 ) then
  begin
       Hotovost := 32000;
       writeln(NLL, 'Nejsem vagón. Víc se do mě nevejde.',
             NL, 'Vracím ', Prijato - Do32, ' Kč');
  end (* if *)
end;

procedure (*****)  Vydani  (*****);
begin
  Plateb5 := Plateb5 + 1;
  SoucetPl := SoucetPl + Platba;
  if( Platba > Hotovost ) then
  begin
       writeln(NLL, #7#7#7'Tolik nemám a dluhy si nemůžeme dovolit. ',
             NL, 'Nedostaneš nic!');
       Halt( 0 );
  end; (* if *)
  if( Platba > 1000 ) then
  begin
       Plateb3  := Plateb3 + 1;
       if( Plateb3 > 3 ) then
       writeln( NLL, '! Nějak moc utrácíš !');
  end; (* if *)
  if( Plateb5 > 5 ) then
       Writeln(NLL, 'A co takhle občas také vydělávat?');
  if( SoucetPl > Minule ) then
       writeln (NLL, 'Utrácíš rychleji než vyděláváš!');
  Hotovost := Hotovost - Platba;
end;

(************** Hlavní program **************)
begin
```

```
  writeln( NL, 'Ukecaná peněženka.',
         NL, '-----------------',
           NLL, 'Kolik máš v peněžence?',
           NLL);
  readln(Hotovost);
  repeat
      writeln(NLL,'Kolik chceš?', NLL);
      read(Platba);
      if( Platba <> 0 ) then
      begin
          if( Platba > 0 ) then
                Vydani
          else
                Prijem;

          writeln(NLL, 'Nový stav hotovosti!',
                 NLL, Hotovost, ' Kč');
      end (* if *)
  until( Platba = 0 );
  writeln(NLL, 'Konec programu');
end.
```

```
/*  Příklad C3 – 5  */
//Ukecaná peněženka
#include <stdlib.h>
#include <iostream.h>

/*** Definice lokálních objektů ***/
int Hotovost = 0;
int Platba  = 0;
int Plateb3 = 0;
int Plateb5 = 0;
int Minule  = 0;
int SoucetPl = 0;

/*** Prototypy lokálních funkcí ***/
void Prijem();
void Vydani();

/**************  Hlavní program  **************/
void /*****/  main  /*****/
()
{
  cout << "\nUkecaná peněženka."
       << "\n-----------------"
       << "\n\nKolik máš v peněžence!"
       << "\n\n\? ";
  cin  >> Hotovost;
  do
  {
       cout     << "\n\nKolik chceš?"
           << "\n\n? ";
       cin >> Platba;
       if( Platba != 0 )
       {
```

```
          if( Platba > 0 )
               Vydani();
          else
               Prijem();
          cout << "\n\nNový stav hotovosti!"
               << "\n\n" << Hotovost << " Kč";
       } /* if */
  }while( Platba );
  cout << "\n\nKonec programu";
}
/*********  Lokální podprogramy  *********/
void /*****/  Prijem  /*****/
()
{
  int Prijato;
  int Do32;
  Prijato = -Platba;
  Plateb3 = 0;
  Plateb5 = 0;
  SoucetPl= 0;
  if(Prijato > 5000 )
       cout << "\n\nTak se mi to líbí. Jsi pašák!";
  if(Prijato <= 50 )
       cout << "\n\nVydělávat neznamená sbírat po ulici drobné!";
  if(Prijato < Minule )
       cout << "\n\nMinule to bylo lepší!";
  Minule = Prijato;
  Do32 = 32000 - Hotovost;
  Hotovost = Hotovost + Prijato;
  if( Prijato > Do32 )
  {
       Hotovost = 32000;
       cout     << "\n\nNejsem vagón. Víc se do mě nevejde."
           << "\nVracím " << Prijato - Do32 << " Kč";
  } /* if */
}
void /*****/  Vydani  /*****/
()
{
  Plateb5 = Plateb5 + 1;
  SoucetPl = SoucetPl + Platba;
  if( Platba > Hotovost )
  {
       cout     << "\n\n\a\a\aTolik nemám a dluhy si nemůžeme dovolit.
  "
           << "\nNedostaneš nic!";
       exit( 0 );
  } /* if */
  if( Platba > 1000 )
  {
       Plateb3  = Plateb3 + 1;
       if( Plateb3 > 3 )
       cout << "\n\n! Nějak moc utrácíš !";
  } /* if */
```

```
  if( Plateb5 > 5 )
      cout << "\n\nA co takhle občas také vydělávat?";
  if( SoucetPl > Minule )
      cout << "\n\nUtrácíš rychleji než vyděláváš!";
  Hotovost = Hotovost - Platba;
}
```

Abyste si trochu „osahali“ možnosti standardního vstupu, zkuste svému programu poslat prázdný vstup – tj. stisknout ENTER, aniž byste cokoliv zadali. Jak se sami přesvědčíte, kurzor se přesune na další řádek a počítač nadále čeká na nějaké zadání.

Ověřte si, že při zadávání dat prostřednictvím standardního vstupu je možné zadávané hodnoty editovat, přičemž inteligence editoru záleží na použitém operačním systému.

Zkuste si také, jak to dopadne, když pošlete evidentně špatnou hodnotu – např. text. Reakce programů se budou lišit podle použitého jazyka. Pascalský program skončí chybovou zprávou

```
Runtime error 106 at 0B2E:0117
```

čímž nám oznamuje, že při vykonávání instrukce na adrese $0117 v segmentu 0B2Eh došlo k chybě číslo 106. Zalovíme v manuálu a zjistíme, že se jedná o chybu *Invalid numeric format* (pokud máme českou verzi manuálu, dozvíme se, že byla „načtena nesprávná číselná hodnota“.

Jestliže počítači zadáme číslo, jehož absolutní hodnota je větší než 32 768, dozvíme se, že nastala chyba

```
Runtime error 201 at 0B2E:0115
```

přičemž o chybě 201 se dozvíme, že se jedná o *Range check error*, neboli (opět citujeme z českého manuálu):

Chyba vzniklá v případě překladu s direktivou {$R+}. Příčin může být několik:

Překročení indexu pole.

Proměnné byla přiřazena hodnota mimo rozsah jejího typu.

Parametrům procedury nebo funkce byla přiřazena hodnota mimo rozsah jejich typu.

V našem případě šlo o druhou jmenovanou příčinu, tedy že proměnné byla přiřazena hodnota mimo rozsah jejího typu, protože celočíselné proměnné typu *integer* mohou mít rozsah pouze z intervalu <-32 768; 32 767>.

Všechny výše uvedené informace se můžeme dozvědět my, neboť máme po ruce manuál (anebo protože nám to někdo řekl), ale co má dělat chudák uživatel našeho programu? (Svede to na nás, programátory – a pro jednou bude mít pravdu.) A proto nikdy nezapomínejte na jednu důležitou zásadu: **Všechny vstupy je třeba náležitě ošetřit!**

Přitom **náležitě** znamená tím dokonaleji, čím většího laika se chystáme k programu posadit. Pokud si píšeme pouze pomocný program sami pro sebe, můžeme tato ošetření odbýt a spolehnout se na vlastní neomylnost. (On už nás život naučí.) Jakmile však děláme program na zakázku, nesmíme na ošetření vstupů šetřit!

Na platnosti zásad z předchozího odstavce se nic nemění ani při programování v C++, které na chybné vstupy reaguje trochu jinak než Pascal. Pokud se operátoru >> nepodaří převzít ze standardního vstupu očekávanou hodnotu, ponechá proměnnou, do níž měl hodnotu načíst, netknutou a pouze si někde uvnitř nastaví příznak, že při posledním čtení došlo k chybě, a dokud je tento příznak nastaven, nic dalšího nepřečte.

Podívejme se, jak bychom se mohli s takovýmito chybami vstupu vypořádat:

Pascal poskytuje možnost vypnout kontrolu chyb při čtení dat a kontrolu přetečení povoleného rozsahu hodnot. Obě můžeme vypnout buď přímo v IDE v okně *Options* | *Compiler*, kde potlačíme volby *Range checking* a *I/O checking* (v příslušných závorkách nesmí být křížek [X]).

Druhou možností je potlačit tyto volby přímo v programu a učinit tak zdrojový text nezávislý na okamžitém nastavení voleb překladače. Tyto volby jakož i ostatní **pokyny pro překladač** se v programu zadávají pomocí **komentářových direktiv** („**dolarových poznámek“)**. To jsou komentáře, které začínají znakem $ (dolar), za kterým následuje (bez vložených bílých znaků) písmeno, označující o jakou volbu jde. Za tímto písmenem napíšeme znak + (plus) v případě, že chceme volbu nastavit, a - (minus) v případě, že chceme volbu potlačit.

V jedné dolarové poznámce můžeme zadat i několik direktiv. V tom případě je oddělujeme čárkami, přičemž mezi ně nesmíme vložit nadbytečné bílé znaky. Najde-li překladač v dolarové poznámce bílý znak, považuje část poznámky za ním za obyčejný komentář.

Naše direktivy bychom tedy mohli zadat buď ve tvaru

```
{$I- Potlačení kontroly formátu čísel}
{$R- Potlačení kontroly rozsahu}
```

nebo

```
{$I-,R- Potlačení kontroly formátu čísel a rozsahu}
```

Obě uvedené volby můžeme během programu libovolně zapínat a vypínat, takže sami ovlivňujeme, v kterých částech programu se dané chyby kontrolují a v kterých nikoliv.

Jakmile tyto volby potlačíme, je ošetření bezchybnosti vstupu již zcela na nás. Ošetření přetečení vyžaduje trochu větší znalosti, než jaké dosud máme, a proto si je necháme na pozdější dobu. Nyní si ve zkratce povíme o ošetření nesprávné číselné hodnoty. Nejprve si ukážeme, jak bychom mohli upravit program s upovídanou peněženkou, aby se uměl vyrovnat se špatným vstupem, a pak si tyto úpravy vysvětlíme. Zde uvedeme jen funkci *Spatne*, která se bude starat o vstup, a část hlavního programu (úplný zdrojový text najdete na doplňkové disketě v souboru *P3-06.PAS*):

```
(*   Příklad P3 – 6   *)
function (*****)  Spatne  (*****): Boolean;
{Funkce má na starosti hlídání vstupu a
 vrací ANO v případě, že vstup proběhl bez chyby}
var
```

```
  ch: char;
begin
  if( IOResult <> 0 )then
  begin
       write( NL,'Co jsi to zadal za hodnotu? Znovu!' );
       while( not Eoln )do            {Dočti znaky do konce řádku}
           read(ch );
       Spatne:= TRUE;
  end
  else Spatne:= FALSE;
end;
 (**************  Hlavní program   **************)
begin
  {$i-}
  write(NLL, NLL, 'Ukecaná peněženka.',
             NL , '------------------' );
  repeat
    write( NLL, 'Kolik máš v penežence: ' );
    read( Hotovost );
  until( not Spatne );
  Minule := Hotovost;
  repeat
    repeat
      write( NLL, 'Kolik chceš: ' );
      read( Platba );
    until( not Spatne );
  { a dále stejně jako minule }
  until( Platba = 0 );
  write( NLL, 'Konec programu' );
end.
```

Převezmeme-li ošetření formátu zadávaných hodnot na vlastní bedra, musíme se po každém vstupu zeptat, nedošlo-li při něm k chybě. K tomu slouží celočíselná funkce *IOResult*, která vrací v případě bezchybného čtení nulu a v opačném případě kód chyby. Dokud tuto funkci nevyvoláme, bude program po chybném zadání odmítat číst dál.

Voláním funkce *IOResult* se vynulují interní příznaky chybného vstupu a systém bude ochoten pokračovat ve čtení. (Vyvoláme-li ji dvakrát bezprostředně za sebou, obdržíme při druhém volání nulu.)

Při ošetřování chybného vstupu jsme dále použili logickou funkci *eoln*, která vrátí ANO v případě, že by dalším přečteným znakem byl znak konce řádku. Této funkce využijeme k tomu, abychom vybrali všechny znaky z řádku, v němž jsme nalezli chybné zadání a připravíme jej tak pro nové zadání. (Znak konce řádku přečíst nesmíme – proč, o tom si povíme později.)

C++ se nesnaží ani naznačovat, že by mohlo chybná zadání kontrolovat za programátora, protože autoři jazyka věděli, že si to chce stejně každý dělat po svém. Mechanismus je však podobný jako v Pascalu. Objeví-li se v zadání chyba, vstup se s vámi „nebaví“ do té doby, dokud mu explicitně nenařídíte smazat vnitřní příznak chyby.

Ukážeme si opět, jak bychom mohli upravit program s upovídanou peněženkou, aby se uměl vyrovnat se špatným vstupem, a pak si tyto úpravy vysvětlíme. Uvedeme si jen funkci *Spatne()*, která se bude starat o vstup, a část funkce *main()*. (Úplný zdrojový text najdete na doplňkové disketě v souboru *C3-06.CPP*.)

```
static int /*****/   Spatne   /*****/ ()
{
  if( !cin.good() )
  {
       cout << "\n\nCo jsi to zadal za hodnotu?  Znovu!" ;
       cin.clear();
       cin.ignore( 999, '\n' );
       return 1;
  }
  return 0;
}
/**************  Hlavní program   **************/
void /*****/   main   /*****/ ()
{
  cout << "\nUkecaná peněženka."
       << "\n------------------";
  do{
       cout << "\n\nKolik máš v peněžence: ";
       cin  >> Hotovost;
  }while( Spatne() );
  Minule = Hotovost;
  do{
       do{
           cout << "\n\nKolik chceš: ";
           cin  >> Platba;
       }while( Spatne() );
// a dále stejně jako minule
  while ( Platba );
  cout << "\n\nKonec programu";
}
```

O tom, zda při čtení vstupní hodnoty nastala chyba, nás zpraví logická funkce *good()*, která, jak ostatně napovídá její název, vrací ANO v případě, že čtení vstupu proběhlo bezchybně, a NE v opačném případě.

Na rozdíl od Pascalu však tato funkce vnitřní příznaky chyby nenuluje. Na to musíme použít jinou funkci – např. tak jako my funkci *clear()*.

Třetí funkcí, kterou můžeme při ošetřování vstupu použít, je *ignore()*, která vyplní náš požadavek na ignorování zbylých znaků na řádku. Musíme předat dvě hodnoty: počet ignorovaných znaků a znak, kterým bude posloupnost ignorovaných znaků ukončena.

Přesný počet zbylých znaků na řádku dopředu neznáme. Protože však víme, že chceme přeskakování znaků zarazit na konci řádku, bezostyšně jí zadáme požadavek ignorovat 999 znaků, čímž máme zajištěno, že funkce vyčistí řádek vždy celý.

Všimněte si změněné syntaxe při volání funkcí. Mohli bychom ji interpretovat tak, že náš program nevolá tyto funkce, ale žádá standardní vstupní proud **cin,** aby si je zavolal. Proč je tomu tak, o tom mluvíme podrobněji v knize *Objektové programování*.

Chyby, o nichž jsme si doposud povídali, jsou chybami vadného formátu vstupní hodnoty. Pokud zadáme příliš malou nebo příliš velikou vstupní hodnotu, musíme k ošetření vzniklé chyby použít prostředky, na jejichž naprogramování nám naše dosavadní znalosti ještě nestačí a o nichž si povíme někdy později.

4. Proměnné a konstanty

4.1 Lokální, globální a externí objekty

Než se začneme věnovat vlastnímu tématu kapitoly, chtěli bychom ještě jednou připomenout rozdíl mezi deklarací a definicí, protože je to otázka, ve které musíme mít při studiu této kapitoly jasno.

Účelem **deklarace** je sdělit překladači základní charakteristiky objektu, s nímž se chystáme pracovat: jméno funkce a typ vracené hodnoty, jméno a typ používané proměnné či konstanty apod. Deklarace tedy ještě nemusí obsahovat všechny informace o daném objektu.

Naopak **definice** všechny informace o daném objektu obsahovat musí, protože jejím účelem je poskytnout překladači dostatečné podklady pro to, aby mohl definovaný objekt skutečně vytvořit (zřídit). Definice funkce tedy musí obsahovat popis jejího těla, definice konstanty musí definovat hodnotu této konstanty atd. Každý objekt smíme **definovat pouze jednou**, kdežto deklarovat jej můžeme kolikrát chceme (aniž bychom jej definovali).

Z uvedeného je zřejmé, že každá definice je zároveň deklarací. Pokud tedy budeme v dalším textu hovořit o deklaracích, bude se to ve většině případů týkat i definic. Pokud tomu tak nebude, výslovně to uvedeme.

Přejděme nyní k vlastnímu tématu kapitoly. O lokalitě a globalitě proměnných jsme se již zmiňovali v souvislosti s deklaracemi v kapitole 2. Nyní si tedy o nich povíme podrobněji.

O objektech, které jsou známé pouze v nějaké uzavřené oblasti programu, říkáme, že jsou v této oblasti **lokální**. Oblastí, která má své lokální objekty, může být **modul**, **procedura**, a v C++ také složený příkaz – **blok**.

O objektech, které jsou deklarovány v některé nadoblasti dané oblasti (tj. v oblasti, jejíž je daná oblast součástí), říkáme, že jsou pro danou oblast **globální**.

V nadpisu se také zmiňujeme o **externích** objektech. Za externí budeme označovat všechny objekty, které mohou být sdíleny několika moduly – tedy všechny vyvážené (exportované) a dovážené (importované) objekty. O externích objektech můžeme říci, že to jsou objekty, které jsou lokální v celém programu a tedy globální pro všechny moduly.

Z toho plyne, že objekt, který je pro danou oblast globální, musí být lokální v některé její nadoblasti (např. objekty globální pro podprogram budou lokální v modulu obsahujícím tento podprogram anebo musí být externí – tj. lokální v programu) a naopak, že objekt lokální v nějaké oblasti je globální vůči všem jejím podoblastem, podoblastem jejích podoblastí atd.

V souvislosti s lokalitou objektů se musíme vždy zabývat otázkou jejich **viditelnosti**, tj. otázkou, kde všude můžeme dané objekty použít. Pro viditelnost objektů platí:

- ✧ Každý objekt je za normálních okolností viditelný v oblasti, v níž je deklarován (a to včetně všech jejích podoblastí), počínaje místem své deklarace. Jinými slovy, každý objekt musíme nejprve deklarovat, a teprve pak jej smíme používat (ale to užívá me).

- ✧ Objekty, které jsou lokální v nějaké oblasti, nejsou z míst mimo tuto oblast viditelné.
- ✧ Objekty, které jsou vůči dané oblasti globální, jsou sice deklarovány mimo danou oblast, avšak jsou z dané oblasti viditelné s výjimkou situací popsaných v bodě 5.
- ✧ Externí objekty jsou viditelné v modulu, který je vyváží, a ve všech modulech, které se explicitně přihlásí k jejich dovozu.
- ✧ Pokud v oblasti definujeme nový objekt, překryje jméno tohoto objektu jména všech stejnojmenných globálních objektů (jsou-li takové) a tyto globální objekty nebudou v dané oblasti ani v jejích podoblastech, v podoblastech těchto podoblastí atd. viditelné.

Budeme-li tedy v dalším textu o nějakém objektu tvrdit, že je vůči nějaké oblasti globální, doplníte si sami, že je globální vůči všem jejím podoblastem, podoblastem těchto podoblastí atd., a že je viditelný ve všech částech programu, vůči nimž je globální, s výjimkou podoblastí, v nichž je překryt deklarací nějakého stejnojmenného objektu.

Poznámka 1:
Výjimkou z pravidla 1 jsou návěští v C++, která jsou viditelná v celém těle funkce, v níž jsou deklarována. To znamená, že jsou viditelná i před místem své deklarace, a že jsou viditelná i v nadřazených blocích (tj. nadoblastech – jsou-li ovšem takové).

Poznámka 2:
V C++ si v některých případech překrytí globálních objektů (bod 5) můžeme pomoci. Pokud je překrytý objekt externí anebo lokální v modulu, můžeme jej použít tak, že před něj napíšeme rozlišovací operátor :: (čtyřtečka) – příklad najdete v příkladu C4 – 1.

Přejdeme nyní k jednotlivým jazykům a vysvětlíme si vše ještě jednou na demonstračních příkladech.

V kapitole o deklaracích jsme si řekli, že pascalské podprogramy mohou mít své vlastní lokální podprogramy. Takovýto lokální podprogram je onou výše zmiňovanou podoblastí svého rodičovského podprogramu.

- ✧ Všechny objekty deklarované v podprogramu jsou v daném podprogramu lokální a jsou globální vůči všem později definovaným lokálním podprogramům tohoto podprogramu.
- ✧ Objekty deklarované v hlavním modulu a v části **implementation** řadových modulů vně podprogramů jsou v těchto modulech lokální a jsou viditelné od místa své deklarace.
- ✧ Objekty deklarované v části **interface** jsou vyvážené externí objekty a jsou viditelné v celém modulu.
- ✧ V daném modulu jsou viditelné všechny externí objekty modulů, k jejichž dovozu se tento modul přihlásil v direktivě **uses**.

Předchozí zásady určování lokality a viditelnosti proměnných konstant a datových typů deklarovaných na různých místech programu si můžete ověřit na příkladu P4 – 1. Každý příkaz *write* tiskne hodnoty všech v daném místě viditelných proměnných a konstant.

```
(*   Příklad P4 – 1   *)
{Viditelnost - hlavní modul }
Program Viditelnost;
Uses P4_01u;

(*****  Lokální objekty modulu  *****)
(* Hodnoty přiřazované proměnným mají následující interpretaci:
  1. cifra = jméno: Externí=1, Globální=2, Zakrytý=3
  2. cifra = místo deklarace: číslování - viz následující bod
  3. cifra = místo poslední modifikace:
     0 - hlavní modul
     1 - procedura Tiskni0 v hlavním modulu
     2 - dovážený modul
     3 - inicializace dováženého modulu (jen modifikace)
     4 - procedura Tiskni0 v dováženém modulu
     5 - dovážená procedura Tiskni1
  4. cifra = celkový počet modifikací
*)
const
  Globalni : integer = 2000;
  { Proměnná Globální deklarovaná zde je jiná proměnná,
    než stejnojmenná proměnná deklarovaná v dováženém modulu }

  Zakryty  : integer = 3000;
  { Proměnná Zakrytý deklarovaná zde je jiná proměnná,
    než stejnojmenná proměnná deklarovaná kdekoliv jinde }

procedure (*****)  Tiskni0  (*****);
{ Tato procedura nemá nic společného se stejnojmennou
 procedurou deklarovanou v dováženém modulu }
const
 Zakryty : integer = 3100;
 { Proměnná Zakrytý deklarovaná zde je jiná proměnná,
   než stejnojmenná proměnná deklarovaná kdekoliv jinde}
begin
 write (NLL, '======Tiskni0=========================',
         NL, 'Objekty viditelné  v proceduře Tiskni0: ',
         NL, 'Externí  = ', Externi,
         NL, 'Globalni = ', Globalni,
         NL, 'Zakryty  = ', Zakryty,
         NL, 'konstanty NL a NLL',
         NL, 'a procedura Tiskni1');
 Tiskni1;
 write( NLL, 'Externi  po Tiskni1: ', Externi,
        NL,  'Globalni po Tiskni1: ', Globalni,
        NL,  'Zakryty  po Tiskni1: ', Zakryty,
        NL,  '======konec Tiskni0==================');
 Externi  := (Externi  div 100 * 100) + (Externi  mod 10) + 11;
 Globalni := (Globalni div 100 * 100) + (Globalni mod 10) + 11;
 Zakryty  := (Zakryty  div 100 * 100) + (Zakryty  mod 10) + 11;
```

```
end;

(**************   Hlavní program   **************)
begin
 write (NLL,'   Hlavní program                    ',
        NL, 'Objekty viditelné v hlavním programu: ',
        NL, 'Externí  = ', Externi,
        NL, 'Globalní = ', Globalni,
        NL, 'Zakrytý  = ', Zakryty,
        NL, 'konstanty NL a NLL',
        NL, 'a procedury Tiskni0 a Tiskni1');
 Tiskni0;
 write( NLL,'Externí  po Tiskni0: ', Externi,
        NL, 'Globalní po Tiskni0: ', Globalni,
        NL, 'Zakrytý  po Tiskni0: ', Zakryty );
 Tiskni1;
 write( NLL,'Externí  po Tiskni1: ', Externi,
        NL, 'Globalní po Tiskni1: ', Globalni,
        NL, 'Zakrytý  po Tiskni1: ', Zakryty,
        NL, '   Konec hlavního programu          ');
end.
```

Jednotka P4_01U vypadá takto:

```
{ Viditelnost - jednotka (řadový modul) }
Unit P4_01U;

Interface
(*************  Vyvážené objekty  **************)
const                    {Sekce deklarace konstant}
  NLL  = #13#10#13#10;     {Vyvážená textová konstanta - vynech řádek}
  NL   = #13#10;               {Nový řádek}
  Externi : integer = 1220;     {Vyvážená proměnná}

procedure Tiskni1;         {Vyvážená procedura}

Implementation

(*******  Lokální objekty modulu  *******)
const                    {Proměnné lokální v modulu}
 Globalni : integer = 2220;
 { Proměnná Globální deklarovaná zde je jiná proměnná,
   než stejnojmenná proměnná deklarovaná v hlavním programu }

 Zakryta  : integer = 3220;
 { Proměnná Zakrytá deklarovaná zde je jiná proměnná,
   než stejnojmenná proměnná deklarovaná kdekoliv jinde }

procedure (*****) Tiskni0  (*****);
{ Tato procedura nemá nic společného se stejnojmennou
 procedurou deklarovanou v hlavním modulu }
const
 Zakryta: integer = 3400;
 { Proměnná Zakrytá deklarovaná zde je jiná proměnná,
   než stejnojmenná proměnná deklarovaná kdekoliv jinde}
begin
 write( NLL, '............Tiskni0.....................',
```

```
        NL, 'Objekty viditelné  v proceduře Tiskni0:',
        NL, 'Externí  = ', Externi,
        NL, 'Globalní = ', Globalni,
        NL, 'Zakrytá  = ', Zakryta,
        NL, 'konstanty NL a NLL',
        NL, '............Konec Tiskni0.............');
 Externi  := (Externi  div 100 * 100) + (Externi  mod 10) + 41;
 Globalni := (Globalni div 100 * 100) + (Globalni mod 10) + 41;
 Zakryta  := (Zakryta  div 100 * 100) + (Zakryta  mod 10) + 41;
end;

(*********  Vyvážené podprogramy  *********)
procedure (*****)  Tiskni1  (*****);
const
 Zakryta : integer = 3500;
begin
 write (NLL, '---------Tiskni1----------------------"',
        NL, 'Objekty viditelné  v proceduře Tiskni0:',
        NL, 'Externí  = ', Externi,
        NL, 'Globalní = ', Globalni,
        NL, 'Zakrytá  = ', Zakryta,
        NL, 'konstanty NL a NLL',
        NL, 'a procedura Tiskni0');
 Tiskni0;
 write( NLL, 'Externí  po Tiskni0: ', Externi,
        NL,  'Globalní po Tiskni0: ', Globalni,
        NL,  'Zakrytá  po Tiskni0: ', Zakryta,
        NL,  '---------Konec Tiskni1-----------------');
 Externi  := (Externí  div 100 * 100) + (Externi  mod 10) + 51;
 Globalni := (Globalní div 100 * 100) + (Globalni mod 10) + 51;
 Zakryta  := (Zakrytá  div 100 * 100) + (Zakryta  mod 10) + 51;
end;

(************ Initialization ************)
begin
 write (NLL,'======Inicializace modulu===============',
        NL, 'Objekty viditelné v inicializaci modulu:',
        NL, 'Externí  = ', Externi,
        NL, 'Globalni = ', Globalni,
        NL, 'Zakrytá  = ', Zakryta,
        NL, 'konstanty NL a NLL',
        NL, 'a procedury Tiskni0 a Tiskni1');
 Tiskni0;
 write( NLL,'Externí  po Tiskni0: ', Externi,
        NL, 'Globalní po Tiskni0: ', Globalni,
        NL, 'Zakrytá  po Tiskni0: ', Zakryta );
 Tiskni1;
 write( NLL,'Externí  po Tiskni1: ', Externi,
        NL, 'Globalní po Tiskni1: ', Globalni,
        NL, 'Zakrytá  po Tiskni1: ', Zakryta,
        NL, '======Konec inicializace modulu=========');
 Externi  := (Externi  div 100 * 100) + (Externi  mod 10) + 31;
 Globalni := (Globalni div 100 * 100) + (Globalni mod 10) + 31;
 Zakryta  := (Zakryta  div 100 * 100) + (Zakryta  mod 10) + 31;
end.
```

V dalším textu se budeme setkávat se dvěma novými klíčovými slovy: **static** a **extern**. Tato klíčová slova patří mezi tzv. **specifikátory paměťové třídy** a budeme je proto někdy nazývat zkráceně pouze specifikátory. (S dalšími specifikátory paměťové třídy se setkáme v příští kapitole.)

Specifikátory paměťové třídy se předřazují před zbytek deklarace objektu. Z formálních důvodů – kvůli umístění v deklaraci – se mezi specifikátory paměťové třídy zařazuje i klíčové slovo **typedef**, i když jeho funkce je od ostatních specifikátorů naprosto odlišná.

Datový typ je vždy lokální v oblasti, v níž je definován. Na rozdíl od Pascalu **nemůže v C++ být datový typ deklarován jako externí**. Budu-li proto v dalším textu hovořit o externích objektech, nebude se to vztahovat na datové typy.

Potřebujeme-li vyvézt datový typ, definujeme jej v hlavičkovém souboru modulu (soubor s příponou .H nebo .HPP), kterou příkazem **#include** vložíme do zdrojového textu všech modulů, které chtějí daný datový typ dovézt.

Přejděme ale k lokalitě a viditelnosti. V C++ sice podprogram nemůže mít své lokální podprogramy, ale na druhou stranu zase může být deklarace součástí kteréhokoliv složeného příkazu – bloku. V dalším textu budeme rozlišovat deklarace, které jsou uvnitř nějakého bloku, a deklarace, které nejsou uvnitř žádného bloku.

Deklarace uvnitř bloku

- ✧ Všechny objekty definované uvnitř bloku jsou implicitně v tomto bloku lokální. Jak jsme si již řekli, **výjimku z tohoto pravidla tvoří návěští**, která jsou vždy viditelná v celém těle funkce, v níž jsou deklarována, tedy i před místem své deklarace a v nadřazených blocích.
- ✧ Jsou-li v různých blocích deklarovány dva stejnojmenné objekty bez použití specifikátoru **extern**, jsou to pro překladač naprosto různé objekty, které mají čirou náhodou stejné jméno.
- ✧ Všechny deklarace stejnojmenných objektů obsahující specifikátor **extern** se vztahují k jednomu a témuž externímu objektu. Zda se jedná o objekt dovážený nebo vyvážený závisí na tom, ve kterém modulu je daný objekt definován (víme, že každý objekt smí být definován pouze jednou).
- ✧ Objekty deklarované se specifikátorem **extern** (tyto objekty **nelze v bloku definovat**) jsou externí objekty, jejichž jméno je však v daném bloku lokální a tedy mimo tento blok neviditelné – ledaže by bylo znovu deklarováno.

Deklarace mimo bloky

- ✧ Proměnné a funkce, které jsou deklarovány mimo těla funkcí (tj. nikoliv uvnitř nějakého bloku) a jejichž deklarace neobsahují žádný specifikátor paměťové třídy, považuje překladač za externí.
- ✧ Proměnné a funkce, které mají být v daném modulu lokální, musíme deklarovat se specifikátorem **static**.
- ✧ Konstanty jsou implicitně považovány za lokální (což ovšem neznamená, že jejich lokalitu nemůžeme zdůraznit specifikátorem **static**). Mají-li být externí, musíme je deklarovat se specifikátorem **extern**.
- ✧ Dovážené konstanty a proměnné musíme deklarovat se specifikátorem paměťové třídy **extern** – nejlépe v hlavičkovém souboru modulu, který tyto objekty váží.
- ✧ Deklaraci konstanty nebo proměnné, která neobsahuje specifikátor **extern**, považuje překladač za definici. Podobně považuje za definici i deklaraci, která obsahuje specifikátor **extern** a zároveň inicializaci.
- ✧ Je-li daná konstanta, proměnná, či funkce deklarována jednou jako lokální a jinde jako externí, považuje ji překladač za externí.
- ✧ Je-li stejnojmenný externí objekt definován v několika modulech, považuje to překladač (přesněji sestavovací program) za chybu, protože se domnívá, že se jedná o několik objektů, které od sebe nedokáže rozlišit.

Předchozí zásady určování lokality a viditelnosti proměnných konstant a datových typů deklarovaných na různých místech programu si můžete ověřit na příkladu C4 – 1. Každý výstupní příkaz tiskne hodnoty všech v daném místě viditelných proměnných a konstant.

```
/*   Příklad C4 – 1   */
//Viditelnost - hlavní modul
//
//Nechcete-li program krokovat, spusťte jej z příkazové
//řádky tak, že mu přesměrujete standardní výstup
//do definovaného souboru nebo na tiskárnu - např.:
//         C4-1  >PRN
//*************************************************************

#include <iostream.h>;
#include "C4-1A.H";

/***** Lokální objekty modulu *****/
//Hodnoty přiřazované proměnným mají následující interpretaci:
//  1. cifra = jméno: Externí=1, Globální=2, Zakrytá=3
//  2. cifra = místo deklarace: číslování - viz následující bod
//  3. cifra = místo poslední modifikace:
//     0 - hlavní modul
//     1 - procedura Tiskni0 v hlavním modulu
//     2 - dovážený modul
//     3 - inicializace dováženého modulu (jen modifikace)
//     4 - procedura Tiskni0 v dováženém modulu
```

```
//     5 - dovážená procedura Tiskni1
//     6 - funkce main v hlavním modulu
//     4. cifra = celkový počet modifikací

int Globalni = 2000;
//Proměnná Globální deklarovaná zde je jiná proměnná,
//než stejnojmenná proměnná deklarovaná v dováženém modulu

int Zakryta  = 3000;
//Proměnná Zakrytá deklarovaná zde je jiná proměnná,
//než stejnojmenná proměnná deklarovaná kdekoliv jinde

static void Tiskni0();

/**************   Hlavní program   **************/

void /*****/  main  /*****/
()
{
 int Zakryta = 3660;
 //Proměnná Zakrytá deklarovaná zde zakrývá stejnojmennou
 //proměnnou deklarovanou v hlavním modulu
 cout << NLL
      << "   Hlavní program                    "
      << "\nObjekty viditelné v hlavním programu: "
      << "\nExterní   = " << Externi
      << "\nGlobalní  = " << Globalni
      << "\nZakrytá   = " << Zakryta
      << "\n::Zakrytá = " << ::Zakryta
      << "\nkonstanta NLL"
      << "\na procedury Tiskni0 a Tiskni1";
 Tiskni0();
 cout << NLL
      <<   "Externí   po Tiskni0: " << Externi
      << "\nGlobalní  po Tiskni0: " << Globalni
      << "\nZakrytá   po Tiskni0: " << Zakryta
      << "\n::Zakrytá po Tiskni0: " << ::Zakryta;
 Tiskni1();
 cout << NLL
      <<   "Externí   po Tiskni1: " << Externi
      << "\nGlobalní  po Tiskni1: " << Globalni
      << "\nZakrytá   po Tiskni1: " << Zakryta
      << "\n::Zakrytá po Tiskni1: " << ::Zakryta
      << "\n   Konec hlavního programu           ";
}

void /*****/  Tiskni0  /*****/
()
//Tato procedura nemá nic společného se stejnojmennou
//procedurou deklarovanou v dováženém modulu }
{
 int Zakryta = 3110;
 //Proměnná Zakrytá deklarovaná zde zakrývá stejnojmennou
 //proměnnou deklarovanou v hlavním modulu
 //Lokální proměnná je pokaždé znovu inicializována
 cout << NLL
      << "======Tiskni0=========================="
```

```
     << "\nObjekty viditelné  v proceduře Tiskni0: "
     << "\nExterní   = " << Externi
     << "\nGlobalní  = " << Globalni
     << "\nZakrytá   = " << Zakryta
     << "\n::Zakrytá = " << ::Zakryta
     << "\nkonstanta NLL"
     << "\na procedura Tiskni1";
 Tiskni1();
 cout << NLL
     <<   "Externí   po Tiskni1: " << Externi
     << "\nGlobalní  po Tiskni1: " << Globalni
     << "\nZakrytá   po Tiskni1: " << Zakryta
     << "\n::Zakrytá po Tiskni1: " << ::Zakryta
     << "\n======konec Tiskni0==================";
 Externi   = (Externi  / 100 * 100) + (Externi  % 10) + 11;
 Globalni  = (Globalni / 100 * 100) + (Globalni % 10) + 11;
 Zakryta   = (Zakryta  / 100 * 100) + (Zakryta  % 10) + 11;
 ::Zakryta = (::Zakryta/ 100 * 100) + (::Zakryta% 10) + 11;
}
```

Podívejme se ještě na hlavičkový soubor *C4-1A.H*:

```
// Soubor C4-1A.H
// Viditelnost  -  hlavičkový soubor řadového modulu

extern const char* NLL;    //Vyvážená textová konstanta - vynech řádek

extern int Externi;        //Vyvážená proměnná

void Tiskni1();            //Vyvážená procedura
```

Řadový modul najdete v souboru *C4 – 1A.CPP*.

```
//Soubor C4 – 1A.CPP
//Viditelnost - řadový modul

#include <iostream.h>
#include "C4-1A.H"

/**  Definice vyvážených objektů  **/

extern const char* NLL     = "\n\n";
extern       int   Externi = 1220;

/*******  Lokální objekty modulu  *******/
static int Globalni = 2220;
//Proměnná Globální deklarovaná zde je jiná proměnná,
//než stejnojmenná proměnná deklarovaná v dováženém modulu

static int Zakryta  = 3220;
//Proměnná Zakrytá deklarovaná zde je jiná proměnná,
//než stejnojmenná proměnná deklarovaná kdekoliv jinde

static void Tiskni0();
//Lokální procedura Tiskni0 je jinou procedurou,
//než stejnojmenná procedura lokální v hlavním modulu
```

```
/*********  Vyvážené podprogramy  *********/
void /*****/   Tiskni1   /*****/ ()
{
 int Zakryta = 3550;
 //Proměnná Zakrytá deklarovaná zde zakrývá stejnojmennou
 //proměnnou deklarovanou jako lokální v modulu
 //Proměnná lokální v bloku je pokaždé znovu inicializována
 cout << NLL
      << "---------Tiskni1----------------------"
      << "\nObjekty viditelné  v proceduře Tiskni0:"
      << "\nExterní   = " << Externi
      << "\nGlobalní  = " << Globalni
      << "\nZakrytá   = " << Zakryta
      << "\n::Zakrytá = " << ::Zakryta
      << "\nkonstanta NLL"
      << "\na procedura Tiskni0";
 Tiskni0();
 cout << NLL
      <<   "Externí   po Tiskni0: " << Externi
      << "\nGlobalní  po Tiskni0: " << Globalni
      << "\nZakrytá   po Tiskni0: " << Zakryta
      << "\n::Zakrytá po Tiskni0: " << ::Zakryta
      << "\n---------Konec Tiskni1-----------------";
 Externi   = (Externi  / 100 * 100) + (Externi  % 10) + 51;
 Globalni  = (Globalni / 100 * 100) + (Globalni % 10) + 51;
 Zakryta   = (Zakryta  / 100 * 100) + (Zakryta  % 10) + 51;
 ::Zakryta = (::Zakryta/ 100 * 100) + (::Zakryta% 10) + 51;
}
/*********   Lokální podprogramy   *********/
static void /*****/   Tiskni0   /*****/ ()
//Tato procedura nemá nic společného se stejnojmennou
//procedurou deklarovanou v hlavním modulu
{
 int Zakryta = 3440;
 //Proměnná Zakrytá deklarovaná zde zakrývá stejnojmennou
 //proměnnou deklarovanou jako lokální v modulu
 //Proměnná lokální v bloku je pokaždé znovu inicializována
 cout << NLL
      << ".............Tiskni0....................."
      << "\nObjekty viditelné  v proceduře Tiskni0:"
      << "\nExterní   = " << Externi
      << "\nGlobalní  = " << Globalni
      << "\nZakrytá   = " << Zakryta
      << "\n::Zakrytá = " << ::Zakryta
      << "\nkonstanta NLL"
      << "\n.............Konec Tiskni0.............";
 Externi   = (Externi  / 100 * 100) + (Externi  % 10) + 41;
 Globalni  = (Globalni / 100 * 100) + (Globalni % 10) + 41;
 Zakryta   = (Zakryta  / 100 * 100) + (Zakryta  % 10) + 41;
 ::Zakryta = (::Zakryta/ 100 * 100) + (::Zakryta% 10) + 41;
}
/***   Inicializace a finalizace   ***/
void /*****/  Modul_Init  /*****/
()
//Inicializace je v důsledku místa své definice a v důsledku
```

```
//absence jakékoliv předběžné deklarace neviditelná z předchozích
//částí modulu. Musí však být definována jako externí,
//aby ji sestavovací program našel a zprostředkoval její
//provedení před vlastním zahájením programu.
{
 int Zakryta = 3330;
 //Proměnná Zakrytá deklarovaná zde zakrývá stejnojmennou
 //proměnnou deklarovanou jako lokální v modulu
 cout << NLL << "======Inicializace modulu==============="
      << "\nObjekty viditelné v inicializaci modulu:"
      << "\nExterní   = " << Externi
      << "\nGlobalní  = " << Globalni
      << "\nZakrytá   = " << Zakryta
      << "\n::Zakrytá = " << ::Zakryta
      << "\nkonstanta NLL"
      << "\na procedury Tiskni0 a Tiskni1";
 Tiskni0();
 cout << NLL
      <<   "Externí   po Tiskni0: " << Externi
      << "\nGlobalní  po Tiskni0: " << Globalni
      << "\nZakrytá   po Tiskni0: " << Zakryta
      << "\n::Zakrytá po Tiskni0: " << ::Zakryta;
 Tiskni1();
 cout << NLL
      <<   "Externí   po Tiskni1: " << Externi
      << "\nGlobalní  po Tiskni1: " << Globalni
      << "\nZakrytá   po Tiskni1: " << Zakryta
      << "\n::Zakrytá po Tiskni0: " << ::Zakryta
      << "\n======Konec inicializace modulu=========";
 Externi   = (Externi  / 100 * 100) + (Externi  % 10) + 31;
 Globalni  = (Globalni / 100 * 100) + (Globalni % 10) + 31;
 Zakryta   = (Zakryta  / 100 * 100) + (Zakryta  % 10) + 31;
 ::Zakryta = (::Zakryta/ 100 * 100) + (::Zakryta% 10) + 31;
}
#pragma startup Modul_Init
```

4.2 Statické, automatické a registrové proměnné

V minulé kapitole jsme si probrali jednotlivé kategorie objektů z hlediska jejich viditelnosti a dosažitelnosti. To byly statické kategorie, které ovlivňovaly proces překladu. V této kapitole se budeme zabývat problematikou **životnosti** (**doby trvání**) objektů a některými otázkami s životností souvisejícími – tedy kategoriemi, které souvisí přímo s během programu.

Na datových typech se v průběhu programu nemá co změnit, takže o jejich životnosti v podstatě nemá smysl hovořit. Celá tato kapitola se proto bude zabývat životností proměnných (a případně konstant).

V podstatě rozeznáváme tři kategorie životnosti: životnost **statickou**, **dynamickou** a **automatickou**:

- ✧ Proměnné se statickou životností „žijí“ nepřetržitě po celou dobu běhu programu, protože vznikají a zanikají společně s celým programem.
- ✧ Proměnné (a v C++ i konstanty) s dynamickou životností žijí od chvíle, kdy je programátor explicitně vytvoří (zřídí), až do chvíle, kdy je explicitně zruší – jejich životnost je tedy plně v jeho moci. Zřizování a rušení proměnných (a v C++ i konstant) však patří k náročnějším oblastem programování, o kterých budeme hovořit později.
- ✧ Proměnné s automatickou životností vzniknou ve chvíli, kdy se aktivuje oblast, v níž jsou lokální, a ruší se ve chvíli, kdy program tuto oblast opouští.

Všechny externí proměnné a proměnné lokální v některém modulu mají životnost statickou, aniž bychom to mohli nějak ovlivnit. Životnost proměnných lokálních v podprogramech a blocích však ovlivnit můžeme.

V Pascalu určujeme životnost proměnné lokální v podprogramu tím, že ji definujeme jako inicializovanou nebo neinicializovanou. Neinicializované proměnné lokální v podprogramech mají v Pascalu automatickou životnost, inicializované proměnné lokální v podprogramech mají statickou životnost. (Připomeňme si, že inicializované proměnné deklarujeme v Pascalu v sekci konstant.)

Chceme-li, aby lokální proměnná měla automatickou životnost, nesmíme ji inicializovat v rámci definice, ale až později, přiřazovacím příkazem v těle programu. Naproti tomu lokální proměnnou, která má mít statickou životnost, musíme inicializovat v definici (tj. musíme ji deklarovat jako tzv. typovou konstantu), a to i v případě, že vzhledem k charakteru programu žádnou inicializaci nepotřebuje (to se ovšem u statických proměnných stává zřídka).

V C++ mají všechny lokální proměnné implicitně automatickou životnost a nezáleží na tom, zda je inicializujeme či nikoliv. Chceme-li, aby některá lokální proměnná měla statickou životnost, musíme ji deklarovat specifikátorem paměťové třídy **static**.

Pokud bychom chtěli zdůraznit automatickou životnost nějaké lokální proměnné, můžeme ji deklarovat se specifikátorem **auto**. Tento specifikátor se ale prakticky nepoužívá, protože jeho uvedením nebo vynecháním nic nezměníme (nenajdete jej ani v demonstračním programu).

Pro lokální proměnné s automatickou životností se však používá specifikátor **register**. Tím oznamujeme překladači, že se domníváme, že dotyčná proměnná se v bloku používá tak hojně, že si zaslouží, aby byla místo v paměti uchovávána přímo v některém volném registru procesoru. Tím by se všechny operace s touto proměnnou výrazně urychlily.

Překladač Borland C++ používá pro registrové proměnné registry SI a DI, příp. ESI a EDI. Pokud je některý z nich k dispozici a pokud se proměnná, o jejíž uložení jsme specifikátorem **register** požádali, do registru vejde, překladač ji tam umístí. Pokud se mu to nepodaří, pracuje s ní jako s běžnou automatickou proměnnou.

Poznámka:
Proběhne-li překlad bez jakýchkoliv varovných zpráv, neznamená to ještě, že všechny registrové proměnné se překladači podařilo umístit v registrech. Neúspěch při usazování registrových proměnných v registrech totiž překladač nepovažuje za chybu ani za důvod k varování, a proto je nehlásí.

Umisťování automatických proměnných do registrů můžeme ovlivnit nastavením přepínače *Register Variables* v dialogovém okně *Options* | *Compiler* | *Optimizations* (BC++ 3.1) resp. *Options* | *Project* | *Compiler* | *Code generation* (BC++ 4.0 a pozdější). Význam jednotlivých nastavení tohoto přepínače je následující:

- ✧ *None* – zakáže překladači umísťovat do registrů jakékoliv proměnné, i ty, které jsme v programu deklarovali se specifikátorem **register**.
- ✧ *Register keyword* – povolí překladači umístit (přesněji pokusit se umístit) do registrů pouze proměnné deklarované se specifikátorem **register**.
- ✧ *Automatic* – překladač vždy umísťuje některé z lokálních automatických proměnných (většinou prvé dvě, ale nastavíte-li v Borland C++ 3.0 a pozdějších některé optimalizace, volí překladač „inteligentněji“) do registrů. Pokud jsou mezi automatickými proměnnými proměnné deklarované se specifikátorem **register**, mají při umísťování přednost.

Proměnné se statickou životností jsou velice užitečné ve chvíli, kdy potřebujeme, aby si podprogram mezi jednotlivými voláními zapamatoval některá data – např. tak, jako v následujících příkladech.

Příklad: filtr pro tisk

Nejprve bychom si měli povědět něco o tom, kterým programům říkáme filtry. Pokud jste pozorně četli příručky operačního systému DOS, víte, že filtry jsou takové programy, které čtou ze standardního vstupu a zapisují na standardní výstup. Mezi programy operačního systému jsou tři filtry – MORE, SORT a FIND.

My si zkusíme naprogramovat filtr, který bude ze standardního vstupu číst nějaký text a na standardní výstup jej bude předávat upravený do tvaru vhodného pro tiskárnu, přičemž sám tiskárnu inicializuje do vhodného stavu. Tím myslíme tisk kondenzovaným písmem (nastaví se povelem Ctrl-O), řádkování zhuštěné na 8 řádek na palec (povel Esc 0) a pro případ, že by nebyla délka stránky definována v palcích, ale v řádcích, nastavíme ještě 96 řádků (Esc C`) na 12" stránce (na 11" stránce by to bylo 88 řádků s řídicí posloupností Esc CX).

Program bude číst znaky ze vstupu jeden po druhém. Na počátku každého řádku napíše číslo tohoto řádku a pak jeho obsah. Bude schopen nahradit tabulátory mezerami k příští tabulační zarážce a bude umět po vytištění 90 řádků sám přejít na novou stránku, přičemž na počátku stránky vždy uvede její pořadové číslo.

Aby byl program dostatečně komfortní, musí umět přejít na novou stránku i v případě, že v textu narazí na znak přechodu na novou stránku (tzv. *form feed* (FF) s kódem 12_D resp. $0C_H$).

Než se pustíme do vlastního programování, musíme si něco říci o práci se standardním vstupem a výstupem.

První, co musíme umět, je převzetí znaku. V Pascalu to již umíme, v C++ by nám však doposud používaný postup při řešení naší úlohy moc nepomohl. C++ totiž při dosud používaném čtení znaku nejprve přeskočí všechny bílé znaky a jako vstup nám předá prvý nebílý znak. To samozřejmě nechceme, a proto budeme číst znaky pomocí funkce *cin.get*, která bílé znaky nepřeskakuje (nelekejte se tečky v identifikátoru – časem si povíme, proč tam musí být).

Druhá věc, bez níž se neobejdeme, je rozpoznání konce vstupujícího textu. V Pascalu k tomu slouží funkce *eof* (*end of file* – konec souboru), která vrací *ANO* v případě, že jsme již veškerá vstupní data vyčerpali. V C++ platí, že od chvíle, kdy vyčerpáme vstup, čteme znaky s hodnotou *EOF*. Aby to však fungovalo, musíte potlačit volbu *Unsigned char* v dialogovém okně *Options* | *Compiler* | *Code generation*. V kapitole věnované znakům si povíme, proč to musí být právě tak.

Jak víme, není v IDE možné přesměrovat standardní vstup programů, a proto budeme muset během ladění zadávat vstup z klávesnice. Konec vstupních dat pak oznámíme zadáním řádku s jedním znakem Ctrl-Z (na obrazovce bude zakreslen jako^Z).

```
(*   Příklad P4 – 2   *)
{ Filtr pro tisk }
const
  RS      =  90;               {Počet řádků tištěných na stránce}
  ST      =   8;               {Počet sloupců tabulátory}
  NL      =  #13#10;
  NLL     =  #13#10#13#10;

  PRINIT = #15#27'0'#27'C'#96;  {Inicializace tiskárny}
(* inicializace tiskárny:
      #15 = Ctrl+O        kondenzované písmo
      #27'0' = Esc-0      8 řádků na palec
      #27'C'#96 = Esc-0-n 96 řádků na stránku
*)

var
  ch: char;                    {Čtený znak}
  Sloupec : integer;           {Číslo aktuálního sloupce}

procedure (*****)   NovyRadek  (*****);
const
  ANO = TRUE;
  NE  = FALSE;
  Poprve : Boolean = ANO;      {První řádek celého textu}
  Stranka: integer = 1;        {Číslo aktuální stránky}
  Radek  : integer = 1;        {Číslo aktuálního řádku}
  RnS    : integer = RS;       {Pořadí řádku na stránce}
```

```
  Mezer  : integer = 4;          {Počet mezer před číslem}
  Zmena  : integer = 10;         {Kdy se zvýší počet cifer}
var
  i: integer;                    {Pomocná proměnná}
begin
  if(  (ch = #12)  or            {Nová stránka}
       (RnS = RS)  )then         { nebo plná stránka}
       begin
           if( Poprve )
           then Poprve := NE     {Před tiskem neodstránkujem}
           else write( #12 );    {Odstránkovat}
           write( 'Stránka: ', stranka, NLL);
           Stranka := Stranka + 1;
           RnS := 0;
       end;
  if( Radek = Zmena ) then       {Přibyla cifra}
       begin
           Zmena := Zmena * 10;  {Kdy očekáváme další}
           Mezer := Mezer - 1;   {Ubude vedoucích mezer}
       end;
  write( NL );
  i := Mezer;                    {Tisk vedoucích mezer}
  while( i > 0 ) do
       begin
           write(' ');
           i := i - 1;
       end;
  write( Radek, '  ' );          {Očíslování}
  Radek   := Radek + 1;
  Sloupec := 0;
end;

(**************   Hlavní program   **************)
(*  Následující  proměnné  by  měly  být  lokální  v  hlavním  programu
  (abychom
   je nemohli omylem použít v některé z procedur a funkcí). To však
   Pascal neumí, takže jediná možnost, jak je skrýt před zbytkem
   programu, je deklarovat je až po všech procedurách.
*)

var
  Mezer   : integer;             {Počet   mezer   k   další   tabulační
  zarážce}

begin
  read( ch );
  if( not EOF )then                          {Není-li soubor prázdný}
  begin
       write( PRINIT );
       NovyRadek;
  end;
  while( not EOF ) do                        {Test na konec souboru}
  begin
       if( ch = #12 )then                    {Nová stránka}
           NovyRadek
       else if( ch = #09 ) then              {Tabulátor}
```

```
      begin
          Mezer := ST - (Sloupec mod ST); {Kolik mezer zbývá?}
          Sloupec := Sloupec + Mezer;               {Nový    aktuální
  sloupec}
          while( Mezer > 0 ) do           {Vytiskni mezery}
          begin
               write( ' ' );
               Mezer := Mezer - 1;
          end;
      end
      else
      begin                               {Ostatni znaky}
          write(ch);                      {jenom opíšeme}
          Sloupec := Sloupec + 1;
      end;
      while( EOLN and not EOF )do
      begin                           {Následuje konec řádku?}
          ch := ' ';                  {Aby je nezmátl případný #12}
          NovyRadek;
          read( ch );
          read( ch );                 {NL = dvojice znaků}
      end;
      read( ch );
  end;
  write( #12 );                       {Na závěr odstránkuj}
end.
```

```
/*   Příklad C4 – 2   */
// Filtr pro tisk
#include <iostream.h>

const int RS = 90;                    //Tištěných řádků na stránce
const int ST =  8;                    //Počet sloupců tabulátory
const POPRVE = -2;                    //Neexistuje znak - příznak 1.str
const char* PRINIT = "\xF\x1B""0\x1B""C`"; //Inicializace tiskárny

int   ch = POPRVE;                    //Čtený znak -
int   Sloupec = 0;                    //Číslo aktuálního sloupce

void NovyRadek();

/*****************   Hlavní program   *****************/
void /*****/   main  /*****/ ()
{
  cout << PRINIT;                     //Inicializujeme tiskárnu
  NovyRadek();                        //ch==POPRVE => neodstránkuje se
  while( (ch = cin.get()) != EOF )    //Test na konec souboru
  {
       switch( ch )
       {
           case '\f':                 //Nová stránka
           case '\n':                 //Nový řádek
                NovyRadek();
                break;
           case '\t':                 //Tabulátor
                                      //Kolik mezer zbývá?
```

```
                int Mezer = ST - Sloupec%ST;
                Sloupec  += Mezer;    //Nový aktuální sloupec
                while( Mezer-- ) cout << ' ';  //Vytiskni mezery
                break;
           default:
                cout << (char)ch;     //Ostatni znaky jenom opíšeme
                Sloupec++;
      }
  }
  cout << '\f';                       //Na závěr odstránkuj
}

void /*****/   NovyRadek  /*****/ ()
{
  static Stranka = 1;           //Číslo aktuální stránky
  static Radek   = 1;           //Číslo aktuálního řádku
  static RnS     = RS;          //Pořadí řádku na stránce
  static Mezer   = 4;           //Počet mezer před číslem řádku
  static Zmena   = 10;          //Kdy se zvýší počet cifer čísla řádku
  if(  (ch  == '\f'  ) ||       //Nová stránka
       (RnS == RS    ) ||       //nebo plná stránka
       (ch  == POPRVE) )        //nebo počátek textu
  {
       if( ch != POPRVE ) cout << '\f';    //Odstránkovat
       cout << "Stránka: " << Stranka++ << "\n\n";
       RnS = 1;                 //Příští řádek bude na stránce první
  }
  else                          //Pouze nový řádek
  {
       cout << '\n';
       RnS++;
  }
  if( Radek == Zmena )          //Přibyla cifra
  {
       Zmena *= 10;             //Kdy  očekáváme  další  změnu  počtu
  cifer
       Mezer--;                 //Ubyde vedoucích mezer
  }
  for( int i = Mezer;           //Tisk mezer před číslem řádku
       i-- > 0;
       cout << ' '
       );
       cout << Radek++ << "  "; //Očíslování řádku
       Sloupec = 0;
}
```

5. Procedury a funkce

Všechny naše dosavadní podprogramy se silně specializovaly na řešení konkrétních úloh. Pokud jsme např. chtěli Karla naučit ohradit nějakou čtvercovou oblast dvorku značkami, museli jsme přesně vědět, jak má být daná oblast velká. Pokud měl Karel ohradit oblast jiné velikosti, museli jsme napsat nový podprogram.

S obdobnými problémy se potýkali i programátoři na počátku počítačové éry. Je jasné, že takovýto stav je nemohl dlouho uspokojovat. Potřebovali mít možnost definovat podprogram tak, aby mohli např. Karlovi v našem předchozím příkladě až na poslední chvíli říci, jak má být ohrazovaná oblast veliká, a aby vystačili pro všechny velikosti ohrazovaných oblastí s jedním univerzálním programem.

Bylo by sice možné definovat několik globálních proměnných, do nichž by volající programy ukládaly hodnoty, které by měnily chování volaného podprogramu (vzpomeňte si např. na proměnnou *Plateb5* v příkladech P3 – 5 a C3 – 5), ale jak sami jistě na první pohled vidíte, takové řešení je přece jen poněkud těžkopádné. Mimo jiné i proto, že těžko zabráníte použití takovýchto proměnných v jiných částech programu, při tom snadno zapomenete na jejich původní účel a přepíšete si důležitá data.

Aby nemohlo docházet ke kolizím, musel by mít každý podprogram své vlastní jednoúčelové globální proměnné, které by však zbytečně zabíraly místo v paměti i v době, kdy by vůbec nebyly třeba.

Zkrátka a dobře, tudy cesta nevede. Světlo světa proto spatřily **parametry** procedur a funkcí, které můžeme prozatím považovat za lokální proměnné volaného podprogramu, které volající program v okamžiku volání inicializuje.

Aby to nebylo tak jednoduché, zavedeme si ještě dva pomocné termíny: identifikátory parametrů v definici podprogramu budeme nazývat **formální parametry**, protože ve skutečnosti pouze formálně označují v těle podprogramu místa, kde bude dotyčný podprogram pracovat se **skutečnými parametry**. Skutečné parametry jsou hodnoty, které se za formální parametry v okamžiku volání dotyčného podprogramu dosadí.[5]

S podprogramy s parametry jsme se již v našich dosavadních programech setkali – patří mezi ně např. procedury **NačtiDvorek** a **Krát**. Proto byste asi sami odvodili, že podprogram s parametry voláme stejně jako podprogram bez parametrů, pouze do závorek za jeho identifikátor napíšeme seznam skutečných parametrů (tj. předávaných hodnot). V tomto seznamu **záleží na pořadí**. Pořadí skutečných parametrů musí totiž odpovídat pořadí deklarací formálních parametrů v deklaraci podprogramu.

---

[5] V některých porevolučních příručkách různých programovacích jazyků jsme se setkali s trochu jinou terminologií: formální parametry jsou zde označovány jako **parametry** a skutečné parametry jsou pak označovány jako **argumenty**. Protože se nám zdá, že se takovéto označování bude zejména začátečníkům plést, setrváme u terminologie, která se u nás používá minimálně od poloviny šedesátých let.

Deklarace podprogramů bez parametrů již známe. Deklarace podprogramů s parametry se od nich liší tím, že v kulatých závorkách za identifikátorem deklarovaného podprogramu musíme uvést seznam deklarací jeho formálních parametrů. Přitom v Pascalu oddělujeme jednotlivé deklarace formálních parametrů středníky, kdežto v C++ čárkami.

Tvar jednotlivých deklarací formálních parametrů závisí na mechanismu předávání deklarovaného parametru. V Pascalu i C++ se používají dva mechanismy předávání parametrů: **předávání hodnotou** a **předávání odkazem** (**referencí**).

5.1 Vstupní parametry – parametry předávané hodnotou

Formální parametr předávaný hodnotou představuje v těle podprogramu lokální automatickou proměnnou, která je při vstupu do podprogramu inicializována hodnotou skutečného parametru. V programech se používají jako tzv. **vstupní parametry**, tedy jako parametry, kterými volanému podprogramu předáváme nějakou hodnotu. V obou jazycích se deklarují stejně, podobně jako obyčejné proměnné:

V Pascalu uvedeme identifikátor formálního parametru (případně seznam identifikátorů oddělených čárkami), dvojtečka a typ daného parametru (parametrů v seznamu). V C++ uvedeme naopak nejprve typ daného formálního parametru a za ním jeho identifikátor; v tomto jazyku musíme formální parametry deklarovat jeden po druhém.

Pro parametry předávané hodnotou platí jedna příjemná věc: skutečný parametr nemusí být stejného typu jako jeho formální protějšek. Stačí, když je s ním kompatibilní vzhledem k přiřazení.

Dosti teorie, podívejme se na příklady. Na doplňkové disketě najdete soubory *P5–0.PAS* resp. *C5–0.CPP*, obsahující definici procedury *Vyznačkuj*. V ní naučíme Karla vyznačkovat obdélník o rozměrech, určených parametry. Zde si jako příklad ukážeme definici funkce *Fakt*, která vypočte a vrátí faktoriál svého parametru.

```
(*   Příklad P5 – 1   *)
function Fakt ( Cislo: integer ): real;
var
  F: real;
begin
  F := 1;
  repeat
      F := F * Cislo;
      Cislo:=Cislo-1;
  until (Cislo < 1);
  Fakt := F;
end;
```

Deklarace téže funkce bude v C++ mít tvar

```
/*   Příklad C5 – 1   */
double Fakt ( int Cislo )
{
  double F = Cislo;
  if( !Cislo )  return 1;
  while( (Cislo = Cislo-1) != 0 )
      F = F * Cislo;
  return( F );
}
```

5.2 Vstupně-výstupní parametry – parametry předávané odkazem

V předchozím odstavci jsme si řekli, že parametry předávané hodnotou se chovají jako lokální proměnné v daném podprogramu. Formální parametry předávané odkazem se chovají tak trochu jako globální proměnné daného podprogramu. Můžeme se na ně dívat jako na lokální jména pro skutečné parametry.[6]

Parametry předávané odkazem se používají v případech, kdy bychom chtěli daný parametr používat nejen jako vstupní, ale také (anebo jenom) jako výstupní, tedy v případech, kdy po podprogramu žádáme, aby hodnotu parametru nějakým způsobem změnil.

Kromě toho se parametry předávané odkazem používají také v případech, kdy předávaný parametr zabírá mnoho místa v paměti, takže považujeme za výhodnější předat pouze jeho adresu (tedy předat jej odkazem), a to i v případě, že jej nechceme v podprogramu měnit.

Probereme-li si doposud známé datové typy, hned nás asi napadne, že o předávání odkazem si asi vzhledem ke své velikosti budou „koledovat" textové řetězce. Rozhodnout se, zda textový řetězec předáme hodnotou či odkazem, můžeme pouze v Pascalu. V C++ se textové řetězce předávají odkazem vždy.

Deklarace parametrů předávaných odkazem v Pascalu začínají klíčovým slovem **var**, za kterým následuje seznam identifikátorů a typ. V C++ vložíme mezi identifikátor typu a identifikátor formálního parametru znak **&** (et, ampersand).

V obou jazycích platí, že skutečným parametrem předávaným odkazem musí být vždy objekt, který má nějakou adresu – např. proměnná – a musí být stejného typu jako formální parametr. V žádném případě nemůžeme předat podprogramu odkazem konstantu a už vůbec ne výraz. (Připomínáme, že pascalské inicializované proměnné nejsou konstanty, i když se mezi nimi v sekci konstant definují.)

---

[6] Ve skutečnosti se při předávání odkazem předává podprogramu adresa skutečného parametru.

Pokud chcete podprogramu předat prostřednictvím parametru předávaného odkazem konstantu nebo výraz, musíte nejprve vytvořit nějakou pomocnou proměnnou, přiřadit jí hodnotu dotyčného výrazu a pak ji předat jako skutečný parametr předávaný odkazem.[7]

Pokud chceme v C++ předávat odkazem i konstanty, musíme daný parametr deklarovat jako konstantu, předávanou odkazem.

Vlastnosti parametrů předávaných odkazem si ukážeme na následujících příkladech. Procedura *ProhodA* prohodí obsahy dvou celočíselných proměnných. Protože to znamená, že chceme měnit obsah skutečných parametrů, musíme použít parametry předávané odkazem.

```
(*   Příklad P5 – 2   *)
procedure (*****)  ProhodA  (****¶*)
( var i1: integer;  var i2: integer );
var
  ip: integer;                                {Pomocná proměnná}
begin
  ip := i1;   i1 := i2;   i2 := ip;
end;
```

Táž funkce v C++:

```
/*   Příklad C5 – 2   */
/*****/  Prohod  /*****/
( int& i1, int& i2 )
{
  register ip = i1;   i1 = i2;   i2 = ip;
}
```

Jsou-li *a* a *b* proměnné typu *integer* (v Pascalu) resp. **int** (v C++), můžeme jejich obsahy prohodit např. příkazem

```
Prohod(a, b);
```

[7] Borland C++ nám dovolí předat odkazem cokoliv. Pokusíme-li se předat odkazem konstantu či výraz, překladač si sám zřídí dočasnou proměnnou, přiřadí jí hodnotu dotyčné konstanty či výrazu a tuto proměnnou předá odkazem volanému podprogramu.
Tato univerzalita však v sobě skrývá jedno nebezpečí: dočasnou pomocnou proměnnou překladač zřizuje i při konverzích typů – např. když reálnému formálnímu parametru předáváme celočíselný skutečný parametr. Pokud podprogram hodnotu svého odkazem předaného parametru změní, našeho skutečného parametru se to nijak nedotkne, protože novou hodnotu dočasné proměnné zpět nepřevezme.
Naštěstí nás na toto nebezpečí dokáže překladač upozornit, takže je v případě, kdy nás nová hodnota bude zajímat, můžeme ošetřit stejně, jako bychom to dělali v Pascalu – tj. explicitní definicí pomocné proměnné, která je (na rozdíl od dočasné proměnné zřízené překladačem) našemu programu dostupná.

Na doplňkové disketě najdete soubory *P5–01–02.CPP* a. *C5–01–05.CPP*, které obsahují kromě uvedených příkladů ještě variantu funkce *Prohod* pro reálné proměnné, funkci *Koreny2*, která vypočte kořeny kvadratické rovnice a vrací logickou hodnotu toho, zda jsou kořeny reálné, a další příklady.

5.3 Přetěžování funkcí

Jednou z novinek, o které rozšířilo C++ schopnosti jazyka C, je možnost **násobných definic funkcí**. Protože terminologie jazyka C++ u nás dosud není ustálena, můžete se setkat také s termíny **rozšíření definice funkce** a nebo s termínem **funkční homonyma**, který jsme používali mj. v časopisecké verzi tohoto kursu. Kromě toho jsme viděli i termíny přepsání funkce a předefinování funkce, ty jsou ale scestné, protože zde o žádné přepsání ani předefinování nejde.

O co tedy jde? Jazyk C++ nám umožňuje definovat několik funkcí se stejným jménem. Tyto definice se však musí lišit v počtu nebo v typech formálních parametrů.

V souboru *C5–01–05.CPP* najdete dvě varianty procedury *Prohod*, z nichž jedna má oba parametry typu **int**, druhá má oba parametry typu **double**. V Pascalu jsme museli použít dvě různá jména.

5.4 Implicitní hodnoty parametrů

Další z příjemných vlastností jazyka C++ je možnost zadat implicitní hodnoty posledních několika parametrů. (Pascal nic podobného nenabízí.) Na deklarace a volání podprogramů se zadanými implicitními hodnotami některých (nebo i všech) parametrů jsou kladeny následující podmínky:

- ✧ Za deklarací parametrů s přiřazenými implicitními hodnotami již nesmí následovat deklarace parametru, který implicitní počáteční hodnotu přiřazenu nemá. Proto někdy hovoříme o zadání implicitní hodnoty posledních parametrů.
- ✧ Pokud při volání daného podprogramu neuvedeme některý ze skutečných parametrů, pro jehož formální protějšek jsme definovali implicitní hodnotu, dosadí překladač za tento skutečný parametr jeho implicitní hodnotu. Pro volání podprogramů platí stejně jako pro deklarace zásada, že parametry ze seznamu smíme vynechávat pouze odzadu. To znamená, že vynechám-li při volání funkce některý parametr, musím vynechat i všechny parametry, které za ním v seznamu formálních parametrů následují. Při definici funkce musíme proto navrhnout takové pořadí formálních parametrů, aby parametry, jejichž vynechávání předpokládáme nejčastější, byly deklarovány jako poslední.
- ✧ Při návrhu podprogramů s implicitními hodnotami některých parametrů si musíme dát pozor, aby vynecháním několika parametrů nemohlo dojít ke kolizi s homonymy dané

funkce. Musíme zkrátka zabezpečit, aby překladač mohl jednoznačně rozhodnout, kterou z přetížených funkcí má v danou chvíli použít.

✧ Implicitní hodnoty parametrů definujeme pouze v první deklaraci funkce v daném oboru viditelnosti. (Je jedno, zda jde o prototyp nebo o definiční deklaraci.)

S implicitními hodnotami parametrů se setkáme v příkladu C5 – 3 v definici funkce *Kořen2*, při jejímž volání můžeme vynechat nulové parametry. Zkuste si ji nejprve definovat sami podle následujícího zadání:

Chceme definovat logickou funkci (tj. funkci vracející logickou hodnotu) *Koren2*, která bude ve svých prvních dvou parametrech *x* a *y* vracet hodnoty kořenů kvadratické rovnice

$$ax^2 + bx + c = 0.$$

Jejími dalšími parametry budou koeficienty *a*, *b*, *c* této rovnice. Funkce vrátí hodnotu *ANO* v případě, že oba kořeny jsou reálné, a hodnotu *NE* v případě, že jsou komplexní. Jsou-li kořeny reálné, vrátí tato funkce v parametru *x* větší a v parametru *y* menší z nich, jsou-li komplexní, vrátí v *x* jejich reálnou a v *y* imaginární část. Funkci definujeme tak, abychom pokud možno nemuseli při volání zapisovat nulové koeficienty.

```
/*   Příklad C5 – 3   */
int /*****/  Koren2  /*****/
( double& x, double& y, double a, double b=0, double c=0 )
{
  double D = b*b - 4.0*a*c;
  a = 2 * a;
  if( D >= 0 )
  {
    D = sqrt( D );
    x = (-b + D) / a;
    y = (-b - D) / a;
    if( a < 0)   Prohod( x, y );
    return( 1 );
  }
  else
  {
    D = sqrt( -D );
    x = -b / a;
    y =  D / a;
    return( 0 );
  }
}
```

Potřebujeme-li vyřešit rovnici

$$2x^2 + 5x = 0,$$

zavoláme tuto funkci příkazem

```
i = Koren2(x, y, 2, 5);
```

který znamená totéž jako

```
i = Koren2(x, y, 2, 5, 0);
```

5.5 Konstantní a registrové parametry

V předchozí kapitole jsme hovořili o možnosti deklarovat v C++ lokální automatické proměnné se specifikátorem **register**. Tato možnost se vztahuje i na parametry procedur, přičemž u parametrů předávaných hodnotou se do registru ukládá hodnota skutečného parametru, kdežto u parametrů předávaných odkazem se tam ukládá adresa skutečného parametru.

Kromě specifikátoru paměťové třídy **register** lze v C++ použít v deklaracích parametrů i specifikátor **const**. Deklarace konstatních parametrů předávaných odkazem zaručuje volajícímu programu, že se předávané skutečné parametry nezmění, a jeho užitečnost oceníte zejména tehdy, rozhodnete-li se v zájmu úspory času a paměti využít odkazem pro paměťově náročné proměnné a konstanty. Bez této záruky by se totiž mohl překladač zdráhat předat odkazem konstantu – viz procedura*Zaramuj* v programu *C5–01–05.CPP*.

Na rozdíl od C++ Pascal neumožňuje předávat konstantní parametry odkazem. Budeme-li chtít tuto proceduru naprogramovat v Pascalu, máme dvě možnosti: buď se smíříme s nižší efektivitou programu a budeme parametr předávat hodnotou, nebo budeme sice předávat parametr odkazem, ale nebudeme používat textové konstanty, resp. budeme muset hodnoty těchto konstant nejprve přiřadit dočasným proměnným.

V definici funkce *Zarámuj* na doplňkové disketě (příklady *P5–01–02.PAS* a *C5–01–05.CPP*) vás možná překvapí neznámá funkce *length* (Pascal) resp. *strlen* (C++). Je to knihovní funkce, jež vrací délku textového řetězce, který jí předáváme jako parametr. Její deklarace v Pascalu je

```
function Length( s: String ) : integer
```

a v C++

```
size_t strlen( const char* s );
```

Typ *size_t* je kompatibilní s typem **int,** a k tomu, abyste tuto funkci mohli používat, musíte do programu vložit (**#include**) soubor *string.h*.

5.6 Proměnný počet parametrů (výpustka)

V profesionálních programech potřebujeme často definovat podprogram, u nějž předem neznáme přesný počet parametrů nebo jejich přesný typ (případně obojí). Typickým příkladem takových podprogramů jsou např. pascalské procedury *read* a *write*. Pascal sice v definici jazyka tyto procedury zavádí, ale programátorovi neumožňuje nadefinovat jejich ekvivalenty. Jedním z udávaných důvodů je snížená bezpečnost výsledného kódu. Pokud

tedy chce programátor v Pascalu definovat podprogram, u nějž předem nezná počet jeho parametrů, musí si pomoci všelijakými figly a triky, které nakonec bezpečnost výsledného kódu sníží ještě mnohem více.

C++ zavádí pro proměnný počet parametrů symbol **...** (výpustka) – tedy tři po sobě jdoucí tečky bez jakéhokoliv vloženého bílého znaku. Výpustkou musí seznam parametrů končit. Pokud má tedy funkce také parametry, jejichž typ a počet předem známe, musíme je v seznamu formálních parametrů deklarovat před výpustkou.

Abychom si v dalším textu usnadnili vyjadřování, budeme parametrům, které jsou v deklaraci souhrnně reprezentovány výpustkou, říkat **výpustkové parametry**.

Hodnoty výpustkových parametrů si volaný program přebírá standardním mechanismem, jehož nástroje jsou popsány v hlavičkovém souboru *stdarg.h*:

1. Nejdříve musíme definovat proměnnou typu *va_list* (jednoúčelový typ pouze pro přebírání hodnot výpustkových parametrů), které budeme říkat **výpustkový ukazatel**, protože bude ukazovat na výpustkový parametr, který jsme ještě nepřevzali.
2. Výpustkový ukazatel musíme inicializovat procedurou[8] *va_start*, která má dva parametry, za něž dosadíme výpustkovou proměnnou.

Poznámka:
V manuálech se dočteme, že druhým parametrem funkce va_start *má být identifikátor posledního pevného parametru před výpustkou. To je však pouze rudiment udržovaný ve snaze po kompatibilitě s ANSI C. C++ totiž, na rozdíl od ANSI C, povoluje výpustku jako jediný parametr funkce. Procedura* va_start *tedy z důvodů kompatibility druhý argument vyžaduje, ale ignoruje jej. Proto považujeme za nejjednodušší předat v obou parametrech výpustkový ukazatel.*

3. Pomocí funkce[9] *va_arg* postupně přiřadíme hodnoty výpustkových parametrů proměnným, jejichž typ bude kompatibilní s typem očekávaného skutečného parametru. Funkce *va_arg* má dva parametry: za první dosadíme výpustkový ukazatel a jako druhý parametr uvedeme očekávaný typ zpracovávaného výpustkového parametru.
4. Z důvodů kompatibility (Borland C++ to však nevyžaduje) je vhodné zavolat na závěr zpracování výpustkových parametrů proceduru[10] *va_end*.

Z uvedeného je zřejmé, že podprogram s výpustkovými parametry musí umět určit jak počet výpustkových parametrů, tak i jejich typ.

Z důvodů, které si vysvětlíme později, však někdy nesmí být správný očekávaný typ výpustkového parametru shodný s typem příslušného skutečného parametru. Jednou z těchto výjimek je typ **char**, který musíme vždy přebírat jako **int** – to znamená, že typ **int**

[8] Ve skutečnosti jde o makro, ale to zatím není důležité.
[9] Opět jde o makro, ale to nevadí, klidně se můžeme na *va_arg* dívat jako na podivnou funkci, jejímž parametrem může být i jméno typu.
[10] Také *va_end* je ve skutečnosti makro.

musí být uveden jako druhý parametr funkce *va_arg*. S ohledem na kompatibilitu vzhledem k přiřazení však můžeme přebíranou hodnotu přiřadit zpět proměnné typu **char**.

Definice a použití podprogramu s proměnným počtem parametrů je v doprovodném programu předvedena ve dvou verzích funkce **Průměr**, která do standardního výstupu vytiskne své parametry a jejich aritmetický průměr. Prvá verze má jako první parametr číslo, které udává počet průměrovaných hodnot, druhá verze dělá průměr z nenulových hodnot a konec seznamu parametrů pozná podle nulové hodnoty parametru. Zde si ukážeme pouze první z nich, druhou najdete spolu s ostatními příklady z této kapitoly na doplňkové disketě v souboru *C5–01–05.CPP*.

```
/*   Příklad C5 – 4   */
double /*****/  PrumerA  /*****/
( int N, ... )                  //Vypočte průměr z N reálných čísel
{
  va_list vu = va_start( vu, vu );    //vu = Výpustkový ukazatel
  int     i    = N;
  double  Suma = 0;
  do
  {
    Suma = Suma + va_arg( vu, double );
  } while( (i = i-1) > 0 );
  va_end( vu );
  return( Suma / N );
}
```

5.7 Vložené funkce

Potřebujeme-li definovat nějakou hodně jednoduchou, ale také hodně používanou proceduru nebo funkci, stojíme často před rozhodnutím, zda tím, že ji definujeme jako podprogram, nesnížíme zbytečně efektivnost programu. S každým voláním podprogramu je totiž spojena jistá režie postihující přípravu a předávání parametrů, vlastní volání podprogramu a vstupní a výstupní posloupnost instrukcí ve vlastním podprogramu (tzv. *standard stack frame*, standardní ošetření zásobníku).

U opravdu jednoduchých akcí se někdy může stát, že tato režie zabere paměťový prostor, který je srovnatelný s paměťovým prostorem a časem vlastní akce a může být dokonce i větší. C++ umožňuje definovat takovéto podprogramy jako **vložené**[11], což znamená, že definice sice bude vypadat jako definice funkce, ale místo volání překladač tělo této funkce na místo, kde je volána, pouze opíše (vloží).

Pokud vám to připomíná definice makroinstrukcí, máte pravdu. Přesto je tu jeden podstatný rozdíl: při volání vložené funkce (na rozdíl od použití makra) překladač kontroluje typy parametrů. Kromě toho můžeme – např. při ladění – překladač požádat, aby vložené

[11] V časopisecké verzi kursu jsme používali termín *fiktivní funkce*. Označení *vložená funkce* je v podstatě doslovným překladem původního termínu *inline function*, funkce (vložená) do řádky programu.

funkce překládal stejně jako funkce „normální“ a tím nám je umožnil krokovat. Po odladění celého programu pak změníme nastavení voleb překladače a přeložíme tyto funkce opravdu jako vložené.

Některé funkce nelze definovat jako vložené; např. funkce, v jejichž těle použijeme cyklus, příkaz **switch** atd., překladač odmítne přeložit jako vložené. Není to ovšem chyba, překladač pouze vypíše upozornění.

Vložené funkce definujeme pomocí specifikátoru **inline**. **Pokud chceme vloženou funkci používat v několika modulech, musíme ji definovat v každém z nich**. Nejlepším řešením tedy je umístit její definici do hlavičkového souboru a tento soubor pak vložit (**#include**) do zdrojových textů modulů, kde funkci potřebujeme.

Jako příklad poměrně typické procedury, která bývá definována jako vložená, může posloužit právě „celočíselná“ varianta procedury *Prohoď* z následujícího příkladu. Zdrojový text je opět součástí souboru *C5–01–05.CPP*.

```
/*   Příklad C5 – 5   */
inline void /*****/  Prohod  /*****/
( int& i1, register int& i2 )
//Vložené funkce musíme definovat před prvním použitím.
//Samotná deklarace nestačí.
{
  register ip = i1;   i1 = i2;   i2 = ip;
}
```

6. Ladění programů s daty

Nyní už víme alespoň nejzákladnější věci o práci s daty. Programy s daty je ovšem také třeba ladit a borlandské prostředí nám k tomu poskytuje řadu nástrojů.

Při ladění programů s daty potřebujeme často průběžně sledovat hodnoty některých proměnných. K tomu můžeme v IDE použít **sledovací okno** (*Window* | *Watches*), do kterého vložíme identifikátory proměnných, jejichž hodnoty chceme sledovat.

Vkládat identifikátory sledovaných proměnných můžeme dvěma způsoby: buď v libovolném okně stiskem klávesy CTRL-F7, nebo ve sledovacím okně stiskem klávesy INS. V obou případech se vynoří dialogové okno, v jehož vstupním poli zadáme identifikátor sledované proměnné. (Můžeme zadat i složitější výraz.)

Pokud chceme některý ze zadaných výrazů opravit či vypustit, otevřeme sledovací okno a najedeme na daný výraz řádkovým kurzorem. Chceme-li jej zrušit, stiskneme klávesu DEL, chceme-li jej pouze opravit, stiskneme ENTER.

Pokud nepotřebujeme hodnotu nějakého výrazu sledovat průběžně, ale stačí nám pouze zjistit jeho současnou hodnotu, nebo pokud naopak potřebujeme hodnotu nějaké proměnné (v C++ i některých konstant) změnit, otevřeme stiskem CTRL-F4 dialogové okno *Evaluate / Modify* (vyhodnoť / změň), které má tři pole a čtyři tlačítka.

Do vstupního pole *Expression* (výraz) zadáváme výraz, který chceme vyhodnotit. Vpravo vedle pole je šipka dolů naznačující, že můžeme využít minulých zadání, jak to již známe např. z dialogového okna pro otevření souboru.

Ve výstupním poli *Result* (výsledek) nám počítač oznámí vypočtenou hodnotu zadaného výrazu.

Vstupního pole *New value* (nová hodnota) můžeme využít, pokud jsme v poli *Expression* zadali identifikátor proměnné (nebo obecně l-hodnotu), jejíž hodnotu bychom chtěli změnit. Do pole *New value* totiž zadáváme novou požadovanou hodnotu.

Tlačítka slouží pouze pro ovládání myší, protože při ovládání z klávesnice stačí po zadání hodnoty do příslušného vstupního pole pouze stisknout klávesu ENTER a nápovědu resp. zrušení okna dosáhneme standardními prostředky, tj. klávesami F1 resp. ESC.

Pokud počítač odmítá některou hodnotu změnit, bývá to většinou proto, že daná hodnota není obsahem nějakého místa v paměti, ale že ji překladač v rámci optimalizací používá v programu jako literál.

Zadávání vyhodnocovaných výrazů je v obou oknech velmi podobné. IDE se nám snaží vyjít vstříc tím, že nám nabídne text, na němž je v danou chvíli kurzor – identifikátory tak můžeme z editačního okna zadávat jednoduše tak, že na ně najedeme kurzorem, stiskneme CTRL-F7 a nabídnutý text již pouze potvrdíme.

Zadávat můžete nejen identifikátory proměnných a konstant, ale jakékoliv výrazy, jejichž hodnota nás zajímá. Pokud chceme zjistit nebo sledovat hodnotu výrazu, který je někde v programu, najedeme na něj kurzorem a po otevření dialogového okna postupnými stisky šipky vpravo kopírujeme do vstupního pole další znaky z editovaného textu.

Vyhodnocované výrazy nesmějí obsahovat volání funkcí a nesmíme v nich používat konstant a proměnných, jejichž oblast platnosti nezahrnuje právě krokovaný příkaz. Jakmile při krokování oblast platnosti daného identifikátoru opustíte, IDE u něj místo hodnoty napíše, že daný výraz není definován. Z toho tedy automaticky vyplývá, že pokud zrovna nekrokujete, budou vyhodnoceny pouze výrazy, v nichž používáte jen literály.

Abyste si ověřili možnosti sledování, zkuste krokovat např. program *C4–01* resp. *P4–01* z doplňkové diskety a přitom průběžně sledovat hodnoty proměnných *Externí*, *Globální* a *Zakrytá*. V C++ můžete průběžně zadáním výrazu

```
::Prekryta
```

sledovat hodnotu globální proměnné i v případě, že je zakryta stejnojmennou lokální proměnnou.

Při zadávání výrazů ve sledovacím resp. vyhodnocovacím a modifikačním okně můžeme ovlivňovat formát, v němž počítač výsledné hodnoty zobrazí. Implicitní formát, v němž počítač hodnotu vyhodnocovaného výrazu zobrazí, je dán typem vyhodnocovaného výrazu. Požadujete-li zobrazení v jiném formátu, napíšete za vyhodnocovaný výraz čárku následovanou popisem požadovaného formátu.
Formátovací příkazy, které nám mohou být již nyní užitečné, shrnuje následující tabulka. S dalším se seznámíme později.

Kromě přímého uvedení formátu za výrazem máme v C++ ještě možnost globálně ovlivnit číselnou soustavu, v níž budou zobrazovány hodnoty celých čísel. V okně *Options* | *Debugger* můžete nastavit přepínač na hodnotu:

- ✧ *Show decimal* pro zobrazení hodnot v desítkové soustavě,
- ✧ *Show hex* pro zobrazení hodnot v šestnáctkové soustavě,
- ✧ *Show both* pro zobrazení každé hodnoty v obou soustavách.

Symbol	Význam
c	Hodnota se zobrazí jako znak, přičemž hodnoty 0 až 31 se v Pascalu zobrazí použitím syntaxe #xx a v C++ pomocí řídicích posloupností (\n, \t, atd.)
d	Hodnota se zobrazí jako číslo v desítkové soustavě
x	Hodnota se zobrazí jako číslo v šestnáctkové soustavě
h	Totéž jako x
$	Pouze pro Pascal – tam totéž jako x
f#	Racionální číslo se zobrazí s přesností na # platných cifer, kde # musí být z intervalu <0;18>. Implicitní hodnota je v Pascalu 7, v C++ 11. Po zadání přesnosti se s touto přesností zobrazují i všechny následující racionální hodnoty až do nového zadání přesnosti

Tab. 6.1 Základní formáty hodnot zobrazovaných při ladění.

7. Pole

S výjimkou textových řetězců řadíme všechny doposud probrané datové typy mezi tzv. **skalární typy**, které reprezentují jednu dále nedělitelnou hodnotu. Proti nim stojí tzv. **strukturované datové typy**, které jsou obecně tvořeny více jednotlivými hodnotami.

Základním strukturovaným datovým typem jsou **jednorozměrná pole** neboli **vektory**, které jsou tvořeny konečnou množinou hodnot stejného typu (tímto typem však může být klidně opět vektor – viz příklady deklarací v jednotlivých jazycích). Jednotlivé hodnoty, které nazýváme **prvky vektoru,** jsou uspořádány a jejich pořadí označujeme **indexem**.

S vektorovými objekty můžeme pracovat buď jako s celky, nebo můžeme pracovat s jejich jednotlivými prvky. Na konkrétní prvek vektoru se odvoláme tak, že napíšeme jméno vektorové proměnné (v C++ to může být i jméno vektorové konstanty) a za ním v hranatých závorkách uvedeme index požadovaného prvku.

Protože vektorové objekty jsou většinou velmi rozměrné, bývá zvykem předávat je odkazem – v C++ je hodnotou ani předávat nelze. Z toho plynou dvě poučení:

✧ Při definici podprogramů s vektorovými parametry nesmíme v Pascalu zapomenout uvést klíčové slovo **var** – leda bychom opravdu trvali na tom, že parametr budeme předávat hodnotou (tedy že se v podprogramu má vytvořit lokální kopie skutečného parametru).

✧ Chceme-li předávat parametr hodnotou (tj. chceme-li, aby program neovlivnil hodnotu předávaného parametru), máme v C++ na vybranou dvě možnosti: buď chceme problém řešit ve volající proceduře, kde pak musíme zřídit dočasnou proměnnou, do níž obsah předávaného vektoru zkopírujeme, nebo jej hodláme řešit v proceduře volané, která si zřídí lokální vektorovou proměnnou, do níž si zkopíruje obsah odkazem předaného vektoru a s níž si pak již může dělat co chce.

Textové řetězce jsou vlastně vektory, jejichž prvky jsou jednotlivé znaky daného textu. Oproti vektorům mají však ještě některé vlastní dodatečné rysy, a proto si o nich povíme ve zvláštní podkapitole. Prozatím si pamatujte, že s textovým řetězcem můžete pracovat jako s vektorem znaků, jehož jednotlivé prvky mají v Pascalu indexy 1 až *n* a v C++ 0 až *n*-1, kde *n* je délka řetězce získaná voláním funkce *length* (Pascal) resp. *strlen* (C++).

Vektorový datový typ definujeme v Pascalu (v sekci **type**) takto: za identifikátorem nově definovaného datového typu napíšeme rovnítko, za ně pak klíčové slovo **array**, otevírací hranatou závorku, nejnižší index prvku, symbol **..** (dvě tečky), nejvyšší index prvku a zavírací hranatou závorku. Celou definici ukončíme středníkem. Například

```
type pole = array [5 .. 77] of integer;
```

je deklarace typu *pole*, což je vektor se 73 prvky typu *integer*, indexovanými od 5 do 77.

Definujeme-li vektorovou proměnnou (inicializovanou v sekci konstant, neinicializovanou v sekci proměnných) resp. skupinu proměnných, můžeme buď uvést v definici jmé-

no typu nebo zde tento typ přímo definovat. **Takto definovaný typ však není kompatibilní** s žádným jiným datovým typem, a to ani tehdy, mají-li oba typy shodné definice.

U inicializovaných proměnných, tj. u proměnných deklarovaných v sekci konstant, zadáváme počáteční hodnotu vektoru tak, že za jménem nebo definicí typu pokračujeme rovnítkem, otevírací kulatou závorkou, seznamem hodnot jednotlivých prvků oddělených čárkami, a skončíme uzavírací kulatou závorkou.

Jednou z velkých nectností Pascalu je nekompatibilita různých vektorových typů, kvůli které není ve starších verzích jazyka možno napsat podprogram, jehož parametrem by mohl být vektor předem neznámé délky. Autoři Turbo Pascalu obešli toto omezení zavedením beztypových parametrů předávaných odkazem. Veškerá typová kontrola, kvůli které se Pascal tak silně bije v prsa, přichází sice v tu chvíli vniveč, ale zato můžete začít psát i prakticky použitelné programy.

Následující postup se sice liší od toho, který najdete v manuálu, ale domníváme se, že je čistší, bezpečnější a praktičtější:

1. Jestliže nebudeme chtít kontrolovat počet prvků v poli, složeném z prvků typu *ttt*, definujeme typ

   ```
   V_ttt = array[ 0 ..N_ttt ] of ttt;
   ```

   Dolní index je nula, protože vektory s tímto dolním indexem se zpracovávají nejefektivněji, a horní index *N_ttt* by měl být natolik velké číslo, aby všechny vektory, které přicházejí v úvahu, byly kratší. Přijatelnou hodnotu získáme ze vzorce

   ```
   L_ttt = 64000 div sizeof( ttt )
   ```

Podle potřeby si tedy takto definujeme typy **V_integer**, **V_char**, **V_real** a další vektorové typy, které budeme v programu potřebovat. Tyto typy bychom sice měli správně definovat jako lokální v podprogramu, ale protože je pravděpodobné, že je budeme chtít použít v několika podprogramech zároveň, a protože jsou programátoři od přírody líní, doporučujeme vám je standardně definovat jako globální.

2. V každém podprogramu budeme ty vektorové parametry, u nichž nechceme kontrolovat jejich rozměr, předávat odkazem a neuvedeme u nich žádný typ.
3. Pro každý beztypový parametr zřídíme lokální proměnnou. Její identifikátor by se měl od identifikátoru odpovídajícího parametru co nejméně lišit – např. pouze úvodním podtržítkem, o něž identifikátor formálního parametru rozšíříme. Tato lokální proměnná bude typu *V-xxx*, kde *xxx* je typ odpovídajícího parametru, a v její deklaraci napíšeme před závěrečný středník klíčové slovo **absolute** následované identifikátorem odpovídajícího parametru – tedy např.:

   ```
   Mereni : V_real absolute _Mereni;
   ```

4. V celém podprogramu používáme místo beztypových parametrů odpovídající vektorové lokální proměnné.

Vše by měly osvětlit příklady v souboru *P7–1.PAS* na doplňkové disketě. Zde si ukážeme funkci *Prumer*, která vypočte průměrnou hodnotu z prvních *n* prvků pole.

```
(*   Příklad P7 – 1   *)
type
  V_real = array[ 0 .. 1000 ] of real;
  {Pomocný typ pro předávání beztypových parametrů}

function (*****)   Prumer   (*****)
( n : integer;  var _d ) : real;
var d : V_real absolute _d;
  i : integer;        {Tyto dvě proměnné nemohou být inicializované,}
  s : real;           {protože je potřebujeme inicializovat při každém}
                      {volání funkce znovu}
begin
  s := d[ 0 ];
  i := 1;
  while( i < n )do
  begin
       s := s + d[ i ];
       i := i + 1;
  end;
  Prumer := s / n;
end;

(**************   Hlavní program   **************)
const d1:array[ 1 .. 4 ] of real = (1.7, 3.5, 2.38, 4.52);

begin
  d2 := Prumer( 2, d1 );
end.
```

V Turbo Pascalu 7.0 a v Delphi můžeme použít tzv. otevřená pole. U parametru tohoto typu specifikujeme typ složek, nikoli však rozsah indexů, např. takto:

```
function prumer(var d: array of real): real;
```

V těle procedury se takovéto pole chová, jako kdybychom ho deklarovali s indexy od 0 do *N*-1, kde *N* je počet složek. Nejvyšší hodnotu indexu zjistíme pomocí funkce *High*. S využitím otevřených polí bychom tedy mohli funkci *prumer* přepsat takto:

```
(*   Příklad P7 – 2   *)
function (*****)   Prumer¶   (*****)
( var d: array of real ) : real;
var
  s: real;
  i: integer;
begin
  s :=  0;
  i := 0;
  while ( i <= High(d) ) do
```

```
      begin
      s := s + d[ i ];
      i := i + 1;
  end;
  Prumer := s / (High(d)+1);
end;
```

V C++ se samostatná definice vektorových datových typů většinou nepoužívá (je však možná a budu o ní hovořit za chvíli). Místo toho se „vektorovost“ definovaných proměnných a konstant vyjadřuje přímo v jejich definicích. Definujeme-li vektorovou proměnnou nebo konstantu, deklarujeme ji „pod typem“ prvků a za její identifikátor uvedeme v hranatých závorkách počet jejích prvků.[12] Například

```
int pole[100];
```

je deklarace vektoru *pole*, složeného ze 100 prvků typu **int**, indexovaných od 0 do 99. (Pozor! Prvek s indexem 100 v tomto vektoru neexistuje!)

Ve srovnání s Pascalem narazíme v C++ na několik odlišností:

1. V C++ lze deklarovat i vektorové konstanty.
2. V C++ jsou všechny vektorové typy se shodnými typy prvků navzájem kompatibilní (to je mimo jiné i důvod, proč se nepoužívají samostatné definice vektorových typů – nejsou třeba). Hlavní výhodou této kompatibility je, že nás při definici podprogramů nemusí zajímat počet prvků jejich vektorových parametrů a překladač je proto schopen kontrolovat kompatibilitu typů formálních a skutečných parametrů i u podprogramů určených pro práci s vektory různých délek.
3. V C++ si nemůžeme vybrat nejnižší a nejvyšší hodnotu indexu – prvky vektoru indexujeme **vždy od nuly**. Poslední prvek vektoru má proto vždy index *n*-1, kde *n* je počet prvků vektoru. O tom, jak lze toto omezení jednoduše obejít, si povíme v kapitole o ukazatelích.

Počáteční hodnotu vektoru zadáváme u vektorových konstant a inicializovaných proměnných tak, že za vlastní deklarací pokračujeme rovnítkem, otevírací složenou závorkou, seznamem hodnot jednotlivých prvků oddělených čárkami a skončíme zavírací složenou závorkou.

Vektory znaků můžeme navíc inicializovat i řetězcem, avšak délka řetězce musí být o jeden znak kratší, než je počet prvků inicializovaného vektoru. Posledním (neviditelným a do délky řetězce nezapočítávaným) znakem řetězce je totiž vždy znak s kódem 0, který se do inicializovaného vektoru musí také vejít. Pokud se tam nevejde, ohlásí překladač chybu.

[12] Indexové závorky mohou být i prázdné, pak se ovšem musí jednat o konstantu či inicializovanou proměnnou, u níž může překladač odvodit počet prvků z počtu inicializátorů.

Počet inicializátorů, tedy prvků inicializačního vektoru, **nesmí být větší než počet prvků vektoru inicializovaného**. Může však být menší – pak se prvky, na něž „nezbylo“, inicializují nulami. Pokud není v deklaraci vektorového objektu ani v definici jeho typu uveden počet prvků, předpokládá se, že inicializovaný vektor má stejný počet prvků jako inicializační hodnota.

Už jsme si řekli, že v C++ se všechny vektory předávají odkazem. Chceme-li zaručit, že podprogram nezmění hodnoty svého vektorového parametru, deklarujeme tento parametr jako konstantní.

I když to není příliš běžné, můžeme v C++ definovat pomocí klíčového slova **typedef** i vektorové typy, a to tak, že za jméno typu prvků vektoru napíšeme identifikátor nově definovaného datového typu, otevírací hranatou závorku, počet prvků vektoru a zavírací hranatou závorku. Celou definici ukončíme středníkem. Např. takto:

```
typedef int pole[10];
```

Proměnné vektorových typů definujeme formálně stejně jako proměnné skalárních typů: napíšeme jméno typu a za něj seznam proměnných tohoto typu s případnými inicializačními hodnotami. (O inicializaci budeme za chvíli hovořit podrobněji.)

Připomeňme si ale, že pomocí klíčového slova **typedef** ve skutečnosti nedefinujeme nové datové typy, ale pouze zavádíme nová mnemotechnická označení. Podívejme se opět na několik příkladů (zdrojové texty najdete spolu s dalšími příklady na doplňkové disketě v souboru *C7–01.CPP*0):

```
/*   Příklad C7 – 1   */
typedef int V7I[ 7 ];
      //Typ vektoru sedmi celých čísel indexovaných od 0 do 6

char vcv[] = { '1', '2', '3', '4', '5' };  //Vektor pěti znaků
      //Velikost vektoru se určí podle počtu inicializátorů

char vxcv[6] = "12345";                    //Vektor šesti znaků
  /* Vektor lze inicializovat řetězcem, který však musí
     být o jeden znak kratší než je deklarovaná délka
     vektoru, protože součástí řetězce je i závěrečný
     znak s kódem 0 */

static double /*****/  Prumer  /*****/
( int n,  double d[] )
//Předpokládáme n > 0 - spočte průměr prvých n prvků vektoru d
{
  int    i = n;
  double s = d[ 0 ];
  while( (i = i-1) > 0 )
      s = s + d[ i ];
  return( s / n );
}
/************** Hlavní program  **************/
void /*****/  main  /*****/ ()
{
  double d1[] = { 1.7,  3.5,  2.38,  4.52 };
  double d2[] = { Prumer( 2, d1 ),    //Inicializační hodnota
                  Prumer( 3, d1 ),    //automatických proměnných
```

```
                Prumer( 4, d1 ) };  //může vznikat výpočtem
                                    //až při běhu programu
  Prumer( 3, d2 );
}
```

Abyste si použití vektorů v programech náležitě procvičili, zkuste si naprogramovat tři příklady: za prvé jednoduchý filtr pro převod z kódu Kamenických do kódu Latin 2, za druhé program, který převádí římská čísla na arabská a za třetí program, který převádí arabská čísla na římská. Řešení posledních dvou úkolů – tedy převodník římských čísel na čísla arabská a naopak – najdete na doplňkové disketě v souborech *P7–03.PAS* a *C7–02.CPP*.

8. Operátory

V této kapitole si probereme většinu zbylých operátorů, se kterými se můžeme v obou jazycích setkat. Některé ovšem stále ještě vynecháme; o nich si povíme, až budeme vědět více o objektových datových typech.

Nejprve si však musíme vysvětlit pojem **pořadový typ** (můžete se také setkat s označením **ordinální typ**). Z dosud probraných typů řadíme mezi pořadové typy celá čísla, znaky a v Pascalu také logické hodnoty. Časem k nim ještě přidáme datové typy definované výčtem možných hodnot. Mezi pořadové typy tedy nepatří reálná čísla ani žádné strukturované typy, jako vektory, řetězce apod.

Než začnete číst následující výklad, musíme vás upozornit, že v něm při výkladu bitových operací budeme hojně používat zápis čísel v šestnáctkové soustavě, protože v ní se dá vlastní průběh operací daleko snadněji pochopit. Pokud šestnáctková soustava není vaší silnou stránkou (profesionální programátor by ji však měl ovládat zcela suverénně), vezměte si na pomoc tabulku 2.1 ze 2. kapitoly.

Přejděme ale k vlastnímu tématu kapitoly. Všechny operátory obou jazyků, včetně těch, které si budeme moci podrobněji vysvětlit až později, jsme seřadili do tabulky 8.1. Podívejme se nejprve, co můžete v jednotlivých sloupcích najít:

První sloupec obsahuje **symbol**, kterým daný operátor znázorňujeme v programu. Má dva podsloupce, levý obsahuje symbol používaný v Pascalu a pravý obsahuje symbol používaný v C++. Pokud není pro daný operátor v některém podsloupci uveden žádný symbol, není tento operátor v příslušném jazyce definován.

Druhý sloupec obsahuje tzv. **aritu** operátoru, tj. počet zpracovávaných operandů. V tabulce jsou operátory unární, které mají pouze jediný operand (např. operátor negace), operátory binární, které mají dva operandy (např. operátor násobení) a operátory ternární se třemi operandy (podmíněný výraz v jazyku C++).

Třetí sloupec má opět dva podsloupce, které tentokrát obsahují **prioritu** operátorů v daném programovacím jazyce. Operátory s nižším číslem mají vyšší prioritu a vyhodnocují se proto ve výrazech před operátory s nižší prioritou (např. násobení se vyhodnocuje před sčítáním). Protože jazyk C++ má mnohem jemnější dělení priorit, jsou v zájmu přehlednosti operátory v tabulce seřazeny podle klesající priority v C++.

U operátorů, které nejsou v daném jazyce definovány, obsahuje odpovídající sloupec v příslušném řádku znak – (pomlčku). Obsahuje-li pascalský podsloupec v některém řádku místo čísla znak ∅, znamená to, že daný operátor má sice v Pascalu svůj ekvivalent, avšak není to operátor, takže jej nelze použít ve výrazech, a proto nemá ani smysl hovořit o jeho prioritě.

Čtvrtý sloupec obsahuje znázornění **asociativity**, nebo chcete-li **pořadí vyhodnocování** v případě, že ve výrazu použijeme vedle sebe operátory se stejnou prioritou (všimněte si, že operátory se stejnou prioritou mají vždy i stejnou asociativitu).
Poslední pátý sloupec popisuje slovy funkci daného operátoru.

Před dalším výkladem bychom vám měli ještě vysvětlit pojem **fixace operátorů**, kterým popisujeme umístění symbolu operátoru vzhledem k jeho operandům. Unární operátory dělíme na **prefixové**, které se píší před operand (např. *!a*) a **postfixové**, které se píší za operand (např. *a*++). Binární operátory se pak v obou jazycích používají téměř vždy jako **infixové**, což znamená, že operátor se zapisuje mezi operandy (např. *A*+*b*). Jedinou výjimkou z tohoto pravidla jsou závorky (operátor volání funkce).

Pro přehlednost jsme operátory v tabulce rozčlenili do skupin podle priority v jazyce C++. To znamená, že všechny operátory v jedné skupině mají stejnou prioritu, která je vyšší než priorita operátorů z následující skupiny a nižší než priorita operátorů ze skupiny předchozí.

Symbol		**Arita**	**Priorita**		**Asociativita**	**Význam operátoru**
Pas	**C++**		**Pas**	**C++**		
( )	( )	?	0	1	→	Volání funkce
[ ]	[ ]	2	0	1	→	Selektor prvku pole (op. indexování)
.	**.**	2	0	1	→	Přímý selektor
^**.**	->	2	–	1	→	Nepřímý selektor
∅	::	1, 2	–	1	→	Rozlišovací a přístupový operátor
not	!	1	1	2	←	Logická negace
not	~	1	1	2	←	Bitová negace
+	+	1	1	2	←	Unární plus (např. +3)
-	-	1	1	2	←	Unární minus (např. -6)
Inc	++	1	–	2	←	Preinkrement resp. postinkrement
Dec	--	1	–	2	←	Preinkrement resp. postdekrement
t( )	(*t*)	1	–	2	←	Přetypování na typ *t*
@	&	1	1	2	←	Získání adresy
∅	*	1	–	2	←	Dereferencování ukazatele
^	∅	1	0	∅	→	Dereferencování ukazatele
sizeof	**sizeof**	1	–	2	←	Velikost objektu nebo typu
new	**new**	1	–	2	←	Vytvoření dynamického objektu
dispose	**delete**	1	–	2	←	Zrušení dynamického objektu
–	**.***	2	–	3	→	Selektor ve třídě
–	->*	2	–	3	→	Nepřímý selektor ve třídě
*	*	2	2	4	→	Násobení
/	/	2	2	4	→	Dělení
div	/	2	2	4	→	Celočíselné dělení
mod	%	2	2	4	→	Dělení modulo, zbytek po dělení
+	+	2	3	5	→	Sčítání
-	-	2	3	5	→	Odečítání
shl	<<	2	2	6	→	Bitový posun vlevo
shr	>>	2	2	6	→	Bitový posun vpravo

Symbol Pas	Symbol C++	Arita	Priorita Pas	Priorita C++	Asociativita	Význam operátoru
<	<	2	4	7	→	Menší než
>	>	2	4	7	→	Větší než
<=	<=	2	4	7	→	Menší nebo rovno
>=	>=	2	4	7	→	Větší nebo rovno
=	==	2	4	8	→	Rovná se (porovnání)
<>	!=	2	4	8	→	Nerovná se
in	∅	2	4	–	→	Přítomnost prvku v množině
and	&	2	2	9	→	Bitové AND (bitový součin)
xor	^	2	3	10	→	Bitové XOR (bitová nonekvivalence)
or	|	2	3	11	→	Bitové OR (bitový logický součet)
and	&&	2	2	12	→	Logické AND (logický součin)
or	||	2	3	13	→	Logické OR (logický součet)
xor	∅	2	3	–	→	Logické XOR (nonekvivalence)
∅	?:	3	–	14	←	Podmíněný výraz
:=	=	2	–	15	←	Prosté přiřazení
∅	*=	2	–	15	←	Přiřazení součinu
∅	/=	2	–	15	←	Přiřazení podílu
∅	%=	2	–	15	←	Přiřazení modulu (zbytku)
Inc	+=	2	x	15	←	Přiřazení součtu
Dec	-=	2	x	15	←	Přiřazení rozdílu
∅	&=	2	–	15	←	Přiřazení bitového součinu
∅	^=	2	–	15	←	Přiřazení bitového součtu
∅	|=	2	–	15	←	Přiřazení bitového logického součtu
∅	<<=	2	–	15	←	Posunutí obsahu vlevo
∅	>>=	2	–	15	←	Posunutí obsahu vpravo
∅	,	2	–	16	→	Postupné vyhodnocení

Tab. 8.1 Operátory v C++ a v Turbo Pascalu

8.1 Typ výsledku

Občas je důležité znát typ výsledku výrazu.

V Pascalu je situace poměrně jednoduchá. Jsou-li oba operandy aritmetického operátoru stejného typu, je téhož typu i výsledek. Pokud ne, převede se nejprve typ „méně přesného“ operandu na typ „přesnějšího“ a teprve pak se operace provede. To znamená, když sečteme

```
a + b
```

kde *a* je typu *shortint* a *b* je typu *integer*, převede se nejprve hodnota *a* na typ *integer* (je „přesnější“, tj. má větší rozsah) a teprve pak se sečtou. Kdyby *a* bylo typu *real* a *b* typu *longint*, převedla by se nejprve hodnota *b* na typ *real* a teprve pak by se sečetla s hodnotou *a*.

V C++ se s číselnými operandy nejprve vždy provedou tzv. celočíselná rozšíření. To znamená, že se hodnoty typů **char**, **unsigned char**, **signed char**, **short**, **unsigned short**, hodnoty výčtových typů a bitových polí převedou na typ **int,** pokud může jejich hodnoty obsáhnout, jinak se převedou na**unsigned int**[13].

Potom se zjistí, zda jsou oba operandy stejného typu. Pokud ano, provede se požadovaná operace a výsledek je stejného typu jako byly operandy.

V případě, že jsou operandy různých typů, převede se hodnota „méně přesného“ typu na typ „přesnější“. Přitom za nejméně „přesný“ se považuje typ **int**, pak následují další v pořadí **unsigned**, **long**, **unsigned long**, **float**, **double** a za nejpřesnější se považuje typ **long double**.

To znamená, že pokud deklarujeme proměnnou *c* příkazem

```
char c = 'a';
```

je sice *c* typu **char**, ale +*c* je typu **int** a *c* + 3.14 je výraz typu **double**. Tyto rozdíly obvykle nehrají roli; mohou ale způsobit protivné zmatky při rozlišování funkčních homonym (přetížených funkcí).

8.2 Operátory s nejvyšší prioritou

Operátor funkčního volání a závorky

Tento operátor nemá definovanou aritu. První operand se zapisuje před závorky a specifikuje funkci, která se má zavolat. Zbylé operandy představují parametry funkce; píší se do závorek a oddělují čárkami.

Operátor funkčního volání je specifikou C++ – pokud nenapíšete za jméno funkce alespoň prázdné závorky, nejedná se o volání funkce, ale pouze o její adresu.

V obou jazycích se však závorky používají ještě k jinému účelu: k určení pořadí vyhodnocování výrazů v případech, kdy potřebujeme pořadí vyhodnocování uspořádat jinak, než jak by velely priority operátorů. (V tom případě se ovšem z hlediska syntaxe nejedná o operátor, ale o „interpunkci“, takže se na ně povídání o prioritě, asociativitě apod. nevztahuje. Přesto si zde o nich povíme několik slov.)

[13] Pozor, starší překladače používaly lehce odlišné pravidlo: hodnoty typu **unsigned short** se převáděly na **unsigned**, ostatní na **int**. Současná úprava je v souladu s normou ANSI.

V souvislosti se závorkami bychom měli upozornit na jednu specifiku C++. Protože priority posunových, relačních a bitových operátorů jsou odstupňovány trochu jinak, než odpovídá běžnému intuitivnímu cítění, lze v překladači C++ nastavit možnost varovné zprávy v případě, že se některé z těchto typů operátorů vyskytují společně ve výrazu a pořadí jejich vyhodnocování není explicitně definováno uzávorkováním.

Použijete-li ve výrazu nadbytečné množství závorek, nemůžete nikdy udělat chybu – samozřejmě pokud se nebude lišit počet levých a pravých. Proto se jejich používání nebojte a použijte je všude tam, kde vám jejich použití výraz zpřehlední.

Operátor indexování (selektor prvku pole)

S tímto operátorem jsme se již seznámili v kapitole o vektorech. Odtud také víme, že jeho prostřednictvím vybíráme z vektoru žádanou položku. Víme také, že se jedná o binární operátor. Prvním operandem je vektor, jehož položku chceme získat, a jeho identifikátor píšeme před hranaté závorky. Druhým operandem je pak index žádané položky a píšeme jej mezi hranaté závorky.

Přímý a nepřímý selektor složky záznamu (struktury)

Záznam bychom mohli chápat jako vektor, jehož prvky, kterým se u záznamu říká složky, mohou být různých typů. Tyto složky neoznačujeme indexem, ale vlastním identifikátorem. Uvedené dva operátory nám při práci se složkami záznamů poskytují podobnou službu, jakou nám poskytuje selektor prvku pole pří práci s prvky vektorů.

Podrobněji se s použitím tohoto operátoru seznámíme při výkladu použití záznamů v programech.

Rozlišovací a přístupový operátor

S operátorem **::** jsme se již setkali při výkladu o lokálních a globálních objektech. Tam jsme jej používali jako unární prefixový operátor. Hlavní použití tohoto operátoru je však jiné a seznámíme se s ním podrobněji, až si budeme vyprávět o objektově orientovaném programování.

8.3 Unární operátory

Operátory negace

V Pascalu se používá pro operátory logické a bitové negace stejný symbol. Operátory se od sebe liší pouze typem operandů (vzpomeňte si na přetěžování funkcí).

Pokud je operand typu **boolean** nebo některého z typů s tímto typem kompatibilních (tj. přejmenovaný **boolean**), funguje **not** jako operátor logické negace.

Pokud je operand některého z celočíselných typů (prozatím známe pouze typ **int**), provádí operátor **not** bitovou negaci – neguje každý bit vnitřní reprezentace daného čísla (podrobněji si o vnitřní reprezentaci povíme v samostatné kapitole věnované celočíselným typům).

```
(*   Příklad P8 – 1   *)
const
  Malo : integer = $26;
  Jeste: boolean = FALSE;

begin
  Malo := not Malo;             {Malo = $FFD9}
  Jeste := not Jeste;           {Jeste = TRUE}
end.
```

C++ nemá zvláštní typ logických hodnot, a proto musí rozlišovat požadovanou akci symbolem operátoru. Pokud sami definici některého z nich nerozšíříte, mohou být operandem bitové negace pouze hodnoty některého pořadového typu, kdežto operandy logické negace mohou být kteréhokoliv z dosposud probraných typů.

Operátor bitové negace ~ (tilda, vlnovka) pracuje stejně jako jeho pascalský ekvivalent: neguje každý bit vnitřní reprezentace svého operandu.

Operátor logické negace **!** (vykřičník) převádí nulu na jedničku a nenulové hodnoty na nulu. Co to znamená, když má textový řetězec hodnotu nula (přesněji ukazatel na něj má hodnotu 0), to se dozvíte, až si budeme povídat o ukazatelích.

```
/*   Příklad C8 – 1   */
  int Malo = 0x26;
  Malo = ~Malo;           //Malo = 0xFFC9
  Malo = !Malo;           //Malo = 0
  Malo = !Malo;           //Malo = 1
  Malo = ~Malo;           //Malo = 0xFFFE
```

Unární plus a minus

Tyto operátory se chovají se v obou jazycích stejně, a to tak, jak jsme z matematiky zvyklí: operátor unární + (plus) vrací hodnotu svého operandu a operátor - (minus) vrací hodnotu opačnou, tj. hodnotu vynásobenou číslem (-1). Ale pozor: předtím proběhnou standardní konverze. To znamená, že ‘a’ je sice konstanta typu **char**, ale +‘a’ je výraz typu **int**. Jsou situace, kdy to může být podstatné.

Inkrementace a dekrementace

Začneme opět Pascalem, kde je vše zcela triviální. Pascal totiž žádný inkrementační ani dekrementační operátor nemá. Aby však o tento užitečný prostředek programátoři nepřišli, byly do knihovny přidány procedury *Inc* a *Dec*, které hodnotu svého celočíselného nebo znakového parametru zvětší (zmenší) o jedničku. Musíme však mít na paměti, že *Inc* i *Dec* jsou pouze procedury, a že je proto nemůžeme používat ve výrazech.

```
(*   Příklad P8 – 2   *)
const  Test : integer = 7;
       Znak : char = 'A';
begin
  Inc( Test );       {Test = 8}
  Dec( Test );       {Test = 7}
  Inc( Znak );       {Znak = 'B'}
end.
```

V C++ je situace trošku složitější, protože každý z operátorů, tj. jak inkrementační označovaný symbolem ++ (plus plus) tak dekrementační označovaný symbolem -- (minus minus), vystupuje ve dvou různých podobách: buď jako prefixový (tj. píše před operand), nebo jako postfixový (nejdříve operand, potom operátor). Všechny čtyři operátory očekávají operand pořadového nebo reálného typu (časem si ukážeme, jak tyto operátory pracují s jinými typy) a jeho hodnotu modifikují – inkrementační k ní přičítají jedničku, kdežto dekrementační ji odečítají.

Prefixové a postfixové verze obou typů operátorů se liší vracenou hodnotou. Prefixové operátory nejprve modifikují svůj operand a vracejí hodnotu modifikovaného operandu, kdežto postfixové operátory vracejí hodnotu nemodifikovaného operandu a modifikují jej až poté.

Všechny čtyři operátory jsou sice v C++ definovány i pro operandy typu **char***, ale prozatím je s těmito operandy nepoužívejte. Počkejte si, až si v samostatné kapitole povíme o textových řetězcích podrobněji – tam se také dozvíte, jak tyto operátory nad textovými řetězci používat. Dozvíte se také, že tyto operátory můžeme používat i na ukazatele.

```
int Beru, Davam = 5;
char Znak = 'A';

Beru = Davam++ + 20;       //Beru==25, Davam==6
Beru = ++Davam – 10;       //Beru==17, Davam==7
Davam--;                   //Beru==7,  Davam==6
--Davam;                   //Totéž ->  Davam==5
Beru = --Davam * 2;        //Beru==8,  Davam==4
Beru = 10 + Davam--;       //Beru==14, Davam==3
Znak++;                    //Znak=='B'
```

Operátor přetypování

Přetypování je sice definováno v obou jazycích, ale (jak lze očekávat) každý z nich je definuje jinak.

Pascal používá dva druhy přetypování: přetypování proměnných a přetypování výrazů. Operátor přetypování vypadá jako funkce, která se jmenuje stejně jako cílový typ (tj. typ, na který přetypováváme). Další charakteristiky se liší podle toho, zda parametrem této funkce je proměnná nebo výraz.

Při přetypovávání proměnných interpretuje operátor vnitřní reprezentaci dotyčné proměnné jako reprezentaci objektu cílového typu. Na svůj operand klade dva požadavky: za prvé to musí být proměnná a za druhé musí typ operandu i cílový typ požadovat pro své objekty stejně velké místo v paměti.

Přetypování výrazů je možné pro výrazy pořadových typů (z typů, které jsme probrali, sem patří celá čísla, znaky a logické hodnoty) a výsledná hodnota vzniká konverzí, při níž se velké hodnoty ořezávají a malé doplňují.

Pascalská interpretace operátorů přetypování je pro nás tak trochu zahalena tajemstvím, protože to, co se z manuálů dozvíte, nebývá vždy pravda. (Asi je to proto, aby Pascal jako jazyk určený pro výuku poskytl studentům dostatek námětů k experimentům.) Nikde se třeba nepíše, podle čeho překladač pozná, zda chci proměnnou přetypovat jako proměnnou nebo jako výraz. Jeho chování v nás vzbuzuje dojem, že proměnné pořadových typů přetypovává jako výrazy a proměnné ostatních typů jako proměnné.

```
(*   Příklad P8 – 3   *)
Type
  Tc6 = array[ 1 ..6 ] of char;                        {6 bajtů}
  Ti3 = array[ 1 ..3 ] of integer;         {6 bajtů}
const
  c6 : Tc6 = 'abcdef';
  i3 : Ti3 = ( 0, 0, 0 );
  i0 : Ti3 = ( 0, 0, 0 );
  rr : real = 123456789;                   {6 bajtů}
begin
{Přetypování proměnných}
  i3 := Ti3( c6 );              {i3 = ($6261, $6463, $6665)}
  c6 := Tc6( rr );              {c6 = (#$81, #0, #0, #0, #0, #$40)}
  i3 := Ti3( rr );              {i3 = ($0081, $0000, $4000)}
  rr := real( I0 );             {rr = 1.0009765625}
{Přetypování výrazu}
  i3[ 3 ] := integer( c6[ 6 ] );     {i3[ 3 ] = $40}
end.
```

Operátor přetypování slouží v C++ k tomu, abychom získali objekt jiného typu, avšak pokud možno s touže hodnotou anebo alespoň s hodnotou logicky související – viz přetypování reálných čísel na celá. (Na rozdíl od Pascalu můžeme v C++ přetypovat i reálnou konstantu nebo výraz.) Není-li přetypování součástí přiřazení, překladač automaticky vytvoří pro přetypovaný objekt dočasnou proměnnou, do níž novou hodnotu uloží.

Operátor přetypování můžeme pro jednoslovné identifikátory cílových typů (zatím ani jiné neznáme) zapsat stejně jako v Pascalu, tj. ve formě funkce. Daleko používanější je však způsob převzatý z jazyka C, kde se cílový typ zapsal do závorek a za tento typ se zapsal (v případě potřeby také do závorek) konvertovaný výraz.

Pro operátory přetypování nemáme prozatím v C++ vhodné použití, protože všechny převody, které zatím používáme, zařídí překladač automaticky. Jediná situace, v níž pro nás může být nyní použití operátoru přetypování užitečné, je změna typu tištěné hodnoty – v následujících příkladech se např. snažíme tisknout obsah znakové i reálné proměnné jako znak.

```
/*   Příklad C8 – 2   */
#include <iostream.h>
double r = 65.7;                //66 == 'B'
int i = 0x500 + 'A';            //= 0x541

void /*****/ main /*****/ ()
{
  cout << i << endl             //Vytiskne 1345 = 0x541
       << char( i ) << endl     //Vytiskne 'A' = 0x41
       << r << endl             //Vytiskne 65.7
       << (char) r << endl;     //Vytiskne 'B' = 66
}
```

Operátory získání adresy a dereferencování

V profesionální praxi potřebujeme v programech velice často zjišťovat adresy různých objektů a naopak pracovat s objekty na definovaných adresách. Pro tyto účely byly zavedeny operátory získání adresy a dereference (inverzní operátor k operátoru získání adresy – mohli bychom říci získání objektu na dané adrese). Podrobněji se s těmito operátory seznámíme, až si budeme vykládat o ukazatelích.

Operátor sizeof

Operátor **sizeof** slouží k získání informace o velikost objektů. Jako parametr můžeme uvést jak identifikátor typu, tak identifikátor proměnné nebo konstanty (časem si povíme i o obecnějších možných typech parametru) a operátor nám vrátí velikost paměti (v bajtech), kterou zabírají objekty daného typu.

V Pascalu se tento operátor chová jako běžná funkce, kdežto v C++ je to běžný operátor, takže pokud jeho parametr není identifikátor typu, nemusíme jej vkládat do závorek.

Operátory pro správu dynamické paměti

Do vyšší školy programátorského umění patří dovednost dynamicky zřizovat v paměti nové objekty a po použití je opět ve vhodnou dobu rušit. Jednou z možností, jak tyto operace realizovat, je použití operátorů pro správu dynamické paměti, které se říká **halda** nebo

hromada (*heap*). Blíže se s těmito operátory seznámíme až v kapitolách, ve kterých se budeme učit s haldou pracovat.

Multiplikativní operátory

S multiplikativními operátory jsme se již setkali ve 3. kapitole, nazvané *Jednoduché výrazy*.

Operátory přístupu ke členům třídy

Operátory přístupu ke členům třídy zaujímají v hierarchii operátorů setříděných podle priority čtvrtou příčku a zabýváme se jimi v dílech, věnovaných objektově orientovanému programování.

Aditivní operátory

Také s aditivními operátory jsme se již setkali ve 3. kapitole. Zde bychom si ještě měli povědět, že operátor + (plus) lze v Pascalu aplikovat i na řetězce. Výsledkem součtu dvou řetězců je řetězec vzniklý spojením levého a pravého operandu. Pokud by délka výsledného řetězce měla překročit maximálně povolených 255 znaků, přebývající znaky se ignorují. Později se dozvíme, že aditivní operátory lze v C++ aplikovat také na ukazatele.

Posunové operátory

Posunové operátory pracují pouze s celými čísly (v C++ se všemi pořadovými typy). Posouvají vnitřní binární reprezentaci svého levého operandu o tolik bitů vpravo nebo vlevo, kolik činí hodnota pravého operandu.

Posun o jeden bit vlevo je ekvivalentní vynásobení čísla dvěma, a naopak, posun o jeden bit vpravo je ekvivalentní celočíselnému vydělení čísla dvěma. (V binární reprezentaci se při posouvání vpravo kopíruje znaménkový bit – viz příklady.) Proto také jednou z metod optimalizace je nahrazování násobení a dělení mocninami dvou odpovídajícími posunovými operacemi. Překladače obou jazyků dělají toto nahrazování automaticky.

V C++ jsou posunové operátory >> a << homonymní s operátory vstupu a výstupu do proudu. Překladač je odliší podle typu levého operandu: je-li levým operandem proud, jedná se o operátor vstupu či výstupu, je-li levým operandem hodnota pořadového typu, jedná se o posunový operátor.

```
(*   Příklad P8 – 4   *)
const
  i : integer = 1028;               {$0404}
  j : integer = -1020;              {$FC04}
begin
{Posun o 4 bity odpovídá pro kladná čísla násobení/dělení 16}
```

```
  i := i shr 4;                          {i = 64 = $0040}
  j := j shr 4;                          {j = 4032 = $FC0}
  i := i shl 4;                          {i = 1024 = $0400}
  j := j shl 4;                          {i =-1024 = $FC00}
end.
```

```
/*   Příklad C8 – 3   */
//Posun o 4 bity odpovídá násobení/dělení 16
int i = 1028;                          //i = 1028 = 0x0404
i = i >> 4;                            //i = 64 = 0x0040
i = i << 4;                            //i = 1024 = 0x0400
int j = -1020;                         //j =-1020 = 0xFC04
j = j >> 4;                            //j = -64 = 0xFFC0
j = j << 4;                            //j =-1024 = 0xFC00
```

Relační a porovnávací operátory

O relačních a porovnávacích operátorech jsme již hovořili v kapitole 3.

Test přítomnosti prvku v množině

Operátor testující přítomnost prvku v množině je specialitou Pascalu. Podrobněji se s ním seznámíme v kapitole, v níž si budeme vysvětlovat práci s množinami.

Bitové binární operátory

Bitové operátory vyžadují v Pascalu operandy celočíselných typů a v C++ typů pořadových typů. Provádějí příslušnou logickou operaci nad každou dvojicí odpovídajících bitů. V Pascalu mají bitové operátory stejné identifikátory jako operátory logické a správný typ operátoru pozná překladač podle typu jeho operandů.

Bitové operátory se používají zejména při nastavování, maskování a testování nejrůznějších binárních příznaků – nejčastěji v hodnotách, jejichž prostřednictvím komunikujeme s jednotlivými obvody počítače.

```
(*   Příklad P8 – 5   *)
const
  FF = -1;                {FF= $FFFF}
  i : integer = $5678;
  j : integer = $7abc;

var
  k : integer;

begin
  k := i and j;                {k = $5238}
  k := i or j;            {k = $7EFC}
  k := i xor j;                {k = $2CC4}
  k := i xor FF;          {k = $A987}
  k := not i;             {k = $A987}
```

```
end.
```

```
/*   Příklad C8 – 4   */
const FF = -1;              //FF= 0xFFFF
int i = 0x5678;
int j = 0x9abc;
int k;
k = i & j;                  //k = 0x1238
k = i && j;                 //k = 1
k = i | j;                  //k = 0xDEFC
k = i || j;                 //k = 1
k = i ^ j;                  //k = 0xCCC4
k = i ^ FF;                 //k = 0xA987
k = ~i;                     //k = 0xA987
k = !i;                     //k = 0
```

Logické binární operátory

Logické operátory známe již z knihy *Základy algoritmizace*. K pascalským operátorům vám nic nového neřekneme – očekávají operandy typu **boolean** a vracejí hodnotu téhož typu. Uživatelé C++ asi přivítají zjištění, že v tomto jazyce mohou být operandy logických operátorů kteréhokoliv z dosposud probraných typů.

Logické operátory se používají především v řídicích příkazech selekce a iterace. Způsob vyhodnocování výrazů s logickými operátory můžete v Pascalu ovlivnit nastavením či potlačením volby *Complete boolean eval* v dialogovém okně *Options* | *Compiler* v bloku *Syntax options*. Je-li tato volba nastavena, vyhodnocuje se výraz vždy celý (původní definice Pascalu), je-li potlačena, vyhodnocuje se zrychleně, tj. stejně jako v C++, které vyhodnocuje logické výrazy vždy zrychleně.

Podstata zrychleného vyhodnocování logických výrazů spočívá v tom, že program přestane vyhodnocovat daný výraz ve chvíli, kdy si je jist výsledkem. To nastane ve dvou případech:

1. Pokud má levý operand logického součtu (v Pascalu **or**, v C++ ||) hodnotu *ANO* (v Pascalu *TRUE*, v C++ nenulovou), bude hodnota součtu *ANO* nezávisle na hodnotě pravého operandu. Pravý operand se proto již nevyhodnocuje.
2. Pokud má levý operand logického součinu (v Pascalu **and**, v C++ &&) hodnotu *NE* (v Pascalu *FALSE*, v C++ nulovou), bude hodnota součinu *NE* nezávisle na hodnotě pravého operandu. Pravý operand se proto již nevyhodnocuje.

Pokud v Pascalu volbu *Complete boolean eval* nepotlačíte, budou se vyhodnocovat vždy oba operandy nezávisle na hodnotě levého.

Hlavní výhodou zrychleného vyhodnocování – kromě zvýšené efektivity – je, že pravým operandem může být výraz, který v situacích blokovaných levým operandem nedává smysl. Např. ve výrazu

```
if( (i <= MaxIndex) && (A[i] > 0) ) ...
```

nemá smysl se ptát na *A*[*i*] ve chvíli, kdy prvek *A*[*i*] neexistuje, protože *i* > *MaxIndex*, tedy ukazuje mimo pole *A*.

V tabulce 8.1 zjistíte, že v C++ není operátor pro logickou nonekvivalenci (logické XOR). Pokud máte zaručeno, že pracujete pouze s hodnotami 0 a 1, můžete jej nahradit operátorem =!. Jakmile se začnou v logických výrazech objevovat i jiné hodnoty, musíte operátor XOR nahradit funkcí:

```
/*   Příklad C8 – 5   */
inline int XOR ( int a, int b )
{
  return( (a==0) != (b==0) );
}
```

Podmíněný výraz

Podmíněný výraz je specialitou C++, která nemá v Pascalu obdobu. Je to operace, kterou zavedl již v padesátých letech jazyk Algol a kterou tvůrce Pascalu zavrhl jako redundantní (nadbytečnou). Jde vlastně o aplikaci konstrukce „**if ... then ... else ...**“ ve výrazu. Místo dlouhého vysvětlování vám raději vše ukážeme rovnou na příkladě:

```
/*   Příklad C8 – 6   */
int i = 1;
int j = 2;
i = i + (j % 2 ? 2 : 1);

//je ekvivalentní příkazu

{
if( j % 2 )                    //j je liché
  i = i + 2;
else
  i = i + 1;
}

//Obdobně

i = 3 + i * (j>0 ? j : -j)           //i * Absolutní hodnota j
    - (j%2 ? (j-1) : (j/2) );

//je ekvivalentní příkazu

{
  if( j > 0 )
       if( j % 2 )
           i = 3 + i*j – (j-1);
       else
           i = 3 + i*j – (j/2);
  else
  if( j % 2 )
       i = 3 - i*j - (j-1);
  else
       i = 3 - i*j - (j/2);
}
```

Pamatujte si, že ve výrazu

```
V1 ? V2 : V3
```

musí být výraz *V1* skalárního typu (všechny doposud probrané typy s výjimkou vektoru) a výrazy *V2* a *V3* musí být navzájem kompatibilních typů. *V1* je podmínka; pokud je splněna, vyhodnotí se a výsledkem bude *V2*, jinak se vyhodnotí (a bude výsledkem) *V3*. Za dvojice výrazů *V2* a *V3* se vyhodnotí vždy jen jeden.

Přiřazovací operátory

Přiřazovací operátory již používáme dlouho; přesto je ještě několik věcí, které jsme si o nich dosud neřekli.

První z nich je fakt, že Pascal umí přiřazovat hodnoty libovolných typů (kromě souborů), kdežto C++ nedokáže přiřadit hodnoty vektorových typů. Je to proto, že všechny vektorové typy, které mají stejný typ svých prvků, jsou v C++ navzájem kompatibilní, takže by pak muselo být možno přiřazovat i vektory nestejné délky.[14] Hodnoty vektorů se proto navzájem přiřazují například pomocí funkce *memcpy*, která má tři parametry: cílový vektor, zdrojový vektor a počet přenášených bajtů. Pokud budete tuto funkci chtít používat, musíte do svého zdrojového programu vložit soubor *mem.h* nebo soubor *string.h*.

```
(*   Příklad P8 – 6   *)
{ Přiřazování polí v Pascalu }
type
  Vec5 = array[ 0 ..5 ] of integer;
  Vec7 = array[ 0 ..7 ] of integer;

const
  v15 : Vec5 = ( 0, 1, 2, 3, 4, 5 );

var
  v25 : Vec5;
  v17 : Vec7;

begin
  v25 := v15;                          {v2 = ( 0, 1, 2, 3, 4, 5 )}
  v17 := v15;                          {CHYBA - nekompatibilní typy}
end;
```

```
/*   Příklad C8 – 7   */
#include <mem.h>
// Přenos polí v C++

int v15[ 5 ] = {0, 1, 2, 3, 4 };
int v25[ 5 ];
```

[14] Důvod je ve skutečnosti ještě hlubší; spočívá v tom, že pole se v C++ v mnoha případech převede na ukazatel na svůj první prvek, takže by takové přiřazení nemělo smysl – ale o tom si povíme podrobně později.

```
int v17[ 7 ];

void /*****/ main /*****/ ()
{
  memcpy( v25, v15, 5*sizeof( int ) ); //v25 = {0, 1, 2, 3, 4 }
  memcpy( v17, v15, 5*sizeof( int ) ); //v17 = {0, 1, 2, 3, 4, ?, ? }
}
```

Pro přenos polí můžeme v C++ také použít cyklu **for**, o kterém si řekneme později. Použití funkce *memcpy* je ale zpravidla efektivnější.

Jak jste si mohli všimnout v tabulce, je v C++ kromě operátoru prostého přiřazení ještě řada kombinovaných přiřazovacích operátorů. Všechny pracují podle stejného principu:

```
x op= V1
```

je ekvivalentní výrazu

```
x = x op (V1)
```

Opět se domníváme, že vše pochopíte nejlépe z příkladů:

```
/*   Příklad C8 – 8   */
int i      = 100;
int j      = 20;
int k      = 3;
int x      = 0;
char c     = 'A';
int v[5]   = {4, 3, 2, 1, 0 };

x += k;                          //x = 3
x *= i - j;                      //x = 240
x >>= 3;                         //x = 30
x |= 7;                          //x = 31
x %= j;                          //x = 11
c += 3;                          //c = 'D'
v[ v[ (i-j+k)%5 ] ] *= j - k;
//v[ v[ 3 ] ] *= 17 --- v[ 1 ] *= 17 --- v[ 1 ] = 68
//Kdyby neexistoval operátor *=, bylo by třeba psát
v[ v[ (i-j+k)%5 ] ] = v[ v[ (i-j+k)%5 ] ] * (j - k);
```

Kombinované přiřazovací operátory si vymysleli líní programátoři, kterým se na jednu stranu nechtělo opisovat jeden výraz zbytečně dvakrát a zvyšovat tak navíc riziko chyby (viz poslední uvedený příkaz) a na druhou stranu si uvědomovali, že použitím kombinovaného přiřazovacího operátoru překladači napovídají, jak by mohl daný výraz optimalizovat.

Protože programátoři v Pascalu jsou také líní a svým kolegům píšícím v jazycích C a C++ tyto operátory záviděli, rozšířili autoři Turbo Pascalu definice procedur *Inc* a *Dec* o možnost inkrementace a dekrementace o libovolné celočíselné hodnoty. Pokud chcete k hodnotě dané pořadové proměnné přičíst či odečíst jinou hodnotu než jedničku, vyvoláte patřičnou proceduru se dvěma parametry: prvním parametrem bude modifikovaná L-hodnota pořadového typu a druhým operandem celočíselná hodnota, kterou funkce *Inc* k hodnotě prvého parametru přičte a funkce *Dec* ji od něj odečte.

Ještě jednou však upozorňujeme na to, že *Inc* i *Dec* jsou procedury, a nelze je proto použít ve výrazech.

```
(*   Příklad P8 – 7   *)
const
  i : integer = 100;
  j : integer = 20;
  a : char = 'A';
  v : array[ 0 ..4 ] of integer = (4, 3, 2, 1, 0);

begin
  Dec( i, j );                        {i = 80}
  Inc( j, i );                        {j = 100}
  Inc( a, 3 );                        {a = 'D'}
  Inc( v[ v[ (i+17) mod 5 ] ], 2*j );
end;
```

Operátor postupného vyhodnocení

I s operátorem postupného vyhodnocení (čárka) jsme se již setkali. V programech se nejčastěji používá v hlavičce cyklu **for**, o kterém si budeme povídat v příští kapitole. Pamatujte si, že operátor postupného vyhodnocení zaručuje – na rozdíl od velké většiny ostatních operátorů – postupné vyhodnocení jednotlivých podvýrazů zleva doprava. (Podrobnosti o pořadí vyhodnocování najdete v příští podkapitole.)

8.4 Pořadí vyhodnocování výrazů

Pořadí vyhodnocování výrazů začne být důležité ve chvíli, kdy se ve výrazech objeví funkce s vedlejším efektem. Asociativita, znázorněná v tabulce 8.1, popisuje pořadí vyhodnocování výrazů se stejnou prioritou, avšak – až na výjimky, o kterých si povíme dále – nic neříká o pořadí vyhodnocování jednotlivých podvýrazů. Pořadí vyhodnocování podvýrazů nelze ovlivnit ani závorkami. Objeví-li se tedy ve vašem programu sekvence. (Pro jednoduchost budeme následující příklady uvádět pouze v C++).

```
j = ++i + 2*(++i) + 5*(++i);
k = i++ + 2*i;
```

a předpokládáme-li, že *i* má počáteční hodnotu nulovou, může *j* nabýt v závislost na pořadí vyhodnocování jednotlivých podvýrazů hodnot:

```
// C++
// 20 = 1 + 2*2 + 5*3
// 17 = 1 + 2*3 + 5*2
// 19 = 2 + 2*1 + 5*3
// 11 = 2 + 2*3 + 5*1
// 14 = 3 + 2*1 + 5*2
// 9 = 3 + 2*2 + 5*1
```

a *k* hodnot 9 (3+2*3) nebo 11 (3+2*4).

Pořadí vyhodnocování jednotlivých podvýrazů totiž překladač volí operativně tak, aby výsledný kód byl co nejlepší. Pokud se nám tedy v programu objeví výrazy, jejichž výsledná hodnota by mohla záviset na pořadí vyhodnocování jednotlivých podvýrazů, musíme je rozložit do několika příkazů. Předchozí úsek programu bychom tedy měli přepsat do tvaru (předpokládám, že jsme požadovali vyhodnocování podvýrazů zleva doprava):

```
// C++
j = ++i;
j += ++i * 2;
j += ++i * 5;              //j=20
k = i++;
k += 2*i;                  //k=11
//nebo
j = (i+1) + 2*(i+2) + 5*(i+3);
i += 4;
k = (i-1) + 2*i;
```

Pořadí vyhodnocování podvýrazů ovlivnilo i výsledky programu 3.11.2. Při tisku hlášení o chybném formátu římského čísla totiž program v C++ napřed vytiskl toto hlášení a teprve pak vlastní římské číslo s nulovým výsledkem. Naproti tomu pascalský program nejprve vytiskl převáděné římské číslo, pak hlášení o chybném formátu a nakonec nulový výsledek.

Z uvedeného si jistě sami odvodíte, že program v C++ nejprve vyhodnotil všechny podvýrazy (během tohoto vyhodnocování se přišlo na chybný formát a vytiskla se příslušná zpráva), a teprve pak hodnoty těchto podvýrazů postupně vytiskl.

Naproti tomu pascalský program vyhodnocuje parametry funkce *write* postupně zleva doprava a hodnotu každého z nich vytiskne hned po té, co jej vyhodnotí.

Výjimky z tohoto pravidla

V předchozím odstavci jsme naznačili, že toto pravidlo má v C++ několik výjimek. Jsou celkem 4:

- ✧ operátor čárka, který zaručuje, že nejprve se vyhodnotí levý operand (jeho hodnota se „zapomene“) a pak se vyhodnotí pravý operand (jeho hodnota je výsledkem),
- ✧ operátor podmíněného výrazu „?:“, ve kterém se vždy nejprve vyhodnotí podmínka a teprve pak jeden ze zbývajících dvou operandů (druhý z nich se nevyhodnotí),
- ✧ operátor logického součtu „||“, ve kterém se nejprve vyhodnotí první operand (vždy), a pokud je nulový (má hodnotu *NE*), vyhodnotí se i druhý operand, jinak se nevyhodnotí,
- ✧ operátor logického součinu „&&“, ve kterém se nejprve vyhodnotí první operand (vždy), a pokud je nenulový (má hodnotu *ANO*), vyhodnotí se i druhý operand, jinak se nevyhodnotí.

8.5 Zanedbání funkční hodnoty

Jazyk C++ trvá na tom, abyste při návratu z funkce, která není pouhou procedurou (tj. vrací jiný typ než **void**), vždy vraceli výslednou funkční hodnotu. Netrvá však již na tom, že tuto funkční hodnotu musí volající program použít. Této možnosti se využívá, pokud danou funkci nevoláte proto, abyste získali její funkční hodnotu, ale proto, že se vám hodí nějaký její vedlejší efekt.

Programátoři v Pascalu si dlouho stěžovali na to, že při volání takovýchto funkcí musí zbytečně zřizovat dočasné pomocné proměnné, které neslouží ničemu jinému, než tomu, aby se vyhovělo syntaktickým pravidlům. Autoři Turbo Pascalu proto umožnili ve verzi 6.0 ignorovat vracené hodnoty funkcí.

Chcete-li v Pascalu využívat možnosti zanedbání funkční hodnoty, musíte nastavit volbu *Extended syntax* v dialogovém okně *Options* | *Compiler* ve skupině označené *Syntax options*.

9. Dva užitečné příkazy

9.1 Cyklus s parametrem

Cyklus s parametrem je velice často používaná forma cyklu, kterou bychom byli již v našich programech mohli nejednou použít. Jedná se vlastně o modifikovanou podobu cyklu **while**. Jelikož se podoby cyklu v obou probíraných jazycích opět liší, probereme si jejich základní charakteristiky samostatně.

Cyklus s parametrem (přesněji příkaz cyklu s parametrem) označujeme klíčovým slovem **for**. Za ním následuje přiřazovací příkaz, v němž přiřazujeme tzv. **počáteční hodnotu parametru cyklu**; parametrem cyklu musí být proměnná pořadového typu. Za tímto přiřazovacím příkazem napíšeme klíčové slovo **to** nebo **downto** následované výrazem, jehož vyhodnocením získáme **ukončovací hodnotu cyklu**. Za tento výraz napíšeme klíčové slovo **do** a za ně příkaz, který se má cyklicky opakovat.

Myslím, že lépe než rozsáhlý slovní výklad vysvětlí podstatu cyklu s parametrem následující úsek programu:

```
{Následuje příkaz cyklu, v němž:
  i je proměnná pořadového typu (z pořadových typů jsme dosud probrali
  typy integer a char), která představuje parametr cyklu,
  Vyraz1 udává počáteční hodnotu parametru cyklu,
  Vyraz2 udává ukončovací hodnotu parametru cyklu,
  Prikaz představuje příkaz, který je tělem cyklu
}
for i:=Vyraz1 to Vyraz2 do Prikaz;
                                {Cyklus s rostoucí hodnotou parametru
  }
                                {Tento příkaz je ekvivalentní
  příkazu:}
begin
  p := Vyraz1;
  k := Vyraz2;
  if( p <= k )then
  begin
       i := p;
       repeat
           Prikaz;              {Vlastní tělo cyklu for}
           Inc( i );            {Hodnota parametru se zvětší}
       until( i <> k );         {Dokud nepřekročí ukončovací hodnotu}
  end;
end;

{Obdobně příkaz:}
for i:=Vyraz1 downto Vyraz2 do
                                {Cyklus s klesající hodnotou
  parametru}
begin
  Prikaz1;
  Prikaz2;
```

```
  ...
  PrikazN;
end;

{je ekvivalentní příkazu: }

begin
  p := Vyraz1;
  k := Vyraz2;
  if( p >= k )then
  begin
       i := p;
       repeat
           Prikaz1;              {Začátek vlastního těla cyklu for}
           Prikaz2;
           ...
           PrikazN;              {Konec vlastního těla cyklu for}
           Dec( i );             {Hodnota parametru se zmenší}
       Until( i <> k )           {Dokud neklesne pod ukonč. hodnotu}
  end;
end;
```

Ekvivalentní posloupnosti příkazů v předchozí ukázce sice neodpovídají tomu, co můžete najít ve firemním manuálu, ale zato ukazují, jak překladač cyklus s parametrem doopravdy překládá.

Pokud jsou výrazy definující počáteční a ukončovací hodnotu parametru cyklu konstantní (tj. nevyskytují se v nich proměnné), překladač je samozřejmě vyhodnotí již ve fázi překladu a podle zjištěného výsledku cyklus patřičně zjednoduší.

Možná vás napadá, proč se v ekvivalentech cyklu s parametrem zavádí proměnné *p* a *k*. Pomocná proměnná *p* se zavádí proto, aby v případě, že se cyklus nemá provádět ani jednou, zůstala hodnota parametru cyklu nezměněna a program se choval stejně, jako kdyby tam žádný cyklus nebyl.

Pomocná proměnná *k* se zavádí proto, že v zájmu efektivity se ukončovací hodnota cyklu vyhodnotí opravdu pouze jednou před vlastním spuštěním cyklu. Případné změny hodnot proměnných, které vystupují ve výrazu definujícím ukončovací hodnotu, ji proto nemohou ovlivnit.

Manuál nás nabádá, abychom při práci s cykly s parametrem měli na paměti následující tři omezení:

1. Parametrem cyklu musí být proměnná, která je v daném podprogramu **lokální**.
   Tato povinnost je sice zakotvena v definici standardního Pascalu, ale Turbo Pascal ji od verze 6.0 nevyžaduje a ani nehlídá. Mezi chybovými hlášeními však tuto chybu pod číslem 97 najdete – zbyla tu však nejspíš jako rudiment z předchozích verzí.
2. Hodnotu parametru cyklu **nesmíme** v průběhu cyklu explicitně měnit. Překladač může totiž pro zvýšení efektivity programu provádět různé akce, jejichž korektnost bychom mohli tímto přiřazením porušit.
   I zde se však jedná pouze o úlitbu bohu standardního Pascalu, kterou sice manuál vyhlásí, ale jejíž dodržování nikdo nekontroluje. Turbo Pascal si nikdy s optimalizací

moc starostí nedělal a pokud jsme si mohli všimnout, překládá cyklus s parametrem vždy standardně podle výše uvedených ekvivalentů. (Nespoléhejte na to, v příštích verzích se to může změnit.)

Jedno nebezpečí však přece jen hrozí: prostřednictvím explicitní modifikace parametru cyklu se můžeme umně vyhnout ukončovací hodnotě a vyrobit pěkně potměšilý nekonečný cyklus. (Jednou se nám program tak dokonale zacyklil, že nepomohlo ani resetovací tlačítko a nezbylo, než počítač vypnout, počkat chvíli, až se vybijí všechny kondenzátory, a opět jej zapnout.) Zkuste si například odkrokovat program:

```
(*   Příklad P9 – 1   *)
var i: integer;

begin
    for i := 1 to 3 do
    begin               {Tento cyklus je ve skutečnosti nekonečný}
        i := i + 1;
        write( i );
    end;
end.
```

3. Hodnota parametru cyklu **není** po řádném opuštění cyklu **nedefinována** (tj. neopustíme-li cyklus explicitně příkazem **goto** vloženým do těla cyklu).

   I zde manuál trochu straší. Vzhledem k podobě přeloženého cyklu (viz výše uvedené ekvivalenty) je zřejmé, že po opuštění cyklu najdete v parametru ukončovací hodnotu cyklu. (Pokud jste však cyklem vůbec neprošli, je v něm táž hodnota jako před cyklem, jak jsme si řekli při výkladu o pomocné proměnné *p*.). Přesto vám však nedoporučujeme na to spoléhat, protože například ve verzi 3.0 to bylo jinak a v některé z dalších verzí to může být opět jinak.

Jako příklad použití cyklu **for** napíšeme proceduru, která bude ověřovat platnost Gaussova vzorce pro součet prvních *n* přirozených čísel. Její zdrojový text spolu s dalšími příklady použití tohoto cyklu najdete na doplňkové disketě v souboru *P9–02.PAS*.

```
(*   Příklad P9 – 2   *)
procedure (*****)  Gauss  (*****)
( N : integer );
var
  Vzorec, Pocet, Soucet, Cislo : integer;
begin
  for Pocet:=1 to N do
  begin
      Vzorec := Pocet * (Pocet + 1) div 2;          {Podle vzorce}
      Soucet := 0;
      for Cislo:=1 to Pocet do
          Soucet := Soucet + Cislo;
      write( NL, 'Soucet cisel od 1 do ', Pocet, ' je ', Soucet,
             ' - podle Gaussova vzorce : ', Soucet, ' - ' );
      if( Soucet <> Vzorec )then
```

```
      write( 'ne' );
      write( 'souhlasi' );
  end;
  write( NL );
end;
```

Programátoři v C++ mají ve srovnání s pascalisty život přece jen trochu jednodušší, protože jazyk na ně v tomto příkazu žádné záludnosti nepřichystal (ty si na sebe připraví až oni sami). Příkaz cyklu s parametrem se v C++ stejně jako v Pascalu označuje klíčovým slovem **for**. Tam však podoba končí.

Za klíčovým slovem **for** následuje hlavička cyklu. V ní uvedeme do závorek tři výrazy oddělené středníkem, v nichž popíšeme:

1. Inicializaci cyklu (tj. co se provede před vlastním spuštěním cyklu).
   Novinkou, kterou C++ zavedlo oproti jazyku C, je, že v rámci inicializace můžeme deklarovat proměnné. Chceme-li této možnosti využít, musí výraz touto deklarací začínat.
2. Podmínku pokračování v cyklu (mohli bychom říci negace ukončovací podmínky), která se bude testovat před každým provedením těla cyklu. V ANSI C++ můžeme i v této podmínce deklarovat proměnné.[15]
3. Modifikaci parametrů pro další průchod cyklem, která se provede po každém vykonání těla cyklu („reinicializaci“).

Pomocí doposud probraných konstrukcí bychom tedy činnost cyklu s parametrem mohli definovat následovně:

```
//Příkaz cyklu:

for( Inic; Podm; Modif ) Příkaz;

//můžeme nahradit ekvivalentním příkazem:

Inic;           //Inicializační výraz (může obsahovat deklaraci)
                //Všimněte si, že vzhledem k umístění případné
                //deklarace jsou deklarované proměnné k disposici
                //i mimo vlastní tělo cyklu
while( Podm )   //Vyhodnocení podmínky pokračování cyklu
{
  Prikaz;       //Tělo cyklu
  Modif;        //Modifikační výrazový příkaz
}
```

K cyklům typu **for** připojíme ještě tři poznámky:

1. Budete-li chtít při krokování programu odlišit jednotlivé akce v záhlaví cyklu (tj. inicializaci, test setrvání v cyklu a modifikaci), musíte každou z nich uvést na samostatném řádku v programu – viz např. funkce *Gauss* v příkladu C9 – 2.

[15] Poprvé se s touto možností setkáme v Borland C++ 5.0.

2. V profesionálních programech se často setkáte s tím, že jednoduché akce, které se mají provádět v těle cyklu, se často provedou přímo v rámci modifikace a vlastní tělo cyklu pak zůstane prázdné. Takto jsou psány i jednoduché cykly ve všech třech funkcích programu C9 – 01.
3. Popsaná náhrada cyklu **for** platí ve starších verzích jazyka (v Borland C++ až po verzi 4.52 včetně). V ANSI C++ jsou proměnné, deklarované v inicializačním příkazu nebo v podmínce, k disposici pouze ve výrazech v hlavičce cyklu a v příkazech těle cyklu.

Ukážeme si dva příklady použití cyklu **for**. Funkce *PisPole()* má za úkol vypsat pole typu **int** s *n* prvky; ukazuje nejprve „obvyklé“ použití cyklu **for**, tedy použití tak, jak se zpravidla předvádí v učebnicích. Následující funkce *Gauss()* ověřuje platnost Gaussova vzorce pro součet prvních *n* přirozených čísel. Ukazuje, že cyklus **for** může mít v C++ několik parametrů (na rozdíl od Pascalu) a že jednoduché akce lze provést přímo v hlavičce cyklu, takže tělo je prázdné. Úplný text obou funkcí a další příklady na použití cyklu **for** najdete v souboru *C9–01–02.CPP* na doplňkové disketě.

```
/*   Příklad C9 – 1   */
static void  /*****/  PisPole  /*****/
(int P[ ], int n)
{      //Tiskne jeden prvek vektoru za druhým a odděluje je mezerou
       //Takto se použití cyklu for předvádí v učebnicích
  for( int i=0; i < n; i++ )
                      cout << P[ i ] << " " ;
  cout << endl;                 //Na závěr odřádkuje
}

/*   Příklad C9 – 2   */
static void /*****/  Gauss  /*****/
( int N )
//Ověření Gaussova vzorce pro prvních N čísel s využitím cyklu for
{
  for( int Pocet=1;   Pocet <= N;   Pocet++ )
  {
       int  Vzorec = Pocet * (Pocet+1) / 2;//Podle vzorce

       for( int Soucet = 0,      //Inicializační výraz může obsahovat
                 Cislo  = 1;           //i deklaraci
            Cislo  <= Pocet;     //Test pokračování cyklu
            Soucet += Cislo++ ); //Modifikace hodnot parametrů cyklu
   //V  cyklu  for  se  jednoduché  akce  často  vykonávají  v  rámci
  modifikace

       cout     << "\nSoučet čísel od 1 do " << Pocet << " je " <<
  Vzorec
            << " - podle Gaussova vzozce: " << Soucet << " - "
            << ((Vzorec == Soucet) ? "" : "ne") << "souhlasí";
  }/* vnější for*/
  cout << endl;
}
```

V našich dosavadních programech byste našli řadu míst, v nichž by bylo použití cyklu s parametrem výhodné. Až si prohlédnete doprovodné programy, zkuste si znovu projít všechny minulé programy v této knize a najděte v nich místa, kde by bylo výhodnější použití cyklu **for**.

9.2 Přepínač

Přepínače jsou programové konstrukce, které umožňují rozvětvit program do několika větví na základě jediného vyhodnocení podmínky. V tom je také jeho základní rozdíl od podmíněného příkazu, který umí rozvětvit program pouze do dvou větví.

V Pascalu se jako přepínač používá příkaz **case**, v C++ příkaz **switch**. Jelikož některé podstatné podrobnosti, stojící v pozadí těchto příkazů, jsou v obou vysvětlovaných jazycích odlišné, vysvětlíme si opět každý zvlášť.

V Pascalu je příkaz **case** tvořen z následujících částí:

1. Začíná klíčovým slovem **case**.
2. Za ním následuje **výběrový výraz** (selektor). Jeho vyhodnocením obdržíme hodnotu pořadového typu, která definuje větev, kterou se bude pokračovat.
3. Za selektorem následuje klíčové slovo **of,**
4. a za ním seznam jednotlivých větví oddělených středníky.
5. Příkaz **case** končí klíčovým slovem **end**.

Každá větev přepínače musí začínat seznamem tzv. **vstupních hodnot** oddělených čárkami. Místo několika po sobě následujících hodnot je možno zapsat i interval (připomínáme: dolní mez, operátor rozsahu (**..**) a horní mez).

Množiny vstupních hodnot jednotlivých větví musí být navzájem disjunktní. Pokud by se totiž některá hodnota vyskytla v několika seznamech, nemohl by počítač jednoznačně určit, kterou větví má dále pokračovat.

Za seznamem vstupních hodnot se píše dvojtečka a za ní příkaz tvořící vlastní tělo větve.

Turbo Pascal navíc umožňuje ukončit seznam větví přepínače větví, kterou se bude pokračovat v případě, že hodnota výběrového výrazu nebyla nalezena v žádném seznamu vstupních hodnot. Tato větev začíná klíčovým slovem **else**, za nímž následuje příkaz tvořící vlastní tělo větve (**bez** oddělující dvojtečky).

Narazí-li počítač v programu na přepínač, vyhodnotí nejprve výběrový výraz. Pokračovat bude větví, mezi jejímiž vstupními hodnotami je i hodnota výběrového výrazu. Pokud taková větev v seznamu není, pokračuje se větví **else**, a pokud v seznamu není ani větev **else**, pokračuje se prvním příkazem za přepínačem.

Jako příklad si uvedeme úsek programu, který se uživatele zeptá, jakým způsobem má zašifrovat zadaný text, a podle toho s ním naloží. Úplný program najdete na doplňkové disketě v souboru *P9–03.PAS*.

```
(*   Příklad P9 – 3   *)
const
  NL       = #13#10;                  {Přechod na nový řádek}
  TAB      = #9;                      {Tabulátor}
  Posunuti = 3;

type
  TSlovo  = string[ 40 ];             {Řetězec pro max. 40 znaků}

var
  Slovo, Sifra : TSlovo;
  Volba        : char;

 (************** Hlavní program **************)
begin
  repeat
       write( NL, NL, NL, 'Zadej šifrované slovo (max. 40 znaků):' );
       read( Slovo );
       write(    NL, 'Zadej požadovaný způsob šifrování:', NL,
                 TAB, 'K - Konec', NL, TAB, 'O - Obráceně', NL,
                 TAB, 'Z - Záměna', NL, TAB, 'P - Posunutí', NL, NL,
                 'Požadovaná akce: ');
       repeat
           read( Volba );
       until( Volba > ' ' );          {Přeskoč bílé znaky}
       write( NL, 'Zašifrováno: ' );

       case UpCase(Volba)  of
           'O' : Obrat( Slovo, Sifra);
           'Z' : Zamen( Slovo, Sifra);
           'P' : Posun( Slovo, Sifra);
           'K' : Exit;
       else
           begin
                write('Špatně zadaná volba -> obracím:');
                Obrat( Slovo, Sifra );
           end;
       end; { od case Volba of ... }

       write( Sifra );
       repeat
           read( Volba );
       until( Volba = #10 );          {Dočti do konce řádku}
  until( FALSE );
end.
```

Jak jsme si již řekli, přepínač v C++ se chová trochu jinak. Na rozdíl od pascalského přepínače totiž mohou jednotlivé větve přepínače v C++ sdílet části kódu. Podívejme se však nejprve na jeho syntax.

Syntax přepínače se v C++ podobá syntaxi podmíněného příkazu: nejprve napíšeme klíčové slovo **switch**, za něj do závorek výběrový výraz, jehož hodnota musí být pořadového typu, a za závorku příkaz, kterému budeme říkat **tělo přepínače**.

Přepínač odlišuje od podmíněného příkazu podoba jeho těla – tj. příkazu za závorkou. Aby mělo použití přepínače smysl, musí být tento příkaz blokem (složeným příkazem), který obsahuje všechny větve přepínače.

Jednotlivé větve jsou označeny návěštími, které začínají klíčovým slovem **case** následovaným konstantním výrazem (tj. výrazem, který lze vyhodnotit již v době překladu) a které je (jako každé návěští) ukončeno dvojtečkou.

Hodnoty výrazů u jednotlivých návěští **case** (budeme jim říkat **vstupní hodnoty větve**) se musí navzájem lišit. Pokud by se totiž některá vstupní hodnota vyskytla v několika návěštích **case**, nemohl by počítač jednoznačně určit, odkud má dále pokračovat.

Pro hodnoty výběrového výrazu, které nesouhlasí s žádnou z hodnot výrazů v návěštích **case**, můžeme definovat větev uvedenou návěštím **default**. Na rozdíl od Pascalu nemusí být tato větev uvedena jako poslední – návěští **default** můžeme umístit prakticky libovolně.

Narazí-li počítač v programu na přepínač, vyhodnotí nejprve výběrový výraz. Pokračovat se bude v těle přepínače od toho návěští **case**, jehož vstupní hodnota je shodná s hodnotou výběrového výrazu. Pokud takové návěští v příkazu není, pokračuje se od návěští **default**, a pokud v seznamu není ani návěští **default**, pokračuje se prvním příkazem za přepínačem.

Upozornění:
Klíčová slova **default** *a* **case** *slouží* ***pouze*** *k označení návěští a v žádném případě tedy neukončují předchozí větev. Větev můžeme ukončit pouze příkazem* **break**, **return** *nebo* **goto** *(ale to je neslušnost). Pokud žádným z těchto příkazů větev neukončíme, bude se pokračovat větví následující – viz doprovodný program.*

Jako příklad si ukážeme úsek programu, který se uživatele zeptá, jakým způsobem má zašifrovat zadaný text, a podle toho s ním naloží. Úplný program najdete na doplňkové disketě v souboru *C9–03.CPP*.

```
/*   Příklad C9 – 3   */
#include <iostream.h>
#include <string.h>
#include <ctype.h>

const POSUN = 3;

static void Obrat( const char Text[], char Sifra[] );
static void Posun( const char Text[], char Sifra[] );
static void Zamen( const char Text[], char Sifra[] );

void /*****/  main  /*****/ ()
{
  char Slovo[ 40 ], Sifra[ 40 ];
  char Volba;
```

```
  while( 1 )                                  // Nekonečný cyklus
  {
    cout << "\n\n\nZadej šifrované slovo (max. 40 znaků): ";
    cin  >> Slovo;
    cout << "\nZadej požadovaný způsob šifrování:\n"
            "\tK - Konec\n\tO - Obrácenĕ\n"
            "\tZ - Záměna\n\tP - Posunutí\n\n"
            "Požadovaná akce: ";
    cin >> Volba;
    cout << "\nZašifrováno: ";

    switch( toupper( Volba ) )
    {
       case 'K':
           return;
       default:
           cout << "Špatnĕ zadaná volba -> obracím: ";
       case 'O':
           Obrat( Slovo, Sifra );
           break;
       case 'P':
           Posun( Slovo, Sifra );
           break;
       case 'Z':
           Zamen( Slovo, Sifra );
           break;
    } /* konec příkazu switch */

    cout << Sifra;
  }
}
```

Všimněte si větve **default**. Jestliže se tato větev použije, vypíše se upozornění a pak se přejde do větve **case** ‘O’, neboť větev **default** neobsahuje žádný z příkazů, které by způsobily ukončení těla příkazu **switch**. Zavolá se tedy funkce *Obrat()* a teprve pak provádění příkazu **switch** skončí.

10. Podrobnosti o skalárních typech

V této kapitole doplníme vědomosti o skalárních datových typech informacemi o jejich jednotlivých variantách. Postupně se podíváme na celočíselné datové typy, znaky, logické hodnoty a reálné datové typy. Na závěr si pak povíme o **výčtových datových typech** a o práci s nimi.

Než se pustíme do vlastního výkladu, musíme vás upozornit, že všechny údaje o paměťovém prostoru a rozsahu hodnot, které zde uvedeme, platí pouze pro překladače Turbo Pascal a Borland C++. Stejné údaje nejspíš platí pro většinu překladačů určených pro počítače typu PC, ale na jiných počítačích to může být jinak.

10.1 Celá čísla

Jak jsme si již řekli, existuje několik celočíselných typů. Objekty doposud probíraných datových typů *integer* (Pascal) a **int** (C++) zaujímaly v paměti 2 bajty a mohly nabývat hodnot z intervalu <-32768; 32768>.[16]

Tento rozsah je však pro některé aplikace příliš malý. Oba programovací jazyky proto definují vlastní typ pro velká celá čísla, který se v Pascalu jmenuje *longint* a v C++ **long** (lze použít i úplnou specifikaci **long int**, většinou se však používá zkrácená verze). Objekty tohoto datového typu zabírají v paměti 4 bajty a mohou nabývat hodnot z intervalu <-2147483648; 2147483647>.

V řadě případů je potřeba pracovat pouze s malými čísly a 2 bajty zabírané každým číslem pro nás mohou být zbytečným přepychem, protože zpracovávané hodnoty by se bohatě vešly do bajtu jednoho. V takovém případě volíme v Pascalu mezi typy *shortint*, který pokrývá interval <-128; 127>, a *byte*, který pokrývá rozsah <0; 255>.

V C++ se pro tyto účely používá znakových typů, které jsou – jak již víme – ve výrazech zcela kompatibilní s celočíselnými typy. Ke konkrétní podobě jejich deklarací se vrátíme v příští podkapitole.

Poslední variantou celočíselných datových typů, o níž bychom se chtěli zmínit, jsou typy bez znaménka. Neznaménkové datové typy se velice často používají při přímé spolupráci programu s hardwarem, ale hodí se i pro jiné účely.

V Pascalu se neznaménkový ekvivalent typu *integer* jmenuje *word*, zabírá v paměti také 2 bajty a pokrývá rozsah hodnot <0; 65535>. V C++ je jeho ekvivalentem typ **unsigned** (lze použít i plnou specifikaci **unsigned int**, ale většinou se nechává na překladači, aby si **int** domyslel), a kromě něj je zde ještě zaveden typ **unsigned long**, který pokrývá rozsah <0; 4294967285>. Typ **unsigned long** nemá v Pascalu obdobu.

---

[16] To platí v programech pro reálný režim DOSu a pro 16bitová Windows. V programech pro chráněný režim (např. pro Windows 95 nebo Windows NT) jsou typy *integer* resp. **int** 4bajtové, tj. pokrývají rozsah <-2147483648; 2147483647> (stejný jako typ *longint* resp. **long**).

Ve výkladu o operátorech bitového posunu jsme si říkali, že se při nich v C++ zachovává při posunu doprava znaménko (v Pascalu ne). To znamená, že posun o jeden bit doprava změní binární reprezentaci 0xFFFF čísel typu **unsigned**, tj. neznaménkových, na 0x7FFF (tj. 32767), kdežto u čísel typu **int**, tj. znaménkových, zachová hodnotu -1. Pascal upraví obě hodnoty na 0x7FFF, tj. na 32767. (Poznamenejme, že 0xFFFF reprezentuje u typu **int** hodnotu -1 a u typu **unsigned** hodnotu 65535.) Pokud chcete v C++ posouvat jiným způsobem, než jak by vyplývalo ze „znaménkovosti" dané hodnoty, musíte použít operátor přetypování – např.

```
i = unsigned( 60000 ) >> 1;
```

Máme ovšem ještě jednu možnost – zadat posouvaný literál jako neznaménkový. C++ nám totiž umožňuje předepsat i typy literálů, a to pomocí tzv. **celočíselných přípon**. Přidáme-li za číslo příponu *U* nebo *u*, označujeme daný literál za neznaménkový, a přidáme-li příponu *L* nebo *l*, definujeme daný literál jako číslo typu **long**. Chceme-li literál definovat jako číslo typu **unsigned long**, musíme uvést přípony obě, přičemž na pořadí ani velikosti písmen nezáleží. Předchozí příklad bychom tedy mohli zapsat

```
i = 60000U >> 1;
```

V programech psaných v C či C++ se setkáte ještě s datovým typem **short int** nebo zkráceně **short** a s typem **unsigned short**. Jelikož vlastnosti tohoto typu jsou v překladačích určených pro reálný režim PC většinou totožné s vlastnostmi typu **int** resp. **unsigned int**, používá se jen velice zřídka.[17]

Je ale důležité vědět, že „velikost", rozsah typů **short**, **int** a **long** není nikde definována a každý překladač je tedy může implementovat po svém. Jediné, co definice jazyka vyžaduje, je platnost symbolické nerovnosti

```
short <= int <= long
```

tedy aby rozsah typu **short** nebyl větší než rozsah typu **int** a rozsah typu **int** aby nebyl větší než rozsah typu **long**. Podobně pro rozsahy typů bez znamének musí platit

```
unsigned short <= unsigned <= unsigned long
```

Přehled všech celočíselných typů uvádí následující tabulka:

Pascal	C++	Rozsah
shortint	**signed char**	-128 .. 127
byte	**unsigned char**	0 .. 255
integer	**int**	-32 768 .. 32 768
word	**unsigned**	0 .. 65 535
longint	**long**	-2 147 483 648 .. 2 147 483 647
	unsigned long	0 .. 4 294 967 285

[17] Stejný rozsah má ale i v programech pro chráněný režim, takže zde je odlišný od typu **int.**

Tab. 10.1 Celočíselné datové typy

10.2 Znaky

V Pascalu není co k našim znalostem dodat, a proto čtenáři, které C++ nezajímá, mohou klidně přejít k další podkapitole.

Jak jste již nepochybně pochopili z předchozí tabulky, C++ rozlišuje znaky se znaménkem a bez něj. Implicitním prvotním nastavením překladače Borland C++ jsou znaky se znaménkem. Je to proto, že původní verze jazyka C pracovaly implicitně se znaky se znaménkem a C++ od nich toto nastavení dědí. Autoři jazyka C totiž znaky s kódy nad 127 prakticky nepoužívali (angličtina je nepotřebuje) a vzhledem k požadované kompatibilitě s typem **int** jim připadala znaménková varianta přirozenější.

Pokud bychom přistoupili na práci se znaménkovými znakovými typy, měla by všechny písmena s diakritickými znaménky kódy menší než 0, a to by nám asi naše programy komplikovalo. Proto použijeme volbu *Options | Compiler | Code Generation* a v nabídnutém dialogovém okně zkontrolujeme, zda je volba *Unsigned characters* opravdu nastavena, tj. zda je v jejích hranatých závorkách **X**.

Když jsme popisovali filtr, který měl upravovat zdrojové texty do žádané podoby pro tiskárnu, nabádali jsme vás, abyste tuto volbu potlačili. Nyní si povíme proč.

Základním problémem bylo, jak zachytit ve vstupním proudu hodnotu *EOF* označující konec souboru. Protože tato hodnota musí mít jiný kód než všechny běžné znaky, vyřešili to autoři jazyka C tak, že operace čtení znaku nevrací znak, ale celé číslo. Tím si rázem zpřístupnili rozsáhlou množinu celočíselných hodnot, kterých běžné znaky nabýt nikdy nemohou. *EOF* mívá většinou hodnotu -1.

Připomeneme, že vnitřní reprezentace čísla -1 má tvar 0xFFFF. Pokud tedy pracujeme se znaky se znaménkem, dosáhneme binární reprezentace 0xFFFF znaménkovým rozšířením vnitřní reprezentace znaku s kódem 0xFF, tedy 255 nebo -1. Při práci se znaménkovými znaky je proto hodnota *EOF* dosažitelná, protože existuje znak, jehož celočíselným rozšířením tuto hodnotu získáme. Tento znak se však nesmí vyskytnout uprostřed souboru, protože by se po jeho přečtení program chybně domníval, že soubor je již vyčerpán.

Problém se zachycením EOF vyřešíme nejlépe tak, že proměnná, do níž načítáme hodnotu vstupu funkcí *cin.get*, bude typu **int**, abychom ji mohli s hodnotou EOF porovnávat. Ostatní znakové proměnné mohou být nadále typu **char**. Pokud potřebujeme hodnotu celočíselné proměnné vytisknout jako znak, pomůžeme si operátorem přetypování.

Znaky v ANSI C++

Předchozí povídání platí beze zbytku ve starších verzích C++. V ANSI C++ je ale situace o něco komplikovanější, neboť ANSI C++ považuje **char**, **signed char** a **unsigned char**

za 3 různé typy, a to i přes to, že typ **char** je implementován jako jeden ze zbývajících dvou. Důsledky tohoto pravidla se uplatní při přetěžování funkcí, kdy se jednotlivá homonyma rozlišují podle počtu a typů parametrů.

10.3 Logické hodnoty

Je známou skutečností, že proměnné logických typů používá většina programátorů pouze zřídka. Není to však tím, že by v jejich programech nebyla pro logické proměnné příležitost, jako spíše tím, že programátoři nechtějí opouštět zabydlený svět celých čísel a proto místo logických proměnných používají velice často proměnné celočíselné.

Jak jsme si již řekli, jazyk C dokonce ani logické proměnné nezavedl. Nenajdeme je ani ve starších verzích jazyka C++; obsahuje je teprve ANSI C++. Prostředky C++ nám však dovolují datový typ logických hodnot definovat i ve starších překladačích. Avšak k tomu, aby tato definice byla opravdu plnohodnotná, nám naše dosavadní znalosti ještě nestačí – to dokážeme až za pomoci prostředků objektově orientovaného programování.

Jedním z důvodů, proč se autoři jazyka C do zavedení logického datového typu nehrnuli, bylo i to, že zavedením tohoto typu nic nezískají (s výjimkou možnosti typové kontroly, ale ta jim byla proti srsti), protože logickým proměnným musí v paměti vyhradit stejně alespoň jeden bajt.

Jiná situace ovšem nastává, když začneme hovořit o polích logických hodnot. Kdybychom totiž každému prvku takového pole vyhradili pouze jeden bit (a logické hodnoty víc paměti nepotřebují), mohli bychom snížit potřebu paměti až 8krát. Toho se však v obou jazycích dosahuje jinými prostředky, o nichž budeme hovořit později.

Typ bool

Typ **bool**, zavedený normou ANSI, má dvě hodnoty, vyjádřené klíčovými slovy **false** a **true**. Patří mezi celočíselné typy a je s ostatními celočíselnými typy plně kompatibilní. To znamená: konstanta **false** se při přiřazení proměnné jiného celočíselného typu konvertuje na 0, konstanta **true** na 1. Při opačném přiřazení se nenulová hodnota konvertuje na **true**, nula na **false**.

Relační operátory vytvářejí v ANSI C++ hodnoty typu **bool**, nikoli typu **int**.

Při praktickém programování můžeme většinou na jeho existenci zapomenout a téměř nic se nestane. Musíme si ale dát pozor při přetěžování funkcí.

Příklad

Abyste si trochu pocvičili práci s logickými proměnnými, připravili jsme pro vás následující úlohu ze sbírky úloh pro gymnázia: V okamžiku, kdy dohlížející učitel na chodbě slyšel třeskot skla, byli ve třídě tři žáci: David, Eda a Filip. Při vyšetřování se zjistilo, že:

1. u okna byl nejvýše jeden z dvojice David - Eda,
2. Filip byl u okna právě tehdy, když tam nebyl David,
3. pokud nebyl u okna Eda, nebyl tam ani David.

Lze určit pachatele v případě, že byl pouze jeden? Pokud ano, který z žáků to byl? (Řešení najdete v programu*P10–00A.PAS* resp. *C10–00A.CPP* na doplňkové disketě.)

10.4 Reálná čísla

Termín „reálná čísla“ se některým programátorům jako označení třídy datových typů nelíbí. Prohlašují, že v počítači reálná čísla zobrazit nelze, a že by se proto tato čísla měla nazývat racionální. Pokud se nad problémem zamyslíte, zjistíte, že i z racionálních (ba dokonce i z celých) čísel můžete v počítači zobrazit pouze nějakou jejich podmnožinu, takže si nebudeme dělat násilí, a budeme se o těchto číslech dále bavit jako o reálných.

Reálná čísla mají stejně jako čísla celá řadu variant, které se navzájem liší potřebným paměťovým prostorem a tím i dosažitelnou přesností. Charakteristiku i identifikaci všech reálných typů, které jsou v probíraných jazycích k dispozici, si můžete přečíst v tabulce 10.2.

Typ		Rozsah exponentu	Platných cifer	Bajtů
Pascal	**C++**			
single	**float**	-38 – 38	7 – 8	4
real	—	-39 – 38	11 – 12	6
double	**double**	-308 – 308	15 – 16	8
extended	**long double**	-4932 – 4932	19 – 20	10
comp	—	—	19 – 20	8

Tab.10.2 Reálné datové typy

V prvních dvou sloupečcích najdete identifikátory reálných typů v Pascalu a v C++, ve třetím rozsah exponentu, ve čtvrtém je zaručený uchovaný počet platných číslic (v průběhu složitých výpočtů se může samozřejmě zmenšovat) a v posledním pak počet bajtů, které zaberou objekty daného typu v paměti.

Podíváte-li se do manuálu k borlandskému Pascalu, zjistíte, že v této tabulce jsou o něco menší rozsahy hodnot exponentů. Je to proto, že zde uvádíme exponenty nejmenších normalizovaných hodnot, tj. nejmenších hodnot se zaručeným počtem platných číslic. Naproti tomu v manuálu najdete nejmenší dosažitelnou hodnotu – u ní však máte zaručenu pouze jednu platnou binární číslici, tj. asi 0,3 číslice desítkové.

Datové typy *single*, *double* a *extended* (Pascal), resp. **float**, **double** a **long double** (C++) jsou totožné s datovými typy, s nimiž pracuje numerický koprocesor 80x87, a odpovídají tedy standardu IEEE 754.

Překladač překládá výrazy, v nichž vystupují objekty těchto typů, vždy tak, jako kdyby byl k dispozici numerický koprocesor. Do inicializační posloupnosti programu pak přidá podprogram, který zjistí, zda je počítač, na němž program běží, opravdu koprocesorem vybaven. Pokud je, posílají se všechny zpracovávané hodnoty koprocesoru, pokud koprocesor k dispozici není, používají se místo něj podprogramy z **emulační knihovny**, které činnost koprocesoru emulují (jednotlivé instrukce koprocesoru – např. násobení dvou reálných čísel – vypočtou pomocí zvláštního podprogramu).

Vnitřní datový formát matematického koprocesoru odpovídá typu *extended* (Pascal) resp. **long double** (C++), který má 64bitovou mantisu a 14bitový exponent. V tomto formátu probíhají i veškeré výpočty podprogramů emulační knihovny. Proto se také použitím „kratších" datových typů zrychlí výpočet pouze o dobu nutnou k manipulaci s větším počtem bajtů, ale zásadního zrychlení tím nedosáhneme. (Doby, za něž na našem počítači proběhlo 5 000 000 násobení reálných čísel jednotlivých reálných typů z programu *C10–0.CPP* resp. *P10–0.PAS* na doplňkové disketě, jsou souhrnně uvedeny v tabulce 10.3). Mezi uvedenými třemi datovými typy si tedy vybíráme spíše podle velikosti dostupné paměti a požadované přesnosti ukládaných mezivýsledků.

Pascal		C++	
Typ	**Čas (s)**	**Typ**	**Čas (s)**
single	4,73	**float**	4,34
double	4,72	**double**	5,38
extended	4,66	**long double**	6,21
real	18,62	—	—
real (emulace)	60,40	—	—

Tab. 10.3 Porovnání doby výpočtu[18]

Upozornění:
Programátoři v C++ si musí při používání výpustkových parametrů typu **float** *(viz procedury s proměnným počtem parametrů) uvědomit, že tyto parametry budou předávány jako hodnoty typu* **double** *a jako takové si je musí volaná procedura přebírat.*

C++ umožňuje určit datový typ i u reálných literálů – jak si asi sami domyslíte, opět pomocí přípon. Pokud nemá reálný literál (tj. literál, který obsahuje desetinnou tečku nebo exponentovou část) žádnou příponu, přisoudí mu překladač typ **double**. Chcete-li, aby byl daný literál považován za číslo typu **float**, musíte jej doplnit příponou *f* nebo *F*, a chcete-li jej definovat jako číslo typu **long double**, musíte k němu připojit příponou *L* nebo *l*.

Standardním reálným typem jazyka Pascal není žádný z uvedených koprocesorových typů, ale typ *real*, jehož objekty zaberou v paměti 6 bajtů. Jeho absolutní hodnoty se mohou

[18] Na počítači s procesorem Pentium, 75 MHz.

pohybovat v rozmezí od 2.9×10^{-39} do 1.7×10^{38}, přičemž čísla jsou uchovávána s přesností na 11 platných cifer.

Tento datový typ se v Turbo Pascalu zavádí v zájmu zachování kompatibility s předchozími verzemi, které ještě neuměly emulovat koprocesor 80x87, a používaly proto takovou vnitřní reprezentaci čísel, která by jim při přijatelné přesnosti dovolila co nejrychlejší výpočet.

Pokud potřebujete provádět rozsáhlejší numerické výpočty a nemůžete si vybavit počítač koprocesorem (je to sice nepravděpodobné, ale...), určitě tento datový typ přivítáte. Pokud však naopak chcete používat koprocesor nebo emulační knihovnu, výpočty s tímto datovým typem budou probíhat pomaleji, protože se hodnoty typu *real* musí napřed konvertovat na typ *extended* a naopak výsledky se musí konvertovat zpět na typ *real* (viz tabulka 10.3).

Pascal poskytuje ještě jeden „koprocesorový“ typ, který je v manuálu zařazován mezi reálné, i když je svojí podstatou vlastně celočíselný. Je to typ *comp*, jehož objekty zabírají v paměti 8 bajtů a mohou nabývat hodnot od $-2^{63}+1$ do $2^{63}-1$.

10.5 Výčtové typy

Vzpomeňte si na robota Karla z knihy *Základy algoritmizace*. Asi si pamatujete, že dokázal otestovat, zda je pod ním značka, zda je před ním zeď a zda je otočen na definovanou světovou stranu. Při svých současných znalostech již asi sami dokážete odhadnout, že Karel si směr, do nějž byl natočen, ukládal do proměnné – nazvěme ji *Směr*.

Tato proměnná mohla nabývat čtyř hodnot odpovídajících východu, severu, západu a jihu. Jinými slovy, měla tu zvláštní vlastnost, že její hodnoty bylo možno vyjmenovat. Je tedy zřejmé, že by pro přehlednost programu bylo vhodné tyto hodnoty pojmenovat a odkazovat pak v textu pouze na jejich identifikátory.

Pokud bychom toužili dané hodnoty pouze pojmenovat, stačily by nám k tomu konstanty – s těmi již pracovat umíme. My bychom však potřebovali něco více: aby překladač mohl zkontrolovat, že jsme dané proměnné nepřiřadili omylem nějakou hodnotu, která v ní ve skutečnosti být vůbec nemůže.

Vrátíme-li se k příkladu s Karlem, můžeme se dohodnout, že vnitřní reprezentací natočení Karla do jednotlivých směrů budou hodnoty 0 (východ) až 3 (jih). Co se však stane, když proměnné *Směr* omylem přiřadíme hodnotu 7?

Protože tato situace není neobvyklá, pomohli si programátoři tak, že zavedli tzv. **výčtové typy** (v literatuře se setkáte také s pojmy vyjmenované nebo enumerativní), tj. typy definované výčtem svých hodnot. Tyto hodnoty mají v programu obdobnou funkci jako literály, a proto je budeme nazývat **výčtové literály**.

Předchozí úvahy bychom mohli shrnout do tvrzení, že výčtové typy byly zavedeny proto, aby bylo možno lépe pracovat s nějakou množinou v podstatě nenumerických hodnot a aby bylo možno kontrolovat korektnost všech prováděných operací.

Někteří programátoři používají konvenci, že všechny identifikátory výčtových typů začínají malým písmenem **e** (*enumerated*) a za ním následuje vlastní jméno datového typu začínající velkým písmenem. I my se této konvence budeme v našich příkladech držet.

Před výkladem specifik obou jazyků ještě připomeneme, že výčtové typy řadíme spolu s typy celočíselnými, znakovými a logickými mezi tzv. **pořadové typy**.

V Pascalu výčtové typy definujeme tak, že v kulatých závorkách uvedeme seznam identifikátorů jejich jednotlivých hodnot (výčtových literálů).

```
type
  eSmer = ( _VYCHOD, _SEVER, _ZAPAD, _JIH );
  eDen = ( PO, UT, ST, CT, PA, SO, NE );
  eBarva = ( CERNY, BILY );
  eFigura = ( NIC, PESEC, JEZDEC, STRELEC, VEZ, DAMA, KRAL );
  eSloupec = ( a, b, c, d, e, f, g, h );
```

Pokud vás zajímá vnitřní reprezentace hodnoty daného výčtového literálu, můžete si ji odvodit z pořadí jeho výskytu v seznamu literálů v definici daného výčtového typu. Vnitřní reprezentací první hodnoty ze seznamu je nula, druhé hodnoty jednička atd. Obecně *n*-tá hodnota z tohoto seznamu je v počítači reprezentována číslem *n*-1.

V praxi se často setkáme se situacemi, kdy sice víme, že hodnoty výčtových literálů jsou svoji podstatou nenumerické, avšak rádi bychom jejich vnitřní reprezentaci ovlivnili tak, aby nám to v dalším programu co nejvíce usnadnilo práci.

Přímé přiřazování vnitřní reprezentace výčtovým literálům, které poznáme např. v C++, Pascal bohužel neumožňuje. Pokud však zrovna nepotřebujeme, aby vnitřní reprezentace nabývala záporných nebo velkých kladných hodnot, ale potřebujeme pouze mít mezi hodnotami vnitřních reprezentací některých literálů menší mezery, stačí, když do seznamu začleníme několik atrap (nepoužívaných identifikátorů) tak, aby následující výčtový literál měl již požadovanou vnitřní reprezentaci (viz příklad P10 – 1).

Pokud bychom potřebovali, aby několik výčtových literálů mělo stejnou vnitřní reprezentaci, můžeme využít jedno z rozšíření, které má Turbo Pascal oproti standardnímu Pascalu. Díky tomu, že Turbo Pascal netrvá na pevném pořadí sekcí deklarací návěští, konstant, typů, proměnných a podprogramů, ale umožňuje tyto sekce dokonce i prokládat, máme možnost deklarovat i konstanty výčtových typů. (V manuálu však o tomto rozšíření žádnou explicitní zmínku nenajdete – musíte umět číst mezi řádky.) Zmíněnou deklarací konstant pak můžete obejít potřebu několika výčtových literálů se stejnou vnitřní reprezentací.

```
(*   Příklad P10 – 1   *)
type                        {Definice typu "binární řád"}
  eBinRad = ( NULA, JEDNA, DVE, B3, CTYRI, B5, B6, SEDM );
  {B3, B5 a B6 jsou atrapy, které v programu nepoužijeme}
```

Jedinými požadavky, které nám Turbo Pascal neumožní splnit, je záporná a hodně velká kladná hodnota vnitřní reprezentace. (Teoreticky by to sice šlo, ale za cenu neúměrně velké pracnosti.) Pokud na těchto požadavcích trváme, nezbývá vám, než oželet výhody ty-

pové kontroly, a definovat plánované výčtové literály jako celočíselné konstanty. (Zvykejte si na to, že v Pascalu je důsledná typová kontrola možná pouze v jednoduchých školních příkladech.)

V souvislosti s výčtovými typy se musíme seznámit se třemi novými funkcemi: *Ord*, *Pred* a *Succ*. Všechny tři funkce vyžadují jeden parametr pořadového typu. Funkce *Ord* vrací celé číslo, které je vnitřní reprezentací parametru (tedy vlastně pořadí v definici výčtového typu), funkce *Pred* vrací předchůdce dané hodnoty v seznamu výčtových literálů (pokud existuje) a funkce *Succ* vrací naopak následovníka svého parametru.

Funkce *Ord*, *Succ* a *Pred* lze použít s parametry jakéhokoliv pořadového typu, tzn. včetně znaků, celých čísel a logických hodnot.

Při výkladu o polích jsme si říkali, že v definici vektorového typu se v hranatých závorkách uvádí rozsah přípustných hodnot indexů. Skutečnost je však trochu obecnější – v hranatých závorkách se uvádí typ indexů a interval je (jak již víme) jedním z možných způsobů definice typu.

Z toho tedy plyne, že pokud chceme indexovat prvky nějakého vektoru hodnotami daného výčtového typu, můžeme v indexových závorkách uvést místo nejmenší a největší přípustné hodnoty přímo identifikátor daného výčtového typu.

V následujícím torzu programu najdete příklady deklarací a použití výčtových typů:

```
(*   Příklad P10 – 2   *)
type
  eDen = ( Po, Ut, St, Ct, Pa, So, Ne );
  eBarva = ( Cerny, Bily );
  eFigura = ( Nic, Pesec, Jezdec, Strelec, Vez, Dama, Kral );
  eSloupec = ( a, b, c, d, e, f, g, h );
  eRadek = 1 ..8;
  tDeska = array[ eSloupec ][ eRadek ] of eFigura;

var
  Den : eDen;
  Sachovnice : tDeska;

procedure /*****/ Tah /*****/
( b:eBarva; f:eFigura, s:eSloupec; r:eRadek );
begin
       {Zde by měl být vlastní algoritmus realizující tah. }
end;

begin
  Tah( Bily, Kral, h, 8 );
  {-----------------------------------------}
  for Den:=Po to Pa do
  Pracuj;
  for Den:=So to Ne do
      Odpocivej;
  Den := Succ( Ut );            {Den = St}
  Den := Pred( Ne );            {Den = So}
  write( Ord( Den ) );          {Vytiskne 5 = Ord( So )}
end;
```

V C++ je definice výčtových datových typů o něco univerzálnější, a proto bude i výklad o maličko delší. C++ totiž umožňuje nejen zadat množinu hodnot daného výčtového typu (výčtových literálů), ale umožňuje přiřadit těmto literálům jejich vnitřní reprezentaci.

Výčtové typy definujeme v C++ tak, že napíšeme klíčové slovo **enum**, za ním jméno definovaného typu, dále pak ve složených závorkách seznam čárkami oddělených identifikátorů jednotlivých zaváděných výčtových literálů. Celou definici ukončíme středníkem.

Proměnné výčtových typů můžeme deklarovat a definovat dvěma způsoby:

1. Napíšeme jméno typu a za ním seznam čárkami oddělených identifikátorů proměnných s případnými inicializacemi. (Budeme tomu říkat „klasický“ způsob, i když na něm nic klasického není.)
2. Seznam definovaných proměnných napíšeme mezi složenou závorku uzavírající seznam definovaných literálů a ukončující středník. (Tohle budeme pro změnu označovat za „zrychlenou“ definici.)

V zájmu jednotnosti se však přikláníme k prvnímu způsobu, i když jsou pak zdrojové programy o nějaký ten řádek delší. Tím, že od sebe oddělíme deklaraci typu a deklaraci proměnných, získá program na přehlednosti.

```
/*   Příklad C10 – 1   */
enum eSmer {VYCHOD, SEVER, ZAPAD, JIH };
enum eDen {PO, UT, ST, CT, PA, SO, NE };
enum eBarva {CERNY, BILY } Barva1;                    //Zrychlen
enum eFigura {NIC, PESEC, JEZDEC, STRELEC, VEZ, DAMA, KRAL };
enum eSloupec {a, b, c, d, e, f, g, h };
eBarva Barva2;                                        //Klasicky
```

Vraťme se ale k vlastní definici. Na počátku jsme řekli, že výčtové typy byly zavedeny proto, aby bylo možno lépe pracovat s nějakou množinou v podstatě nenumerických hodnot a aby bylo možno kontrolovat korektnost všech prováděných operací. V profesionální praxi se však často setkáváme se situacemi, kdy sice víme, že ony zpracovávané hodnoty jsou svoji podstatou nenumerické, avšak rámci bychom jejich vnitřní reprezentaci ovlivnili tak, aby nám to v dalším programu co nejvíce usnadnilo práci.

C++ proto umožňuje přiřadit jednotlivým výčtovým literálům jejich vnitřní reprezentaci, a to v celém rozsahu hodnot typu **int**. (Pokud se nám to tedy bude zdát užitečné, můžeme jim přiřadit i záporné hodnoty.) Navíc dovoluje, aby několik výčtových literálů sdílelo tutéž vnitřní reprezentaci – tj. aby měly přiřazenu stejnou hodnotu.

Pokud výše uvedené možnosti přiřazení vnitřní reprezentace nevyužijeme, bude vnitřní reprezentací prvního v seznamu uvedeného literálu číslo nula a u každého dalšího literálu to bude číslo o jedničku větší, než u literálu předchozího.

Pokud za nějakým literálem napíšeme rovnítko následované konstantním celočíselným výrazem vyhodnotitelným v době překladu, bude jeho vnitřní reprezentací hodnota tohoto výrazu. A dále opět platí, že u každého dalšího literálu, který nemá explicitně přiřazenu

svoji vnitřní reprezentaci, bude jeho vnitřní reprezentací číslo o jedničku větší než u literálu předchozího.

```
/*   Příklad C10 – 2   */
//Definice typu "binární řád"
enum eBinRad {NULA, JEDNA, DVE, CTYRI=4, SEDM=7 );
enum ePrachy {BURA=5, PETKA=10, STOVKA=100, KILO=STOVKA, TAC=1000 };
```

Opusťme však nyní definice a podívejme se na použití objektů výčtových typů. Obecně platí, že hodnoty výčtových typů můžeme přiřadit kterékoliv celočíselné proměnné. (To také často potřebujeme a pascalská kontrola typu je zde spíše na obtíž.)

Opačné přiřazení však už není tak snadné. Proměnným výčtových typů totiž můžeme přiřazovat pouze hodnoty jejich typu. Pokud se jim pokusíme přiřadit hodnotu jiného typu, vydá překladač varovné hlášení (není-li ovšem potlačeno). My si pak v programu můžeme na překladačem označených místech zkontrolovat, zda je ona podezřelá operace korektní, nebo zda se jedná o chybu v programu.

Pokud vás tato varovná hlášení obtěžují, máte tři možnosti, jak je potlačit:

1. Potlačit volbu *Assigning 'type' to 'enumeration'* v dialogovém okně *Options* | *Compiler* | *Messages* | *ANSI volations*. Překladač pak žádná varovná hlášení tohoto typu nebude vydávat. Tuto možnost však nedoporučujeme, protože se tím připravujete o možnost typové kontroly.
2. Vůbec daný výčtový typ nezavádět a místo výčtových literálů definovat odpovídající celočíselné konstanty. Toto řešení považuji za ještě horší než předchozí. Programátor se pak musí hlídat sám a to většinou nekončí dobře. Nepřehánějte to se sebedůvěrou a umožněte překladači, aby vás ohlídal všude, kde je to jen trochu možné.
3. Poslední možností je úprava programu do tvaru, který již překladač přijme bez námitek. To je řešení, za které se přimlouváme. Často pomůže pouhé přetypování; nepomůže však při operacích ++, -- nebo **op**=. Tady se budeme muset ještě chvíli smířit s varovnými hlášeními, protože naše dosavadní vědomosti na vyřešení tohoto problému nestačí.

Asi jste si všimli, že všechny seznamy výčtových literálů v doprovodných programech končí literálem, který se jmenuje stejně jako definovaný typ, pouze má na počátku navíc jedno podtržítko. Tento dodatečný identifikátor zavádíme proto, že v něm získáme konstantu, jejíž hodnota je rovna počtu prvků daného výčtového typu. Tuto konstantu pak používáme především v deklaracích vektorů s indexy odpovídajícího výčtového typu. Příklady takového použití najdete v následující ukázce.

```
/*   Příklad C10 – 3   */
enum eDen {PO, UT, ST, CT, PA, SO, NE );
enum eBarva {Cerny, Bily, _eBarva );
enum eFigura{ Nic, Pesec, Jezdec, Strelec, Vez, Dama, Kral );
enum eSloupec{ a, b, c, d, e, f, g, h, _eSloupec );
const NRadek = 8;
```

```
typedef eFigura tDeska[ _eSloupec ][ NRadek ];

eDen Den;
tDeska Sachovnice;

void /*****/ Tah /*****/
( b:eBarva; f:eFigura, s:eSloupec; r:eRadek );
{
  // Zde by měl být vlastní algoritmus realizující tah.
}

void /*****/ main /*****/ ()
{
  Tah( Bily, Kral, h, 8 );
  /*************************************************/
  for( Den=PO; Den <= PA; Den++ ) Pracuj();
  for( Den=SO; Den <= NE; Den++ ) Odpocivej();

  //Následující dva příkazy sice překladač přeloží,
  //avšak vydá varovné hlášení o tom, že proměnné
  //výčtového typu je přiřazena celočíselná hodnota.
  Den = UT+1;                          //Den = ST !
  Den--;                               //Den = UT !!

  cout << Den;                         //Vytiskne 5
}
```

V řádcích, označených vykřičníky v komentáři, ohlásí překladač varování

```
Assigning int to eDen
```

V prvním z nich pomůže přetypování:

```
Den = (eDen)(UT+1);
```

Příklad

Na závěr kapitoly o skalárních datových typech vám dáme úkol. Zkuste napsat program, jehož cílem je simulovat práci semaforů na křižovatce. Program se vás nejprve zeptá, jak dlouho má semafor pracovat a v jakém režimu. Semafory mohou pracovat ve třech režimech:

1. Standardní režim, kdy se na semaforech postupně rozsvěcí červená, oranžová a zelená. Zvolíte-li tento režim, počítač se vás ještě zeptá, jak dlouho má zelená svítit na hlavní a jak dlouho na vedlejší ulici. Při přepínání světel počítač nejprve rozsvítí na dvě sekundy místo zelené oranžovou, po těchto dvou sekundách rozsvítí červenou a v druhém směru nejprve přidá ke svítící červené oranžovou a po jedné vteřině obě zhasne a rozsvítí zelenou. Doba tohoto přepínání se do zadaných dob nepočítá.
2. Blikající oranžová, kdy na semaforech bliká s periodou dvě sekundy (tj. sekundu svítí a sekundu nesvítí) oranžové světlo.
3. Vypnutý semafor, kdy se po zadanou dobu neděje nic.

Trvání jednotlivých operací naprogramujte za pomoci procedury *delay*, jejímž jediným celočíselným parametrem je počet milisekund, po které má počítač počkat. Programátoři v C++ mohou kromě toho použít proceduru *sleep*, které mohou dobu čekání zadávat v sekundách. Abyste mohli tyto procedury používat, musíte v Pascalu dovézt modul *crt* (příkazem **uses**) a v C++ vložit (**#include)** hlavičkový soubor *dos.h*. Řešení najdete opět na doplňkové disketě v souborech *C10–00D.CPP* resp. *P10–00D.PAS*.

11. Podrobnosti o vstupu a výstupu

11.1 Přímý vstup z klávesnice a výstup na obrazovku

Veškerá komunikace našich programů s uživateli probíhala doposud prostřednictvím standardního vstupu a výstupu. Toto řešení s sebou nese řadu výhod, ale také řadu nevýhod. Mezi výhody patří možnost využívání výhodných vlastností přesměrování vstupů a výstupů – např. při tvorbě nejrůznějších filtrů (připomínáme, že filtry jsou programy, které čtou data ze standardního vstupu, nějakým způsobem je transformují a transformovaná je posílají na standardní výstup), mezi nevýhody pak řadíme zejména nedostatečnou tvárnost vstupu z klávesnice.

Pokud se podíváte na profesionální programy, zjistíte, že standardní vstup a výstup používají buď filtry nebo programy, které chtějí přesměrování vstupu a výstupu umožnit, a které bychom mohli klasifikovat jako maskované filtry. Naprostá většina programů však komunikuje s uživatelem přímo.

V této kapitole si vysvětlíme, co musíte udělat, abyste mohli na jedné straně bezprostředně přebírat vstupy zadávané z klávesnice a na druhé straně v zájmu maximálního zrychlení posílat výstupy přímo na obrazovku. Zároveň si povíme, jaké nové možnosti nám tato přímá komunikace nabízí, a ukážeme, jak jich můžeme využít.

Protože podpora přímé komunikace s konzolou (klávesnice + obrazovka) nepatří mezi základní vlastnosti jazyka, vybavili autoři obou probíraných jazyků své systémy moduly, které tuto komunikaci umožňují.

Pascalský modul zprostředkující přímou komunikaci s konzolou se jmenuje *Crt*. Pokud chcete jeho služeb využívat, musíte jej nejprve dovézt prostřednictvím příkaz**uses**:

```
(*   Příklad P11 – 1   *)
Program xxx;
Uses crt;
{...}
begin
  {...}
end.
```

V programu pak již nemusíme provádět žádné další změny. Čteme i nadále příkazem *read* a zapisujeme příkazem *write*, pouze se nám navíc otevřou možnosti řízeného umísťování kurzoru (pro tuto činnost se vžil název **přímá adresace kurzoru**), přímého ovládání barev, podoby kurzoru a dalších funkcí. Kromě toho se všechny výstupy zrychlí, na starších počítačích opravdu viditelně.

Jazyk C, ze kterého C++ vychází, nedoporučuje žádný konkrétní způsob přímé komunikace s konzolou, protože by tím byla výrazně narušena přenositelnost programů (větší počítače totiž často přímou komunikaci s konzolou řadovému uživateli vůbec neumožňují). Každý tvůrce překladače si proto tyto funkce implementoval po svém.

Tvůrci překladače Turbo C zavedli pro tento typ komunikace vlastní systém funkcí, velice podobný systému funkcí pro komunikaci prostřednictvím standardního vstupu a výstupu. Prototypy funkcí pro přímou spolupráci s konzolou najdete v hlavičkovém souboru *conio.h*.

Překladače Borland C++ tento systém samozřejmě zdědily. Jeho použití však není pro programy v C++ optimálním řešením. Jedná se přece jen o systém funkcí vyvinutých pro jazyk C, takže nám nemohou poskytnout výhody objektově orientované definice vstupu a výstupu. A tak přestože nám tyto funkce nabízejí mnohem výkonnější prostředky než jejich pascalské ekvivalenty z modulu *Crt*, univerzálnosti a flexibility datových proudů jazyka C++ nedosahují. (Jsou ovšem situace, kdy je oceníme …)

Abychom stejně jako programátoři v Pascalu nemuseli měnit při přímé komunikaci s konzolou své zvyky, potřebujeme komunikovat prostřednictvím datových proudů. (Modul pro přímý výstup na obrazovku zařadila firma Borland do svých překladačů až od verze 3.0.)

Nyní by již mělo být vše připraveno a můžeme začít. Protože si myslím, že přímý vstup z klávesnice je problémem poněkud palčivějším, začneme nejprve s ním.

Princip spolupráce systému s klávesnicí

Než se pustíme do výkladu o vlastních programových prostředcích pro přímý vstup z klávesnice, měli bychom si nejprve pohovořit o celkové koncepci komunikace programů s klávesnicí.

Pokud vás to v současné chvíli příliš nezajímá, můžete následující pasáž přeskočit, pokračovat oddílem o přímém vstupu z klávesnice a k technickým podrobnostem se vrátit později.

Převzetí a zpracování informace z klávesnice systémem

V počítačích řady IBM PC je klávesnice samostatným zařízením, které s počítačem komunikuje po sériové lince – u PC AT a novějších dokonce obousměrně, takže počítač může nejen zprávy od klávesnice přijímat, ale také jí je posílat. V klávesnici je mikroprocesor, který si pamatuje stav každé klávesy (stisknuta / puštěna) a registruje jeho změny. Informace o těchto změnách, tj. o stisku a puštění jednotlivých kláves, posílá po sériové lince do počítače.

Procesor klávesnice má na starosti i tzv. **autorepeat**, což je opakované vysílání informace o stisku klávesy po celou dobu, po níž je klávesa držena. U počítačů řady AT lze díky obousměrné komunikaci nastavit počáteční prodlevu mezi stiskem klávesy a nastartováním vlastního autorepeatu i frekvenci následně ohlašovaných stisků (počáteční prodlevu od 0,25 do 1 sec, frekvenci od 2 do 30 znaků za sekundu).

Vždy, když je informace z klávesnice zkompletována, vyšle obvod pro komunikaci s klávesnicí (v PC AT je to dokonce samostatný mikroprocesor) žádost o přerušení

(*interrupt*). CPU při první příležitosti přeruší právě prováděný program a předá řízení obsluze přerušení od klávesnice (přerušení č. 9).

Obsluha přerušení od klávesnice přečte z patřičných obvodů informaci dodanou klávesnicí, potvrdí klávesnici její převzetí a nejprve se zeptá, zda má tuto informaci zpracovat (vyvolá přerušení 15h, kam různé programy mohou „věšet“ své požadavky na modifikaci zpracování vstupu z klávesnice). Pokud není žádných námitek, pokračuje ve zpracování převzaté informace.

Nyní se tato procedura podívá, zda se nejedná o stisk nebo puštění klávesy s nějakým speciálním významem (přeřaďovače, PRINT SCREEN, SYSTEMREQUEST), které ihned ošetří, nebo o puštění „obyčejné“ klávesy, které ignoruje.

Pokud se jedná o stisk běžné klávesy, program se podívá do tabulek, kde má zapsány informace o tom, jaký má daná klávesa při aktuálním nastavení přeřaďovačů význam, a z toho odvodí kód, který by bylo třeba předat dále. Tento kód uloží do fronty kódů čekajících na další zpracování a oznámí tuto skutečnost systému tím, že vyvolá přerušení 15h.

Když obsluha přerušení klávesnice ukončí svoji činnost, všechno po sobě uklidí a vrátí řízení zpět původnímu přerušenému programu.

Převzetí kódu od systému programem

Jak jsme si řekli v předchozí pasáži, obsluha přerušení od klávesnice zjistí, jakou informaci uživatel z klávesnice poslal, a uloží její kód do fronty. Do této fronty se za normálních okolností vejde 15 kódů. V této frontě kódy čekají, až bude chtít právě probíhající program přečíst nějaký znak z klávesnice.

Když chce váš program přečíst znak z klávesnice, zavolá správce výše zmíněné fronty znaků, kterým je obsluha přerušení 16h. Pokud tento správce zjistí, že ve frontě znaků již nějaký kód čeká, odebere jej z fronty a předá volajícímu programu.

Pokud zjistí, že fronta je prázdná, rozhodne se počkat, než mu obsluha přerušení od klávesnice nějaký kód do fronty připraví. Aby však nemusel čekat celý počítač, oznámí nejprve systému (klasicky – vyvoláním přerušení 15h), že se bude čekat na stisk klávesy, takže je možno spustit nějaký program, který běží v pozadí, a který se opět přeruší ve chvíli, kdy se ve frontě objeví nějaký znak.

Jakmile se ve frontě objeví nějaký kód, správce fronty si jej vyzvedne a předá programu, který o něj požádal.

Kódy, které se do fronty ukládají, jsou dvoubajtové a dělí se do tří skupin:

1. Skupina kódů, které mají spodní bajt nenulový (obsahuje kód zadaného znaku). Horní bajt kódů z této skupiny obsahuje tzv. **polohový kód** (*scan code*) stisknuté klávesy, což není nic jiného než pořadové číslo dané klávesy na klávesnici. Do této skupiny patří kódy všech zobrazovaných znaků včetně semigrafických a kódy základních řídicích znaků vyvolávaných při stisknutém přeřaďovačiCTRL.
2. Skupina kódů, které mají spodní bajt nulový. Horní bajt kódů z této skupiny obsahuje kód funkce dané klávesy. Do této skupiny patří kódy funkčních kláves, kláves kurzo-

rového a editačního pole a také kódy oznamující stisk dané klávesy při stisknutém přeřaďovači ALT. Jejich přehled najdete v tabulce 11.1.

3. Kód vysílaný při zadávání řídicího znaku CTRL-@, který by sice v podstatě patřil do první skupiny (patří mezi základní řídicí znaky), ale protože jeho kód je nulový, má nulový spodní bajt a musí se zpracovávat stejně jako znaky z druhé skupiny.

Přímý vstup z klávesnice

Nyní už tedy víme, že přímé čtení znaků z klávesnice vlastně příliš přímé není – mezi námi a klávesnicí je obsluha ošetření přerušení od klávesnice a správce fronty čekajících kódů. Přímou se tato komunikace nazývá proto, že se při ní již za tyto dva zprostředkovatele nezařazuje ještě operační systém, který mnohé z vyslaných kódů spolkne (např. kódy editačních kláves), takže je váš program nemá možnost využívat.

Při přímé komunikaci s klávesnicí budeme ve svých programech používat zejména dvě funkce:

- ✧ Funkci pro test, zda byla stisknuta nějaká klávesa (v Pascalu funkce *keypressed*, v C++ funkce *kbhit()*), která bude vracet logickou hodnotu v případě, že ve frontě čeká nějaký znak na zpracování, a logickou hodnotu *NE* v případě, že je fronta prázdná. Nezapomeňte, že tento znak je třeba z fronty také převzít – viz následující odstavec.
- ✧ Funkci pro převzetí znaku (přesněji části kódu) zadaného z klávesnice – přesněji funkce pro převzetí kódu čekajícího ve frontě (v Pascalu funkce *readkey*, v C++ *getch()*). Tyto funkce vracejí hodnotu spodního bajtu tohoto kódu, tj. u kódů z prvé skupiny (zobrazitelné znaky a základní řídicí znaky s výjimkou znaku CTRL-@) kód daného znaku a u ostatních kódů nulu.

Pokud vám tyto funkce vrátí nulu, musíte je zavolat ještě jednou a ony vám pak vrátí obsah horního bajtu daného kódu, v němž je, jak jsme si řekli, kód funkce dané klávesy.

C++ nabízí v *conio.h* kromě funkce *getch()* ještě funkci *getche()*, která se liší pouze tím, že přečtený znak zároveň vypíše na obrazovku. Přesněji: nevypíše přečtený znak, ale znak, jehož kódem je vracené číslo. Pokud tedy zadáme na klávesnici např. šipku vzhůru, pošle *getche()* na obrazovku nejprve znak s kódem 0 (*NULL*, CTRL-@) a při druhém volání znak s kódem 72, tj. písmeno *H*.

Jak vidíte, přímé přebírání znaků z klávesnice je spojeno s jistými komplikacemi. Jednou z možností, jak se jim do jisté míry vyhnout, je definovat si vlastní transformační funkci, která bude kódy znaků z klávesnice transformovat na celá čísla ležící v rozsahu <0; 511>. Tato funkce by mohla být definována v Pascalu takto:

```
(*   Příklad P11 – 2   *)
function (*****) Klavesa (*****)
: integer;
var Kl : char;

begin
```

```
  Kl := ReadKey;
  if( Kl = #0 )then
  begin
       Klavesa := 256 + integer( ReadKey );
  end
  else
       Klavesa := integer( Kl );
end;
```

Její obdoba v C++ může mít tvar

```
void /*****/ Klavesa /*****/
()
{
  int i = getch();
  if( i )
       return( i );
  else
       return( 256 + getch() );
}
```

Prozatím jsme si ukazovali, jak lze z klávesnice přímo získávat jednotlivé znaky. Pokud budete chtít přebírat čísla nebo řetězce, bude situace složitější, protože nemáte k dispozici žádný editor zadávaného textu. V zájmu maximálního komfortu obsluhy se proto používají různé dodatečně definované procedury a funkce, které takovýto komfortní editovaný vstup z klávesnice zprostředkují.

Kód	Funkční klávesa	Kód	Funkční klávesa
1	ALT - ESC	117	CTRL - END
2	nepřiřazeno	118	CTRL - PAGE Down
3	CTRL-@ (Znak s kódem 0)	119	CTRL - HOME
4 – 13	nepřiřazeno	120 – 131	ALT - 1, 2, 3, 4, 5, 6, 7, 8, 9, =[19]
14	ALT - BACKSPACE	132	CTRL - PAGE UP
15	SHIFT - TAB	133 – 134	F11, F12
16 – 25	ALT - Q, W, E, E, R, T, Y, U, I, O, P	135 – 136	SHIFT - F11, SHIFT - F12
26 – 28	ALT - [,]	137 – 138	CTRL - F11, CTRL - F12
29 – 30	nepřiřazeno	139 – 140	ALT - F11, ALT - F12
30 – 38	ALT - A, S, D, F, G, H, J, K, L	141	CTRL - ŠIPKA NAHORU/8

[19] Na US klávesnici jsou na posledních dvou klávesách znaky „-“ a „=“.

Kód	Funkční klávesa	Kód	Funkční klávesa
39 – 41	ALT - , . MINUS[20]	142	CTRL - ŠEDÉ MINUS
42 – 54	nepřiřazeno	143	CTRL - CENTRÁLNÍ TLAČÍTKO/5
55	ALT - ŠEDÁ * (Klávesa v numerickém poli)	144	CTRL - ŠEDÉ +
56 – 58	nepřiřazeno	145	CTRL - ŠIPKA DOLŮ/2
59 – 68	F1 až F10 (Funkční klávesy)	146	CTRL - INS / 0
69 – 70	nepřiřazeno	147	CTRL - DEL / .
71	HOME	148	CTRL - TAB
72	ŠIPKA NAHORU	149	CTRL - ŠEDÉ /
73	PAGE UP	150	CTRL - ŠEDÁ *
74	ALT - ŠEDÉ MINUS (Klávesa v numerickém poli)	151	ALT - HOME (Pouze klávesa v edit. poli)
75	Šipka vlevo	152	ALT - ŠIPKA NAHORU (Pouze klávesa v edit. poli)
76	CENTRÁLNÍ TLAČÍTKO[21]	153	ALT - PAGE UP (Pouze klávesa v edit. poli)
77	ŠIPKA VPRAVO	154	nepřiřazeno
78	ALT - ŠEDÉ + (Klávesa v numerickém poli)	155	ALT - ŠIPKA VLEVO (Pouze klávesa v edit. poli)
79	END	156	nepřiřazeno
80	ŠIPKA DOLŮ	157	ALT - ŠIPKA VPRAVO (Pouze klávesa v edit. poli)
81	PAGE DOWN	158	nepřiřazeno
82	INS	159	ALT - END (Pouze klávesa v edit. poli)
83	DEL	160	ALT - ŠIPKA DOLŮ (Pouze klávesa v edit. poli)
84 – 93	SHIFT - F1 až SHIFT - F10	161	ALT - PAGE DOWN (Pouze klávesa v edit. poli)
94 – 103	CTRL - F1 až CTRL - F10	162	ALT - INS (Pouze klávesa v edit. poli)

[20] Na US klávesnici je zde znak „/“.
[21] „Kurzorový“ význam tlačítka s číslicí 5 v numerickém poli.

Kód	Funkční klávesa	Kód	Funkční klávesa
104 – 113	ALT - F1 až ALT - F10	163	ALT - DEL (Pouze klávesa v edit. poli)
114	CTRL - PRINT SCREEN	164	ALT - ŠEDÉ /
115	CTRL - ŠIPKA VLEVO	165	ALT - TAB
116	CTRL - ŠIPKA VPRAVO	166	ALT - ENTER

Tab. 11.1 Kódy funkčních kláves podle dokumentace k IBM PC XT/AT

Přímý výstup na obrazovku

Přímý výstup na obrazovku je záležitostí mnohem jednodušší. Používáme ho za prvé kvůli jeho rychlosti a za druhé kvůli dalším funkcím, jako je přímá adresace kurzoru, možnost ovlivňování barev, možnost ovlivňování zobrazovacího režimu, práce s okny apod. Některé z těchto možností budeme sice mít i při výstupu na obrazovku prostřednictvím standardního výstupu, ale musíme mít instalován ovladač *ANSI.SYS*, a i pak bude ovládání daných funkcí daleko těžkopádnější.

V úvodu kapitoly jsme si řekli, co máme dělat, abychom své výstupy posílali přímo na obrazovku. Pokud tedy v Pascalu dovezeme modul *Crt*, můžeme proceduru *write* používat nadále stejně jako dosud a starost o to, že výstupy jdou přímo na obrazovku, můžeme přenechat systému.

V C++ musíme použít hlavičkový soubor *constrea.h*, který obsahuje prostředky pro práci s proudy, orientovanými na konzolu. (Tento soubor se postará i o vložení souboru *iostream.h*, takže budeme mít k disposici i všechny prostředky, které jsme měli dosud.) Dále si musíme deklarovat potřebný proud. Nazveme jej třeba *crt*:

```
constream crt;
```

Data, určená k přímému výstupu na obrazovku, budeme nyní posílat do proudu *crt*. Na rozdíl od Pascalu vám však zůstává standardní výstup i nadále otevřen. (V Pascalu sice můžete také používat oba najednou, ale musíte pro to nejprve něco udělat – co, to si povíme později.)

Abyste si mohli sami vyzkoušet možnosti přímé komunikace s klávesnicí (a částečně i s obrazovkou), nabízíme vám následující úlohu: Napište program, který bude sloužit jako test reakční pohotovosti. Program oznámí testované osobě číslo, při jehož dosažení má stisknout jakoukoliv klávesu a tím program zastavit. Po tomto oznámení počká na stisk klávesy, kterým testovaná osoba ohlásí svoji připravenost. Jakmile je klávesa stisknuta, program zobrazí číslo 0000, které rychle po jedné zvyšuje. Ve chvíli, kdy je klávesa stisknuta podruhé, se počítání zastaví, oznámí testované osobě, o kolik se spletla, a zeptá se, zda chce test zkusit ještě jednou. Pokud osoba pošle znak *a* nebo *A*, spustí celý test znovu, v opačné případě program svoji činnost ukončí.

Program byste sice mohli řešit tak, že budete prostě tisknout napočítávaná čísla, ale to byste mohli testovat reakční schopnosti pouze velice hrubě, protože čísla by se měnila

příliš pomalu, neboť při tisku čísel se musí provádět řada násobení a dělení, které práci počítače zdržují.

Výhodnější a hlavně rychlejší řešení je tisknout číslo po jednotlivých řádech tak, jak to dělá náš program. Program mění postupně číslice v jednotlivých řádech: když se vystřídají všechny číslice v řádu jednotek, vrátí se na řád desítek, vytiskne zde další číslici a znovu vytiskne na řádu jednotek postupně všechny číslice. Když pak již vyčerpá všechny číslice i na řádu desítek, vrátí se k řádu stovek, tam zvětší číslici o jednu a celý postup opakuje.

Abyste mohli tento algoritmus naprogramovat, musíte vědět, že znak s kódem 8 je tzv. BACKSPACE, který vrátí kurzor o jednu pozici zpět (tento znak funguje i při použití standardního výstupu).

Možná řešení tohoto úkolu najdete na doplňkové disketě v souborech *P11–00A.PAS* a *C11–00A.CPP*. Program je řešen dvěma způsoby: prostřednictvím vnořených cyklů a rekurzí. Řešení prostřednictvím vnořených cyklů vám bude asi připadat pochopitelnější, ale jak sami vidíte, jedná se o řešení „na tvrdo“. Naproti tomu rekurzivnímu řešení je úplně jedno, zda se bude čítat do deseti, do deseti tisíc nebo do deseti miliónů.

V literatuře se většinou dočtete, že rekurzivní řešení problémů bývají sice kratší a elegantnější, ale na druhou stranu pomalejší. V tomto případě to neplatí, protože při testech vycházela obě řešení jako stejně rychlá.

11.2 Ovládání výstupu na obrazovku

V tomto oddílu si ukážeme několik funkcí, které nám umožní mnohem efektnější výstupy na obrazovku, než jaké jsme uměli doposud.

Abyste mohli níže popisované funkce ve svých programech používat, musíte v Pascalu dovézt prostřednictvím **uses** modul *Crt* a v C++ je potřeba pomocí **#include** vložit hlavičkový soubor *constrea.h* – jak je ostatně dále naznačeno v definicích prototypů popisovaných funkcí.

Textové okno

Textové okno (*text window*) je obdélníková oblast obrazovky, do níž směřuje výstup na obrazovku. Okno se chová, jako by to byla celá obrazovka: jestliže se kurzor dostane k pravému okraji okna, přesune se na začátek následujícího řádku, a dostane-li se kurzor na dolní okraj okna, posune se při přechodu na nový řádek celý obsah okna (ale jenom okna) o řádek vzhůru, přičemž první řádek se ztratí.

Toto ale bude fungovat pouze tehdy, budeme-li používat přímý výstup na obrazovku. V Pascalu stačí importovat modul *Crt* a veškerý výstup bude už prováděn přímo. V C++ je třeba důsledně používat pro výstup proud *crt*. (Můžete si jej pojmenovat i jinak, ale musí to být objekt typu *constream*; v dalším textu budeme ale předpokládat, že se jmenuje *crt*.) Standardní výstupní proud *cout* nebere na definovaná okna žádný ohled.

Textové okno definujeme („otevíráme“) v Pascalu funkcí

```
procedure window( lhx, lhy, pdx, pdy : byte );
```

a v C++ pomocí funkce[22]

```
void crt.window( int lhx, int lhy, int pdx, int pdy );
```

Parametry *lhx*, *lhy* představují souřadnice *x* a *y* levého horního rohu okna (tj. sloupec a řádek) a *pdx*, *pdy* představují souřadnice *x* a *y* pravého dolního rohu. Souřadnicemi se míní pozice na obrazovce, přičemž levý horní roh obrazovky má souřadnice [1, 1] a pravý dolní zpravidla [80, 25]. Jestliže zadáme nějaké nesmyslné souřadnice, bude volání funkce *window* ignorováno.

Otevřením okna se nijak nezmění ani neporuší dosavadní obsah obrazovky. Okno tedy není nějaký grafický objekt, ale pouze předpis pro chování výstupu. Po otevření okna se kurzor umístí do jeho levého horního rohu.

Při spuštění programu se automaticky definuje základní okno – celá obrazovka.

Pozice kurzoru

V našich programech jsme zatím nedokázali výrazněji ovlivnit, kde na obrazovce se náš výstup objeví. Dosud jsme pokračovali vždy tam, kde jsme posledně skončili, a dokázali jsme řídit pouze přechod na novou řádku.

Změnit pozici kurzoru, tedy místa, kam se bude zapisovat, nám v Pascalu umožní funkce:

```
procedure gotoxy( x, y : byte );
```

a v C++ pomocí manipulátoru s parametrem

```
setxy( int x, int y )
```

který vložíme do proudu, např. takto:

```
crt.window(2, 2, 10, 20);
crt << setxy(2,2) << "Ahoj";
```

Parametry *x*, *y* určují souřadnice sloupce a řádku, kam bude kurzor umístěn. Pozor! Tyto souřadnice udávají pozici v naposled definovaném (tedy aktivním) okně, nikoli na celé obrazovce. I zde platí, že souřadnice [1,1] určují levý horní roh okna.

Jestliže jsme v programu před přemístěním kurzoru pomocí funkce *gotoxy* resp. manipulátoru *setxy()* nikde nepoužili funkci *window*, pracuje se s celou obrazovkou.

Pokud zadáme souřadnice mimo definované okno, bude volání *gotoxy* resp. použití *setxy()* ignorováno.

Současnou pozici kurzoru **v aktuálním okně** můžeme v Pascalu zjistit funkcemi

[22] Jméno funkce je připojeno tečkou ke jménu proudu; s tímto způsobem zápisu jsme se již setkali u funkce *cin.get()*.

```
function wherex : byte;
function wherey : byte;
```

V C++ je zápis poněkud složitější, aleznamená totéž:

```
int crt.rdbuf()->wherex();
int crt.rdbuf()->wherey();
```

wherex vrací číslo sloupce (souřadnici *x*), *wherey* číslo řádku (souřadnici *y*).

Práce s obsahem okna

Už tedy víme, jak definovat okno a jak do něj posílat výstup na místo, které chceme. Občas ale také potřebujeme to, co jsme na obrazovku zapsali, opět vymazat. Nejradikálnější postup nabízejí procedury

```
procedure clrscr;
```

a

```
void crt.clrscr();
```

Procedura *clrscr*[23] smaže veškerý obsah aktuálního okna (vyplní je aktuální barvou pozadí) a umístí kurzor do jeho levého horního rohu.

Potřebujeme-li smazat řádek, ve kterém je právě kurzor, použijeme v Pascalu funkci

```
procedure delline;
```

a v C++ vložíme do proudu *crt* manipulátor bez parametrů *delline* (*delete line*). Text řádku, na němž se nachází kurzor, se vymaže a celý obsah okna pod tímto řádkem se posune nahoru. Jedná-li se o poslední řádek v okně, pouze se smaže.

Jestliže naopak potřebujeme vložit řádek, použijeme v Pascalu funkci

```
procedure insline;
```

a v C++ vložíme do proudu *crt* manipulátor bez parametrů *insline* (zkratka slov *insert line*). Tím vytvoříme na místě, kde se nachází kurzor, prázdný řádek. Text pod ním se posune dolů. Text nacházející se těsně nad dolním okrajem okna bude umazán.

Procedura

```
procedure clreol;
```

v Pascalu a manipulátor bez parametrů *clreol* v C++ vymažou znaky od pozice kurzoru až do konce řádku, a to včetně znaku, na kterém je kurzor umístěn. Pozice kurzoru se přitom nezmění.

---

[23] Její název vznikl jako zkratka ze slov *clear screen*, vyčisti obrazovku.

Barvy

Výstup bude přehlednější, jestliže informace rozlišíme nejen jejich umístěním na displeji, ale také vhodným barevným podáním. Informace o nastavení barevných atributů se v počítačích řady PC vyjadřuje jedním bajtem, jehož jednotlivé bity mají přidělený význam. Struktura této stavové slabiky je na následujícím obrázku 11.1.

Bity **ppp** určují barvu popředí, tj. barvu znaků, které na obrazovku zapisujeme. Tři bity nám poskytují možnost kódovat 8 barev. V původních grafických kartách CGA měl každý bit přiřazenu jednu barvu (rgb – *red* (červená), *green* (zelená), *blue* (modrá)). Pozdější adaptery (EGA, VGA, a další) sice umožňovaly barvy všelijak kódovat, nicméně většinou se používá původní paleta osmi barev, na něž v obou jazycích pamatují i předdefinované konstanty (viz tabulka 11.2).

Obr. 11.1 Uložení informací o barvách v textovém režimu

Čtvrtý bit označený **i** má význam atributu intenzity (přesněji měl jej na CGA, ale většinou se tak používá i nadále). Každá z výše uvedených osmi barev tedy měla svůj světlejší a tmavší odstín. Na barevných monitorech tedy například hodnota spodních čtyř bitů 2 (binárně 0010) znamená zelenou a hodnota 10 (binárně 1010) světle zelenou. Na monochromatických monitorech určuje tento intenzitní bit normální resp. vysoký jas zobrazovaných znaků.

Další trojice bitů (**zzz**, tj. bity 4 – 6) určuje barvu pozadí, na které budou znaky tisknuty. Pro barvy pozadí platí totéž, co jsme si řekli o základní osmici barev popředí.

Sedmý bit obsahuje atribut blikání. Je-li nastaven, zobrazí se pozadí (patřičnou barvou) a na něm bude (v barvě popředí) blikat znak.

Pascal	C++	Hodnota	Význam
Black	*BLACK*	0	černá
Blue	*BLUE*	1	modrá
Green	*GREEN*	2	zelená
Cyan	*CYAN*	3	tyrkysová
Red	*RED*	4	červená
Magenta	*MAGENTA*	5	fialová
Brown	*BROWN*	6	hnědá
LightGray	*LIGHTGRAY*	7	světle šedá
DarkGray	*DARKGRAY*	8	*tmavě šedá
LightBlue	*LIGHTBLUE*	9	*světle modrá

Pascal	C++	Hodnota	Význam
LightGreen	*LIGHTGREEN*	10	*světle zelená
LightCyan	*LIGHTCYAN*	11	*světle tyrkysová
LightRed	*LIGHTRED*	12	*světle červená
LightMagenta	*LIGHTMAGENTA*	13	*růžová
Yellow	*YELLOW*	14	*žlutá
White	*WHITE*	15	*bílá
Blink	*BLINK*	128	*blikání

Tab. 11.2 Předdefinované „barevné konstanty“; hodnoty označené * jsou použitdné pouze pro popředí

Na monochromatických displejích, které barvy nezobrazují (třeba Hercules), přísluší některým kombinacím nastavení stavových bitů různé jiné efekty – například tučné nebo podtržené písmo. Přehled těchto efektů spolu s příslušnými hodnotami stavové slabiky najdete v tabulce 11.3. V tabulce 11.4 najdete význam těchto atributů pro černobílé monitory připojené na některý barevný grafický adaptér.

Atribut	Význam
01H	podtržené
07H	normální (bílé na černém)
09H	jasné podtržené
0fH	tučné (jasně bílé na černém)
70H	inverzní (černé na bílém)
81H	blikající podtržené
87H	blikající normální
89H	blikající jasné podtržené
8fH	blikající tučné

Tab. 11.3 Význam barevných atributů na monochromatických monitorech TTL (kompatibilních s monitorem Hercules)

Atribut	Význam
07H	normální (bílé na černém)
08H	šedé na černém
0fH	tučné (jasně bílé na černém)
70H	inverzní (černé na bílém)
78H	šedivé na bílém
7fH	jasné bílé na bílém
87H	blikající normální
8fH	blikající tučné

Tab. 11.4 Textové atributy na černobílých monitorech připojených na některý barevný adaptér

Kdesi v paměti počítače je tedy umístěn bajt s parametry barev a podle něho se řídí výstup. Existují funkce, které nám umožní nastavení těchto parametrů změnit. Tyto funkce mají jednu společnou vlastnost: nijak neovlivňují zobrazení textu, který už byl na obrazovku zapsán před jejich voláním. Jestliže parametry nějak nastavíme, ovlivní toto nastavení až následující výstup.

Jednoduchou možnost zvýraznění textu poskytují v Pascalu procedury

```
procedure highvideo; {Pascal – Crt}
procedure lowvideo;
```

a v C++ manipulátory *highvideo* a *lowvideo*, přičemž *highvideo* nastavuje bit intenzity (při výstupu se bude text zobrazovat s vysokým jasem) a *lowvideo* tento bit „shodí“ – nastaví normální jas.

Chceme-li nastavit barvu textu nebo barvu pozadí, použijeme v Pascalu procedury

```
procedure textcolor( barva : byte );
procedure textbackground( barva: byte );
```

a v C++ manipulátory *setclr(int)* nebo *setbk(int)*.

Procedura *textcolor* a manipulátor *setclr()* nastavují barvu popředí (dolní čtyři bity stavové slabiky). Parametr *barva* může nabývat hodnoty v rozmezí 0 až 15 a můžeme pro něj použít předdefinované konstanty z tabulky 11.2; *textcolor* také umožňuje nastavit parametr blikání, a to tak, že k hodnotě zvolené barvy přičteme konstantu *Blink* (Pascal) resp. *BLINK* (C++). Například příkaz

```
textcolor( Red + Blink );                    {Pascal}
```

resp.

```
crt << setclr( RED + BLINK );                //C++
```

způsobí, že vystupující text bude blikat a bude mít červenou barvu.

Funkce *textbackground* a manipulátor *setbk()* nastavují barvu pozadí (bity 4 – 6 stavové slabiky). Pro pozadí můžeme použít pouze barvy o hodnotách 0 až 7, tj. pouze prvních osm konstant z tabulky 11.2.

Pokud chceme zároveň nastavit barvy pozadí i popředí, získáme potřebnou hodnotu tak, že požadovanou barvu pozadí vynásobíme 16 (posuneme o 4 bity vlevo) a přičteme k zadávané barvě popředí. Tuto hodnotu přiřadíme v Pascalu proměnné

```
var TextAttr : byte; {Pascal – Crt}
```

kdežto v C++ použijeme manipulátor *setattr(int NovyAtribut)*. Červeného blikajícího textu na modrém pozadí tedy v Pascalu dosáhneme přiřazením

```
TextAttr := (BLUE shl 4) + LIGHTRED + BLINK;
```

a v C++ příkazem

```
crt << setattr( (BLUE << 4) + LIGHTRED + BLINK );
```

K nastavení „normálních“ atributů slouží v Pascalu procedura

```
procedure normvideo;
```

a v C++ manipulátor *normvideo* (bez parametrů). Manipulátor *normvideo* nastaví takovou barvu popředí i pozadí, jakou mělo při spuštění programu políčko, na němž se právě nacházel kurzor.

Ovládání zvuku

Počítače řady PC ještě v nedávné době nedisponovaly žádnými úchvatnými možnostmi generování zvuku. Standardním vybavením byl pouze jednotónový zvukový generátor napojený na malý reproduktor. Tyto skromné technické prostředky umožňovaly vyluzování různých pípání nebo krátkých melodií.

V Pascalu získáme přístup k funkcím ovládajícím zvuk dovezením známého modulu *Crt*. V C++ si je zpřístupníme vložením hlavičkového souboru *dos.h*.

Zvukový generátor zapojí procedury

```
procedure sound( frekvence : word );  {Pascal}
```

resp.

```
void sound( unsigned frekvence );     //C++ – dos.h
```

Parametrem je frekvence tónu v hertzích. Připomeňme si, že lidské ucho je schopno vnímat zvuk v rozsahu frekvencí přibližně 16 – 16 000 Hz, přičemž z reproduktorků počítače uslyšíte nejintenzivněji tóny okolo 2 KHz. Jestliže zavoláme funkci *sound*, bude příslušný zvuk znít nepřetržitě (na pozadí běhu programu), dokud nějakou jinou funkcí status zvukového generátoru nezměníme.

Znějící zvuk vypneme zavoláním procedury

```
procedure nosound;                    {Pascal}
```

resp.

```
void nosound();                       //C++
```

Jestliže chceme „zahrát“ několik po sobě následujících tónů, musíme mít možnost nastavit trvání tónu na nějakou přesně danou dobu. To nám umožní procedury

```
procedure delay( ms : word );         {Pascal – Crt}
```

a

```
void delay( unsigned ms );                  //C++ – dos.h
```

které zastaví běh programu na *ms* milisekund. Dvojnásobné pípnutí můžeme tedy naprogramovat například takto (kód je v Pascalu, ale céčkaři ho jistě bez problémů přečtou):

```
soud( 1000 );
delay( 100 );
nosound;
delay( 200 );
sound( 1000 );
delay( 100 );
nosound;
```

Nakonec ještě jednu poznámku: Používání zvukových signálů při komunikaci s uživatelem může používaní vašeho programu zpříjemnit, nebo také právě naopak. Mějte na paměti, že většina lidí nemá na svém počítači možnost ovládat hlasitost zvukového výstupu nebo ho dokonce vypnout. Každý váš program, který pípá, hraje a podobně, by proto měl

dát uživateli možnost tyto zvukové efekty vypnout. Jinak je totiž uživatel proti zvuku vyluzovanému vaším programem naprosto bezbranný. Leda, že by vzal páječku a odpojil ve svém počítači reproduktor od vodičů.

Látku této kapitoly si zkuste procvičit na jednoduchoučkém prográmku, který na počátku umístí do středu prázdné obrazovky kolečko (písmeno O) a bude pípat tónem o frekvenci 1000 Hz v rytmu 0,5 sec zvuk, 0,5 sec ticho atd. Program bude přebírat vstup z klávesnice a čekat na stisk kurzorových šipek. Kurzorovými šipkami budeme ovládat jak posun kolečka po obrazovce, tak i výšku a rytmus pípání. Šipka vzhůru posouvá kolečko vzhůru a zvyšuje frekvenci tónu, šipka dolů je posouvá dolů a frekvenci tónu snižuje. Šipka vpravo posouvá kolečko vpravo a rytmus pípání zrychluje, šipka vlevo posouvá kolečko vlevo a rytmus pípání zpomaluje. Řešení této úlohy v Pascalu může být:

```
(*   Příklad P11 – 3   *)
{Hlasité cestování}
Program Pascal;
Uses crt;

(***********  Lokální objekty modulu  ***********)
const
  ZNAK = 'O';         {Cestující znak}

  MAX_X = 80;         {Předpokládáme obrazovku 80 x 25}
  MAX_Y = 25;         {Časem se naučíme zjišťovat skutečný formát}

  POC_TON = 1000;     {Frekvence počátečního tónu v Hz}
  POC_DEL = 50;       {Počáteční délka zvuku / ticha v milisekundách}

  {Výšky i délky tónu nebudou tvořit aritmetickou řadu (tj. sousední
   členy nebudou mít konstantní rozdíl), ale geometrickou řadu
   (sousední členy budou mít konstantní podíl) }
  KTON = 1.1;         {Zvýšení tónu při změně řádku}
  KDEL = 1.05;        {Prodloužení délky zvuku/ticha při změně sloupce}

  Esc = #27;          {Kód klávesy Escape}
  SNa = #72;          {Kód šipky nahoru}
  SLe = #75;          {Kód šipky vlevo}
  SPr = #77;          {Kód šipky vpravo}
  SDo = #80;          {Kód šipky dolů}

var
  c     : char;       {Přečtený kód}
  x     : integer;    {Souřadnice zobrazovaného znaku}
  y     : integer;
  nx    : integer;    {Nové souřadnice zobrazovaného znaku}
  ny    : integer;
  Ton   : real;       {Výška tonu v Hz}
  Delka : real;       {Délka tónu a mezery v ms}
  t     : integer;    {Pomocné celočíselné proměnné pro zadávání}
  d     : integer;    {Výšky a délky tónu}

(********************  Hlavní program¶  ********************)
begin
  c     := char( 1 );{Aby si nemyslel, že mohla být stisknuta šipka}
  x     := MAX_X div 2;    {Souřadnice středu obrazovky = počáteční}
```

```
y     := MAX_Y div 2;    {souřadnice cestujícího znaku}
nx    := x;              {Na    počátku    jsou    nové    souřadnice
cestujícího}
ny    := y;              {znaku stejné jako staré}
Ton   := POC_TON;
Delka := POC_DEL;

clrscr;                  {Smažeme obrazovku}
gotoxy( x, y );          {a do jejího středu umístíme cestující znak}
write( ZNAK );
repeat                   {Dokud cyklus neukončíme stiskem Esc}
    if( c = char( 0 ) ) {Před vlastním kódem šipky musí předcházet
                         nula;}
                        {pokud nepředchází, nemohla to být šipka}
    then begin          {Vlastní kód je již připraven ve frontě}
        c := ReadKey;   {Čteme vlastní kód}
        case( c )of
        SNa:                 {Šipka nahoru - zvyšujeme tón}
             if( y > 1 )
             then begin
                  ny  := y - 1;
                  Ton := Ton * KTON;
             end;

        SLe:                 {Šipka vlevo - zkracujeme tón}
             if( x > 1 )
             then begin
                  nx    := x - 1;
                  Delka := Delka / KDEL;
             end;

        SPr:                 {Šipka vpravo - prodlužujeme tón}
             if( x < MAX_X )
             then begin
                  nx    := x + 1;
                  Delka := Delka * KDEL;
             end;

        SDo:                 {Šipka dolů - snižujeme tón}
             if( y < MAX_Y )
             then begin
                  ny  := y + 1;
                  Ton := Ton / KTON;
             end;

        else
             c := char( 0 ); {Abychom poznali, že to nebyla šipka}
        end;
    end;
    if( c <> char( 0 ) )     {Byla to šipka}
    then begin
        gotoxy( x, y );      {Smazání znaku na původní poloze -}
        write( ' ' );        {přepíšeme jej mezerou}
        x := nx;             {Nastavení nové polohy znaku}
        y := ny;
        gotoxy( x, y );
        write( ZNAK );       {Vytištění znaku v nové pozici}
```

```
      end;
      t := round( Ton );        {Aby se v následujícím cyklu nemusely}
      d := round( Delka );      {reálné hodnoty pořád přepočítávat}
                                {na celé}
      repeat                    {Pípáme až do stisku nějaké klávesy}
          sound( t );           {Tón}
          delay( d );           {d milisekund}
          nosound;              {Ticho}
          delay( d );           {d milisekund}
      until( KeyPressed );      {Pípání přerušíme stiskem klávesy}
      c := ReadKey;             {Zjistíme, co to bylo za klávesu}
  until( c = Esc );             {Stisk klávesy Esc ukončuje program}
end.
```

V C++ bude tento program vypadat následovně:

```
/*   Příklad C11 – 1   */ ¶
//Hlasité cestování
#include <constrea.h>
#include <dos.h>

constream crt;              // Deklarace konzolového proudu crt

/***********  Lokální objekty modulu  ***********/
const char ZNAK = 'O';      //Cestující znak;

const MAX_X = 80;           //Předpokládáme obrazovku 80 x 25
const MAX_Y = 25;           //Časem se naučíme zjišťovat skutečný formát

const POC_TON = 1000;       //Frekvence počátečního tónu v Hz
const POC_DEL = 50;         //Počáteční     délka     zvuku/ticha     v
  milisekundách

//Výšky i délky tónu nebudou tvořit aritmetickou řadu (tj. sousední
//členy  nebudou  mít  konstantní  rozdíl),  ale  geometrickou  řadu
  (sousední
//členy budou mít konstantní podíl)

const double KTON = 1.1;    //Zvýšení tónu při změně řádku
const double KDEL = 1.05;   //Prodloužení zvuku/ticha při změně sloupce

const Esc = 27;             //Kód klávesy Escape
const SNa = 72;             //Kód šipky nahoru
const SLe = 75;             //Kód šipky vlevo
const SPr = 77;             //Kód šipky vpravo
const SDo = 80;             //Kód šipky dolů

/******************  Hlavní program  ******************/
void /*****/  main  /*****/ ()
{
  char c = 1;               //Přečtený tón - počáteční hodnotu
                            //přiřazujeme proto, aby si hned napoprvé
                            //nemyslel, že mohla být stisknuta šipka
  int  x = MAX_X / 2;       //Souřadnice středu obrazovky = počáteční
  int  y = MAX_Y / 2;       //souřadnice cestujícího znaku
  int nx = x;               //Nové souřadnice cestujícího znaku -
  int ny = y;               //na počátku jsou stejné jsko staré
  double Ton   = POC_TON;   //Výška tónu
```

```
double Delka = POC_DEL;  //Délka tónu

crt.clrscr();            //Smažeme obrazovku
//gotoxy( x, y );        //a do jejího středu umístíme cestující znak
crt << setxy( x, y) << ZNAK;
do                       //Dokud cyklus neukončíme stiskem Esc
{
    if( !c )       //Před vlastním kódem šipky musí předcházet nula,
    {              //pokud nepředchází, nemohla to být šipka
                   //Vlastní kód je již připraven
        switch( c = getch() )      //Čteme vlastní kód
        {
        case SNa:                  //Šipka nahoru - zvyšujeme tón
             if( y > 1 )
             {
                  ny = y-1;
                  Ton *= KTON;
             }
             break;

        case SLe:                  //Šipka vlevo - zkracujeme tón
             if( x > 1 )
             {
                  nx = x-1;
                  Delka /= KDEL;
             }
             break;

        case SPr:                  //Šipka vpravo - prodlužujeme tón
             if( x < MAX_X )
             {
                  nx = x+1;
                  Delka *= KDEL;
             }
             break;

        case SDo:                  //Šipka dolů - snižujeme tón
             if( y < MAX_Y )
             {
                  ny = y+1;
                  Ton /= KTON;
             }
             break;

        defalult:
             c = 0;          //Abychom poznali, že to nebyla šipka
        } /* switch */
    }
    if( c )                  //Byla to šipka
    {
                             //Smazání znaku na původní poloze
        crt << setxy( x, y ) << ' ';    //- přepíšeme jej mezerou
        crt << setxy( x=nx, y=ny )      //Nastavení   nové   polohy
znaku
            << ZNAK;             //a jeho vytištění
    }
    int t = Ton;             //Aby se v následujícím cyklu nemusely
```

```
      int d = Delka;           //reálné hodnoty pořád přepočítávat
      do                       //na celé
      {                        //Pípáme až do stisku nějaké klávesy
          sound( t );          //Tón
          delay( d );          //d milisekund
          nosound();           //Ticho
          delay( d );          //d milisekund
      }while( !kbhit() );      //Pípání přerušíme stiskem klávesy
  } while( (c = getch()) != Esc );   //Stisk Esc ukončuje program
}
```

11.3 Formátovaný výstup

Dosud jsme se u všech výstupních textů starali pouze o vystupující hodnoty a vlastní formát vystupujícího čísla nás nezajímal. Přesněji řečeno nesměl nás zajímat, protože jsme jej neuměli žádným způsobem ovlivnit.

Formátování výstupu se v obou jazycích naprosto zásadně liší, a to nejen způsobem zadávání formátu, ale celou filozofií přístupu k problému. Začneme jako obyčejně nejprve Pascalem, jehož všechny dvě možnosti ovlivňování podoby výstupu jsou poměrně průzračné.

V Pascalu můžeme u všech „nereálných“ typů vystupujících dat ovlivnit pouze velikost prostoru, který si program pro vystupující hodnotu vyhradí, a to tak, že za vystupující hodnotu se napíše dvojtečka a za ní celočíselný výraz udávající počet rezervovaných znaků. Program pak vytiskne vystupující hodnotu zarovnanou k pravému okraji vyhrazeného pole a prostor před ní zaplní mezerami. Pokud se vystupující hodnota do vyhrazeného pole nevejde, „vyteče“ z něj doprava.

S reálnými čísly je to trochu komplikovanější. Velikost rezervovaného prostoru se sice zadává stejně, ale minimální požadovaný prostor je 8 znaků, a pokud zadáme prostor menší, program se tváří, jako kdybychom zadali 8. Vystupující čísla jsou přitom tištěna v semilogaritmickém tvaru.

Pokud chceme tisknout reálná čísla v přímém tvaru, napíšeme za počet vyhrazených míst ještě jednu dvojtečku a za ní výraz udávající počet zobrazovaných desetinných míst. Abyste si vše procvičili, spusťte si příklad P11 – 4 a podívejte se, jaký formát mají údaje vystupující na obrazovku.

```
(*   Příklad P11 – 4   *)
{příklady formátování výstupu}
const
  a: integer = 456;
  b: real    = 3.141592653589;
  c: char    = 'k';
  d: boolean = true;
begin
  writeln(a);       {Implicitní formát}
  writeln(a:1);     {Málo místa}
```

```
  writeln(a:5);
  writeln(b);       {Implicitní formát}
  writeln(b:4);     {Málo místa}
  writeln(b:4:2);   {Čtyři místa celkem, z toho dvě desetinná}
  writeln(c:7);
  writeln(d:7);
end.
```

V C++ je formátovacích možností neporovnatelně více a z toho zákonitě vyplývá i poněkud větší obtížnost jejich zvládnutí. Trochu nepříjemné je navíc i to, že chceme-li plně využít schopností datových proudů, musíme používat i některé objektové konstrukce, které jsme si ještě nevysvětlovali.

Pokud tedy nebudete náhodou něčemu v dalším textu rozumět, zkuste si to sami vyzkoušet, a pokud zjistíte, že to funguje, vezměte to jako fakt, aniž byste se pídili po důvodech. Při výkladu objektově orientovaného programování se ke všemu ještě dostaneme.

Formát vystupujících dat můžeme ovlivnit třemi způsoby:

a) nastavením nebo shozením formátovacích příznaků,

b) použitím tzv. manipulátorů,

c) voláním formátovacích funkcí.

Podívejme se nyní na uvedené možnosti jednu po druhé.

Nastavení a shození formátovacích příznaků

V rámci objektového typu *ios* (název vznikl jako zkratka slov *input output stream* – vstupní a výstupní proud) je definován nepojmenovaný výčtový typ, jehož literály symbolizují použitelné formátovací příznaky. Jejich seznam je v tabulce 11.5.

Stav těchto příznaků můžeme ovlivňovat buď pomocí funkcí nebo prostřednictvím tzv. manipulátorů. O obou možnostech si povíme v následujících pasážích. Hodnoty těchto příznaků označujeme uvedenými identifikátory, před které připojíme operátorem „::“ jméno typu *ios* (např. *ios::left*).

Příznak	Význam
skipws	Je-li příznak nastaven, budou se na vstupu přeskakovat úvodní bílé znaky. Je-li příznak vynulován, bude následujícím vstupujícím znakem první dosud nepřečtený znak bez ohledu na to, zda se jedná o bílý znak či nikoliv.
left	Nastaví zarovnávání vystupujících hodnot k levému okraji vyhrazené oblasti a prostor vpravo od této hodnoty do konce vyhrazené oblasti se zaplní výplňovými znaky (implicitně mezerami).

Příznak	Význam
right	Nastaví zarovnávání vystupujících hodnot k pravému okraji vyhrazené oblasti a prostor od počátku vyhrazené oblasti k vystupující hodnotě se zaplní výplňovými znaky.
internal	Nastaví vnitřní zarovnávání vystupujících hodnot, při němž se znaménko doráží k levému okraji vyhrazené oblasti a vystupující hodnotu k pravému. Prostor mezi znaménkem a vystupující hodnotou se zaplní výplňovými znaky.
dec	Nastavuje vstup a výstup celočíselných hodnot v desítkové soustavě.
oct	Nastavuje vstup a výstup celočíselných hodnot v osmičkové soustavě.
hex	Nastavuje vstup a výstup celočíselných hodnot v šestnáctkové soustavě
showbase	Je-li příznak nastaven, bude se k vystupujícím hodnotám přidávat příznak použité číselné soustavy – vedoucí nulu k hodnotám v osmičkové soustavě a předponu 0x k hodnotám v šestnáctkové soustavě. Je-li příznak vynulován, budou se čísla ve všech číselných soustavách zobrazovat bez těchto dodatečných informací.
showpoint	Je-li příznak nastaven, bude se při výstupu reálných hodnot vždy zobrazovat desetinná tečka.
uppercase	Je-li příznak nastaven, budou se pro znaky A, B, C, D, E, F v hodnotách v šestnáctkové soustavě používat velká písmena, je-li vynulován, budou se používat písmena malá.
showpos	Je-li příznak nastaven, bude se u před kladnými čísly zobrazovat znaménko +. Je-li vynulován, bude se znaménko zobrazovat pouze u záporných čísel.
scientific	Nastavuje výstup reálných hodnot v semilogaritmickém tvaru.
fixed	Nastavuje výstup reálných hodnot v přímém tvaru.
unitbuf	Po výstupu se všechny proudy spláchnou – význam tohoto příznaku si vysvětlíme později.
stdio	Po výstupu spláchni proudy *stdout* a *stderr* – význam tohoto příznaku si vysvětlíme později.

Tab. 11.5 Formátovací příznaky

Použití manipulátorů

Základní metodou ovlivňování formátu vystupujících dat je používání tzv. manipulátorů, což jsou speciální objekty, definované tak, abychom je mohli vkládat do řetězu operací vstupu a výstupu, jako by se jednalo o vstupující nebo vystupující data.

Pokud chcete ve svých programech používat manipulátory s parametry, musíte do programu vložit soubor *iomanip.h*. V základní výbavě jsou k dispozici následující manipulátory:

Manipulátor	Význam
dec	Nastaví vstup a výstup v desítkové soustavě.
hex	Nastaví vstup a výstup v šestnáctkové soustavě.
oct	Nastaví vstup a výstup v osmičkové soustavě.
resetioflags(long)	Vynuluje příznaky označené v parametru.
setioflags(long)	Nastaví příznaky označené v parametru.
setbase(int)	Nastaví číselnou soustavu vstupu a výstupu. Povolené hodnoty parametru jsou 0, 8, 10 a 16. Hodnota 0 nastavuje implicitní režim, při němž jsou vystupující hodnoty předávány v desítkové soustavě a pro vstupní hodnoty platí stejná pravidla jako pro celočíselné literály.
setfill(int)	Nastaví výplňový znak, kterým se bude zaplňovat zbytek vyhrazeného prostoru neobsazený vystupující hodnotou.
setprecision(int)	Nastaví přesnost vystupujících reálných hodnot.
setw(int)	Nastaví velikost rezervovaného prostoru, tj. minimální počet znaků, které se při výstupu příští položky vyšlou do výstupního proudu. **Pozor! Toto nastavení platí pouze pro následující položku a pak se opět automaticky nastavuje 0.**
ws	Na vstupu se přeskočí všechny následující bílé znaky.
endl	Pošle na výstup přechod na novou řádku a spláchne.
ends	Do vystupujícího řetězce přidá ukončující nulu.
flush	Spláchne obsah vyrovnávací paměti do výstupního proudu. K operaci spláchnutí se ještě vrátíme později, až si budeme vyprávět o vyrovnávacích pamětech.

Volání formátovacích funkcí

Používání manipulátorů má jednu nevýhodu: můžeme sice s jejich pomocí nastavit nový stav, ale nemůžeme pak vrátit stav původní, protože jste si jej nezapamatovali, a proto jej neznáte. V takovém případě (a v řadě dalších) pak použijeme s výhodou některou z následujících funkcí, které vám umožní zapamatovat si původní nastavení měněných atributů a po vaší operaci je znovu obnovit.

Následující funkce jsou definovány jako tzv. **metody** datových proudů, a proto musíme při jejich volání používat poněkud jinou syntax. (Už jsme se s ní několikrát setkali, ale neuškodí, když si ji zopakujeme.) Pokud budeme chtít ve svých programech volat některou z níže uvedených funkcí, musíme napsat identifikátor datového proudu, v němž chcete ovlivnit formátovací atributy, následovaný tečkou a identifikátorem volané funkce s případnými parametry, např.:

```
cout.fill( '*' );
```

Ve zbytku této pasáže si vyjmenujeme funkce, které bychom mohli při programování formátovaného výstupu upotřebit. U každé funkce si uvedeme její prototyp a význam atributu, jehož hodnotu prostřednictvím této funkce zjišťujeme nebo měníme.

Všimněte si, že v některých prototypech nejsou uvedeny identifikátory parametrů. Norma jazyka to povoluje a identifikátory parametrů chápe spíše jako komentáře, které mají blíže objasnit účel daného parametru, avšak které překladač ke své bezchybné funkci nepotřebuje. (Zkuste např. v prototypu a v definici funkce udat jiné identifikátory parametrů a uvidíte, že to překladač akceptuje.)

Některé z popisovaných funkcí najdete ve dvou variantách, které se liší tím, zda mají či nemají nějaký parametr. Pokud zavoláte funkci bez parametrů, vrátí vám hodnotu atributu, který obhospodařuje. Pokud ji zavoláte s parametrem, vrátí dosavadní hodnotu daného atributu a nastaví jeho novou hodnotu na hodnotu parametru.

```
char fill();
char fill( char );
```

Zjišťuje a nastavuje výplňový znak, kterým se zaplňuje zbytek vyhrazeného prostoru neobsazený vystupující hodnotou.

```
long flags();
long flags( long );
```

Zjišťuje a nastavuje nastavení formátovacích příznaků.

```
int precision();
int precision( int );
```

Zjišťuje a nastavuje přesnost vystupujících reálných hodnot.

```
long setf( long Nastav );
```

Nastaví formátovací příznaky zadané v parametru *Nastav*. Vrací jejich původní nastavení.

```
long setf( long Nastav, long Nuluj );
```

Vynuluje příznaky zadané v parametru *Nuluj* a pak nastaví příznaky zadané v parametru *Nastav*. Vrací původní hodnotu celého komplexu příznaků.

```
int width();
int width( int );
```

Zjišťuje a nastavuje velikost prostoru vyhrazeného pro výstup.

```
long unsetf( long );
```

Vynuluje zadané příznaky.

Z vyjmenovaných funkcí si zasluhují zvláštní pozornosti pouze *setf()*. Tyto funkce se totiž pokouší dbát na to, že u některých příznaků může být platný pouze jeden příznak z určité množiny: u příznaků zarovnávání jeden z příznaků *ios::left*, *ios::right* a *ios::internal*, u příznaků použité číselné soustavy jeden z příznaků *ios::dec*, *ios::oct* a *ios::hex* a u příznaků tvaru reálných čísel jeden z příznaků*ios:scientific* a *ios:fixed*.

Pokud tedy některý z příznaků nastavovaných funkcí *setf()* patří do jedné ze tří výše uvedených množin, funkce nejprve všechny příznaky dané množiny vynuluje a teprve pak zadaný příznak nastaví.

Dvouparametrická verze funkce *setf()* je zavedena proto, aby se snížil počet potřebných testů a celý proces se urychlil. Umožňuje explicitně zadat množiny příznaků, které je třeba před nastavením požadovaných příznaků vynulovat, takže odpadnou všechny dodatečné testy.

Abychom tyto množiny mohli snadno zadávat, jsou součástí definice datových proudů i tři konstanty, označující množinu vzájemně se vylučujících příznaků, které definují hodnotu formátovacího parametru. Těmito konstantami jsou *ios::adjustfield,* označující příznaky způsobu zarovnávání, *ios::basefield* označující příznaky číselné soustavy a *ios::floatfield* označující příznaky tvaru reálných čísel.

Chcete-li tedy nastavit ve standardním výstupním proudu šestnáctkovou číselnou soustavu a vkládání nul jako výplňových znaků mezi znaménko a vlastní vystupující číslo a po vlastním výstupu opět obnovit původní stav, můžete toho dosáhnout následujícím fragmentem programu:

```
/*   Příklad C11 – 2   */
//...Předchozí program
//Zapamatujeme si původní hodnoty a nastavíme hodnoty požadované
char c = cout.fill( '0' );
long f = cout.setf( ios::hex + ios::internal,
                    ios::basefield | ios::adjustfield);
//Vzhledem k hodnotám příznaků a konstant je jedno, zda
//je sčítáme nebo s nimi provádíme bitové OR (|).
//...Zde by měla být část programu
//realizující potřebné výstupní operace
//Obnovení původních hodnot
cout.fill( c );
cout.flags( f );
//...Další části programu
```

12. Náhodná čísla

Při simulaci procesů reálného života velice často potřebujeme, aby se simulovaný proces nechoval zjevně deterministicky (předem určeně), ale aby se okolnímu světu (pozorovateli, ostatním částem programu atd.) jevil jako více méně náhodný. Této zdánlivé (a někdy i skutečné) náhodnosti simulovaného procesu dosahujeme tak, že některé hodnoty řídící další průběh výpočtu generujeme pomocí tzv. **generátoru náhodných čísel**.

Generátory náhodných čísel jsou v podstatě dvojího druhu: hardwarové a softwarové. Hardwarové generátory jsou vlastně generátory tzv. bílého nebo růžového šumu a generují neopakovatelné nekonečné posloupnosti opravdu náhodných čísel. Tyto generátory jsou však většinou zbytečně drahé, je třeba je neustále dolaďovat, aby generovaly opravdu „ten správný“ šum, a proto se používají opravdu výjimečně.

Většina aplikací vystačí se softwarovými generátory, i když se ve skutečnosti jedná pouze o generátory pseudonáhodných čísel. Jsou to jednoduché prográmky generující posloupnosti čísel, jež vykazují mnohé vlastnosti opravdu náhodné posloupnosti čísel – většinou s rovnoměrným rozložením. Tato posloupnost ovšem zákonitě obsahuje jen konečný počet různých hodnot, a proto se začne po jisté (dlouhé, avšak konečné) době opakovat.

Napsat generátor náhodných čísel je velmi snadné (mnohé programy se tak často chovají i proti naší vůli) – obtíže začnou ve chvíli, kdy potřebujete zjistit, jak dobře náhodná čísla tento generátor generuje. Dobrý generátor náhodných čísel s rovnoměrným rozložením[24] musí např. generovat nejen co nejdelší posloupnost, ale měl by zároveň generovat takovou posloupnost, jejíž členové dávají po dělení libovolným číslem co nejrovnoměrnější rozložení všech zbytků, jejich dvojic, trojic, … , n-tic atd.

Oba jazyky (přesněji řečeno všechny jejich zmiňované implementace) proto poskytují prověřené a přitom dostatečně rychlé generátory náhodných čísel. Když některou z funkcí těchto generátorů (pseudo)náhodných čísel zavoláme, obdržíme jako funkční hodnotu pseudonáhodné číslo, které je hodnotou nějaké matematické funkce, jejíž průběh je natolik „divoký“, že její jednotlivé hodnoty můžeme interpretovat jako nezávislé a předpokládat, že pravděpodobnost obdržení dané funkční hodnoty je pro všechny hodnoty z rozsahu, ze kterého je generujeme, stejná[25]. Ukážeme si nyní, jaké možnosti máme.

---

[24] V našem povídání se bude objevovat pojem „rovnoměrné rozložení“ (náhodných čísel). Pokusíme se vysvětlit, o co vlastně jde. Jednoduchým příkladem rovnoměrně rozdělených náhodných čísel jsou výsledky vrhů kostkou (samozřejmě poctivou). Možné výsledky jsou 1, 2, ... 6, a všechny padnou se stejnou pravděpodobností $^1/_6$. Obecněji budeme hovořit o rovnoměrném rozložení v případě, že náhodný pokus má n možných výsledků a všechny mají stejnou pravděpodobnost $^1/_n$. Jestliže může být výsledkem náhodného pokusu libovolné číslo z intervalu <0, 1), budeme říkat, že má rovnoměrné rozložení, jestliže pravděpodobnost, že se toto náhodné číslo „strefí“ do intervalu (a, b), který je podmnožinou <0, 1), je rovna délce tohoto intervalu, tedy $b - a$.

[25] Poznamenejme, že žádný generátor pseudonáhodných čísel nemůže generovat všechna čísla z intervalu <0, 1) např. proto, že v počítači ani nelze všechna čísla z tohoto intervalu zobrazit.

Základním prostředkem pro generování náhodného čísla v Pascalu je funkce *random*, která má dva tvary:

```
function Random : real
function Random( Rozsah : word ) : word
```

Verze bez parametrů vrátí pseudonáhodné **reálné** číslo z intervalu <0; 1) (všimněte si, že interval je zprava otevřený, takže jedničku nikdy nezískáte), kdežto verze s parametrem vrátí číslo typu *word*, které bude z intervalu <0; *Rozsah*) (opět zprava otevřeného).

S generátorem můžeme pracovat dvěma způsoby: buď potřebujeme, aby se při každém spuštění nějaké činnosti posloupnost získávaných pseudonáhodných čísel opakovala (např. při ladění nebo šifrování a dešifrování), nebo naopak potřebujeme, aby byla pokud možno pokaždé jiná (např. generování hodu kostkou apod.). Obou možností dosáhneme vhodnou inicializací.

Pokud chceme, aby se posloupnost generovaných čísel pokaždé opakovala, inicializujeme generátor tak, že zadáme pokaždé stejné číslo do systémové proměnné

```
var RandSeed : longint;
```

jejíž obsah charakterizuje momentální stav generátoru a jednoznačně determinuje další posloupnost (často se jí říká „kvásek“). Když do *RandSeed* běžným přiřazovacím příkazem uložíme nějakou hodnotu (třeba 1), uvedeme generátor do stavu příslušejícímu tomuto číslu. To můžeme kdykoliv opakovat a uvést tak generátor do stejného stavu jako minule, a tak také získat stejnou posloupnost generovaných čísel.

Pokud naopak chceme posloupnost jedinečnou a pokaždé jinou (samozřejmě v rámci omezení daného generátoru), použijeme pro inicializaci generátoru proceduru

```
procedure Randomize;
```

jež inicializuje systémovou proměnnou *RandSeed* podle okamžitého stavu vnitřních hodin (čítají s frekvencí asi 18.2 Hz). Ten bude zákonitě při každém spuštění programu jiný a tím bude jiná i posloupnost generovaných čísel.

Abychom mohli používat funkce generátoru náhodných čísel, musíme nejprve vložit do programu hlavičkový soubor *stdlib.h*, který obsahuje jejich deklarace. Základním prostředkem pro získání náhodného čísla je funkce

```
int rand();                //#include <stdlib.h >
```

která vrací pseudonáhodné číslo z intervalu <0; *RAND_MAX*), kde *RAND_MAX* je konstanta definovaná v *stdlib.h* a má hodnotu (2^{15}-1). Tato funkce se však vzhledem k intervalu výstupních hodnot používá poměrně zřídka, protože většinou musíme získanou hod-

---

Kvalita generátorů pseudonáhodných čísel se obvykle vyšetřuje statistickými metodami: z generátoru získáme větší množství čísel a snažíme se zjistit, zda jde o „opravdu náhodná“ čísla, nebo zda se od nich nějak liší.

notu „normalizovat“ (upravit na žádaný rozsah) – nejčastěji pomocí operátoru % (dělení modulo) a případného sčítání. Například příkaz

```
Prom = 100 + rand() % 900;
```

uloží do *Prom* nějaké číslo z rozsahu <100; 999). Tuto normalizaci má v sobě částečně zabudovanou funkce

```
int random( int Max );          //#include <stdlib.h >
```

která vrací číslo z intervalu <0; *Max*) a s jejíž pomocí bychom mohli předchozí příklad přepsat do tvaru

```
Prom = 100 + random( 900 );
```

S generátorem můžeme pracovat dvěma způsoby: buď potřebujeme, aby se při každém spuštění nějaké činnosti posloupnost získávaných pseudonáhodných čísel opakovala (např. při ladění nebo šifrování a dešifrování), anebo naopak potřebujeme, aby byla pokud možno pokaždé jiná (např. generování hodu kostkou apod.). Obou možností dosáhnete vhodnou inicializací.

Systém inicializuje kvásek hodnotou 1 a generátor proto posílá vždy stejnou posloupnost. Budeme-li chtít tuto posloupnost nahradit jinou, použijeme funkci

```
void srand( unsigned Kvasek );  //#include <stdlib.h >
```

která přiřadí kvásku hodnotu svého parametru. Volání *srand()* můžeme kdykoliv opakovat a uvést tak generátor znovu do stejného výchozího stavu. Tím získáme znovu stejnou posloupnost generovaných čísel.

Pokud naopak chceme posloupnost jedinečnou a pokaždé jinou (samozřejmě v rámci omezení daného generátoru), použijeme pro inicializaci generátoru proceduru

```
void randomize();               //#include <stdlib.h >
```

která uvede generátor do náhodného stavu, daného momentální hodnotou čítače času (čítá s frekvencí asi 18.2 Hz). Ten bude zákonitě při každém spuštění programu jiný a tím bude jiná i posloupnost generovaných čísel.

Při použití této funkce potřebuje mít překladač k dispozici také některé z informací, uložených v souboru *time.h*, a proto jej nesmíme zapomenout také vložit.

Příklad: vrhcáby

Jako příklad na procvičení všeho, o čem jsme v této knize zatím mluvili, sestavíme program, který bude řídit hru vrhcáby pro proměnný počet hráčů a bude jim přitom poskytovat maximální „účetní“ komfort.

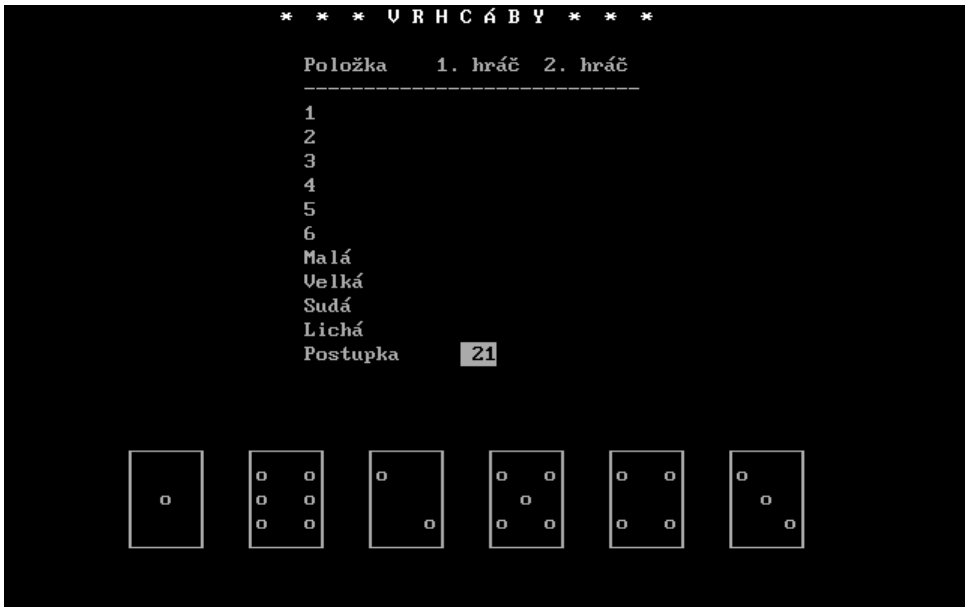
Podívejme se nejprve na pravidla hry: Vrhcáby může hrát libovolný počet hráčů, optimální množství je však 2 až 4. Hráč, který je na řadě, hodí šesti kostkami a po dopadu se podle výsledku hodu rozhodne pro zaplnění jedné z jedenácti figur, které se liší výběrem

hodnot, jež se sčítají. V průběhu hry se může pro každou figuru rozhodnout jen jednou. Ve chvíli, kdy hráči zaplnili všechny figury, hra končí.

Figury jsou jedničky až šestky, malá (1,2,3), velká (4,5,6), sudá (2,4,6), lichá (1,3,5) a postupka (někdy se říká pyramida). Hráč, který se rozhodne pro některou figuru, připíše na své konto součet bodů na těch kostkách, na nichž padly hodnoty odpovídající dané figuře. (Rozhodne-li se pro postupku, smí každou cifru započítat pouze jednou, takže maximálního součtu dosáhne při hodu 123456 nebo při nějaké jeho permutaci.) Vítězí hráč, který má na konci na svém kontě nejvíce bodů.

Program se nejdříve zeptá, kolik hráčů bude hrát. Potom nakreslí tabulku, v níž budou sloupce odpovídat jednotlivým hráčům a řádky budou obsahovat jednotlivé součtové položky.

Potom je postupně jeden hráč po druhém vyzván, aby stiskl nějaké tlačítko a spustil tak míchání: program neustále generuje čísla na „kostkách“, přitom vydává nějaký zvuk a zobrazuje blikající text KOSTKY SE MÍCHAJÍ. Míchaní pokračuje až do dalšího stisku tlačítka, pak se zastaví a počítač na obrazovku nakreslí hodnoty šesti kostek, přičemž zobrazení bude vypadat např. tak, jako na obr. 12.1 (ve spodní části).



Obr. 12.1 Výstup programu Vrhcáby

Potom bude hráč pohybovat pomocí šipek kurzorem po ještě nezaplněných figurách (samozřejmě pouze v rámci svého sloupce), přičemž se mu vždy zobrazí součet, který by byl přičten v případě, že by danou figuru zvolil. Stiskem tlačítka ENTER hráč zvolí figuru, na níž je právě kurzor, a počítač vyzve ke hře dalšího hráče.

Po jedenácti kolech počítač vypíše celkové součty a určí vítěze. (Pamatujte na možnost, že se o první místo bude dělit více hráčů.) Vzorové řešení představují programy *P12–01.PAS* resp. *C12–01.CPP* na doplňkové disketě; obrázek 12.1 ukazuje jejich výstup na obrazovku.

13. Dodatek

13.1 Obcházení typové kontroly parametrů v Turbo Pascalu

Jak víte, Pascal je jazyk, který se honosí přísnou typovou kontrolou. Tato kontrola však v praxi mnohdy brání zdárnému vyřešení úkolu. Autoři Turbo Pascalu proto zavedli několik rozšíření, která umožňují tuto typovou kontrolu buď obejít nebo dokonce úplně vyřadit z činnosti a která z jejich překladače udělala vhodný nástroj i pro praktické programátory. (95% pascalských programů na PC je psáno v Turbo Pascalu – to nepotřebuje žádný další komentář.)

Jedním z těchto rozšíření je možnost přetypování, o níž jsme již této knize hovořili. Druhým, neméně užitečným rozšířením je pak možnost neuvádět typy parametrů předávaných odkazem, tj. parametrů označovaných v deklaracích podprogramů klíčovým slovem **var**. Je to v podstatě totéž, jako kdybychom předávali hodnotou parametr typu *Pointer* (tj. ukazatel na cokoliv), ale řešení pomocí netypových parametrů je přehlednější.

Když u parametru neuvedeme jeho typ, tak nás sice nebude překladač obtěžovat hlášeními o porušení typové disciplíny, ale vzhledem k nedostatku informací o objektu jej také nebude moci nijak smysluplně použít – leda bychom jej neustále ověšovali přetypovávacími doplňky. Aby pro nás byly netypové parametry doopravdy užitečné, musíme mít prostředky pro dodatečnou definici jejich typu v podprogramu. Jen tak totiž s nimi budeme moci pracovat jako s obyčejnými proměnnými.

Tímto prostředkem je klíčové slovo **absolute**, které vám umožňuje umístit deklarovanou proměnnou kdekoliv v paměti. (Občas se nám po podobném prostředku v C++ stýská.) Když toto klíčové slovo uvedeme za deklarací proměnné (ale před závěrečným středníkem) a za ním uvedete nějakou adresu, překladač deklarovanou proměnnou na danou adresu umístí.

Jak jste jistě správně vyrozuměli, za klíčové slovo **absolute** (přesněji mezi ně a závěrečný středník) můžete napsat jakoukoliv adresu, takže si tímto způsobem můžete zpřístupnit i části operačního systému. Tyto fajnovosti však zatím stranou a nyní se omezíme na základní způsob zadání adresy, což je zápis identifikátoru proměnné, s níž bude deklarovaná „absolutně umísťovaná“ proměnná sdílet společný prostor. Přesněji řečeno obě proměnné budou v paměti začínat na stejné adrese.

Této možnosti se nejvíce používá právě ve spojitosti s netypovými parametry. Podívejte se na následující ukázku:

```
procedure (*****) Prohod (******)
( var p1; var p1; Bytu:integer );
type
  AoB = array[ 0..30000 ] of byte;

var
  x : AoB absolute p1;
  y : AoB absolute p2;
```

```
  i : integer;
  b : byte;
begin
  for i:=0 to Bytu-1 do
  begin
       b := x[i];
       x[i] := y[i];
       y[i] := b;
  end;
end;
```

Tato procedura prohodí obsah dvou proměnných jakéhokoli typu (pokud mají obě stejnou délku). Tyto proměnné se předávají jako beztypové parametry *p1* a *p2*; parametr *Bytu* udává velikost skutečných parametrů v bajtech. V programu skutečné parametry pomocí deklarace **absolute** ztotožníme s dostatečně dlouhými poli bajtů a pak jejich obsah bajt po bajtu prohodíme.

13.2 Obcházení typové kontroly parametrů v C++

Aby se programátoři C++ necítili ochuzeni, řekneme si stručně o možnostech obcházení typové kontroly i v jazyku C++. Pokud v C++ potřebujete definovat funkci, která má parametr, jehož typ není v době překladu této funkce ještě znám, máte dvě možnosti řešení: buď se bude jednat o parametr, který se nachází na konci seznamu parametrů v prototypu dané procedury, a pak můžeme použít výpustku (o způsobu předávání těchto parametrů jsme si již vyprávěli), nebo deklarujeme v definici funkce tento parametr jako parametr typu **void***, tj. jako ukazatel na cokoliv.

```
void /*****/ Prohod /*****/
( void *p1, void *p2, unsigned N )
{
//Následující ukazatele nastavíme na konec proměnných px
//čímž si zjednodušíme testy výstupní podmínky cyklu
  register char *x = ((char*)p1 + N-1);
  register char *y = ((char*)p2 + N-1);
  char b;
  do
  {
       b = *x;
       *x = *y;
       *y = b;
  } while( x--, y-- != p2 );
}
```

Na závěr bychom chtěli jen dodat, že víme, že uvedené procedury nejsou napsány optimálně – to bychom museli použít řadu konstrukcí, které ještě neznáte. Budiž to výzvou pro ty zkušenější z vás, aby se je pokusili napsat lépe –klidně i ve strojním kódu.

13.3 Parametry příkazového řádku

Většina profesionálních programů nám dovoluje zadávat v příkazovém řádku parametry, na jejichž podkladě program modifikuje svoji funkci. Určitě bychom rádi využili těchto možností i ve svých programech. Není to nic obtížného.

V Pascalu slouží k práci s parametry příkazového řádku funkce

function *ParamCount* : *word*;

a

function *ParamStr*(i: *integer*) : *String*;

Funkce *ParamCount* vrátí celkový počet parametrů předávaných programu v příkazovém řádku, přičemž předpokládá, že jednotlivé parametry jsou odděleny mezerami nebo tabulátory nebo uzavřeny v uvozovkách.

K vlastnímu získávání hodnot těchto parametrů slouží funkce *ParamStr*, které předáme pořadí žádaného parametru a která nám vrátí odpovídající text. Pracujete-li pod operačním systémem, jehož číslo verze je minimálně 3.0, můžete tuto funkci požádat také o hodnotu nultého parametru, který se do celkového počtu parametrů nezapočítává a jehož hodnotou je název spuštěného programu spolu s úplnou cestou.

Domníváme se, že práce s parametry příkazového řádku je natolik jednoduchá, že nepotřebuje další komentář (obtížnější může být zpracování vlastních hodnot těchto parametrů). V následující ukázce je příklad programu, který v příkazovém řádku očekává parametry ve tvaru dvojice čísel. Prvé číslo označuje výšku tónu v Hz a druhé číslo jeho délku v milisekundách. Posloupnost těchto dvojic definuje fanfáru, kterou program zahraje.

Abychom následující ukázku pochopili, musíme se nejprve seznámit s funkcí *val*, jejíž deklarace má tvar

function *val*(*Text*: *String*; *var Cislo*; **var** *Kod*);

Tato funkce převádí text v řetězci *Text* na číslo. Typ skutečného parametru *Cislo* pak určuje, zda se tento text bude převádět jako číslo celé či reálné. (Abychom měli k dispozici obě možnosti, musí být parametr deklarován jako netypový.) V parametru *Kód* pak vrací nulu v případě, že převod proběhl bez problému. Pokud v tomto parametru najdete nenulové číslo, jedná se o index znaku, který při převodu řetězce způsobil chybu.

```
(*   Příklad P13 – 1   *)
Program Fanfara;
Uses crt;

var
  i:integer;
  Vyska : integer;
  Delka : integer;
  Poz : integer;
```

```
begin
{Počet parametrů nezahrnuje nultý parametr
 s názvem programu a úplnou cestou}
  for i:=0 to ParamCount do
      writeln( i, ': ', ParamStr( i ) );
  if( (ParamCount mod 2) <> 0 )then
  begin
      writeln( 'Lichý počet parametrů' );
      halt( 1 );                    {Ukonči program s chybovým kódem
  1}
  end;
  for i:=1 to ParamCount div 2 do
  begin
      val( ParamStr( 2*i-1 ), Vyska, Poz );
      val( ParamStr( 2*i ), Delka, Poz );
      sound( Vyska );
      delay( Delka);
      Nosound;
  end;
end.
```

V C++ získáme informace o parametrech příkazového řádku pomocí parametrů funkce *main()*. Pro deklaraci funkce *main()* platí následující pravidla:

1. Návratovým typem smí být pouze **int**. Návratová hodnota představuje kód ukončení programu, který lze použít např. při dávkovém zpracování pomocí testů *ERRORLEVEL*. (Poznamenejme, že mnohé překladače, mj. i Borland C++, tolerují i funkci *main()* deklarovanou s typem **void**.)
2. Funkce *main()* můžeme deklarovat bez parametrů, s jedním nebo dvěma parametry (v Borland C++ také se třemi parametry).
3. Pokud deklarujeme ve funkci *main()* parametry, musí být první z nich typu **int** (tradičně se mu dává jméno *argc*). Jeho hodnotou bude počet parametrů příkazového řádku, přičemž v tomto počtu je započítán i nultý parametr obsahující název programu včetně úplné cesty.
4. Pokud deklarujeme funkci *main()* s alespoň dvěma parametry, musí být druhý parametr typu *char* ***[ ]** a tradičně se označuje *argv*. (Je to pole textových řetězců; vzhledem k tomu, že pole se při předávání jako parametr transformuje na ukazatel na první prvek, můžeme typ tohoto parametru také deklarovat jako **char****.) Prvky tohoto vektoru ukazují na jednotlivé parametry příkazového řádku. (Parametry jsou v příkazovém řádku odděleny mezerami nebo tabulátory nebo jsou uzavřeny v závorkách.)
5. Třetí parametr, pokud jej deklarujeme, má stejný typ jako druhý; je specialitou Borland C++ a obsahuje ukazatel na pole textových řetězců, obsahující kopii proměnných operačního systému (nastavovaných příkazem SET). Toto pole končí prázdným řetězcem.

V následující ukázce je příklad programu, který v příkazovém řádku očekává parametry ve tvaru dvojice čísel. Prvé číslo označuje výšku tónu v Hz a druhé číslo jeho délku v milisekundách. Posloupnost těchto dvojic definuje fanfáru, kterou program zahraje.

```
/*   Příklad C13 – 1   */
#include <dos.h>
#include <process.h>
#include <strstream.h>

void /*****/ main /*****/
( int Pocet, char *Param[] )
{
//Počet parametrů zahrnuje i nultý parametr
//s názvem programu a úplnou cestou
  for( int i=0;
       i < Pocet;
       cout << i << ": " << Param[ i++ ] << endl );
  if( !(Pocet & 1) )            //Test lichosti počtu parametrů
  {             //Celkový počet parametrů (včetně nultého) je sudý
       cout << "Lichý počet parametrů";
       exit( 1 );               //Ukončí program s chybovým kódem 1
  }

//V C++ se místo indexování používá rychlejších ukazatelů
  for( char **p = &Param[ 1 ];
  p != &Param[ Pocet ];
  /*Modifikace se provádí v těle cyklu*/ )
  {
       int Vyska;
       int Delka;
       istrstream TxtV(*p++);
       TxtV >> Vyska;
       istrstream TxtD(*p++);
       TxtD >> Delka;
       sound( Vyska );
       delay( Delka);
       nosound();
  }
}
/********** main **********/
```

13.4 Vstupní a výstupní operace v jazyce C

V úvodu k povídání o datových proudech jazyka C++ v kapitole 15.3 jsme si řekli, že v C++ můžeme také používat veškeré prostředky pro vstupy a výstupy z jazyka C. Protože opravdový programátor musí ovládat obojí, nezbývá, než si o nich povědět alespoň v dodatku.

Výstup do souboru stdout

Prostředky pro výstup do souboru *stdout* najdeme v hlavičkovém souboru *stdio.h*. (Připomeňme si, že pod označením *stdout* se skrývá standardní výstup. Na PC to je obrazovka monitoru, lze jej ale příkazem operačního systému přesměrovat do souboru nebo na některé jiné zařízení, např. na tiskárnu.)

Funkce *printf()*

Funkce *printf( )* je z prostředků pro výstup do *stdout* bezesporu nejznámější. Vzhledem k tomu, že se v dalším povídání setkáme s funkcemi, které budou mít velmi podobné vlastnosti, povíme si o ní podrobněji. Její prototyp má tvar

int *printf*(**const char * format**, ...);

Tři tečky označují výpustku (viz kap. 5.6). První parametr této funkce musí být ukazatel na znakový řetězec. (To znamená, že to může být opravdu proměnná typu **char ***, pole znaků nebo řetězcová konstanta.) Za ní může následovat libovolný počet parametrů (dále si povíme, jakých mohou být typů.)

Podívejme se nejprve na nejjednodušší případ, kdy má funkce *printf( )* jen jeden parametr. Příkaz

```
printf("A jede se dále");
```

způsobí, že se do standardního výstupního proudu vloží řetězec

```
A jede se dále
```

a pokud výstup programu nepřesměrujeme, vypíše se na obrazovku monitoru. Přesně platí toto: Pokud řetězec, zadaný jako první parametr, neobsahuje znak %, vloží se do výstupního proudu beze změny. Tento řetězec může obsahovat i řídicí znaky, např. '\n', předepisující přechod na nový řádek, takže příkazem

```
printf("A jede se dále\nmočálem černým\nkolem bílých skal.");
```

vypíšeme Werichův citát na tři řádky:

```
A jede se dále
močálem černým
kolem bílých skal.
```

Konverze

Funkce *printf( )* ovšem umí tisknout i jiné věci než jen znakové řetězce. Hodnoty číselných a znakových typů, ukazatelů a řetězců můžeme zadat jako další parametry, předávané prostřednictvím výpustky. Funkci *printf( )* ovšem musíme na tyto parametry upozornit a musíme jí říci, kde a jak je má vytisknout. K tomu slouží znaky „%“, které zapíšeme v řetězci *format* (tj. v prvním parametru funkce *printf( )*).

Znak „%“ v řetězci *format* označuje místo, na které chceme vložit znakovou podobu hodnoty odpovídajícího parametru. Přitom první „%“ odpovídá prvnímu parametru za řetězcem *format*, druhý znak „%“ odpovídá druhému parametru za řetězcem *format* atd.[26]

Za znakem „%“ musí následovat údaje, které funkci *printf( )* řeknou, jakého typu je vystupující hodnota a jak ji má pro výstup upravit. (Např. zda chceme vytisknout celé číslo v desítkové nebo v šestnáctkové soustavě atd.) Tyto údaje se označují jako **specifikace konverze**, neboť funkci *printf( )* říkají, jak konvertovat vystupující hodnotu na znakový řetězec. Funkce *printf( )* odstraní z řetězce *format* specifikaci konverze a nahradí ji znakovou podobou vystupující hodnoty.

Obecný formát specifikace konverze je

```
%[příznak][šířka][.přesnost][velikost]typ
```

a závorky „[ ]“ zde označují položky, které můžeme vynechat. Jak je vidět, stačí v nejjednodušším případě uvést pouze typ vystupující hodnoty. Jejich přehled najdete v tabulce 13.1

Typ	Parametr	Vystoupí
d	Celé číslo	Celé číslo se znaménkem v desítkové soustavě
i	Celé číslo	Celé číslo se znaménkem v desítkové soustavě
o	Celé číslo	Celé číslo bez znaménka v osmičkové soustavě
x	Celé číslo	Celé číslo bez znaménka v šestnáctkové soustavě (jako číslice se použijí znaky 1, – 9, a, b, c, d, e, f, 0)
X	Celé číslo	Celé číslo bez znaménka v šestnáctkové soustavě (jako číslice se použijí znaky 1, ... , 9, A, B, C, D, E, F, 0)
u	Celé číslo	Celé číslo bez znaménka desítkové soustavě
f	Reálné číslo	Číslo s pevnou řádovou tečkou (tvar *-ddd.dddd*)
e	Reálné číslo	Číslo v semilogaritmickém tvaru (tvar *d.ddde+ddd*)
E	Reálné číslo	Číslo v semilogaritmickém tvaru (jako **e**, ale ve výstupu se použije znak E)
g	Reálné číslo	Podle okolností se použije buď konverze **f** nebo **e**
G	Reálné číslo	Podle okolností se použije buď konverze **f** nebo **E**
c	Celé číslo	Znak
s	ukazatel na řetězec	Znaky řetězce
p	ukazatel	adresa ve tvaru *Seg:Ofs* nebo jen *Ofs*
n	ukazatel na	do proměnné, na kterou ukazatel ukazuje, se uloží počet znaků,

[26] Dále uvidíme, že to není stoprocentně pravda, ale zatím se s tím spokojíme.

Typ	Parametr	Vystoupí
	int	které dosud vystoupily

Tab.13.1 Typ ve formátovacím řetězci

Podívejme se na příklad. Napíšeme kratičký program, který vytiskne tabulku čísel od 100 do 111 v osmičkové, desítkové a šestnáctkové soustavě.

```
/*   Příklad C13 – 2   */
#include <stdio.h>
int main(){
 printf("oct   dec   hex\n---------------\n");
 for(int i = 100; i < 112; i++)
  printf("%o   %d   %x\n", i,i,i);
 return 0;
}
```

První řádky výstupu tohoto programu budou

```
oct   dec   hex
---------------
144   100   64
145   101   65
 ...
```

Některé základní datové typy v tabulce 13.1 nenajdeme. Připomeňme si ale, že znakové typy, výčtové typy a typy **short** a **unsigned short** se při předávání na místě výpustky rozšíří na typ **int** a typ **float** se rozšíří na typ **double**.

Pro výstup hodnot typu **long**, **unsigned long** a **long double** musíme použít specifikaci velikosti.

Také pro rozlišení blízkých a vzdálených ukazatelů – pokud neodpovídají implicitní velikosti pro daný paměťový model – musíme použít modifikátor velikosti.

Poznámka:
Chceme-li vytisknout znak „%“, musíme jej zdvojit. Obsahuje-li proměnná x *typu* ***int*** *hodnotu 5, vytiskneme příkazem*

```
printf("pokles o %d%%\n",x);
```

sdělení

```
pokles o 5%
```

Velikost typu

Velikost typu vystupující hodnoty zadáváme jednopísmenovým modifikátorem před specifikací typu. Tyto modifikátory shrnuje tabulka 13.2.

Modifikátor	Význam
F	Vzdálený ukazatel

Modifikátor	Význam
N	Blízký ukazatel
h	Při konverzích d, i, o, u, x, X bude odpovídající parametr pokládán za hodnotu typu **short**
l	Při konverzích d, i, o, u, x, X bude odpovídající parametr pokládán za hodnotu typu **long** resp. **unsigned long**. Při konverzích e, E, f, g, G bude odpovídající parametr pokládán za hodnotu typu **double**.
L	Při konverzích e, E, f, g, G bude odpovídající parametr pokládán za hodnotu typu **long double**

Tab. 13.2 Modifikátory velikosti typu

Podíváme se opět na jednoduchý příklad. Vytiskneme si tabulku funkce faktoriál[27] pro čísla od 9 do 12. Program bude vypadat takto:

```
/*   Příklad C13 – 3   */
#include <stdio.h>

unsigned long Fakt(int n){
  unsigned long s = 1;
  for(int k = 1; k <= n; k++) s *= k;
  return s;
}

int main(){
  printf("N    N!\n---------\n");
  for(int i = 9; i < 13; i++)
  printf("%d  %lu\n",i,Fakt(i) );
  return 0;
}
```

Dostaneme

```
N    N!
---------
9  362880
10  3628800
11  39916800
12  479001600
```

Zkuste si, co by se stalo, kdybychom modifikátor velikosti **l** vynechali.

Šířka

Poslední tabulka ukazuje, že potřebujeme umět také zadat počet znaků, které má vystupující hodnota zabírat. Zadáváme ji jako celé číslo bez znaménka za znakem „%“.

Jestliže upravíme v předchozím příkladu příkazy pro výstup takto,

```
printf(" N          N!\n--------------\n");
```

[27] Faktoriál přirozeného čísla *N* se značí *N*! a je to součin 1.2 ... *N*; faktoriál 0 je roven 1.

```
for(int i = 9; i < 13; i++)
printf("%2d%12lu\n",i,Fakt(i) );
```

dostaneme již daleko elegantnější výstup

```
 N          N!
--------------
 9      362880
10     3628800
11    39916800
12   479001600
```

Předepíšeme-li šířku *n* znaků, vystoupí vždy **alespoň** *n* znaků. Má-li vystupující hodnota po konverzi méně znaků, doplní se zleva mezerami. (Použijeme-li příznak „-“, doplní se zprava.)

Jestliže zadáme šířku číslem tvaru 0*n* (začne nulou), použijí se k doplnění nuly.

Šířku lze také zadat nepřímo, prostřednictvím jednoho z parametrů funkce *printf( )*, předávaných na místě výpustky. Jestliže zadáme jako specifikaci šířky znak „*“ (hvězdičku), předpokládá se, že šířku obsahuje parametr, který „je na řadě“ (který by jinak měl právě vystupovat). Za parametrem, obsahujícím šířku, bude teprve následovat vystupující hodnota.

Přesnost

Přesnost zadáváme vždy ve tvaru

```
.n
```

tedy jako celé číslo bez znaménka, před kterým je tečka. (Ta odlišuje specifikaci přesnosti od zadání šířky.)

Význam přesnosti se pro jednotlivé konverze liší. Podívejme se nejprve, co způsobí zadání přesnosti .*n* pro *n* nenulové.

Pro konverze **e**, **E** a **f** předepisuje, že vystoupí *n* číslic za desetinnou tečkou, přičemž poslední vznikne zaokrouhlením.

Pro konverze **g** a **G** předepisuje, že vystoupí nejvýše *n* platných číslic.

Pro konverze **d**, **i**, **o**, **u**, **x** a **X** předepisuje, že vystoupí alespoň *n* číslic; v případě potřeby bude vystupující hodnota doplněna zleva nulami. Pokud má vystupující hodnota více číslic než stanoví přesnost, vystoupí všechny.

Pro konverzi **s** přesnost určuje, že nebude vytištěno více než *n* znaků.

Jestliže přesnost nezadáme, použijí se implicitní hodnoty, uvedené v tabulce 13.3.

Konverze	Implicitní přesnost
d, i, o, u, x, X	1
e, E, f	6
g, G	Všechny platné číslice
s	Všechny znaku po první '\0'

Konverze	Implicitní přesnost
c	Nemá význam

Tab. 13.3 Implicitní hodnota přesnosti

Zadáme-li přesnost .0, znamená to pro konverze **d**, **i**, **o**, **u**, **x** a **X** použití implicitní hodnoty 1 a pro typy **e**, **E** a **f**, že nebude vystupovat desetinná tečka.

Zadáme-li přesnost zápisem „.*“, přečte si ji funkce *printf()* z parametru, který je právě „na řadě“, podobně jako v případě nepřímého zadání šířky. Formátován bude až následující parametr.

Příznaky

V Borland C++ můžeme jako příznaky použít znaky „-“, „+“, „ “ (mezera) a „#“.

Příznak „-“ (minus) předepisuje zarovnání výstupu ve výstupním poli vlevo (připadá v úvahu, pokud jsme předepsali větší šířku výstupního pole než byl počet znaků vystupující hodnoty).

Příznak „+“ způsobí, že při výstupu čísel, která mohou mít znaménko (konverze **d**, **i**, **e**, **E**, **f**, **g**) se bude vypisovat i kladné znaménko.

Příznak „ “ (mezera) způsobí, že při výstupu hodnot se znaménkem se znak „+“ nahradí mezerou. Jinými slovy, kladná čísla budou vystupovat bez znaménka. Použijeme-li zároveň příznaky „+“ a „ “, platí „+“.

Nejsložitější je význam příznaku „#“. Při výstupu čísel v osmičkové soustavě (konverze **o**) způsobí, že číslo bude začínat znakem 0. Při výstupu čísel v šestnáctkové soustavě (konverze **x** a **X**) způsobí, že číslo bude začínat znaky 0x resp. 0X.

Při výstupu reálných čísel (konverze **e**, **E**, **f**) bude výsledek vždy obsahovat desetinnou tečku, i v případě, že za ní nenásledují žádné číslice. Při výstupu pomocí konverzí **g** nebo **G** navíc nebudou odstraněny koncové nuly.

Jako příklad si tentokrát vezmeme program, který nám vytiskne tabulku funkce sinus s krokem 10°. (Prototyp funkce *sin()* a řady dalších běžných matematických funkcí najdeme v hlavičkovém souboru *math.h*. Tam najdeme i makro *M_PI*, které se rozvine v hodnotu Ludolfova čísla π.)

```
/*   Příklad C13 – 4   */
#include <stdio.h>
#include <math.h>
#include <conio.h>

int main(){
  clrscr();
  double x = M_PI/18;
  printf(" x   sin x\n--------------\n");
  for(int i = 0; i < 10; i++)
  printf("%2i\xf8%10.6f\n", i*10, sin(x*i) );
  return 0;
}
```

První řádky výstupu tohoto programu budou

```
 x   sin x
-------------
 0°  0.000000
10°  0.173648
20°  0.342020
```

Změníme-li specifikaci formátu výstupu hodnoty sinu na

```
"%-10.6f"
```

tj. použijeme-li příznak „-“, změní se výstup na

```
 x   sin x
-------------
 0°0.000000
10°0.173648
```

neboť vystupující čísla budou zarovnána doleva. Zkuste si s tímto programem experimentovat, měnit šířku a přesnost výstupu a používat různé příznaky, abyste si zvykli na tento způsob výstupu.

Další funkce pro výstup do stdout

O dalších funkcích, které lze použít k výstupu do standardního výstupního proudu, se zmíníme pouze telegraficky.

Pro výstup jediného znaku můžeme použít funkci[28]

int *putchar*(**int** *c*);

tato funkce vrátí buď hodnoty vypsaného znaku nebo *EOF*.

Pro výstup znakového řetězce můžeme použít funkci

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

Tato funkce vloží do standardního výstupního proudu řetězec *s* až po prázdný znak '\0' a přidá znak '\n' (přechod na nový řádek).

Vstup ze souboru stdin

Také prostředky pro vstup ze standardního vstupního souboru *stdin* najdeme v hlavičkovém souboru *stdio.h*. (Připomeňme si, že standardní vstup je implicitně napojen na klávesnici, lze jej ale příkazy operačního systému přesměrovat na jiný soubor nebo zařízení.)

Funkce *scanf()*

Základním nástrojem, používaným v jazyku C pro vstup, je funkce *scanf( )*. V mnoha ohledech se s ní zachází podobně jako s funkcí *printf( )* (také má jako první parametr for-

[28] Ve skutečnosti jde o makro, které se rozvine ve volání jiné funkce.

mátovací řetězec), je tu ale jeden velice důležitý rozdíl: **její parametry musí být adresy proměnných, do kterých chceme přečtené hodnoty uložit** Její prototyp je

int *scanf*(**const char** **format*, ...);

Tato funkce čte data ze vstupu znak po znaku, na základě informací, nalezených v řetězci *format*, je konvertuje na hodnotu předepsaného typu a výsledek uloží do proměnné, jejíž adresu dostane jako další parametr.

Podívejme se na jednoduchý příklad. Chceme-li ze standardního vstupu přečíst hodnotu typu **int** a uložit ji do proměnné *i*, napíšeme v programu příkaz

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

Ve formátovacím řetězci se mohou vyskytnout tři typy údajů:

- ✧ Bílé znaky: způsobí, že funkce *scanf( )* přeskočí na vstupu všechny následující bílé znaky.
- ✧ Formátové specifikace (konverze): začínají znakem „%“, podobně jako u funkce *printf( )*. Určují, jakého typu bude následující hodnota, jak má být konvertována a zda bude uložena do proměnné, určené následujícím parametrem.
- ✧ Ostatní znaky: pokud najde funkce *scanf( )* ve formátovacím řetězci jiné znaky než bílé nebo specifikaci konverze, očekává, že tytéž znaky najde i ve vstupním proudu. Přečte je a zapomene je.

Obecný tvar formátové specifikace (specifikace konverze) pro funkci *scanf( )* je

```
%[*][šířka][modifikátor_ukazatele][modifikátor_velikosti]typ
```

a závorky „[ ]“ opět označují části, které lze vynechat. Podívejme se na jednotlivé součásti.

Hvězdička v úvodu potlačuje přiřazení. To znamená, že hodnota se sice ze vstupního proudu přečte, ale nepřiřadí se následující proměnné ve vstupním seznamu (proměnné, jejíž adresu jsme funkci *scanf()* předali jako parametr na místě výpustky a do které by měla jinak přečtenou hodnotu uložit.

Šířka určuje maximální počet znaků, které se ve vstupním proudu přečtou. Čtení může skončit i dříve, pokud funkce *scanf( )* narazí na bílý znak nebo na znak, který nepatří do zápisu čtené hodnoty.

Jako *modifikátor_ukazatele* můžeme použít **N** nebo **F** (jako u funkce *printf( )*). Určuje explicitně, zda je předáván blízký nebo vzdálený ukazatel na proměnnou, do které se má přečtená hodnota uložit).

Jako *modifikátor_velikosti* můžeme použít znaků **l**, **L** a **h**. Modifikátor **h** ve spojení s celočíselnými typy označuje typ **short**. Modifikátor **l** ve spojení s konverzemi **d**, **o**, **i** označuje typ **long**, ve spojení s konverzí **u** typ **unsigned long** a ve spojení s konverzemi **e**, **E**, **f**, **g** a **G** typ **double**. Modifikátor **L** označuje ve spojení s konverzemi **e**, **E**, **f**, **g** a **G** typ **long double**.

Označení typu, které je kromě znaku „%“ jedinou povinnou částí formátové specifikace, shrnuje tabulka 13.4

Typ	Očekávaný vstup	Očekávaný typ parametru
d	Desítkové celé číslo	Ukazatel na **int**
D	Desítkové celé číslo	Ukazatel na **long**
o	Osmičkové celé číslo	Ukazatel na **int**
O	Osmičkové celé číslo	Ukazatel na **long**
x	Šestnáctkové celé číslo	Ukazatel na **int**
X	Šestnáctkové celé číslo	Ukazatel na **long**
i	Desítkové, osmičkové nebo šestnáctkové celé číslo	Ukazatel na **int**
I	Desítkové, osmičkové nebo šestnáctkové celé číslo	Ukazatel na **long**
u	Desítkové celé číslo bez znaménka	Ukazatel na **unsigned**
U	Desítkové celé číslo bez znaménka	Ukazatel na **unsigned long**
e, E	Reálné číslo	Ukazatel na **float**
f	Reálné číslo	Ukazatel na **float**
g, G	Reálné číslo	Ukazatel na **float**
s	Řetězec znaků	Ukazatel na **char** (musí ukazovat na dostatečně velké pole)
[*množina***]**	Řetězec znaků	Ukazatel na **char** (musí ukazovat na dostatečně velké pole)
c	Znak	Ukazatel na **char**
p	Ukazatel[29]	Ukazatel na ukazatel
n		Ukazatel na **int**[30]

Tab. 13.4 Specifikace typu pro funkci scanf()

Dodejme ještě, že:

- ✧ Místo konverze %**D** můžeme použít %**ld** – obojí znamená čtení do proměnné typu **long int**. Podobně i pro ostatní celočíselné konverze.
- ✧ Konverze %**d**, %**D**, %**x**, %**X**, %**o** a %**O** očekávají na vstupu celé číslo v odpovídající číselné soustavě. Číslo v osmičkové soustavě nemusí začínat 0, číslo v šestnáctkové soustavě nemusí začínat 0*x* nebo 0*X*. Konverze %**i** a %**I** si poradí se kteroukoli ze tří

[29] Zápis ukazatele ve tvaru *Seg*:*Ofs* nebo jen *Ofs* (podle paměťového modelu) v šestnáctkové soustavě.
[30] Do **n* se uloží počet dosud úspěšně přečtených znaků.

dovolených soustav, číslo v šestnáctkové soustavě musí ale začínat 0*x* nebo 0*X* a číslo v osmičkové soustavě musí začínat 0.

- ✧ Pro vstup reálných čísel jsou konverze **e**, **E**, **f**, **g** a **G** ekvivalentní. Všechny ukládají přečtenou hodnotu do proměnné typu **float**. Na vstupu očekávají řetězec, který je platným zápisem reálného čísla v programu.
- ✧ Chceme-li reálné číslo uložit do proměnné typu **double**, musíme je číst konverzí %**lf**, %**le** apod. Chceme-li je uložit do proměnné typu **long double**, musíme je přečíst konverzí %**Lf**, %**Le** apod.
- ✧ Při všech vstupech s výjimkou konverze **c** se nejprve přeskočí bílé znaky a vstupní pole začíná prvním nebílým znakem. Pak se přečtou znaky, které tvoří zápis vstupující hodnoty. Čtení skončí, jestliže funkce *scanf()* narazí na bílý znak, na znak, který nepatří do zápisu čtené hodnoty nebo jestliže se přečte *n* znaků, kde *n* je zadaná šířka.
- ✧ Použijeme-li konverzi %**c**, přečte se první následující znak, ať je bílý či jiný. Použijeme-li konverzi **%***n***c**, přečte se *n* znaků (včetně bílých) a ty se uloží do pole.
- ✧ Při čtení pomocí konverze **s** se přeskočí úvodní bílé znaky a přečte se řetězec. Čtení skončí, narazí-li funkce *scanf()* na bílý znak (mezeru, nový řádek atd.) nebo vyčerpá-li se povolená šířka.

V následujícím příkladu prostě jen přečteme několik hodnot a pak je zase vypíšeme. Doporučujeme vám, abyste si tento program spustili a zkoušeli mu zadávat různé (i chybné) hodnoty a sledovali, jak se bude chovat.

```
/*   Příklad C13 – 5   */
// Použití funkce scanf
#include <stdio.h>

int i;
long l;
double r;
char C[100];
char D[100];

int main(){
  printf("zadej celé číslo: ");
  scanf("%i",&i);
  printf("zadej celé číslo typu long: ");
  scanf("%I", &l);
  printf("zadej reálné číslo (double): ");
  scanf("%lf", &r);
  printf("zadej řetězec bez mezer: ");
  scanf("%s", C);
  printf("zadej řetězec s mezerami (10 znaků): ");
  scanf("%10c", D);
  printf("\zadané hodnoty: %d,  %ld,  %e, \n"
         "%s\n%s", i, l, r, C, D);
  return 0;
}
```

Všimněte si, že pokud zadáte např. jako celé číslo typu **long** hodnot 12L, přečte se jen 12 a L zůstane ve vstupním bufferu. Při následujícím čtení hodnoty typu **double** od vás počítač nebude nic chtít: v bufferu má stále "L", takže proměnné *r* přiřadí 0.0 a začne rovnou číst řetězec *C*. Navíc při zadávání posledního řetězce *D* (v němž jsou dovoleny mezery) bude nekompromisně vyžadovat oněch 10 znaků.

Jestliže ukončíte řetězec *C* (ten čteme jako první, nedovoluje mezery ani bílé znaky) stisknutím klávesy ENTER, bude řetězec *D* znakem ENTER – přechodem na nový řádek – začínat. To proto, že tento znak sice čtení řetězce *C* ukončil, ale sám zůstal nepřečtený ve vstupním proudu, a konverze **c** (jako jediná) čte i bílé znaky.

Další funkce pro vstup ze stdin

Z dalších funkcí pro vstup ze souboru *stdin* se zmíníme o dvou. Chceme-li přečíst jediný znak, můžeme použít funkci[31]

int *getchar*(**void**);

Přečtený znak vrátí – jak je v jazyku C obvyklé – jako hodnotu typu **int**.

Pro čtení celého řetězce můžeme použít funkci

char* *gets*(**char** **s*);

Vstupující řetězec může obsahovat mezery, ukončí jej až přechod na nový řádek. Přečtený řetězec uloží do pole, na jehož počátek ukazuje *s*. Pokud poběhne čtení bez problémů, vrátí ukazatel na přečtená data (tedy *s*), jinak vrátí 0.

Práce se soubory

Jazyk C nabízí dvě možnosti práce se soubory. První, historicky starší, používá identifikační čísla (*handle*), druhá, novější, je založena na předefinované datové struktuře *FILE*. Začneme druhou, používanější.

Soubory a struktura FILE

V souboru *stdio.h* je definován datový typ *FILE*. Popisuje strukturu, do které se uloží data, potřebná pro práci s externím souborem.

Struktura *FILE*

Chceme-li tedy pracovat se soubory, začneme tím, že v programu definujeme **ukazatel na strukturu** *FILE*, např.[32]

---

[31] Opět jde ve skutečnosti o makro, které volá jinou funkci.

[32] Typ *FILE* je deklarován prostřednictvím deklarace **typedef**. To znamená, že i v jazyku C se na něj odvoláváme pouze identifikátorem *FILE*, nikoli **struct** *FILE*.

```
FILE * f;
```

Tato proměnná *f* představuje logický datový proud. Hraje podobnou roli jako logický soubor v Pascalu: budeme ji předávat jako parametr funkcím, které budou se souborem něco provádět.

Otevření a zavření souboru

Dále potřebujeme sdělit programu, na který fyzický soubor chceme náš datový proud připojit a jakým způsobem chceme soubor otevřít. K tomu nám poslouží funkce

FILE * *fopen*(**const char** **jmeno_soub*, **const char** **rezim*);

Tato funkce otevře soubor, jehož jméno je v řetězci *jmeno_soub*, v režimu *rezim*, a připojí ho k našemu datovému proudu. To znamená: pokud se tato operace podaří, vytvoří v paměti strukturu *FILE*, naplní ji daty a vrátí ukazatel na ni. Pokud se operace nepodaří, vrátí *NULL*.

Režim otevření souboru je určen řetězcem *rezim*. Jeho možné hodnoty najdete v tabulce 13.6.

rezim	Význam
"r"	Otevře soubor pouze pro čtení (vstup). Pokud neexistuje, nastane chyba.
"w"	Vytvoří a otevře soubor pro zápis. Pokud soubor neexistuje, vytvoří ho, pokud existuje, smaže jeho obsah.
"a"	Otevře soubor pro připisování na konci. Pokud soubor neexistuje, vytvoří ho.
"r+"	Otevře soubor pro aktualizaci (pro čtení i zápis). Pokud neexistuje, nastane chyba.
"w+"	Vytvoří a otevře soubor pro aktualizaci. Pokud soubor již existuje, smaže jeho obsah.
"a+"	Otevře soubor pro aktualizaci na konci. Pokud neexistuje, vytvoří jej.
"b"	Tento řetězec se připojuje k předchozím. Předepisuje otevření souboru v binárním režimu.
"t"	Podobné jako **"b"**, předepisuje však otevření v textovém režimu.

Tab. 13.6 Význam parametru *retezec* ve funkci *fopen( )*

Chceme-li tedy otevřít soubor *C:\WORK\DATA.DTA* v textovém režimu pro čtení, napíšeme

```
f = fopen("C:\\WORK\\DATA.DTA", "rt");
```

Poté musíme zkontrolovat, zda se operace podařila. Pokud došlo k chybě, tj. pokud se soubor nepodařilo otevřít (třeba proto, že neexistuje), bude v proměnné *f* hodnota *NULL*. Často se otevření souboru vkládá přímo do podmínky:

```
FILE *f;
char *jmeno = "C:\\WORK\\DATA.DTA";
if(!(f = fopen(jmeno, "rt")))
{
  printf("Nepodařilo se otevřít soubor %s\n", jmeno);
  // ... a další akce ...
}
// ...
```

Jakmile práci se souborem skončíme, uzavřeme jej. K tomu použijeme funkci

int *fclose*(*FILE* proud*);

jejímž parametrem je uzavíraný datový proud (tedy ukazatel na strukturu *FILE*, která tento proud určuje).

Poznámka:
Pokud při volání funkce fopen() *v parametru* rezim *neuvedeme, zda chceme soubor otevřít v binárním nebo textovém režimu, použije se nastavení, předepsané globální proměnnou* _fmode, *do které můžeme uložit jednu z hodnot* O_TEXT *nebo* O_BINARY.

Formátovaný vstup a výstup

Formátovaný vstup a výstup se používá zpravidla pro práci s textovými soubory, není to ovšem podmínkou. Pro formátovaný výstup do otevřeného souboru slouží funkce

int *fprintf*(*FILE *proud*, **const char ***format, ...);

Jejím prvním parametrem je proud, do kterého zapisujeme, další parametry (a také ostatní vlastnosti) jsou stejné jako u funkce *printf()*.

Pro formátované čtení ze souboru slouží funkce

int *fscanf*(*FILE *proud*, **const char ***format, ...);

Jejím prvním parametrem je proud, ze kterého čteme, další parametry jsou stejné jako u funkce *scanf()*.

Pro výstup jednotlivých znaků můžeme použít funkci

int *fputc*(**int** *c*, *FILE* proud*);

která vypíše do zadaného souboru znak *c*. Pokud se zápis podařil, vrátí *c*, jinak vrátí *EOF*.

Chceme-li přečíst jeden znak, můžeme použít funkci

int *fgetc*(*FILE* proud*);

která vrátí buď hodnotu přečteného znaku nebo *EOF*.

Někdy se stane, že potřebujeme právě přečtený znak vrátit do vstupního proudu – tedy tvářit se, jako kdybychom jej nepřečetli. Zde nám jazyk C nabízí funkci

int *ungetc*(**int** c, *FILE* proud*);

Tato funkce vrátí daný znak do vstupního proudu, odkud jej přečte příští volání *fgetc()* nebo některé jiné z čtecích funkcí, které využívají služeb *fgetc()*.

Vrátíme-li více znaků (tj. nebude-li po volání *ungetc()* následovat čtení, ale další volání *ungetc()*), vrátí se do proudu pouze poslední znak, předaný funkci *ungetc()*.

Chceme-li vypsat celý řetězec, můžeme použít funkci

int *fputs*(**const char*** *retezec*, *FILE* proud*);

Vypisovaný řetězec *retezec* musí končit prázdným znakem '\0'. Pokud se akce podaří, vrátí tato funkce poslední vypsaný znak, jinak vrátí *EOF*.

Celý znakový řetězec můžeme přečíst pomocí funkce

char * *fgets*(**char *** *retez*, **int** *n*, *FILE* proud*);

Tato funkce přečte ze vstupního proudu *proud* znakový řetězec o délce nejvýše *n*-1 znaků (pokud narazí na přechod na novou řádku, skončí dříve), připojí k němu '\0' a výsledek uloží do pole *retez*. Pokud se vše podaří, vrátí ukazatel na *retez*, jinak vrátí 0.

Neformátovaný vstup a výstup

Neformátované operace se používají obvykle při práci s binárními soubory. Pro neformátovaný výstup můžeme použít funkci[33]

size_t fwrite(**const void ***ptr, *size_t vel*, *size_t n*, *FILE* proud*);

Tato funkce předpokládá, že zapisujeme data z pole do souboru; první parametr, *ptr*, je ukazatel na začátek tohoto pole. Druhý parametr udává velikost jedné složky pole, třetí parametr pak počet složek pole, které chceme zapsat, a čtvrtý parametr je proud, do kterého zapisujeme. Pokud se tedy vše podaří, vypíše se celkem (*n * vel*) bajtů. Na rozdíl od formátovaného zápisu (funkce *fprintf()*) funkce *fwrite()* prostě kopíruje obsah paměti do souboru. Vrátí počet vypsaných položek (nikoli bajtů).

Pro čtení můžeme použít funkci

size_t fread(**void ***ptr, *size_t vel*, *size_t n*, *FILE *proud*);

Tato funkce předpokládá, že čteme data ze souboru do pole, a první parametr pokládá za adresu jeho počátku. Význam dalších parametrů je stejný jako u funkce *fwrite()*. Celkem se přečte (*n * vel*) bajtů. Na rozdíl od formátovaného zápisu tato funkce prostě kopíruje binární data ze souboru do paměti. Vrátí počet opravdu přečtených položek.

---

[33] *size_t* je datový typ (typedef) pro vyjadřování velikosti. V Borland C++ je totožný s typem **unsigned**.

Pozice v souboru

Předchozí funkce nám umožňovaly sekvenční zpracování souborů – tedy čtení nebo zápis jeden bajt po druhém resp. jeden záznam po druhém. Vzhledem k tomu, že soubory na disku jsou semisekvenční, potřebujeme také umět zjistit aktuální pozici v souboru a případně ji změnit. K tomu můžeme použít mj. funkce

int *fgetpos*(*FILE *proud*, *fpos_t *pozice*);
int *fsetpos*(*FILE *proud*, **const** *fpos_t *pozice*);

Funkce *fgetpos( )* uloží do proměnné *pozice* aktuální pozici v souboru. Formát této informace není blíže specifikován, takže ji lze použít pouze v následujícím volání funkce *fsetpos( )*. V případě úspěchu vracejí obě tyto funkce hodnotu 0, v případě neúspěchu nenulovou hodnotu.

Vedle toho můžeme použít funkci

long *ftell*(*FILE *proud*)

jež vrátí aktuální polohu v binárním souboru v bajtech od jeho počátku. Pokud se operace nepovede, vrátí -1.

K nastavení polohy můžeme použít funkci

int *fs0eek*(*FILE *proud*, *long oKolik*, **int** *Odkud*);

Tato funkce nastaví aktuální pozici v daném proudu na novou hodnotu. Její druhý parametr určuje, o kolik bajtů se má pozice změnit, a třetí parametr určuje, odkud se má změna počítat. Můžeme zde použít jednu z hodnot, uvedených v tabulce 13.7

Hodnota	Identifikátor	Význam
0	*SEEK_SET*	*oKolik* se počítá od počátku souboru
1	*SEEK_CUR*	*oKolik* se počítá od aktuální polohy v souboru
2	*SEEK_END*	*oKolik* se počítá od konce souboru

Tab. 13.7 Význam třetího parametru funkce *fseek( )*

Pokud se přemístění podaří, vrátí tato funkce 0, jinak vrátí nenulovou hodnotu. Ve skutečnosti ovšem ohlásí chybu pouze v případě, že zapomeneme otevřít soubor; opírá se totiž o služby DOSu, a tak nemůže ohlásit chyby, které jí neohlásí DOS.

Příklad

Práci se soubory pomocí struktury *FILE* si ukážeme na jednoduchém příkladu. Vytvoříme v aktuálním adresáři binární soubor *DATA.DTA* a zapíšeme do něj čísla od 0 do 9 (v binárním tvaru). Pak tento soubor otevřeme pro čtení, jeho obsah přečteme (a budeme se přitom tvářit, že nevíme, kolik čísel v souboru je) a zapíšeme je do textového souboru *DATA.TXT*. Při zápisu do textového souboru budeme výstup pochopitelně formátovat: každé číslo bude na jednom řádku a bude mít vyhrazeno 5 znaků.

```
/*   Příklad C13 – 6   */
/* Příklad použití neformátovaných a formátovaných vstupů
   a výstupů z jazyka C pomocí struktury FILE
*/
#include <stdio.h>
#include <stdlib.h>

void chyba(int kod){ /* ... */ }

int main(){
  FILE* f;
  FILE* g;           /* Deklarace proudů a otevření jednoho z nich */
  int i;

  f = fopen("data.dta", "wb");
  if(!f) chyba(1);
  for(i = 0; i < 10; i++)            /* Neformátovaný zápis do */
  fwrite(&i, sizeof(int), 1, f);          /* binárního souboru */
  if(fclose(f))chyba(2);             /* Pak soubor uzavřame */

  /* a znovu otevřeme, tentokrát pro čtení */
  /* Zároveň otevřeme další soubor pro zápis v textovém režimu */
  if(((f = fopen("data.dta", "rb")) == NULL)) chyba(1);
  if(((g = fopen("data.txt", "wt")) == NULL)) chyba(1);

  /* Dokud se čtení daří, vrací funkce fread nenulovou hodnotu
     při prvním pokusu o čtení za koncem souboru vrátí 0 - podmínka
     opakování cyklu nebude splněna */
  while(fread(&i, sizeof(int), 1, f)){
      fprintf(g,"  %5d\n", i);
  }
  if(fclose(f))chyba(2);
  if(fclose(g))chyba(2);
  return 0;
}
```

Soubory a identifikační čísla

Ovládání souborů pomocí identifikačních čísel (*handle*) se obvykle používá při programování „nižší úrovně“ (tj. blíže hardwaru). Základní nástroje, které se k tomu používají, jsou deklarovány v hlavičkovém souboru *io.h,* některé další ve *fcntl.h* a v *sys\stat.h*.

Otevření a uzavření souboru

K otevření souboru použijeme funkci

int *open*(**const char** **cesta*, **int** *pristup*, .../* **unsigned** *rezim* */);

která vrátí identifikační číslo souboru (nebo -1, pokud se akce nepodaří).

První parametr udává jméno a cestu souboru. Druhý způsob specifikuje zacházení se souborem; sestavíme ho jako bitový součet (pomocí operátoru „|“) z hodnot, uvedených v tabulkách 13.8 a 13.9; jsou definovány ve *fcntl.h*.

Příznak	Význam
O_RDONLY	Otevře soubor pouze pro čtení
O_WRONLY	Otevře soubor pouze pro zápis
OR_DWR	Otevře soubor pro čtení i zápis

Tab. 13.8 Otevření souboru pro čtení nebo zápis funkcí *open()*

Příznak	Význam
O_APPEND	Před každým zápisem bude ukazatel na aktuální pozici nastaven na konec souboru
O_CREAT	Pokud soubor existuje, nemá význam. Pokud neexistuje, vytvoří se
O_EXCL	Výlučné otevření. Používá se pouze spolu s *O_CREAT*. Pokud soubor již existuje, vrátí se příznak chyby
Příznak	**Význam**
O_TRUNC	Pokud soubor již existuje, smaže se jeho obsah
O_BINARY	Otevírá soubor v binárním režimu
O_TEXT	Otevírá soubor v textovém režimu

Tab. 13.9 Další příznaky pro otevření souboru funkcí *open()*

Použijeme-li možnost *O_CREAT*, můžeme jako třetí parametr můžeme uvést číslo *rezim* typu **unsigned**, které udává počáteční stav souboru. Tyto příznaky ukazuje tab. 13.10 a jsou definovány v *sys\stat.h*.

Příznak	Význam
S_IWRITE	Do souboru můžeme zapisovat
S_IREAD	Ze souboru smíme číst
S_IREAD|S_IWRITE	Povoluje čtení i zápis

Tab. 13.10 Možné hodnoty parametru *rezim* funkce *open()*

Poznamenejme, že standardní soubory *stdin*, *stdout* a *stderr*[34] mají po řadě identifikační čísla 0, 1 a 2.

Chceme-li vytvořit nový soubor, můžeme také použít funkci

int *creat*(**const char** **cesta*, **int** *rezim*);

Tato funkce vytvoří nový soubor nebo přepíše existující. První parametr obsahuje cestu a jméno souboru, druhý popisuje způsob zacházení se souborem (použijeme opět příznaků z tab. 13.10). Podle obsahu globální proměnné *_fmode* se určí, zda se má soubor otevřít ja-

[34] Soubor pro chybový výstup, na PC směřuje na obrazovku a je nepřesměrovatelný.

ko textový nebo binární. Tato proměnná může obsahovat jednu z konstant *O_TEXT* nebo *O_BINARY*.

Otevřený soubor uzavřeme voláním funkce

int *close*(**int** *handle*);

V případě, že se soubor podaří uzavřít, vrátí tato funkce 0; pokud ne, vrátí -1. Pozor, tato funkce nevloží na konec textového souboru znak '\1a', který označuje konec souboru.

Čtení a zápis

Pro zápis do souboru slouží funkce

int *write*(**int** *handle*, **void** **buf*, **unsigned** *delka*);

Druhý parametr, *buf*, je ukazatel na počátek oblasti v paměti, která obsahuje zapisovaná data. Do souboru se přenese *delka* bajtů počínaje touto adresou. V případě úspěchu vrátí tato funkce počet opravdu zapsaných bajtů, v případě chyby vrátí -1.

Ke čtení můžeme použít funkci

int *read*(**int** *handle*, **void** **buf*, **unsigned** *delka*);

která se pokusí přečíst *delka* bajtů z daného souboru a uložit je v paměti počínaje adresou *buf*. V textovém režimu pokládá znak '\1A' za konec souboru. V případě úspěšného čtení vrátí tato funkce počet přečtených bajtů; pokud narazí na konec souboru, vrátí 0, při jiné chybě vrátí -1.

Pozice v souboru

Aktuální pozici v souboru (v bajtech) můžeme zjistit pomocí funkce

long tell(**int** *handle*);

V případě chyby vrátí tato funkce -1.

Chceme-li aktuální pozici v souboru změnit, použijeme funkci

long *lseek*(**int** *handle*, **long** *oKolik*, **int** *Odkud*);

Tato funkce nastaví ukazatel na aktuální pozici v souboru na bajt s relativní polohou *oKolik* vztaženou k poloze *Odkud*. K zadání třetího parametru můžeme použít konstant z tabulky 13.7.

Pokud se přesun aktuální pozice podaří, vrátí tato funkce novou aktuální polohu, vztaženou k počátku souboru. V případě chyby vrátí -1L.

Velikost souboru

Pomocí funkce
long *filelength*(**int** *handle*);
zjistíme velikost souboru v bajtech. (Pokud se operace nezdaří, vrátí tato funkce -1.)

Příklad

Podívejme se na jednoduchý příklad. Bude velice podobný příkladu v závěru povídání o práci se soubory pomocí struktury *FILE*. Vytvoříme v aktuálním adresáři binární soubor *DATA.DTA* a zapíšeme do něj čísla od 0 do 19 (v binárním tvaru). Pak tento soubor otevřeme pro čtení, jeho obsah přečteme – budeme se přitom opět tvářit, že nevíme, kolik čísel v souboru je – a zapíšeme je do textového souboru *DATA.TXT*. Tentokrát ovšem použijeme k zápisu do textového souboru funkci *write( )*, která nejenže neumožňuje formátování, ale kopíruje do souboru binární obsah paměti. Pouze před číslo 10 vloží 1 bajt, obsahující číslo 13 (10 je kód znaku '\n', při výstupu do textového souboru se před něj přidá znak '\r'…).

```
/*   Příklad C13 – 7   */
/* Příklad použití neformátovaných vstupů a
   výstupů z jazyka C pomocí identifikačních čísel (handlů)
*/
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>

void chyba(int kod){ /* ... */ }

int main(){
   int hf = open("DATA.DTA", O_CREAT|O_BINARY, S_IREAD|S_IWRITE );
   int hg;            /* deklarace handlů a otevření jednoho souboru */
   int i;
   if(hf == -1) chyba(1);
   for(i = 0; i < 20; i++)            /* Neformátovaný zápis */
       write( hf, &i, sizeof(int) );  /* do binárního souboru */
   if( close(hf) == -1) chyba(2);          /* Pak soubor uzavřeme */

   /* a znovu otevřeme, tentokrát pro čtení */
   /* Zároveň otevřeme další soubor pro zápis v textovém režimu */
   hf = open("DATA.DTA", O_CREAT|O_BINARY);
   if((hf == -1)) chyba(1);
   if(((hg = open("data.txt", O_CREAT|O_RDWR|O_TEXT,S_IWRITE|S_IREAD))
                                                          == -1))
       chyba(1);

   /* Dokud se čtení daří, vrací funkce fread nenulovou hodnotu
      při prvním pokusu o čtení za koncem souboru vrátí 0 - podmínka
      opakování cyklu nebude splněna
   */
   while(read( hf, &i, sizeof(int)) > 0)
       write( hg,&i, sizeof(int) );
   if(close(hf))chyba(2);
   if(close(hg))chyba(2);
   return 0;
}
```

Paměťové proudy

V povídání o datových proudech jazyka C++ jsme hovořili také o datových proudech, které umožňují číst ze znakového řetězce nebo zapisovat do něj. Podobné nástroje nabízí i jazyk C; slouží k tomu funkce *sprintf( )* a *sscanf( )*, které se chovají velice podobně jako funkce *fprintf( )* a *fscanf( )*; najdeme je v hlavičkovém souboru *stdio.h*. Prototyp první z nich je

int *sprintf* (**char** **retezec*, **const char** **format* …);

Prvním parametrem je řetězec (ukazatel na znakové pole), do kterého chceme zapisovat – musí to být dostatečně dlouhé znakové pole, aby se do něj vešel celý výstup. Další parametry jsou stejné jako u funkce *fprintf( )*. Tato funkce vezme hodnotu parametru předaného na místě výpustky, vytvoří její znakovou reprezentaci podle specifikace v řetězci *format* a výsledek uloží do pole *retezec*. Na konec výstupu, po vypsání hodnot všech parametrů, připojí znak '\0'.

Tato funkce vrátí počet zapsaných bajtů, do kterého ale nepočítá koncový znak '\0'.

Podívejme se na příklad. Do znakového pole *reci* uložíme znakovou podobu Ludolfova čísla a tento řetězec pak vypíšeme funkcí *puts( )*. Pak do téhož pole uložíme nějaké další sdělení a výsledek opět vypíšeme. Protože chceme, aby se nový text připojil za původní, musíme začít zapisovat až za něj. K tomu použijeme hodnotu, vrácenou funkcí *sprintf( )*.

```
/*   Příklad C13 – 8   */
// Použití funkce sprintf()
#include <stdio.h>
#include <math.h>

char reci[100];

int main(){
  int poloha = sprintf(reci, "Číslo pí je %10.8f\n", M_PI);
  puts(reci);

/* Pomůžeme si adresovou aritmetikou, abychom nepřepsali první zápis
  */
  sprintf(reci+poloha, "A to je všechno, co vím");
  puts(reci);
  return 0;
}
```

Prototyp funkce *sscanf( )* je

int *sscanf* (**const char** **retzec*, **const char** **format*, …);

Prvním parametrem je znakový řetězec, ze kterého chceme číst. Druhým parametrem je řetězec *format*, jehož význam je stejný jako u funkce *scanf()*, a další parametry jsou adresy proměnných, do kterých chceme přečtené hodnoty uložit.

Tato funkce čte jednotlivé znaky z řetězce *retezec*, konvertuje je na vnitřní reprezentaci typu, předepsaného zápisem konverze v řetězci *format*, a výsledek uloží do odpovídajícího parametru, předaného na místě výpustky.

Jako příklad napíšeme program, kterému zadáme jako parametr v příkazové řádce reálné číslo. Pomocí funkce *sscanf( )* je ze znakového řetězce přečte, vypočte jeho druhou mocninu a tu vytiskne.

```
/*   Příklad C13 – 9   */
#include <stdio.h>

int main(int argc, char* argv[])
{
  double d = -1;
  if(argc < 2)                  /* Nedali jsme žádný parametr */
       printf("To chce zadat číslo...");
  else
       sscanf(argv[1], "%lf", &d);
  printf("\ndruhá mocnina parametru je %f\n", d*d);
  return 0;
}
```

Práce s konzolou

Borland C++ obsahuje také prostředky pro přímé čtení z klávesnice a přímý[35] výstup na obrazovku – tedy pro práci s konzolou. Najdeme je v hlavičkovém souboru *conio.h*. Většina z nich je velice podobná nástrojům, se kterými jsme se seznámili v kapitole 11, a proto si o nich povíme jen velice stručně.

Poznamenejme, že knihovna (modul) *conio* je velice přesnou analogií pascalské knihovny *crt*; je tu ale jeden důležitý rozdíl. Jestliže se v Pascalu rozhodneme použít knihovnu *crt*, ztratíme možnost používat standardní vstupy a výstupy. Naproti tomu použití funkcí z knihovny *conio* naprosto nevylučuje možnost zároveň používat funkce z knihovny *stdio*.

Základní funkce

Pro zápis na obrazovku máme k dispozici funkci

int *cprintf*(**const char** **format*, …);

[35] Výstup na obrazovku může probíhat buď přímo, tj. tak, že dále uvedené funkce zapisují bezprostředně do obrazovkové paměti, nebo prostřednictvím služeb operačního systému (což je pomalejší). Systém určí, která z možností se použije, podle hodnoty proměnné *directvideo*. Hodnota 0 (implicitní) znamená použití služeb BIOSu, 1 znamená přímý zápis.

Její vlastnosti jsou podobné jako u funkce *printf( )*, až na to, že znak '\n' nepřevádí na dvojici '\n' '\r'. (Výstup na konzolu se nepovažuje za výstup do textového souboru.) To znamená, že příkaz

```
cprintf("Jede se dále\nmočálem černým");
```

vypíše na obrazovku

```
Jede se dále
           močálem černým
```

Abychom dostali nápisy pod sebe, musíme napsat

```
cprintf("Jede se dále\n\rmočálem černým");
```

Tento výstup také není přesměrovatelný prostředky operačního systému. Na druhé straně respektuje nastavení barev, oken apod. pomocí dalších funkcí z této knihovny.

Pro čtení z konzole můžeme použít funkci

int *cscanf*(**char** **format*, …);

První parametr je formátovací řetězec, stejný jako u funkce *scanf( )*; další parametry pak jsou adresy proměnných, do kterých se budou přečtené hodnoty ukládat.

Chceme-li zapsat jediný znak na pozici kurzoru, použijeme funkci

int *putch*(**int** *c*);

V případě úspěchu vrátí hodnotu *c*, jinak vrátí *EOF*. Pozor, ani tato funkce netransformuje znak '\n' na dvojici '\r'\n'.

Chceme-li přečíst právě jeden znak, použijeme jednu z funkcí

int *getch*(**void**);
int *getche*(**void**);

Tyto funkce způsobí, že systém bude čekat na stisknutí klávesy[36].

Jakmile stiskneme klávesu, vrátí odpovídající ASCII-kód. Funkce *getche( )* navíc vypíše odpovídající znak na obrazovku. Tyto funkce představují nebafrovaný vstup, to znamená, že zadaný znak nemůžeme po stisknutí editovat – náš program jej přebírá okamžitě.

Funkce

int *ungetch*(**int** *c*);

umožňuje vrátit jeden znak zpět do konzolové fronty. Pokud uspěje, vrátí *c*, jinak vrátí *EOF*.

Chceme-li zapsat na konzolu celý řetězec, použijeme funkci

int *cputs*(**const char** **s*);

[36] Pokud čeká nějaký znak ve frontě, vezmou si jej tyto funkce z fronty.

Tato funkce vypíše řetězec, ukončený nulou. Podobně jako ostatní funkce z knihovny *conio* nekonvertuje znak '\n' na dvojici '\r''\n'.

Chceme-li naopak přečíst celý řetězec, použijeme funkce

char ***cgets**(**char** ***str**);

která přečte řetězec z konzole a uloží jej do znakového pole *str*. Tato funkce očekává, že v prvku *str[0]* najde maximální možnou délku řetězce (tj. kolik znaků má maximálně číst). Čtení skončí tím, že narazí na dvojici znaků *CR-LF*, nebo vyčerpáním povolené délky. Narazí-li na dvojici *CR-LF* (vznikne stisknutím klávesy ENTER), nahradí ji znakem '\0'.

V prvku *str[1]* najdeme po skončení počet skutečně přečtených znaků. To znamená, že přečtený řetězec začíná až prvkem *str[2]*. V případě, že se čtení podaří, vrátí ukazatel na *str[2]*.

Funkce pro ovládání obrazovky

Funkce, o kterých zde budeme hovořit, jsou přesnou analogií stejnojmenných funkcí z pascalské knihovny *crt*, resp. metod proudu *constream*, se kterými jsme se seznámili v kap. 11. Proto se omezíme na heslovitý výčet.

Textové okno definujeme pomocí funkce

void *window*(**int** *Lhx*, **int** *LHy*, **int** *PDx*, **int** *Pdy*);

Chceme-li umístit textový kurzor do bodu *(x, y)*, zavoláme funkci

void *gotoxy*(**int** *x*, **int** *y*)

Poznamenejme, že tato funkce používá – stejně jako obě následující – relativní souřadnice vzhledem k levému hornímu rohu aktuálního textového okna.

Ke zjištění souřadnic textového kurzoru v platném textovém okně použijeme funkce

int *wherex*(**void**);
int *wherey*(**void**);

Barvu zapisovaného textu nastavíme pomocí funkce

void *textcolor*(**int** *barva*);

kde *barva* je celé číslo v rozmezí 0 – 15. Můžeme použít také konstant, definovaných v tabulce 11.2.

Barvy pozadí nastavujeme pomocí funkce

void *textbackground*(**int** *barva*);

kde *barva* je celé číslo v rozmezí 0 – 7.

Pokud chceme nastavit atributy vystupujícího textu pro popředí (znaky) i pro pozadí zároveň, použijeme funkci

void *testattr*(**int** *atribut*)

Funkce

void *gettext* (**int** *Lhx*, **int** *Lhy*, **int** *Pdx*, **int** *Pdy*, **void** **kam*);

okopíruje obsah okna, jehož levý horní a pravý dolní roh určují první 4 parametry, do pole *kam*. (Poznamenejme, že tato funkce – stejně jako následující – používá absolutní souřadnice – vzhledem k obrazovce, nikoli vzhledem k aktuálnímu oknu).

Obsah okna, získaného pomocí funkce *gettext()*, můžeme umístit kamkoli na obrazovku pomocí funkce

void *puttext* (**int** *Lhx*, **int** *Lhy*, **int** *Pdx*, **int** *Pdy*, **void** **kam*);

Význam parametrů je podobný jako u předchozí funkce.

Chceme-li smazat aktuální textové okno, použijeme funkci

void *clrscr*(**void**);

Tato funkce vyplní okno barvou pozadí.

Chceme-li smazat řádku, ve které je kurzor, zavoláme funkci

void *delline*(**void**);

Chceme-li naopak vložit novou řádku na místo kurzoru, použijeme funkci

void *insline*(**void**);

Potřebujeme-li smazat v řádce text od kurzoru do konce, poslouží nám funkce
void *clreol*(**void**);

Funkce

void *textmode*(**int** *mod*);

slouží k nastavení nového textového režimu. Jako parametry můžeme použít hodnot z tabulky 13.11.

Hodnota	Význam
LASTMODE	Obnoví předchozí režim
BW40	Černobílý, 40 sloupců
BW80	Černobílý, 80 sloupců
C40	Barevný, 40 sloupců
C80	Barevný, 80 sloupců
MONO	Monochromatický, 80 sloupců
C4350	Barevný, 43 řádek na EGA, 50 řádek na VGA

Tab. 13.11 Textové režimy

Rejstřík

S

Š

T

U

V

W

Z