

Obsah

Předpoklady	12
Terminologie	12
Typografické konvence	13
Předmluva	13
1. Ukazatele	14
1.1 Ukazatele na skalární proměnné	14
1.2 Ukazatele na pole, pole ukazatelů	19
1.3 Příklad: simulované bakterie	22
1.4 Aritmetika ukazatelů.....	22
Adresová aritmetika v C++.....	23
Adresová aritmetika v Pascalu	26
Adresování v reálném režimu procesoru 80*86.....	28
Blízké, vzdálené a normalizované ukazatele.....	29
Paměťové modely	33
1.5 Ukazatele na procedury a funkce	35
Blízké, vzdálené a robustní funkce.....	38
1.6 Dynamická alokace paměti	40
1.7 Konstantní ukazatele a ukazatele na konstanty	44
2. Záznamy, struktury a množiny	46
2.1 Deklarace záznamů a struktur	46
2.2 Příkaz with	49
2.3 Ukazatele na záznamy a struktury	50
2.4 Záznamy s variantní částí a unie	51
Pascal: záznamy s variantní částí.....	52
+: unie.....	54
2.5 Inicializace strukturovaných objektů	59
2.6 Bitová pole.....	62
Množiny	62
2.7 Množiny v Pascalu	63
Hodnoty typu množina.....	63
Operace s množinami.....	64
3. Práce se soubory a proudy	66

3.1	Obecně o souborech a proudech	66
3.2	Práce se soubory v Pascalu	72
	Příklad.....	81
3.3	Práce s datovými proudy v C++	83
	Režimy otevření proudu.....	86
	Chybové stavy.....	87
	Specifika externích datových proudů	88
	Specifika paměťových datových proudů.....	89
	Vstup dat	90
	Výstup dat	91
	Ostatní operace	92
	Příklad.....	93
3.4	Formátovaný vstup a výstup v C++	95
	Rozšíření operátorů << a >> pro typy definované uživatelem.....	96
	Šířka výstupního pole.....	97
	Zobrazení znaménka u kladných čísel.....	98
	Způsob zarovnávání.....	99
	Výplňový znak.....	99
	Základ číselné soustavy	100
	Velikost znaku při hexadecimálním výstupu a zobrazení základu číselné soustavy.....	101
	Formáty tisku reálných čísel.....	102
	Přeskakování bílých znaků na vstupu.....	104
	Splachování	104
3.5	Ladění komunikace s datovými proudy	106
4.	Podmíněný překlad a makra	108
4.1	Dolarové poznámky a příkazy preprocesoru	108
4.2	Řízení překladače v Pascalu	109
	Globální přepínače	109
	Lokální přepínače.....	111
4.3	Řízení překladače v C++.....	113
	Speciální direktivy.....	113
	Povolení a potlačení generování chybových hlášení	115
	Vlastní modifikace práce překladače.....	121
4.4	Makra.....	124
	Jednoduchá makra.....	124
	Makra s parametry	127
4.5	Podmíněný překlad	129
	Podmíněný překlad v Pascalu	129
	Podmíněný překlad v C++	131
4.6	Vkládání souborů	133
	Vkládání souborů v Pascalu.....	133
	Vkládání souborů v C++	134
5.	Podrobnosti o ladění programů	135
	Inspekční okno v prostředí Borland C++	137

6. Používání funkcí z jazyka C v C++ **138**

7. Základy práce s grafikou **140**

7.1 Zpracování chyb.....	140
7.2 Inicializace grafického systému	142
Obrazovka	142
Ovládání celého grafického systému.....	145
7.3 Barvy	148
Karty a barvy	148
Nastavování barev.....	149
7.4 Grafický kurzor	153
7.5 Kreslení.....	154
7.6 Okna	159
7.7 Výřezy	160
7.8 Výstup textu.....	161
Textový kurzor v grafickém režimu.....	162
Grafický výstup textu.....	164
7.9 Práce s videostránkami.....	164
7.10 Vyplňování	165
Vyplněné obrazce	165
Způsob vyplňování.....	165

8. Piškorky **168**

8.1 Úvodní kroky	168
Příprava.....	174
Zobraz Start	180
8.2 Hra	182

1. Ukazatele **14**

1.1 Ukazatele na skalární proměnné	14
1.2 Ukazatele na pole, pole ukazatelů	19
1.3 Příklad: simulované bakterie	22
1.4 Aritmetika ukazatelů.....	22
Adresová aritmetika v C++.....	23
Adresová aritmetika v Pascalu	26
Adresování v reálném režimu procesoru 80*86.....	28
Blízké, vzdálené a normalizované ukazatele.....	29
Paměťové modely	33
1.5 Ukazatele na procedury a funkce	35
Blízké, vzdálené a robustní funkce.....	38
1.6 Dynamická alokace paměti	40
1.7 Konstantní ukazatele a ukazatele na konstanty	44

2. Záznamy, struktury a množiny	46
2.1 Deklarace záznamů a struktur	46
2.2 Příkaz with	49
2.3 Ukazatele na záznamy a struktury	50
2.4 Záznamy s variantní částí a unie	51
Pascal: záznamy s variantní částí.....	52
+: unie.....	54
2.5 Inicializace strukturovaných objektů	59
2.6 Bitová pole.....	62
Množiny	62
2.7 Množiny v Pascalu	63
Hodnoty typu množina.....	63
Operace s množinami.....	64
3. Práce se soubory a proudy	66
3.1 Obecně o souborech a proudech	66
3.2 Práce se soubory v Pascalu	72
Příklad.....	81
3.3 Práce s datovými proudy v C++	83
Režimy otevření proudu.....	86
Chybové stavy.....	87
Specifika externích datových proudů	88
Specifika paměťových datových proudů.....	89
Vstup dat	90
Výstup dat	91
Ostatní operace	92
Příklad.....	93
3.4 Formátovaný vstup a výstup v C++	95
Rozšíření operátorů << a >> pro typy definované uživatelem.....	96
Šířka výstupního pole.....	97
Zobrazení znaménka u kladných čísel.....	98
Způsob zarovnávání.....	99
Výplňový znak.....	99
Základ číselné soustavy	100
Velikost znaku při hexadecimálním výstupu a zobrazení základu číselné soustavy.....	101
Formáty tisku reálných čísel.....	102
Přeskakování bílých znaků na vstupu.....	104
Splachování	104
3.5 Ladění komunikace s datovými proudy	106
4. Podmíněný překlad a makra	108
4.1 Dolarové poznámky a příkazy preprocesoru	108
4.2 Řízení překladače v Pascalu	109
Globální přepínače	109
Lokální přepínače.....	111
4.3 Řízení překladače v C++.....	113

Speciální direktivy.....	113
Povolení a potlačení generování chybových hlášení.....	115
Vlastní modifikace práce překladače.....	121
4.4 Makra.....	124
Jednoduchá makra.....	124
Makra s parametry	127
4.5 Podmíněný překlad	129
Podmíněný překlad v Pascalu.....	129
Podmíněný překlad v C++	131
4.6 Vkládání souborů	133
Vkládání souborů v Pascalu.....	133
Vkládání souborů v C++	134
5. Podrobnosti o ladění programů	135
Inspekční okno v prostředí Borland C++	137
6. Používání funkcí z jazyka C v C++	138
7. Základy práce s grafikou	140
7.1 Zpracování chyb.....	140
7.2 Inicializace grafického systému	142
Obrazovka.....	142
Ovládání celého grafického systému.....	145
7.3 Barvy	148
Karty a barvy	148
Nastavování barev.....	149
7.4 Grafický kurzor	153
7.5 Kreslení.....	154
7.6 Okna	159
7.7 Výřezy	160
7.8 Výstup textu.....	161
Textový kurzor v grafickém režimu.....	162
Grafický výstup textu.....	164
7.9 Práce s videostránkami.....	164
7.10 Vyplňování	165
Vyplněné obrazce	165
Způsob vyplňování.....	165
8. Piškorky	168
8.1 Úvodní kroky	168
Příprava.....	174
Zobraz Start	180
8.2 Hra.....	182
Rejstřík.....	192

Předmluva

Tato kniha bezprostředně navazuje na knihu *Práce s daty I*. Naučíte se v ní mimo jiné pracovat s ukazateli a s dynamickými proměnnými, používat struktury resp. záznamy, pracovat se soubory. Seznámíte se také s nástroji pro grafický výstup.

Do této knihy jsme také zařadili kapitolu, věnovanou složitějšímu projektu – programu Piškorcky.

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

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 používat proměnné a konstanty,
- ✧ znají základní vstupní a výstupní operace orientované do standardních vstupních a výstupních souborů a na konzolu,
- ✧ umějí zacházet s procedurami a funkcemi,
- ✧ 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 *přetížená 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:

for	Tučně píšeme klíčová slova.
ukazatel	Tučně píšeme nově zaváděné termíny a také pasáže, které chceme z jakýchkoli důvodů zdůraznit.
<i>free()</i>	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.
<i>option</i>	Kurzivou také píšeme anglické názvy.
CTRL+F9	Kapitálky používáme pro vyznačení kláves a klávesových kombinací.
<code>getch()</code> ;	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. Ukazatele

Ukazatel (můžete se setkat také s termíny směrník nebo pointer) je objekt (konstanta, proměnná nebo výraz), jehož hodnotou je adresa nějakého objektu v paměti. Ve starších programovacích jazycích byly ukazatele chápány pouze jako adresy, tj. adresy čehokoliv; moderní programování však chápe ukazatele jako adresy objektu určitého typu. Tím umožňuje překladači typovou kontrolu se všemi příznivými dopady na produktivitu práce i spolehlivost výsledných programů.

Práce s ukazateli již patří do vyšší školy programátorského umění. V této kapitole si probereme pouze její základy a vrátíme se k ní ještě někdy později, až budeme mít větší zkušenosti, abychom mohli všechny její „fajnovosti“ náležitě docenit.

1.1 Ukazatele na skalární proměnné

Začněme nejprve s ukazateli na proměnné jednoduchých skalárních (tj. nestrukturovaných) typů. Ukážeme si, jak takovéto proměnné deklarujeme a jaká jsou základní pravidla jejich použití.

V úvodu kapitoly jsme si řekli, že hodnoty ukazatelů jsou chápány jako adresy objektů předem daného typu. Každý ukazatelový typ je tedy pevně svázán se svým **základním typem (doménovým typem)** – datovým typem, na jehož konstanty a proměnné objekty daného ukazatelového typu ukazují.

Každé pravidlo však má své výjimky. Při praktickém programování například občas vyvstává potřeba proměnných, které budou uchovávat adresy objektů předem neznámých typů. Pro tento účel zavádí oba jazyky speciální datový typ, označující ukazatel na cokoliv, tj. ukazatel, u něž není přesně specifikován typ objektu, na který ukazuje. Ukazatele tohoto speciálního typu budeme označovat jako **volné** (nesvázané s žádným základním datovým typem – občas budeme také říkat **ukazatele bez doménového typu, netypové** nebo **bezdoménové**), kdežto ukazatele všech ostatních ukazatelových typů budeme označovat jako **vázané** (alternativní termíny jsou **typové** a **doménové**).

V Pascalu se volný ukazatelový typ nazývá *pointer* a proměnné tohoto typu se deklarují stejně jako proměnné kteréhokoliv z dosud probraných skalárních typů. Význačnou vlastností tohoto datového typu je, že proměnným typu *pointer* můžeme přiřadit hodnotu kteréhokoliv ukazatelového typu. Bohužel to v Pascalu (na rozdíl od C+) jde i naopak – zde se tvůrci Turbo Pascalu rozhodli (proti duchu zbytku jazyka) ponechat zodpovědnost za správné přiřazení na programátorovi.

Vázané (typové) ukazatelové typy označujeme v deklaracích tak, že napíšeme šipku vzhůru (ve většině znakových generátorů je symbolizována stříškou „^“) následovanou identifikátorem základního datového typu. Vše vysvětluje následující ukázka:

```

type
  TUKNaInt = ^integer;           {Typ "ukazatel na integer"}
var
  {Čtyři ukazatele na proměnnou typu integer}
  UKNaInt1, UKNaInt2 : TUKNaInt;
  UKNaInt3, UKNaInt4 : ^integer;
  {POZOR - typy těchto proměnných nejsou kompatibilní!}
  UKNaNeco : Pointer;          {Ukazatel na cokoliv}

```

Poznamenejme, že proměnné *UKNaInt1* a *UKNaInt2* nejsou kompatibilní vzhledem k přiřazení s proměnnými *UKNaInt3* a *UKNaInt4*, neboť jejich typy nepovažuje Pascal za stejné (nemají stejný identifikátor nebo jeden z nich nevznikl přejmenováním druhého – zápis *^integer* nepovažuje Pascal za identifikátor typu).

V C++ jsme se již s ukazateli setkali. Proměnné typu **char ***, které jsme používali pro práci s textovými řetězci, nejsou totiž nic jiného než ukazatele na místa v paměti, kde leží vlastní posloupnost znaků ukončená prázdným znakem. Z toho si také můžete odvodit, že ukazatel na nějaký typ definujeme v C++ tak, že v definici napíšeme před identifikátor ukazatelové proměnné hvězdičku. Chcete-li v jedné definici definovat více proměnných, musí být hvězdička před každým identifikátorem ukazatele.

Chceme-li ukazatelový typ pojmenovat, napíšeme klíčové slovo **typedef** následované identifikátorem základního typu, hvězdičkou a identifikátorem.

Na rozdíl od Pascalu není v C++ formálně odlišena deklarace vázaných a volných ukazatelů (tj. ukazatelů na určitý datový typ a „ukazatelů na cokoliv“). Za ukazatele bez doménového typu se totiž v C++ považují ukazatele na prázdný typ **void**. I zde platí, že proměnným typu **void*** můžeme přiřadit hodnotu kteréhokoliv ukazatelového typu. Na rozdíl od Pascalu to však nejde naopak.

```

typedef int *TUKNaInt;           //Typ "ukazatel na int"
                                //Čtyři ukazatele na proměnnou typu integer
TUKNaInt UKNaInt1, UKNaInt2;

int *UKNaInt3, *UKNaInt4;
                                //Deklarace dvou celočíselných proměnných (i1 a i2)
                                //a dvou ukazatelů na int (u1 a u2).
int i1, *u1, i2, *u2;
void *UKNaNeco;                //Ukazatel na cokoliv

```

V předchozích ukázkách jsme definovali typ *TUKNaInt* jako ukazatel na *integer* (Pascal), resp. **int** (C++). Vzápětí jsme definovali čtyři proměnné, a to *UKNaInt1* a *UKNaInt2* typu *TUKNaInt* a proměnné *UKNaInt3* a *UKNaInt4*, které byly přímo deklarovány jako ukazatele na objekty typu *integer* (Pascal), resp. **int**(C++).

V Pascalu jsou tyto dva ukazatelové typy vzájemně nekompatibilní, a to přesto, že oba dva ukazují na stejný typ. Je to v podstatě stejná situace, s jakou jsme se setkali u vektorů – 100% kompatibilní jsou pouze takové typy, z nichž jeden vznikl přejmenováním druhého. Důsledky této nekompatibility si ukážeme za chvíli.

C++ naproti tomu konstatuje, že v obou dvou případech se jedná o ukazatel na stejný typ, a při vzájemném porovnávání a přiřazování vám žádné problémy nedělá. Základní typy však musí být v obou případech buď totožné, anebo musí jeden z nich být typ vzniklý pouhým přejmenováním druhého. Pouhá kompatibilita vůči přiřazení – ukazateli na **int** nelze jen tak beze všeho přiřadit ukazatel **nadouble**.

Ukazatele můžeme používat dvěma způsoby: buď můžeme pracovat s adresou, která je v ukazateli uložena (můžeme ji však pouze přiřadit nebo ji porovnat s jinou adresou téhož typu), nebo můžeme pracovat s hodnotou, na kterou ukazatel ukazuje.

Pokud pracujeme s adresami, nemusíme na svých zvycích nic měnit; musíme pouze dávat větší pozor na to, abychom navzájem přiřazovali a porovnávali pouze hodnoty vzájemně kompatibilních odkazových typů.

Pro zdárnou práci s hodnotami, na něž ukazatele ukazují, se musíme ještě něco přiučit. První, co ještě neumíme, je rozlišení výše uvedených dvou způsobů práce s ukazateli v textu programu. To však není nic složitého. Nechceme-li pracovat s hodnotou ukazatele (adresou), ale s hodnotou, na niž ukazatel ukazuje, oznámíme to počítači pomocí operátoru dereferencování, což je v Pascalu stříška (šipka nahoru), v C++ hvězdička. Operátor je v obou jazycích unární (má pouze jeden operand – dereferencovaný ukazatel), liší se však fixací: v Pascalu je postfixový, tj. operátor (stříšku) napíšeme **za** identifikátor ukazatele, kdežto v C++ je prefixový, tj. operátor (hvězdičku) píšeme před identifikátor ukazatele.

Jak víme, hodnotami ukazatelů jsou adresy. Potřebujeme proto také nějaký prostředek, který nám umožní adresu daného objektu získat a ukazateli ji pak přiřadit. Tímto prostředkem je prefixový unární operátor získání adresy, který v Pascalu označuje znak **@** (zavináč, šnek) a v C++ znak **&** (et, and, ampersand).

Při práci s hodnotami, na něž ukazatele ukazují, musíme mít na paměti, že na jednu proměnnou může ukazovat několik ukazatelů a že změnou hodnoty této proměnné vlastně měníme hodnotu, na kterou ukazují všechny tyto ukazatele. Změníme-li naopak hodnotu ukazatele, ukazuje změněný ukazatel od této chvíle někam úplně jinam a jakákoliv změna hodnoty proměnné, na kterou ukazoval původně, se ho netýká.

Ukazatele nemusí vždy někam ukazovat. Často potřebujeme naopak zdůraznit, že ukazatel ještě (nebo už) nikam neukazuje. Pro tento účel jsou v obou jazycích zavedeny speciální hodnoty, které přiřazujeme ukazatelům ve chvíli, kdy je chceme označit jako „ukazatele neukazující nikam“. Ukazatele, které mají tuto speciální hodnotu, nazýváme **prázdné ukazatele**.

V Pascalu používáme k definici prázdných ukazatelů klíčové slovo **nil** (můžeme je chápat jako literál). V C++ pro tento účel můžeme používat nulu.

V jazyku C bylo zvykem používat místo ní konstantu **NULL**, definovanou v hlavičkových souborech *alloc.h*, *mem.h*, *stddef.h* (ten je z nich nejkratší), *stdio.h* a *stdlib.h*. Pokud

byste tuto konstantu chtěli používat, dovezte kterýkoliv z výše uvedených souborů pomocí příkazu `#include`.¹

Poslední věc, o které si před ukázkovým příkladem povíme, je používání volných (netypových) ukazatelů (tj. ukazatelů typu *pointer* resp. **void***). Řekli jsme si, že těmto ukazatelům můžeme přiřadit hodnotu ukazatele libovolného typu. Neřekli jsme si však zatím, jak to zařídit, abychom mohli v C++ přiřadit hodnotu volného ukazatele ukazateli vázanému (jak jsem již řekl, Pascal v tomto případě na typovou kontrolu rezignuje) nebo jak přímo použít hodnotu, na niž volný ukazatel ukazuje (tato hodnota totiž nemá přiřazen žádný typ).

Asi vás již napadlo, že se z takovýchto situací budeme „vyhlávat“ prostřednictvím operace přetypování.

Při práci s hodnotami volných ukazatelů (tj. s adresami) to mají pascalisté zdánlivě jednodušší. Autoři překladače na chvíli „zapomněli“ na to, že Pascal je jazyk, který vždy provádí přísnou typovou kontrolu, a typ *pointer* definovali tak, že je kompatibilní vůči přiřazení se všemi ukazatelovými typy. Tím sice zjednodušili zápis programů, avšak na druhou stranu připravili půdu pro řadu nepříjemných a záludných chyb z přehlédnutí. Je to sice trochu komické, když dva ukazatelové typy vázané na stejný základní typ spolu kompatibilní nejsou, kdežto s volným „ukazatelem na cokoliv“ kompatibilní jsou, ale na takové paradoxy si v Pascalu musíte zvyknout.

C++ je při typové kontrole přísnější. Volnému ukazateli sice můžete přiřadit hodnotu kteréhokoliv ukazatelového typu, avšak vázanému ukazateli můžete přiřadit pouze hodnotu ukazatele, který ukazuje na stejný základní typ. Volný ukazatel přetypujeme na ukazatel vázaný na daný doménový typ tak, že před identifikátor tohoto volného ukazatele napíšeme do závorek identifikátor patřičného základního typu následovaný hvězdičkou.

V obou jazycích nejprve přetypujeme volný ukazatel na ukazatel vázaný na požadovaný základní typ, přetypovaný ukazatel dereferencujeme a výsledek přiřadíme. Poznamenejme ale, že v Pascalu musíme nejprve zavést pro vázaný ukazatel jednoslovný identifikátor (deklarací **type**).

Podívejte se na následující programky a zkuste si je krokovat. Před tím si však nechte do sledovacího okna *Watches* vypisovat všechny důležité informace: adresy použitých celočíselných proměnných (`@i1`, `@i` a `@i3` v Pascalu, `&i1`, `&i2` a `&i3` v C++), jejich hodnoty (`i1`, `i2`, `i3`), hodnoty ukazatelů (`u1`, `u2`, `u3` a `uu`), a hodnoty, na něž ukazatele ukazují (`u1^`, `u2^` a `^u3` v Pascalu, `*u1` `*u2` a `*u3` v C++). (Sledovací okno si budete muset patřičně zvětšit.) V průběhu krokování pak sledujte, jak se budou jednotlivé hodnoty měnit.

¹ V C++ se velice důrazně doporučuje používat k vyjádření prázdného ukazatele `0`, nikoli `NULL`. Standard jazyka C totiž nestanoví, jakého typu konstanta `NULL` je, a ten se může lišit nejen implementací od implementace, ale i v různých situacích v rámci jedné implementace. Např. v Borland C++ 4.5 je `NULL` podle okolností typu `int`, `long` nebo `void*`.

```

(* Příklad P1 - 1 *)
Program PASCAL;
type
  TU = ^integer;
var
  i1, i2, i3 : integer;
const
  uu : Pointer = NIL;      {Ukazatel je inicializován jako prázdný}
  u1 : TU = @i1;
  u2 : ^integer = @i2;
  u3 : ^integer = @i3;
{Předchozí deklarace inicializovaly pouze hodnoty ukazatelů, avšak
nikoliv odkazovaných proměnných. Proměnné ix a tedy ani odkazované
hodnoty *ux ještě obecně inicializovány nejsou.}
begin
  i1 := 1;                {Totéž jako "*u1 := 1;"}
  i2 := 22;               {Totéž jako "*u2 := 22;"}
  i3 := 333;              {Totéž jako "*u3 := 333;"}
  u1^:= u2^;              {OK: integer := integer}
                          {Totéž jako "i1 := i2;"}
  u1^:= 1;                {Totéž jako "i1 := 1;"}
  {u1 := u2; - NELZE, nekompatibilní typy}
  u1 := TU( u2 );         {OK: TU := TU}
                          {Od této chvíle ukazuje u1 na i2}
  {u3 := u1; - NELZE, nekompatibilní typy}
  {u3 := integer^(u1); - takto přetypovat nelze}
  u3 := @u1^;             {OK: integer^ := @integer}
                          {Od této chvíle ukazuje u3 na i2}
  u2^:= 30;               {Totéž jako "i2 := 30"}
                          {nebo "u1^ := 30"}
                          {nebo "u3^ := 30"}
  u1 := uu;               {!!! - přiřazení hodnoty volného ukazatele vázanému}
  u1 := @i1;
  uu := @u1;              {V uu je adresa u1}
  uu := Pointer(uu^);     {V uu je obsah u1 = adresa i1}
  i2 := 5 * integer(uu^); {Do i2 přiřadíme 5násobek hodnoty,}
                          {na niž ukazuje uu.}
end.

```

Poznámka:

Pokud programátoři v C++ deklarují v následujícím příkladu proměnné i1, i2 a i3 jako globální, tj. deklarují je mimo funkci main(), nemusí ve sledovacím okně vypisovat jejich adresy. Pokud totiž některý z ukazatelů, jejichž hodnoty ve sledovacím okně sledujeme, ukazuje na globální objekt, systém vám ve sledovacím okně nevypisuje adresu (tj. hodnotu ukazatele) číselně, ale napíše přímo identifikátor této odkázané proměnné s přidaným prefixovým &. Kromě toho připomínáme, že globálně deklarované proměnné jsou inicializovány po spuštění programu nulovou hodnotou.

```

/* Příklad C1 - 1 */
#include <stddef.h>

```

```

void /*****/ main /*****/ ()
{
    typedef int* TU;
    int i1, i2, i3;
    void *uu = NULL;
    TU u1=&i1;
    int *u2=&i2, *u3=&i3;
    //Předchozí deklarace inicializovaly pouze hodnoty ukazatelů, avšak
    //nikoliv hodnoty odkazovaných proměnných. Proměnné i1 a tedy ani
    //odkazované hodnoty *u2 ještě inicializovány nejsou.

    i1 = 1; //Totéž jako "*u1 = 1"
    i2 = 22; //Totéž jako "*u2 = 22"
    i3 = 333; //Totéž jako "*u3 = 333";
    *u1 = *u2; //Totéž jako "i1 = i2;"
    // - přiřazení odkazovaných hodnot
    *u1 = 1; //Totéž jako "i1 = 1;"
    u1 = TU( u3 ); //Povolené, ale zbytečné
    u1 = 0; //Prázdný ukazatel - ukazatel nikam
    //Místo 0 bychom mohli použít NULL
    u1 = (TU)u3; //Jinak zapsaný předminulý řádek
    u1 = u2; //Takhle je to nejjednodušší
    //Od této chvíle ukazuje u1 na i2
    u3 = u1; //Obráceně to funguje také
    //Od této chvíle ukazuje u3 na i2
    *u2 = 30; //Totéž jako "i2 = 30"
    // nebo "*u1 = 30"
    // nebo "*u3 = 30"
    //u1 = uu; CHYBA - hodnotu volného ukazatele nelze přiřadit vázanému
    u1 = (int*) uu; //Takto je to již správné
    u1 = &i1;
    uu = &u1; //V uu je adresa u1
    uu = *(void**)uu; //V uu je obsah u1 = adresa i1
    i2 = 5 * *(int*)uu; //Do i2 přiřadíme 5násobek hodnoty,
    //na niž ukazuje uu
}
/***** main *****/

```

1.2 Ukazatele na pole, pole ukazatelů

Ukazatele mohou samozřejmě ukazovat i na proměnné vektorových typů. Protože však má práce s ukazateli na vektory a vícerozměrná pole svá specifika (např. bychom mohli omylem místo ukazatele na vektor deklarovat vektor ukazatelů), věnujeme jí samostatnou podkapitulu.

Pascalská syntax deklarácí je ve srovnání se syntaxí jazyka C++ ve svých požadavcích mnohem přísnější. Díky tomu je však pro průměrného programátora mnohem průzračnější, takže i pravděpodobnost chybné deklarace je mnohem menší.

Vzhledem k přísné typové kontrole se vyžaduje, aby základním typem vázaného ukazatelového typu byl nějaký dříve definovaný typ. Ukazatel na pole nemůžeme tedy dekla-

rovat přímo, ale musíme si nejprve deklarovat typ pole, na něž chceme ukazovat, a tento typ pak v deklaraci ukazatele uvést.

V následující ukázce deklarujeme celá čísla, pole celých čísel, pole ukazatelů na celá čísla, ukazatel na pole celých čísel a ukazatel na pole ukazatelů na celá čísla. U každého typu objektu uvedeme i příklad použití. Zkuste si tento příkládek ~~zk~~zkoušet.

```
(*   Příklad P1 - 2   *)
type
  p2ui = array[ 1..2 ] of ^integer;  {Vektor dvou ukazatelů na
integer}
                                     {Ukazatel na pole dvou prvků typu integer}
  p2i = array[ 1..2 ] of integer;    {Deklarace pole}
  up2i = ^p2i;                       {Deklarace ukazatele na pole}

var
  i11, i12 : integer;
  i21, i22 : integer;
  pi1, pi2 : p2i;                    {Pole dvou celých čísel}
  pu1, pu2 : p2ui;                  {Pole dvou ukazatelů na celá čísla}
  up : up2i;                         {Ukazatel na pole dvou celých čísel}
  upu : ^p2ui;                      {Ukazatel na pole dvou ukazatelů
na celá čísla}

begin
  pi1[ 1 ] := 11; pi1[ 2 ] := 12;
  pi2[ 1 ] := 12; pi2[ 2 ] := 22;
  pu1[ 1 ] := @i11; pu1[ 2 ] := @i12;
  pu2[ 1 ] := @i21; pu2[ 2 ] := @i22;
  upu := @pi1;
  upu := @pu2;
  i11 := 110;
  pu1[ 2 ]^ := 120;                    {i12 := 120}
  upu^[ 1 ] := @pi1[ 1 ];
  upu^[ 1 ]^ := 1100;                 {pi1[ 1 ] := 1100}
  upu^[ 2 ]^ := 22000;                {pu2[ 2 ]^ = i22 := 22000}
end.
```

Syntax deklarácí je v jazyku C++ ve srovnání s Pascallem trochu volnější, avšak zároveň ve své obecnosti také poněkud obtížněji pochopitelná. Do žádných extravagancí nás však nenutí a vždy nám nabízí možnost používat deklarace **typedef** a pracovat pak s konstrukcemi obdobnými pascáským.

Při deklaracích ukazatelů na pole, polí ukazatelů či dokonce ukazatelů na pole ukazatelů musíme mít neustále na paměti priority operátorů, jejichž identifikátory v deklaracích používáme. Protože mají hranaté závorky větší prioritu než hvězdička, bude deklarace

```
int * p [5];
```

totožná s deklarací

```
int * (p [5]);
```

kteřá zavádí p jako pole pěti ukazatelů na **int**. Pokud bychom chtěli deklarovat ukazatel na pole pěti prvků typu **int**, museli bychom použít závorek:

```
int (*p) [5];
```

Ukazatel na pole pěti ukazatelů **na int** pak můžeme deklarovat zápisem

```
int * (*p) [5];
```

nebo

```
int * ((*p) [5]);
```

Už jsme si však řekli, že těmto deklaracím se můžeme vyhnout vhodným použitím deklarace typů (**typedef**).

V následující ukázce deklarujeme celá čísla, pole celých čísel, pole ukazatelů na celá čísla, ukazatel na pole celých čísel a ukazatel na pole ukazatelů na celá čísla. Každý typ objektu deklarujeme jak přímo (identifikátory s velkými písmeny), tak pomocí typedefů (identifikátory s malými písmeny). U každého typu objektu uvádíme i příklad použití. Zkuste si tento příkládek krokovat.

```
/*   Příklad C1 - 2   */
//Vektor dvou ukazatelů na integer
typedef int * p2ui[ 2 ];

//Ukazatel na pole dvou prvků typu integer
typedef int p2i[ 2 ];           //Deklarace pole
typedef p2i * up2i;           //Deklarace ukazatele na pole

int i11, i12, i21, i22;
p2i pi1, pi2;                 //Pole dvou celých čísel

int PI1 [2], PI2 [2];
p2ui pu1, pu2;               //Pole dvou ukazatelů na celá čísla

int *PU1 [2], *PU2 [2];
up2i up;                     //Ukazatel na pole dvou celých čísel

int (*UP) [2];
p2ui *pu;                    //Ukazatel na pole dvou ukazatelů
//na celá čísla

int * (*PU) [2];

void /*****/ main /*****/ ()
{
    pi1[ 0 ] = 11; pi1[ 1 ] = 12;
    pi2[ 0 ] = 12; pi2[ 1 ] = 22;
    pu1[ 0 ] = &i11; pu1[ 1 ] = &i12;
    pu2[ 0 ] = &i21; pu2[ 1 ] = &i22;
    up = &pi1;
    pu = &pu2;
    i11 = 110;
    *pu1[ 1 ] = 120;           //i12 = 120
    (*up)[ 0 ] = 1100;        //pi1[ 0 ] = 1100
    *(*pu)[ 1 ] = 22000;      //*pu2[ 1 ] = i22 = 22000
}
```

|| }

Neklesejte na mysli, pokud jste vše hned nepochopili. Již na počátku kapitoly jsme vám říkali, že práce s ukazateli patří do vyšší školy programátorského umění a netrvejte proto na tom, že ji musíte zvládnout hned po prvním přečtení této kapitoly. V doprovodných programech budete postupně nacházet různá použití ukazatelů a budete proto mít ještě řadu příležitostí se s nimi důvěrně seznámit.

1.3 Příklad: simulované bakterie

Než budeme ve výkladu o ukazatelích pokračovat, zkuste si vyřešit jednoduchou domácí úlohu. Jedná se o konstrukci programu simulujícího život kolonie bakterií. Programy tohoto typu byly jeden čas velice populární a svého času jsme četli studii, která tvrdila, že v některých výpočetních střediscích spotřebuje jejich ilegální provozování až 40 % strojového času. (To bylo ještě před DOOMem a jinými zběšlostmi.)

O co jde? Představte si, že životním prostorem těchto simulovaných bakterií (budeme je nazývat **siby**) je vaše obrazovka. Na každém znakovém poli může být nejvýše jedna siba. Její další osud závisí na počtu sousedů, tj. na počtu sib v sousedních polích (8 polí, které se daného pole dotýkají alespoň růžkem).

Pokud má siba kolem sebe málo sousedů (např. méně než 3), umírá z nedostatku společnosti. Pokud má siba kolem sebe příliš mnoho sousedů (např. více než 5), umírá z nedostatku potravy. Pokud má vhodný počet sousedů, přežívá do další generace. Pokud má prázdné políčko správný počet sousedů (např. 2), narodí se na něm nová siba. Pokud má políčko sousedů méně, zůstane do příští generace prázdné.

Zkuste si tento příklad nejprve naprogramovat pomocí toho, co již znáte – tj. pomocí matic (dvourozměrných polí). Zároveň si v programu připravte prostředky pro to, abyste mohli změřit čistý čas výpočtu – tj. bez času tisků na obrazovku.

Abyste to měli jednodušší, trochu vám napovíme: Mějte na paměti, že nové hodnoty jednotlivých polí (tj. informace o přítomnosti či nepřítomnosti siby v další generaci) nemůžete do polí zaznamenávat hned ve chvíli, kdy je zjistíte, protože původní obsah tohoto pole budete pravděpodobně ještě potřebovat při zjišťování počtu sousedů některého z polí následujících.

Ukázkové řešení tohoto úkolu najdete na doplňkové disketě v souboru *CI-00A.CPP* resp. *CI-00A.PAS*.

1.4 Aritmetika ukazatelů

Aritmetika ukazatelů neboli adresová aritmetika je pojem specifický pro C++. Protože se však jedná a velice užitečnou vlastnost jazyka, ukážeme si zde i některé prostředky, kte-

rými budeme moci některých jejích efektů dosáhnout i v Pascalu. Tato podkapitola bude proto mít dvě zvláštnosti: za prvé bude výklad specifický pro jazyk C++ předcházet výkladu ekvivalentních konstrukcí specifickému pro jazyk Pascal, a za druhé chci všem pascalistům poradit, aby pro tentokrát zkusili výklad specifik jazyka C++ nepřeskakovat, ale pokusili se jej alespoň zběžně přečíst.

Adresová aritmetika v C++

Adresová aritmetika jazyka C++ je určena především pro práci s poli. Proto si musíme nejprve vysvětlit, jak to vlastně s poli a s jejich vztahem k ukazatelům je.

Pole a ukazatele v C a v C++

Autoři jazyka C zakomponovali do svého dítko jeden geniální nápad: sjednotit práci s vektory prvků daného typu a ukazateli na prvky daného typu. A protože to byla myšlenka opravdu geniální, jazyk C++ ji akceptoval.

Základní idea spočívá v tom, že syntax jazyka nerozlišuje vektory prvků daného typu a ukazatele na daný typ. Přesněji řečeno: **pole se v jazyku C téměř vždy konvertuje na ukazatel na první prvek.**² Na první pohled to vypadá jako neuvěřitelný nesmysl, ale je to tak – a pokud si toto pravidlo zapamatujeme, ušetříme si mnohá nepříjemná překlopení.

Podívejme se, co z toho plyne. Vezměme pole P deseti prvků typu `double`,

```
double P[10];
```

Napišeme-li

```
double d = P[3];
```

konvertuje se podle uvedeného pravidla P nejprve na ukazatel na první prvek pole a teprve na tento ukazatel se použije operátor indexování. Plyne z toho snad, že operátor indexování můžeme používat na libovolný ukazatel?

Ano, plyne. Na libovolný ukazatel s doménovým typem můžeme opravdu použít operátor indexování. **S ukazatelem můžeme zacházet jako s vektorem, jehož nultý prvek leží na adrese určené hodnotou v něm uloženou.**

² Existují 3 výjimky, kdy se pole opravdu chová jako pole (v jazyku C pouze 2):

1. Použijeme-li na ně operátor `sizeof`. Je-li např. P pole deseti prvků typu `double`, platí rovnost `sizeof(P) == 10 * sizeof(double)`.
2. Použijeme-li na ně operátor získání adresy „&“. Je-li P opět pole deseti prvků typu `double`, je `&P` adresa **pole jako celku**, tedy hodnota typu `double (*)[10]` (ukazatel na pole deseti prvků typu `double`).
3. Při inicializaci reference – o tom si budeme povídat v následujícím dílu. (Reference najdeme jen v C++.)

Ukazatele tedy můžeme indexovat. A protože si přiřazením vhodné hodnoty ukazateli můžeme počátek vektoru umístit téměř kamkoliv, může mít smysl i použití záporných indexů (toho se využívá při realizaci vektorů, jejichž indexy nezačínají nulou).

Konečně adresová aritmetika

Nyní se konečně dostáváme k aritmetice ukazatelů, o níž má být tato podkapitola. Vlastností této aritmetiky patří k dalším velice příjemným rysům jazyků C a C++. Vezměme nějaký ukazatel s doménovým typem.

Jak jsme si řekli, pokládá jej překladač za ukazatel na začátek pole. Základní idea adresové aritmetiky spočívá v tom, že připočtení jakéhokoliv celého čísla (tedy i záporného) k ukazateli změní jeho hodnotu tak, že získáme adresu prvku pole, jehož index je roven přičítanému číslu. Dereferencováním této adresy získáme hodnotu tohoto prvku stejně, jako kdybychom použili indexovaného výrazu. Je tedy úplně jedno, zda získáme hodnotu i -tého prvku pole P vyhodnocením výrazu

```
P[ i ]
```

nebo

```
* ( P + i )
```

Zároveň je ale také jedno, zda P je ukazatel nebo identifikátor pole.

Podobně můžeme na ukazatele použít operátory inkrementace ($++$) a dekrementace ($--$). Asi se shodneme na tom, že bychom mohli operátor $++$ překřtít na „další“ a operátor $--$ na „předchozí“, protože po jejich aplikaci na ukazatel bude ukazatel ukazovat na další, resp. předchozí prvek původního pole.

Pro přičítání a odečítání celočíselné konstanty můžeme také použít přiřazovací operátory $+=$ a $-=$. Chceme-li tedy ukazatel u „posunout“ o 5 prvků dále, dosáhneme toho jedním z příkazů

```
u += 5;
u -= -5;
```

a chceme-li jej naopak posunout o 3 prvky zpět, dosáhneme toho jedním z příkazů

```
u -= 3;
u += -3;
```

Pro jistotu ještě zdůrazníme, že operátory probrané v předchozích odstavcích, tj. operátory $++$, $--$, $+=$ a $-=$, nelze použít na identifikátory polí. Tyto operátory mění hodnotu svého operandu, takže vyžadují l-hodnotu, a identifikátor pole l-hodnotu nepředstavuje.

Jestliže přičtením celého čísla k ukazateli dostaneme nový ukazatel, měli bychom logicky odečtením dvou ukazatelů získat celé číslo. A opravdu, je to tak. Pokud odečteme dva ukazatele ukazující do stejného pole, obdržíme počet prvků, které leží mezi těmito dvěma adresami.

Protože C++ neumí bez dodatečných informací zkontrolovat, zda oba ukazatele ukazují do stejného pole, dovolí nám překladač od sebe odečítat jakékoliv dva ukazatele se

stejným základním typem, a je na nás, jak obdrženy výsledek interpretujeme v případě, že ukazatele neukazují do téhož pole. Ukazatele s různými doménovými typy od sebe odečítat nelze.

Prvek za polem

Na závěr této části si ještě povíme o rozsahu smysluplných hodnot ukazatelů na prvky vektorů. Představte si, že jsme deklarovali vektor

```
double vd [ N ];
```

Z předchozího výkladu víme, že se jedná o vektor o N prvcích, které mají indexy 0 až $N-1$. Pokud bychom chtěli pracovat s prvkem $vd[N]$, překladač by nám to dovolil, avšak pracovali bychom pouze s nějakým smetím, protože takový prvek ve skutečnosti neexistuje. (Nebo ještě hůře, přepisovali bychom si data, uložená v jiných proměnných.. C++ neumí na rozdíl od Pascalu bez dodatečných informací hlídat překročení mezí polí.)

Prvek následující po posledním prvku vektoru sice ve skutečnosti neexistuje, avšak často je užitečné mít možnost pracovat s jeho adresou – např. při definici vstupních či výstupních podmínek cyklů. Jazyk C++ proto zaručuje, že kromě adres jednotlivých prvků bude smysluplná i adresa hypotetického prvku, který leží za posledním prvkem vektoru.

Tato záruka v našem případě znamená, že ve svém programu můžeme používat nejen adresy prvků $vd[0]$ až $vd[N-1]$, které jsou prvky daného vektoru, ale i adresu hypotetického prvku $vd[N]$, který již prvkem tohoto vektoru není. Jinými slovy: překladač zaručuje, že se adresa tohoto prvku v programu pro PC „nezlomí“ přes hranice segmentu (viz odstavec *Adresování v reálném režimu procesoru 80*86*) nebo že použitím této adresy nedojde k jiné chybě, a že tedy budou pravdivé výrazy:

```
&V[ N ] - &V[ 0 ] == N
&V[ 0 ] < &V[ N-1 ] < &V[ N ]
```

Smysluplnost adresy prvku s indexem -1 nebo $N+2$ již obecně zaručena není, takže výrazy podobné výše uvedeným nemusí platit.

Aby však nedošlo k mýlce, musíme zdůraznit, že jde o adresy v poli deklarovaném obdobně, jako jsme deklarovali vektor Vd . Jde nám o adresy, které lze popsat jako „adresa prvku před nultým prvkem pole“ a „adresa prvku následujícího za prvkem následujícím po posledním prvku pole“. Jinými slovy, z adres prvků, které nejsou skutečnými prvky daného vektoru, překladač zaručuje smysluplnost a konzistenci pouze pro adresu onoho výše zmíněného „prvku po posledním prvku“. Pokud tedy ve svých programech pracujeme s vektory a víme, že prvky se zápornými indexy opravdu existují, nic nám nebrání tyto záporné indexy využívat.

Některé základní vlastnosti aritmetiky ukazatelů si předvedeme v následující ukázce:

```
/* Příklad C1 - 3 */
int vi [] = {00, 10, 20, 30, 40, 50, 60, 70, 80, 90 };
int *ui;
```

```

void /*****/ main /*****/ ()
{
    ui = vi+5;           //&vi[ 5 ];
    *ui++ += 5;         //Totéž, jako *(ui++) += 5
                        //1. *ui = vi[5] = 55
                        //2. ui++ -> ui = &vi[ 6 ]
    *++ui += 5;         //1. ++ui -> ui = &vi[ 7 ]
                        //2. *ui = vi[7] = 75
    (*ui)++;           //vi[7]++ -> vi[7] = 76

    //Následující cykly jsou rozepsány do více řádků,
    //aby je bylo možno snadno krokovat
    //Po skončení následujícího cyklu budou hodnoty prvků
    //pole vi rovny 0, 100, 200, 300, 400, 550, 600, 760, 800, 900
    for( int *u=vi, s=0; //Totéž jako int*u = &vi[0]
        u < &vi[10];   //Adresa &vi[10] je ještě korektní
        s += (*u++ *= 10) //1. *u = *u * 10
        );              //2. s = s + *u
                        //3. u = u + 1

    //Po skončení následujícího cyklu budou hodnoty prvků
    //pole vi: 00, 10, 20, 30, 40, 55, 60, 76, 80, 90
    for( int i=9;
        i >= 0;
        s += (*(vi+i--) /= 10) //1. vi[i] = vi[i] / 10
        );                    //2. s = s + vi[i]
                        //3. i = i - 1

    //Po skončení následujícího cyklu budou hodnoty prvků
    //pole vi rovny 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    //a proměnná s bude obsahovat jejich součet (45)
    for( i=10, s=0;
        --i;                //(i = i-1), (i != 0)
        s += (vi[i] /= 10)  //1. vi[i] = vi[i] / 10
        );                  //2. s = s + vi[ i ]
                        //3. i = i - 1
}

```

Adresová aritmetika v Pascalu

Předpokládáme, že jste četli pasáž věnovanou jazyku C++ a máte tedy základní představu o vlastnostech céčkovské aritmetiky ukazatelů. Vynikající možnosti práce s ukazateli jsou jednou z vlastností jazyka C++, kvůli které mu dáváme v profesionální praxi přednost před Pascalem, který aritmetiku ukazatelů ve starších verzích neobsahuje vůbec a ve verzi 7.0 a v Delphi jen pro typ *PChar* (více o něm najdete v dodatku knihy *Objektové programování I*).

Abychom umožnili ochutnat alespoň některé ze slasti a strasti ukazatelové aritmetiky i programátorům v Pascalu, připravili jsme pro ně modul *UKAZATEL*, který najdete na doplňkové disketě. Nijak jsme však neprověřovali užitečnost funkcí definovaných v tomto modulu a nemůžeme ani vyloučit, že různá další omezení Pascalu práci s těmito funkcemi natolik znepríjemní, že bude výhodnější je nepoužívat. Když jsme však uvážili, že dopro-

vodný program *PI-00A* z doplňkové diskety se po pouhém převedení algoritmu z maticové verze na verzi pro ukazatele podstatně zrychlil, řekli jsme si, že za pokus to stojí.

Základem jsou funkce *PInc* a *IncP*, které posunují obecný ukazatel o zadaný počet bajtů. Funkce *PInc* je preinkrementační (**P**oužijí a **I**nkrementují), což znamená, že vrací původní hodnotu svého parametru (před inkrementací). Naproti tomu funkce *IncP* je postinkrementační (**I**nkrementují a **P**oužijí), což znamená, že svůj parametr nejprve inkrementuje a vrátí hodnotu změněnou.

Nevýhodou funkcí skupiny *PInc* a *IncP* ve srovnání s jejich ekvivalenty v jazyku C++ je, že nemůžete pouze oznámit počet prvků, o něž chcete ukazatel posunout, ale musíte jí předat spočtený počet bajtů, o něž se má hodnota ukazatele změnit. Proto jsme množinu funkcí rozšířili o verze s parametry vázaných ukazatelových typů.

Konstrukce názvů funkcí v tomto modulu je poměrně jednoduchá. Název se skládá ze dvou částí: symbolu operace a symbolu použitého datového typu. Symbolem *PP* označujeme ekvivalent operátoru ++, symbolem *MM* ekvivalent operátoru -- a symbolem *Inc* ekvivalent operátoru +=. Symbol *i* označuje datový typ *integer*, symbol *c* označuje datový typ *char* a symbol *r* označuje datový typ *real*. Kromě toho existují ještě výše zmíněné verze funkcí *XInc* a *IncX* se symbolem *P* označujícím volný ukazatelový typ *Pointer*. Tuto verzi použijeme v případě, kdy potřebujeme pracovat s ukazatelem na datový typ, pro nějž uvedené funkce definovány nejsou a nám se nevyplatí pro něj definovat funkci vlastní.

Vzájemná poloha symbolu datového typu a symbolu operace v názvu funkce symbolizuje, zda daná funkce vrací hodnotu svého parametru před provedením požadované operace anebo po něm – podobně jako u funkcí *PInc* a *IncP*.

Některé z možností použití těchto funkcí najdete v následující ukázce:

```
(*   Příklad P1 - 3   *)
Program Pascal;
Uses ukazatel;

type
  int = integer;
  Tvi = array[ 0..9 ] of int;

const
  vi : Tvi = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 );
  di : int = sizeof( int );

var
  ui : TuInt;
  i : int;
  s : int;
  uv : ^Tvi;

begin
  ui := @vi[5];
  Inc( iPP(ui)^, 5 );           {Ekvivalent *(ui++) += 5}
  Inc( MMi(ui)^, -6 );        {Ekvivalent *--ui -= 6}
  Inc( ui^ );                  {Ekvivalent (*ui)++}
```

```

{Po skončení následujícího cyklu budou hodnoty prvků}
{pole vi 0, 100, 200, 300, 400, 550, 600, 760, 800, 9}
  ui := @vi; {@vi = @vi[0]}
  s := 0;
  repeat
    ui^ := ui^ * 100;
    Inc( s, iPP(ui)^ );
  until( ui = @vi[9] );
  {poslední prvek musíme změnit ručně ...}
  vi[9] := vi[9] * 100;

{Pascal vám nedovolí vytvořit v době překladu adresu neexistujícího}
{prvku pole. Navíc vám neumožní porovnávat ukazatele na větší a
  menší,}
{ale pouze na rovnost nebo nerovnost.}
{Proto bylo nutno změnit typ{předchozího cyklu.}
{Po skončení následujícího cyklu budou hodnoty prvků}
{pole vi: 00, 10, 20, 30, 40, 55, 60, 76, 80, 90}
  for i:=9 downto 0 do
    begin
      vi[i] := vi[i] div 10;           {V Pascalu není možno}
      Inc( s, vi[i] );                {přičítat - pouze indexovat}
    end;

{Po skončení následujícího cyklu budou hodnoty prvků}
{pole vi: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  uv := @vi;
  for i:=9 downto 0 do
    begin
      uv^[i] := uv^[i] div 10;        {Abych mohl v Pascalu}
                                      {ukazatel indexovat, musí to být}
                                      {ukazatel na vektor}

      Inc( s, uv^[i] );
    end;
end.

```

V programech *P13-00A.PAS* a *C13-00.CPP* na doplňkové disketě najdete maticové i ukazatelové řešení úlohy o simulaci bakterií. Pascalisty bychom měli asi upozornit na to, že jsme zde použili jeden obrat, který je v C++ běžný, avšak v Pascalu velmi neobvyklý. Jde o použití polí ukazatelů *Nad*, *Tu* a *Pod*. Jejich zavedením jsme se vyhnuli vyhodnocování adresy prvku v matici, které je vždy spojeno s násobením a které proto celý výpočet silně zdržuje.

Adresování v reálném režimu procesoru 80*86

Než se začneme věnovat specifikům adresové aritmetiky na osobních počítačích, povíme si základní informace o adresovacích možnostech mikroprocesorů řady 80*86 v tzv. reálném režimu.

Mikroprocesor 8086 přišel s tzv. segmentovou architekturou paměti. Segment je oblast paměti, která může být až 64 KB velká (přesně $64 \cdot 1024 = 65536$ bajtů) a může začínat

na kterékoliv adrese dělitelné šestnácti (tj. na adrese, která má nejnižší čtyři bity nulové). Různé segmenty se přitom mohou překrývat.

Adresa každého místa paměti sestává ze dvou částí: ze 16bitové adresy počátku segmentu (říká se jí **segmentová část adresy** nebo prostě jen **segment**) a 16bitové relativní adresy vůči počátku segmentu (označuje se většinou jako **ofset**³) a zapisuje se ve tvaru *Segment:Ofset*. Výsledná 20bitová adresa (tedy pořadové číslo bajtu od počátku paměti) se spočte podle vzorce

$$\text{Adresa} = (\text{segment} * 16) + \text{ofset}$$

Tato koncepce umožňuje šetřit v jednodušších programech čas i paměť. Procesor si totiž ve čtyřech speciálních registrech pamatuje počátky segmentů, a pokud program zpracovává adresy v některém z nich, stačí pracovat pouze s ofsetem. Tím se šetří jak místo v paměti zaujímané programem, tak čas spotřebovaný na adresové operace.

Poznámka:

Následující verze tohoto procesoru, označované 80286 a 80386, přidaly další možnosti (procesory 80486, Pentium a Pentium Pro nové adresovací možnosti nepřidaly), avšak v počítačích pracujících pod operačním systémem DOS se donedávna v zájmu zpětné kompatibility na žádném z typů počítačů tyto možnosti téměř nevyužívaly. Nyní je (konečně) využívají 32bitové operační systémy jako Windows 95, OS2 a další a programy pro toto prostředí.

V přeloženém programu se rozlišují obvykle tyto segmenty:

- ✧ **Kódový segment.** Obsahuje „kód“ programu, tedy instrukce, ze kterých se program skládá. Segmentová část adresy jeho počátku je uložena v registru *CS* (ofsetová část adresy počátku je pochopitelně 0).
- ✧ **Datový segment.** Obsahuje globální a lokální statické proměnné. Segmentová část adresy jeho počátku je uložena v registru *DS*.
- ✧ **Zásobníkový segment.** Obsahuje tzv. zásobník, vyhrazenou oblast paměti, ve které se ukládají lokální automatické proměnné, skutečné parametry při volání funkcí a některé pomocné údaje. Segmentová část adresy jeho počátku je uložena v registru *SS*.

Program ovšem může mít několik kódových a datových segmentů.

Blízké, vzdálené a normalizované ukazatele

Z předchozího povídání plyne, že v programech můžeme podle okolností používat dva druhy ukazatelů:

- ✧ Ukazatele, obsahující pouze ofsetovou část (segmentovou si procesor doplní sám z příslušného registru). Tyto ukazatele lze použít v případě, že se pohybujeme v roz-

³ Anglicky *offset* znamená „posunutí“.

mezi segmentu, jehož adresa je uložena v příslušném registru. Tyto ukazatele budeme označovat jako **blízké** (*near pointers*).

- ✧ Ukazatele, obsahující jak segmentovou tak i ofsetovou část adresy. Potřebujeme je v případě, že pracujeme s objekty, které leží mimo segment, jehož adresa je uložena v příslušném registru (jestliže např. voláme funkci v jiném kódovém segmentu, jestliže pracujeme s daty mimo aktuální datový segment atd.). Tyto ukazatele budeme označovat jako **vzdálené** (*far pointers*).

Při použití vzdálených ukazatelů ale narazíme na problém: vyjádření adresy ve tvaru *Segment: Ofset* není jednoznačné. Např. adresu 17. bajtu v paměti můžeme vyjádřit zápisem 0:16 nebo 1:1. To samozřejmě komplikuje adresovou aritmetiku. Proto se zavádí tzv. **normalizovaná adresa**, pro kterou platí, že $0 \leq \textit{Ofset} \leq 15$. Normalizovaná adresa je jednoznačná.

Potřebujeme-li ukazatel, který obsahuje normalizovanou adresu, použijeme **normalizovaný ukazatel** (*huge pointer*).

Většina překladačů C++ pro PC nabízí programátorovi možnost v deklaraci explicitně předepsat, zda chce blízký, vzdálený nebo normalizovaný ukazatel. K tomu se používají modifikátory `__near`, `__far` resp. `__huge`. (V některých překladačích se tato klíčová slova používají bez úvodních podtržitek, tedy `near`, `far` a `huge`. V Borland C++ 3.1 a pozdějších můžeme použít obě varianty.) Tyto modifikátory se uvádějí před hvězdičkou, označující ukazatel. Např. deklarací

```
int __near * uni, __far * ufi, __huge * uhi;
```

zavádíme blízký ukazatel *uni*, vzdálený ukazatel *ufi* a normalizovaný ukazatel *uhi*, všechny s doménovým typem `int`.

Pokud v deklaraci „velikost“ ukazatele neuvedeme, tedy pokud napíšeme pouze

```
int *ui;
```

použije překladač blízký nebo vzdálený ukazatel podle toho, v jakém paměťovém modelu program překládáme. (O paměťových modelech si povíme za chvíli.)

Adresová aritmetika na PC

Adresová aritmetika funguje naprosto bez problémů pro normalizované ukazatele. Pro zbývající dva druhy musíme počítat s jistými omezeními.

Použijeme-li blízké ukazatele, je vše stále ještě poměrně jednoduché: adresová aritmetika funguje, pokud nenarazíme na hranici segmentu. Překročíme-li ji, vrátíme se zase na počátek. Podívejme se na jednoduchý příklad:

```
#include <iostream.h>
double * d; //V malém modelu totéž jako double near * d;
void main(){
    d = (double*)0xFFFFE; //O tomto přiřazení si povíme dále
```

```

    cout << d++ << endl;
    cout << d << endl;
}

```

Zde uložíme do blízkého ukazatele *d* adresu 0xFFFFE, která je 2 bajty pod hranicí segmentu. Přičtením 1 se má adresa zvětšit o 8 (velikost typu **double**), takže překročíme hranici segmentu. Tento prográmeček, přeložený v malém modelu, výše

```

0xffffe
0x0006

```

Výpočet adresy při adresové aritmetice tedy probíhá modulo 65536. Máme-li tedy zaručeno, že nepřesáhneme hranici segmentu, můžeme adresovou aritmetiku pro blízké ukazatele klidně používat.

Nejproblematictější je situace u vzdálených ukazatelů. Přičteme-li nějaké celé číslo ke vzdálenému ukazateli, přičte se pouze k jeho ofsetové části, segmentová se nemění. I zde přičítání probíhá modulo 65536, takže dosáhneme-li konce segmentu, vrátíme se na jeho počátek.

Vedle toho ale můžeme narazit i na problémy při porovnávání ukazatelů. Vzdálené ukazatele se totiž porovnávají jako 4bajtová celá čísla, takže obsahuje-li ukazatel *ufa* hodnotu 0:17 a ukazatel *ufb* hodnotu 1:1, bude

```

ufa != ufb,

```

i když oba ukazují na totéž místo v paměti. Proto musíme v případě, že chceme ukazatele porovnávat, nebo v případě, že můžeme překročit hranici segmentu, používat normalizované ukazatele.

Konverze ukazatelů

Konverze blízkého ukazatele na vzdálený je automatická, překladač prostě doplní segmentovou část z patřičného registru. Obrácená konverze už automatická není, neboť při ní se ztrácí důležitá informace – segmentová část adresy. Podrobněji si o tom povíme v odstavci *Ukazatele a registry*.

Ukazatele a čísla

Občas se stane, že známe číselné vyjádření adresy a potřebovali bychom je použít jako ukazatel. To jde. Nezáporné celé číslo můžeme bez problémů přetypovat na ukazatel; příklad jsme viděli v předchozí ukázce.

Přiřadíme-li takovéto číslo blízkému ukazateli, vezme je počítač jako ofsetovou část – segmentovou má, jak známo, v příslušném registru. Přiřadíme-li je vzdálenému ukazateli, doplní se segmentová část nulou.

Jestliže známe jak segmentovou tak i ofsetovou část adresy, můžeme z nich sestavit vzdálený ukazatel bez doménového typu zápisem

```

MK_FP(Segment, Ofset)

```

(Chceme-li tento zápis používat, musíme do zdrojového programu vložit hlavičkový soubor *dos.h*.)

Například obsah obrazovky je v textovém režimu na grafických kartách EGA, VGA a novějších uložen počínaje adresou 0xB800:0x0, vždy dva bajty na jeden znak. (V prvním je ASCII-kód znaku, ve druhém atributy, určující barvu znaku a pozadí – podrobně jsme si o tom povídali v kapitole 11 v knize *Práce s daty I.*) To znamená, že příkazem

```
char far* obr = (char far*)MK_FP(0xb800, 0);
*obr = '1';
```

zapišeme znak '1' do levého horního rohu obrazovky.

Na druhé straně občas potřebujeme získat samostatně segmentovou nebo ofsetovou část vzdálené adresy. K tomu nám pomohou zápisy *FP_OFF(x)* resp. *FP_SEG(x)*.

Ukazatele a registry

Při práci s blízkými ukazateli je občas třeba rozlišit, ke kterému segmentu se daný ofset vztahuje. V Borland C++ můžeme k tomuto účelu použít modifikátory `__ss`, `__cs`, `__ds` a `__es`, které předepisují, že deklarovaný ukazatel obsahuje ofset, vztažený k segmentové části v registru *SS*, *CS*, *DS* a *ES*⁴.

Ukazatel deklarovaný příkazem

```
int __near *ui;          //Totéž jako int __ds ui;
```

se považuje za ukazatel, vztažený k registru *DS*.

Ve velkém modelu proto bude následující konstrukce chybná:

```
int near *ui;
int J;                //Globální proměnná - leží v datovém segmentu
int f(){
    int j;            //Lokální proměnná - leží v zásobníkovém segmentu
    ui = &J;          //OK - vztahují se k témuž segmentu
    ui = &j;          //Chyba - vztahují se k různým segmentům
}
```

V předposledním příkazu přiřazujeme ukazateli *ui*, sdruženému implicitně s datovým segmentem, adresu globální proměnné *J* – to je v pořádku. V posledním řádku však přiřazujeme témuž ukazateli adresu lokální proměnné *j*, tj. proměnné, která leží v zásobníkovém segmentu – ale *ui* je vztahován k datovému segmentu. To je chyba.

Pokud bychom chtěli ve velkém modelu používat blízké ukazatele na lokální proměnné, museli bychom je deklarovat příkazem

```
int __ss *ulp;
```

Pak bude přiřazení

⁴ *ES* je další z tzv. segmentových registrů procesorů 80*86. Není obvykle sdružen se žádným pevným segmentem, proto jsme o něm dosud nemluvili.

```
ulp = &j;
```

správné, ale přiřazení

```
ulp = &J;
```

nikoli, neboť proměnná *J* leží v datovém segmentu, ale *ulp* se vztahuje k zásobníkovému.

Poznamenejme, že Borland C++ umožňuje také deklarovat ukazatele, které obsahují pouze segmentovou část adresy. K tomu se používá modifikátor `__seg`. Pro tyto ukazatele nelze používat běžnou adresovou aritmetiku. Lze je však sčítat s blízkými ukazateli, a výsledkem je vzdálený ukazatel, jehož segmentová část je dána ukazatelem s modifikátorem `__seg` a ofsetová část přičteným blízkým ukazatelem.

Na závěr si dovolíme jedno doporučení: Dokud nezískáte dostatek zkušeností s ukazateli vůbec, používejte raději jen implicitní velikost ukazatelů.

Paměťové modely

Nevýhodou segmentové architektury je relativní těžkopádnost adresových operací ve chvíli, kdy velikost programu nebo dat překročí magickou hodnotu 64 KB. Pak musíme umísťovat jednotlivé moduly do různých segmentů a přitom mohou nastat problémy při vzájemném porovnávání ukazatelů.

Modul musíme navrhout vždy tak, aby se všechny jeho funkce vešly do jednoho segmentu a aby se všechna jeho statická data vešla také do jednoho segmentu. (V Pascalu se musí do jednoho segmentu vejít všechna statická data všech modulů dohromady.) To je nejobecnější řešení, ale také řešení nejpomalejší a nejnáročnější na paměť. C++ proto nabízí šest paměťových modelů, které můžeme používat podle okolností a potřeb svého programu.

V Borland C++ 3.1 nastavujeme paměťový model v dialogovém okně *Code Generation*, které vyvoláme příkazem menu *Option | Compiler* (v BC++ 4.0 a pozdějších *Option | Project | 16-bit Compiler | Memory Modell*).

Jednotlivé paměťové modely se liší implicitním zacházením s ukazateli (přitom se rozlišují ukazatele na data a ukazatele na kód, tj. na funkce), maximální možnou velikostí jednotlivých částí programu a dalšími podrobnostmi. Mohou se také v drobnostech lišit i mezi jednotlivými výrobci překladačů, základní koncepce však zůstává.

Drobný model (tiny)

V tomto modelu se musí celý program (kód, data, zásobník) vejít do jednoho segmentu. To znamená, že jeho celková velikost je omezena na 64 KB a registry *SS*, *DS* a *CS* obsahují tutéž adresu. Všechny ukazatele jsou implicitně blízké.

Tento model se hodí pro velmi malé programy; jako jediný umožňuje vytvořit spustitelný soubor nejen ve formátu EXE, ale také COM.

Malý model (small)

V tomto modelu máme k dispozici 64 KB paměti pro kód a 64 KB pro data (datový segment a zásobník dohromady). Segmentové registry nyní mohou obsahovat různé hodnoty. Všechny ukazatele jsou implicitně blízké. Podobně jako v drobném modelu zde můžeme na segmentové části adres prostě zapomenout, počítač se o ně stará sám.

Tento model se hodí pro středně velké programy. Borlandské překladače jej používají jako implicitní. V našich dosavadních programech jsme používali právě tento model.

Střední model (medium)

Tento model se hodí pro rozsáhlé programy, ve kterých nepoužíváme příliš mnoho dat. Kód může být v několika segmentech (celková velikost kódu je omezena 1 MB), data se ale musejí vejít do jednoho 64KB segmentu. Ukazatele na kód jsou proto implicitně vzdálené, ukazatele na data blízké.

Kompaktní model (compact)

Kompaktní model je v jistém smyslu opakem středního modelu: data (včetně dynamicky alokovaných) mohou zabírat až 1 MB, ale kód se musí vejít do 64 KB. Celkový objem globálních a lokálních statických proměnných ovšem nesmí překročit 64 KB. Ukazatele na kód jsou proto implicitně blízké, ukazatele na data vzdálené. Hodí se pro nepříliš rozsáhlé programy, které zpracovávají velké objemy dat.

Velký model (large)

Ve velkém modelu je velikost kódu i celková velikost dat (včetně dynamicky alokovaných) omezena hranicí 1 MB. Celkový objem globálních a lokálních statických proměnných ovšem stále nesmí překročit 64 KB. Všechny ukazatele jsou implicitně vzdálené. Hodí se pro rozsáhlé aplikace.

Rozsáhlý model (huge)

V tomto modelu je velikost kódu i dat omezena hranicí 1 MB. Navíc celkový prostor, nárokováný globálními proměnnými a lokálními statickými proměnnými dohromady, smí překročit 64 KB. Všechny ukazatele jsou implicitně vzdálené. Všechny funkce se implicitně překládají jako robustní (*huge* – viz dále).

Smíšené programování

Pokud píšeme celý program v jednom modelu, nemusíme se příliš starat o velikost ukazatelů, stačí dát si pozor na adresovou aritmetiku. Občas se ale stane, že potřebujeme k programu, napsanému např. v malém modelu, připojit knihovnu, přeloženou ve velkém modelu. V takovém případě si musíme uvědomit, že v připojované knihovně jsou všechny ukazatele a reference implicitně vzdálené, a ve svém programu je jako takové deklarovat. (Podobně musíme i funkce deklarovat jako vzdálené – ale o tom si povíme dále.)

1.5 Ukazatele na procedury a funkce

Ukazovat můžeme nejenom na data, ale také na procedury a funkce. Tyto ukazatele využijeme v situacích, kdy v době psaní programu ještě přesně nevíme, kterou z dané množiny procedur a funkcí budeme chtít na daném místě použít, a program si bude zjišťovat až v průběhu výpočtu, která to bude. Máme pak dvě možnosti: buď v dané části programu použijeme přepínač a podle hodnoty nějakého příznaku vybereme z množiny vhodných funkcí nebo použijeme ukazatel a vyvoláme tu proceduru či funkci, na kterou v danou chvíli tento ukazatel ukazuje. Na konkrétní situaci záleží, které řešení je vhodnější.

Druhou oblastí, v níž jsou ukazatele na procedury a funkce přímo neocenitelným pomocníkem, je tvorba funkcí s modifikovatelnými vlastnostmi. Typickým příkladem jsou například funkce pro třídění vektorů podle nějakého kritéria. Existují obecné třídící algoritmy a bylo by tedy žádoucí, abychom uměli naprogramovat také obecné třídící funkce. (Jednu takovou třídící funkci najdete v doprovodném programu.)

Podívejme se nyní na specifika těchto ukazatelů v obou probíraných jazycích.

Autoři Pascalu skutečnou podstatu těchto objektů před programátory zamlčují a v manuálu hovoří o **procedurálních objektech** nebo o objektech typu procedura či funkce. Když se však na věc podíváte podrobněji, zjistíte, že se ve skutečnosti nejedná o nic jiného, než o ukazatele na procedury a funkce. Syntax jazyka ovšem tuto skutečnost úspěšně maskuje a při práci s proměnnými procedurálních typů se nepoužívá operátor dereferencování⁴.

Definice (popis) procedurálních typů je jednoduchá: napíšeme hlavičku podprogramu, ve které vynecháme jeho identifikátor, avšak uvedeme seznam všech parametrů a u funkce také typ funkční hodnoty. Z této hlavičky překladač odvodí, zda se jedná o (ukazatel na) proceduru nebo funkci, kolik má daná procedura či funkce parametrů a jakého typu jednotlivé parametry jsou. Například takto:

```
type
  fun = function (var i: integer): integer;
```

Proměnným procedurálních typů můžeme přiřazovat procedury či funkce, jejichž specifikace odpovídá definici příslušného typu (ve skutečnosti se však přiřadí pouze adresa podprogramu). Kromě toho můžeme proměnné procedurálních typů navzájem porovnávat a můžeme je volat. Chceme-li volat proceduru či funkci, na niž ukazuje proměnná procedurálního typu, napíšeme identifikátor této proměnné a za něj do závorek seznam skutečných parametrů volané funkce.

Aby to však na druhou stranu nebylo zase tak jednoduché, funkce nemohou v Turbo Pascalu vracet hodnoty procedurálních typů. Chcete-li proto vybrat žádanou proceduru či funkci nějakým podprogramem, musíte mu předat parametr procedurálního typu, který budete specifikovat jako výstupní (**var**).

Asi by bylo vhodné zdůraznit, že identifikátory parametrů v definici procedurálního typu slouží pouze k vyhovění syntaktickým požadavkům a překladač je vzápětí zapomíná. Tyto identifikátory se tedy mohou lišit od identifikátorů odpovídajících parametrů procedur a funkcí, které proměnným daného procedurálního typu přiřazujeme.

Všechny vyložené vlastnosti procedurálních typů shrnuje následující ukázka:

```
(*   Příklad P1 - 4   *)
Program Pascal;
{$F+}                                {Funkce a procedury překládáme jako
vzdálené}
type
  int = integer;
  Tifii = function( a:int; b:int ):int;
  {Funkce se dvěma celočíselnými parametry vracející celé číslo}
  Tproc = procedure( var f:Tifii; n:char );
  {Procedura se dvěma parametry: výstupním parametrem typu Tifii
   a vstupním parametrem typu char}
var
  Operace : Tproc;
function (*****) Krat( a:int; b:int ):int; (*****)
begin
  Krat := a * b;
end;
function (*****) Plus( x:int; y:int ):int; (*****)
begin
  Plus := x + y;
end;0
procedure (*****) VratFci (*****)
{Nastaví parametr Fce na požadovanou funkci}
( var Fce:Tifii; c:char );
begin
  case c of
    '+': Fce := Plus;
    '*': Fce := Krat;
  end;
end;
procedure (*****) Vytiskni (*****)
{Provede nad parametry "x" a "y" operaci o a vrátí výslednou hodnotu}
( x:int; o:char; y:int );
var
  op:Tifii;
begin
  Operace( op, o );
  writeln( x, ' ', o, ' ', y, ' = ', op( x, y ) );
end;
(*****) HLAVNI PROGRAM (*****)
begin
  Operace := VratFci;
  Vytiskni( 3, '*', 5 );
  Vytiskni( 3, '+', 5 );
end.
```

V C++ si ukazatele na funkce na nic nehrají a zůstávají i nadále ukazateli se všemi odpovídajícími vlastnostmi. Deklarují se obdobně jako ukazatele na vektory, pouze místo in-

dexových závorek zapíšeme kulaté závorky a v nich seznam formálních parametrů. (Identifikátory parametrů jsou zde nepovinné, stačí uvádět pouze jejich typy.)

Chceme-li použít funkci, na kterou ukazuje daný ukazatel, použijeme na ukazatel operátor dereferencování a na výsledek pak operátor volání funkce. Přitom musíme ukazatel spolu s dereferenčním operátorem uzavřít do závorek, neboť operátor „*“ má nižší prioritu než operátor „()“. Ukazuje-li např. ukazatel *Uk* na funkci se dvěma celočíselnými parametry, musíme pro parametry 7 a 5 zavolat tuto funkci příkazem

```
(*Uk) ( 7, 5 );
```

Tuto možnost znaly již nejstarší verze jazyka C. V ANSI C a v C++ můžeme ovšem závorky vynechat a aplikovat operátor volání funkce přímo na ukazatel. Jinými slovy, můžeme napsat

```
Uk ( 7, 5 );
```

Poznamenejme, že kdybychom použili první možnost a zapomněli na závorky, tedy kdybychom napsali

```
* Uk( 7, 5 );
```

požadovali bychom po překladači, aby zavolał funkci *Uk*, předal jí parametry 7 a 5 a vrácenou funkční hodnotu dereferencoval.

Před příkladem předvádějícím použití ukazatelů na funkce ještě dodáme, že na rozdíl od Pascalu může funkce v C++ vracet i ukazatel na funkci. Zároveň připomínáme, že pokud v programu použijeme samotný identifikátor funkce, tj. bez závorek s parametry, pracuje s ním překladač jako s konstantou, jejíž hodnotou je adresa dané funkce.

```
/* Příklad C1 - 4 */
#include <constrea.h>

typedef int (* Tifii)( int, int );
//Objekty typu Tifi jsou ukazatele na funkci se
//dvěma celočíselnými parametry, která vrací celé číslo

typedef void (* TPROC)( Tifii&, int );
//Objekty typu TPROC jsou ukazatele na proceduru se dvěma
//parametry: vstupně-výstupním parametrem typu Tifii
//a vstupním parametrem typu char

int (*Uifii)( int, int );
//Proměnná Uifii je ukazatel na funkci se dvěma
//celočíselnými parametry, která vrací celé číslo

int *Fifii( int, int );
//Fifii je funkce se dvěma celočíselnými parametry,
//která vrací ukazatel na celé číslo

TPROC PasProc; //Proměnná typu TPROC
Tifii (*Operace)( char );
//Proměnná Operace je ukazatel na funkci s jedním celočíselným
//parametrem, která vrací hodnotu typu Tifii
```

```

int (* (*Op2)(char) )( int, int );
//Týž typ definovaný přímo bez pomoci deklarace typedef

int /*****/ Krat /*****/
( int a, int b )
{
    return( a * b );
}
/*****/ Krat *****/

int /*****/ Plus /*****/
( int x, int y )
{
    return( x + y );
}
/*****/ Plus *****/

Tifii /*****/ VratFci /*****/
//Nastaví parametr Fce na požadovanou funkci
( char c )
{
    switch( c )
    {
        case '*': return Krat;
        case '+': return Plus;
    }
    return 0;
}
/*****/ VratFci *****/

constream crt;          // Proud na konzolu

void /*****/ Vytiskni /*****/
( int x, char o, int y )
{
    Tifii op = Operace( o );
    crt << x << ' ' << o << ' ' << y << " = "
        << (*op)( x, y ) << endl;
    //Předchozí dva příkazy by zkušenější programátor nahradil příkazem
    //crt << x << ' ' << o << ' ' << y << " = "
    // << (*Operace(o))( x, y ) << endl;
}
/*****/ Vytiskni *****/

void /*****/ main /*****/ ( )
{
    Operace = VratFci;
    Vytiskni( 3, '*', 5 );
    Vytiskni( 3, '+', 5 );
}

```

Blízké, vzdálené a robustní funkce

Při volání funkcí i při návratu z nich se používají ukazatele, a ty mohou být – stejně jako ostatní – blízké nebo vzdálené. (Adresová aritmetika nemá pro ukazatele na kód smysl,

proto se nezavádějí ani normalizované ukazatele na kód.) Podle toho se rozlišují blízké a vzdálené funkce (*near* resp. *far function*).

Blízké a vzdálené funkce v Borland C++

Implicitně určí druh funkce překladač při překladu podle paměťového modelu: v drobném, malém a kompaktním modelu se použijí blízké funkce, ve středním a velkém vzdálené funkce. (O rozsáhlém modelu si povíme dále.)

V případě potřeby můžeme ale předepsat, jak se má určitá funkce přeložit. K tomu použijeme známé modifikátory `__near` a `__far`, které zapíšeme bezprostředně před identifikátor funkce. Například takto:

```
void __far f(void);
```

Blízké a vzdálené funkce v Pascalu

V Pascalu jsou implicitně překládány jako blízké všechny funkce, deklarované v hlavním programu a v části **implementation** jednotek. Všechny funkce, deklarované v části **interface**, jsou implicitně vzdálené.

Chceme-li toto nastavené změnit, máme několik možností. Můžeme použít volbu *Force far calls* v okně *Compiler options* (vyvoláme je z menu příkazem *Options | Compiler*) nebo použít v programu direktivu

```
{$F+}
```

(viz kap. 16). Počínaje Turbo Pascalem 6.0 můžeme také použít direktiv *near* a *far*, které se zapisují za hlavičku:

```
procedure p(x: real); far;
var
  y: real;
begin
  { ... }
end;
```

Robustní funkce

Robustní (*huge*) funkce se implicitně používají v rozsáhlém (*huge*) modelu. Program v tomto modelu může mít několik datových segmentů, a proto si každá funkce musí před začátkem výpočtu nastavit do registru *DS* segmentovou adresu toho datového segmentu, se kterým pracuje. Těsně před ukončením zase obnoví původní hodnotu, která byla v *DS* uložena.

Robustní funkce jsou tedy vzdálené funkce, které se navíc starají o aktualizaci registru *DS*.

Chceme-li nějakou funkci přeložit jako robustní bez ohledu na paměťový model, použijeme modifikátor `__huge` (jeho syntax je podobná syntaxi modifikátorů `__near` a `__far`).

Smíšené programování

Pokud napíšeme celý program v jednom paměťovém modelu, nemusíme se o „vzdálenost“ funkcí starat, překladač si s tím poradí sám. Modifikátory `__near`, `__far` a `__huge` přijdou ke cti především v situaci, kdy chceme např. k programu v malém modelu připojit modul nebo knihovnu, přeloženou ve velkém modelu. (Typickým příkladem může být borlandská grafická knihovna BGI, která existuje jen v jedné verzi, se všemi funkcemi vzdálenými.)

1.6 Dynamická alokace paměti

Dosud jsme se setkávali se dvěma druhy objektů: se statickými objekty, které existovaly po celou dobu běhu programu, a s automatickými objekty, které automaticky vznikaly při vstupu programu do jejich oblasti platnosti a při výstupu z ní zanikaly.

V programech se používá ještě třetí druh objektů: tzv. **dynamické objekty**, které vznikají a zanikají ve chvíli, kdy k tomu programátor dá příkaz. Dynamické objekty se alokují (přidělí se jim paměť) na tzv. haldě (*heap*), což je oblast paměti, která slouží právě k tomuto účelu.

Ani Pascal ani C++ nezavádí dynamické proměnné (jako např. PL/1); místo toho používají pro práci s dynamickými objekty ukazatelů, do nichž se v okamžiku alokace uloží adresa zřízovaného objektu. Z toho však plyne nebezpečí, že pokud hodnotu tohoto ukazatele nějakým způsobem změníme, objekt od této chvíle přestane být přístupný a my jej dokonce ani nemůžeme zrušit. Objekt pak na haldě „zaclání“ a blokuje paměť pro ostatní objekty.

V Turbo Pascalu se pro práci s haldou používá šesti procedur a dvou funkcí, které si postupně probereme:

procedure *New* (**var** *p*: *Pointer*);

Ačkoliv to deklarace nenaznačuje, očekává tato procedura jako parametr vázaný ukazatel. Vyhradí v paměti prostor pro jeden objekt základního typu tohoto ukazatele a do *p* uloží jeho adresu, takže *p* bude ukazovat na nově vytvořený objekt.

procedure *Dispose*(**var** *p*: *Pointer*);

Uvolní paměť alokovanou pro objekt, na nějž ukazuje **vázaný** ukazatel *p*. **Nedoporučuje se používat současně s procedurami *Mark* a *Release*!** Pomocí procedury *Dispose* bychom měli uvolňovat paměť, alokovanou procedurou *New*.

procedure *GetMem* (**var** *p*: *Pointer*; *n*: *word*);

Vyhradí na haldě objekt o velikosti *n* bajtů a jeho adresu uloží do ukazatele *p*. Tuto proceduru využijete např. v situaci, kdy chcete alokovat pouze část deklarovaného pole. (De-

klarujete pole např. maximální přípustné velikosti, ale v průběhu programu zjistíte, že z něj budete potřebovat pouze několik prvních položek – viz uká.))

procedure *FreeMem*(var *p*: *Pointer*; *n*: *word*);

Uvolní paměť *n* bajtů velkého objektu, na nějž ukazuje ukazatel *p*. **Nedoporučuje se používat současně s procedurami *Mark* a *Release*!** Pomocí této procedury bychom měli uvolňovat paměť, alokovanou procedurou *GetMem*.

procedure *Mark* (var *p*: *Pointer*);

Nastaví ukazatel *p* na počátek volného prostoru na haldě pro možnost jeho pozdějšího opětovného uvolnění procedurou *Release*.

procedure *Release*(var *p*: *Pointer*);

Uvolní haldu od místa označeného ukazatelem *p*. Předpokládá se, že tento ukazatel byl nastaven procedurou *Mark*.

function *MaxAvail* : *word*;

Vrátí velikost největšího souvislého bloku volné paměti – tj. např. velikost největší alokovatelné proměnné.

function *MemAvail* : *word*;

Vrátí součet velikostí všech volných bloků paměti, tedy celkovou velikost volné paměti v haldě.

V předchozích popisech procedur pro správu dynamické paměti jsme uvedli, že se nedoporučuje používat procedury *Dispose* a *FreeMem* současně s procedurami *Mark* a *Release*. To proto, že procedury *Mark* a *Release* jsou schopny uvolnit pouze konec haldy, kdežto procedury *Dispose* a *FreeMem* dokáží uvolnit paměť i kdekoliv uprostřed. Pokud byste použili proceduru *Release* po použití procedury *Dispose* nebo *FreeMem*, mohla by zničit některé informace, které si tyto procedury uschovaly, a další chování systému by se mohlo stát nepředvídatelným. Dohodněme se proto, že v naší knize procedury *Mark* a *Release* používat nebudeme, a kdo je bude chtít použít, najde si potřebné informace v příručce.

Při používání alokačních procedur může nastat situace, že chcete alokovat nějaký objekt, ale systém již nemá k dispozici dostatek volné paměti. V takovém případě způsobí alokační procedury v Pascalu běhovou chybu 203 (*Heap overflow error*, tedy vyčerpání haldy). Před alokací paměti bychom si tedy vždy měli nejprve zjistit, zda máme k dispozici dostatek volné paměti.

Příklad použití dynamické alokace paměti je v následující ukázce:

```
(* Příklad P1 - 5 *)
{Alokace pole předem neznámé délky}
type
```

```

TVec = array [ 1..10000 ] of real;
var
  N : integer;
  V : ^TVec;
begin
  write( 'Zadej počet čísel: ' );
  read( N ); {Zjistíme, je-li dostatek paměti}
  if(MaxAvail < N*sizeof(real)) then
  begin
    writeln( 'Nedostatek paměti!' ); {a pokud není,}
    halt(1); {ukončíme program s chybovým kódem 1}
  end;
  GetMem( V, N*sizeof(real) ); {Je-li vše OK, alokujeme paměť}
  {
    Zde by měl být vlastní program pracující s vektorem V^[ 1..N ]
  }
  FreeMem( V, N*sizeof( real ) );{Nakonec paměť uvolníme}
end.

```

V C++ pracujeme s haldou podobně jako v Pascalu. I zde existuje několik možností alokace. Nejvhodnější je použití operátoru **new**, který se chová jako prefixový unární operátor, jehož parametrem je typ alokovaného objektu a který vrací ukazatel vázaný na tento typ, přičemž vrací adresu alokovaného objektu, nebo v případě, že se objekt nepodařilo alokovat, prázdný ukazatel (0).

Pokud se chystáme alokovat vektor prvků daného typu, uvede se rozměr tohoto vektoru v hranatých závorkách za identifikátorem typu prvků vektoru – např. ~~příkzem~~

```
UkaNaInt = new int [ 7 ];
```

alokujeme pole sedmi celých čísel a jeho adresu (tj. adresu jeho nultého prvku) přiřazujeme proměnné *UkaNaInt*, je typu **int***.

Při alokaci skalárních typů nám dokonce operátor **new** nabízí inicializace vytvářené proměnné. Počáteční hodnotu píšeme do kulatých závorek za identifikátor typu alokovaného objektu. Celé číslo s počáteční hodnotou 12 bychom tedy alokovali např. ~~příkzem~~

```
ui = new int (12);
```

K uvolnění (dealokaci) paměti slouží operátor **delete**, za nímž uvedeme identifikátor ukazatele na uvolňovaný objekt. Pokud je alokovaným objektem vektor, liší se požadovaná syntax jazyka podle toho, kterou verzi definice jazyka C++ váš překladač implementuje; mezi verzemi Cfront⁵ 2.0 a 2.1 totiž došlo ke změně. Dříve se mezi operátor **delete** a identifikátor ukazatele na dealokovaný objekt psaly složené závorky, v nichž byl uveden počet prvků dealokovaného vektoru. Od verze 2.1 se tyto závorky nechávají prázdné – velikost dealokované paměti si překladač umí zjistit sám.

⁵ Vývojové verze jazyka C++ se označují např. Cfront 2.1 nebo AT&T 2.1. Cfront je označení překladače, který převádí program z C++ do jazyka C, a který je vyvíjen v Bellových laboratořích firmy AT&T.

Poznámka:

Pozor! Nepleťte si verzi jazyka a verzi překladače. Borlandské překladače Turbo C++ 1.0 a Borland C++ 2.0, implementovaly verzi jazyka C++ označovanou Cfront 2.0. Počínaje Borland C++ 3.0 implementují verzi Cfront 2.1. Tuto verzi implementuje i Microsoft C/C++ 7.0.

Předchozí dva objekty bychom tedy uvolnili příkazy (na pořadí nezáleží)

```
delete ui;
delete [N] UkNaInt;           //Do verze 2.0
delete [] UkNaInt;           //Od verze 2.1
```

Pokud použijeme první podobu příkazu (tj. podobu pro starší verze jazyka) v borlandských překladačích od verze 3.0, vydá překladač varovné hlášení „*Array size for 'delete' ignored*“ a bude se tvářit, že jsme mezi složené závorky nic nenapsali.

Následující příklad předvádí ukázkou dynamické alokace paměti.

```
/*   Příklad C1 - 5   */
#include <constrea.h>

constream crt;                //Definice datového proudu

int /*****/ main /*****/ ()
{
    int N;
    crt << "Zadej počet čísel: ";
    cin >> N;
    double *V = new double[ N ];
    if( !V )                    //Test úspěšnosti alokace
    {
        crt << "Nedostatek paměti!" << endl;
        return 1;              //Ukončí program s chybovým kódem 1
    }
    //
    //Vlastní program pracující s vektorem (*V)[N];
    //
    delete [] V;                //Ve starších verzích C++ bychom zde psali
    //delete [N] V;
    return 0;                    //Ukončí program s chybovým kódem 0
}
/*****/ main *****/
```

Všimněte si testu úspěšnosti alokace

```
if( !V ) ...
```

Pokud se alokace nepodařila, obsahuje V hodnotu 0, tedy prázdný ukazatel (ukazatel nikam). Ukazatele můžeme – podobně jako čísla – používat v C++ v podmínkách. Obsahuje-li ukazatel platnou adresu, interpretuje se jako *ANO*, obsahuje-li 0, interpretuje se jako *NE*. Na ukazatele můžeme také aplikovat logické operátory $\&\&$, $\|$ a $!$. Např. výraz $!V$ má hodnotu *ANO*, jestliže V je prázdný ukazatel, a *NE*, obsahuje-li V platnou adresu.

Už jsme se zmínili, že C++ nabízí ještě druhou možnost – alokovat objekty pomocí knihovnických funkcí, zděděných po jazyku C. Knihovna jazyka C nám pro správu dynamické paměti (haldy) nabízí následující funkce (v komentářích jsou uvedeny hlavičkové soubory, v nichž najdeme jejich prototypy):

```
unsigned coreleft(); //alloc.h
```

Vrátí počet bajtů od konce haldy do konce vyhrazeného paměťového prostoru.

```
void * malloc( size_t PocetBajtu ); //alloc.h, stdlib.h
```

Vyhradí na haldě prostor *PocetBajtu* velký a vrátí ukazatel na jeho počátek. Pokud na haldě není dostatek místa, vrátí hodnotu *NULL*.

```
void * realloc( void *Blok, size_t PocetBajtu ); //alloc.h, stdlib.h
```

Změní (zvětší nebo zmenší) velikost paměťového prostoru vyhrazeného pro objekt, který je již na haldě a na nějž ukazuje parametr *Blok*, na *PocetBajtu*, a v případě, že bude objekt přemístěn, přesune do nového paměťového prostoru i jeho původní obsah. Pokud blok nemůže být realokován nebo pokud je *PocetBajtu* = 0, vrátí hodnotu *NULL*. Pokud je *Blok* prázdný ukazatel (má hodnotu *NULL*), chová se *realloc* jako *malloc*.

```
void free( void *Blok ); //alloc.h, stdlib.h
```

Uvolní z haldy objekt, na nějž ukazuje parametr *Blok*.

Ke všem těmto funkcím existují v Borland C++ ještě „vzdálené“ varianty (*farcoreleft()*, *farmalloc()*, *farrealloc()*, *farfree()*), určené pro práci se vzdálenými ukazateli.

Kromě výše uvedených funkcí najdete v knihovně ještě několik dalších, z nichž některé se liší opravdu jen nepatrně. Pokud budete mít opravdu speciální požadavky, může si jejich popis najít v referenční příručce nebo v nápovědě. Všem však vřele doporučujeme používat operátory **new** a **delete** ve všech situacích, kdy s nimi vystačíte, a k výše uvedeným funkcím se uchýlovat jen v případech, které jinak řešit nelze.

1.7 Konstantní ukazatele a ukazatele na konstanty

Již na počátku této knihy, když jsme se začali zabývat daty, jsme si řekli, že objekty, s nimiž v našich programech pracujeme, můžeme rozdělit na konstanty a proměnné. Veškerý náš předchozí výklad o ukazatelích se však doposud zabýval pouze proměnnými. V této podkapitole se tedy budeme věnovat konstantám. Tím se ovšem okruh zájemců automaticky omezuje pouze na programátory v C++, protože Pascal konstantní ukazatele ani ukazatele na konstanty nezavádí.

Pokud hovoříme o konstantách v souvislosti s ukazateli, musíme rozlišovat tři druhy konstantnosti:

1. Ukazatele na konstanty. Jejich hodnota se může měnit a v průběhu programu mohou postupně ukazovat na různé konstanty. Překladač vám však nedovolí změnit hodnotu objektu, na nějž tyto ukazatele ukazují.
2. Konstantní ukazatele, jejichž hodnota se změnit nedá, takže ukazují neustále na stejný objekt. Můžeme však změnit hodnotu odkazovaného objektu, protože ten konstantní není.
3. Konstantní ukazatele na konstanty, u nichž nemůžeme změnit vůbec nic.

To, který ze tří výše uvedených druhů konstantnosti deklarujeme, rozlišíme umístěním klíčového slova **const** v deklaraci. Obecně bychom mohli říci, že konstantní bude to, co je deklarováno vlevo od klíčového slova **const**, a pokud je v deklaraci uvedeno klíčové slovo **const** jako první, je to totéž, jako kdyby bylo uvedeno jako druhé. To znamená, že deklaracemi

```
int const * UkNaKonst;
```

a

```
const int * UkNaKonst2;
```

deklarujeme ukazatel na konstantu typu **int**, kdežto deklarací

```
int * const KonstUk;
```

deklarujeme konstantní ukazatel na celé číslo. Pokud bychom chtěli deklarovat konstantní ukazatel na konstantu typu **int**, museli bychom použít deklaraci

```
int const * const KonstUkNaKonst;
```

nebo

```
const int * const KonstUkNaKonst2;
```

2. Záznamy, struktury a množiny

Záznamy (Pascal) a **struktury** (C++) jsou po vektorech druhým strukturovaným datovým typem, s nímž se podrobněji seznámíme. Někdy se jim říká heterogenní pole. U záznamů a struktur však nemluvíme o prvcích, ale o **složkách**, a na rozdíl od polí (přesněji homogenních polí), které mají všechny prvky stejného typu, může obecně mít každá složka záznamu nebo struktury jiný typ.

Typickým příkladem záznamu (struktury), uváděným snad ve všech učebnicích, je záznam o pracovníkovi, který kromě textových údajů (jméno, adresa) obsahuje i údaje číselné (měsíční plat, počet dnů nevybrané dovolené, atd.), nebo dokonce údaje speciální (např. stav: svobodný, ženatý, rozvedený, vdovec).

Poznámka:

*V dalším výkladu budeme rozlišovat termíny **strukturový typ** a **strukturovaný typ**. **Strukturovými typy** budeme rozumět datové typy deklarované v Pascalu pomocí klíčového slova **record** a v C++ pomocí některého z klíčových slov **struct**, **union** a **class** (s posledními dvěma se seznámíme později). Naproti tomu mezi strukturované typy budeme kromě strukturových typů zařazovat i vektory a další typy s definovanou vnitřní strukturou (řetězce, množiny, atd.).*

2.1 Deklarace záznamů a struktur

Než se pustíme do výkladu o deklaraci záznamu, musíme zdůraznit, že v Pascalu **funkce nemohou vracet hodnoty strukturovaných typů**; mohou vracet pouze ukazatele na tyto hodnoty⁶.

V Pascalu deklarujeme strukturové typy tak, že napíšeme klíčové slovo **record**, za ním deklarujeme jednotlivé složky záznamu, a celou deklaraci ukončíme klíčovým slovem **end**.

Záznam nemusíme vždy definovat jako samostatný datový typ v sekci **type**, ale můžeme jej deklarovat (stejně jako pole a jiné typy) přímo v definici zaváděné proměnné. Musíme se však smířit s tím, že při tomto postupu vyvstávají známé problémy s nekompatibilitou datových typů. Navzájem kompatibilní jsou pak pouze proměnné, které byly deklarovány společně. Obecně však tento způsob deklarací vřele nedoporučujeme.

Složkou záznamu nemusí být pouze objekty skalárních typů. Může jí být i pole, řetězec nebo další záznam, přičemž datové typy těchto složek mohou být buď již předem definované, nebo mohou být definované přímo v deklaraci odpovídajících složek záznamu. Znovu však musíme upozornit na možnou nekompatibilitu.

⁶ Funkce v Turbo Pascalu smějí vracet pouze skalární typy, ukazatele (nikoli však ukazatele na procedury a funkce, neboť s nimi Pascal nezachází jako s ukazateli) a typ *string*.

Podívejme se na příklady deklarací záznamů:

```

type
  TStav = ( SVOBODNY, ZENATY, ROZVEDENY, VDOVEC );

  TDatum = record
    Den : byte;
    Mes : byte;
    Rok : word;
  end;

  TJmena = record
    Krestni : String[ 10 ];
    Prijmeni : String[ 20 ];
    Rodne : String[ 20 ];
    Tituly : String[ 10 ];
  end;

  TZazOPrac = record
    Jmeno : TJmena;
    Narozen : TDatum;
    Stav : TStav;
    Nastoupil : TDatum;
    Mzda : integer;
  end;

var
  Zam: array[ 1..MAX_ZAM ] of TZazOPrac;
  Prac: TZazOPrac;
  uPrac: ^TZazOPrac;
  Zlomek1, Zlomek2, Zlomek3: record Citatel, Jmenovatel: integer; end;

```

V C++ deklarujeme strukturové typy (zkráceně struktury) tak, že napíšeme klíčové slovo **struct** následované identifikátorem definovaného typu a za ním ve složených závorkách deklarujeme jednotlivé složky deklarované struktury. Deklarace musí končit středníkem.

Kromě toho lze v C++ deklarovat strukturové datové typy také za pomoci klíčových slov **union** (unie) a **class** (třídy); k uniím se ještě vrátíme, se třídami se můžete seznámit v knize o objektově orientovanému programování (*Objektové programování 1*, Grada 1996).

Na rozdíl od Pascalu, kde může být definice datového typu součástí deklarace proměnné, je tomu v C++ právě naopak: deklarace proměnných může být součástí definice strukturového typu (s tímto přístupem jsme se již setkali při výkladu výčtových datových typů).

Složkou záznamu nemusí být pouze objekty skalárních typů. Může jí být i pole, řetězec nebo další záznam. Datové typy těchto složek však musí být již předem definované.

Na rozdíl od Pascalu **mohou funkce v C++ vracet hodnoty strukturových typů** (nikoli strukturovaných, neboť funkce nemohou vracet hodnoty vektorových typů). Jazyk však, bohužel, neposkytuje mechanismus, jak pracovat v těle funkce přímo s vrácenou hodnotou, takže pokud nechceme používat nějaké implementačně závislé konstrukce, musíme definovat lokální proměnnou návratového typu, do ní vrácenou hodnotou uložit a tuto

proměnnou pak v příkazu **return** předat. Přitom překladač zabezpečí, aby se její hodnota zkopírovala na místo, kde ji volající podprogram očekává.

Podívejme se na příklad deklarací struktur v jazyce C++:

```
enum eStav = {SVOBODNY, ZENATY, ROZVEDENY, VDOVEC };

struct sDatum
{
    unsigned char Den;
    unsigned char Mes;
    unsigned Rok;
};

struct sJmeno
{
    char Krestni [ 11 ];
    char Prijmeni [ 21 ];
    char Rodne [ 21 ];
    char Tituly [ 11 ];
};

struct sZazOPrac
{
    sJmeno Jmeno;
    sDatum Datum;
    eStav Stav;
    eDatum Nastoupil;
    unsigned Mzda;
};

sZazOPrac Zam[ MAX_ZAM];
sZazOPrac Prac;
sZazOPrac uPrac;

struct sZlomek
{
    int Citatel;
    unsigned Jmenovatel;
} Zlomek1, Zlomek2, Zlomek3;
```

V obou jazycích platí, že na rozdíl od polí, v nichž má každá složka definované pořadí a odkazujeme na ni prostřednictvím indexů, má v záznamu každá složka svůj vlastní identifikátor a tím se na ni také odkazujeme. Na rozdíl od polí, u nichž se index položky uvádí v hranatých závorkách za jménem pole, se při práci se složkami záznamu používá infixový operátor tečka (nazývaný selektor složky záznamu). Jeho levým operandem je záznam, jehož položku chceme získat, a pravým operandem identifikátor zpracovávané položky, tedy např.

```
while( IdZaznamu.IdPolozky > 0 ) ...
```

přičemž identifikátor záznamu nazýváme **kvalifikátorem** a celou dvojici pak **kvalifikovaným objektem** – proměnnou nebo konstantou.

2.2 Příkaz with

V programech se často setkáváme se situacemi, kdy potřebujeme po nějakou dobu pracovat s různými složkami téhož záznamu. Abychom nemuseli psát v příkazech stále dokořka identifikátor téhož záznamu, s jehož složkami pracujete, tj. abychom nemuseli složky tohoto záznamu pokaždé kvalifikovat, nabízí nám jazyk Pascal strukturovaný příkaz **with**.

Použitím příkazu **with** si však často nejen ušetříte práci při psaní programu, ale zároveň i zvýšíte efektivitu programu. Pokud je označený záznam např. prvkem nějakého pole nebo získáváte jeho adresu vyhodnocením složitějšího ukazatelového výrazu, provede se potřebná indexace či vyhodnocení adresového výrazu pouze jednou při vstupu do příkazu.

Syntax příkazu **with** je formálně podobná syntaxi příkazu **while**. Nejprve napíšeme klíčové slovo **with**, za ním identifikátor záznamu, s jehož složkami chceme pracovat, pak klíčové slovo **do** a nakonec příkaz (nejspíše složený), v němž chceme složky tohoto záznamu použít.

V těle příkazu **with** se překladač před interpretací každého identifikátoru nejprve podívá, zda nejde o identifikátor některé ze složek záznamu uvedeného v hlavičce příkazu **with**. Je-li tomu tak, je tento objekt identifikován jako složka záznamu, a to i v případě, že v dané oblasti platnosti existuje proměnná stejného jména. (Ta je uvnitř příkazu **with** nepřístupná.)

Chceme-li si ušetřit kvalifikaci složek několika záznamů, můžeme do sebe vnořit několik příkazů **with**, přičemž z logiky věci vyplývá, že identifikátory v těle příkazu **with** se budou nejprve porovnávat s identifikátory složek identifikačního záznamu nejnvnitřnějšího **with**, pak s identifikátory složek vnějšího **with** a tak dále, až se vyčerpají všechny úrovně vnoření.

Stejného efektu jako při vnořování příkazu **with** dosáhneme i tak, že v hlavičce příkazu **with** uvedeme seznam všech záznamů, s jejichž složkami chceme pracovat bez explicitní kvalifikace, přičemž jednotlivé identifikátory v seznamu oddělíme čárkami. Překladač pak s takovýmto příkazem pracuje stejně jako s posloupností vnořených příkazů **with**, přičemž za nejnvnitřnější považuje ten, který je v seznamu uveden jako poslední.

Před ukázkou použití příkazu **with** zbývá ještě podotknout, že kvalifikačním záznamem vnořeného **with** může být i záznam, který je složkou kvalifikačního záznamu některého nadřazeného **with**.

```
for i:=1 to Zamestnancu do
  with Zam[i], Jmeno, Narozen do
  begin
    write( i:4, '. ' );
    write( Prijmeni, ' ', Krestni, ' ', Tituly );
    write( ' ':55-Prijmeni[0]-Krestni[0]-Tituly[0] );
    writeln( Den:2, '.', Mes:2, '. ', Rok );
  end;
```

C++ žádný ekvivalent pascalského příkazu **with** nezavádí a pro účely zkrácení zápisu i zefektivnění výsledného programu využívá ukazatele na struktury (záznamy); věnujeme jim následující podkapitolu.

2.3 Ukazatele na záznamy a struktury

Ukazatele na záznamy se na první pohled nijak neliší od ostatních ukazatelů. Na druhý pohled však zjistíme, že práce s nimi s sebou nese některá specifika. Povězme si o nich.

První zvláštnost plyne z toho, že složkou záznamu sice nesmí být objekt dosud nedefinovaného typu, ale může jí být ukazatel na takovýto objekt. V požadavcích na informace o typu odkazovaného objektu se však oba jazyky liší.

Pascal, který neumožňuje datové typy pouze deklarovat, ale nedefinovat, nemá jinou možnost, než dovolit používat ukazatele i na typy, které budou deklarovány a definovány až později.

Naproti tomu C++ požaduje, aby byl odkazovaný datový typ alespoň deklarován. To zařídíme tak, že napíšeme klíčové slovo **struct** (případně **union** či **class**), za ním uvedeme identifikátor deklarovaného typu a tuto deklaraci ukončíme středníkem. Definice deklarovaného datového typu, v níž podobně popíšeme jeho strukturu, může následovat **později**.

Možná vás napadá, že definice lze seřadit vždy tak, abych v každé z nich používal pouze dříve definované datové typy. Bohužel, nemáte pravdu. Představte si program, který bude hrát piškorky. V něm budeme potřebovat datové typy *TPole* a *TPisk* pro pole a piškorku. Každé políčko potřebuje vědět, které piškorky jím prochází, a obsahuje proto ukazatele na piškorky, a na druhou stranu musí každá piškorka vědět, kterými prochází políčky, a obsahuje proto ukazatele na políčka. Podívejme se na jejich deklarace:

```
const
  PISKORKA = 5;           {Počet políček v piškorce}
  POCITAC = FALSE;
  HRAC = TRUE;

type
  THrac = boolean;
  TPole = record          {Charakteristika pole šachovnice}
    Obsah : char;         {Nic nebo znak hráče}
    PoPrPi : integer;     {Počet procházejících piškorek}
    ProPi : array[ 0..(PISKORKA*4-1) ]of ^TPisk;
                          {Adresy procházejících piškorek}
    Vaha : array[ THrac ]of integer;   {Důležitost pole}
  end;

  TPisk = record          {Charakteristika piškorky}
    Majitel: integer;     {Nikdo - Počítač - Hráč}
    Proch : array[ 0..(PISKORKA-1) ]of ^TPole;
  end;
  {Kudy prochází}
```

```
const int PISKORKA = 5;           //Počet políček v piškorce
enum eHrac{ POCITAC, HRAC, _eHrac };
```

```

struct TPisk; //Pouze předběžná deklarace
struct TPole //Charakteristika pole šachovnice
{
    char Obsah; //Nic nebo znak hráče
    int PoPrPi; //Počet procházejících piškorek
    TPisk* ProPi[ PISKORKA*4 ]; //Adresy procházejících piškorek
    int Vaha [ _eHrac ]; //Důležitost pole pro hráče
};

struct TPisk //Charakteristika piškorky
{
    int Majitel; //Nikdo - Počítač - Hráč
    TPole* Proch[ PISKORKA ]; //Kudy piškorka prochází
};

```

Další specifika ukazatelů na strukturové typy se týká již pouze C++. V minulé podkapitole jsme si pověděli, že C++ nenabízí žádný ekvivalent pascalského příkazu **with**, a že se místo něj v programech využívá ukazatelů na struktury. Protože je však použití ukazatelů na struktury velmi časté a protože je vzhledem k prioritám operátorů nutno použít ve výrazu závorky, zavedli autoři jazyka pro zjednodušení zápisu a zvýšení přehlednosti programů operátor nepřímé selekce \rightarrow (minus a větší než), který spojuje funkci operátoru de-reference ($*$ – hvězdička) a operátoru selekce ($.$ – tečka). Pomocí tohoto operátoru můžeme nahradit zápis

```
(* UkNaZaznam) .Slozka
```

jednodušším a přehlednějším zápisem

```
UkNaZaznam -> Slozka
```

2.4 Záznamy s variantní částí a unie

Zatím jsme nehovořili o tzv. záznamech s variantní částí (Pascal) a uniích (C++). To jsou datové struktury na první pohled podobné klasickým záznamům (strukturám), liší se však od nich tím, že jejich složky nejsou v paměti seřazeny za sebou, ale zabírají všechny též úsek paměti.

Záznamy s variantní částí a unie slouží ke dvěma účelům: za prvé šetří paměť v případě, kdy daná složka záznamu resp. struktury může obsahovat hodnoty, jejichž typ se může v závislosti na konkrétní situaci lišit, a za druhé se využívají v případě, kdy chceme jednu hodnotu interpretovat několika způsoby.

Podívejme se nyní, jak jsou tyto datové struktury implementovány v probíraných programovacích jazycích.

Pascal: záznamy s variantní částí

Pascal používá pro výše uvedené účely záznamy s variantní částí. To jsou záznamy, jejichž poslední složka může nabývat hodnot několika typů. Syntax definice této poslední, variantní složky (jak jsem již řekl, všechny předchozí složky musí být pevné, tj. takové, jaké jsme dosud používali) je podobná syntaxi přepínače:

record

<Pevná část>

case <rozlišovací typ variantní části> **of**

<specifikace varianty 1>;

<specifikace varianty 2>;

...

<specifikace varianty n>;

end;

Jak vidíte, je definice variantní části uvedena klíčovým slovem **case**, za nímž následuje název některého pořadového typu (tzv. rozlišovací typ variantní části), klíčové slovo **of** a seznam jednotlivých variant. Specifikace každé varianty má tvar

<seznam rozlišovacích konstant> : (<seznam položek>)

Specifikace každé varianty je tedy – stejně jako větve přepínače – nejprve uvedena seznamem hodnot daného pořadového typu následovaným dvojtečkou. Vlastní definice této varianty dané složky (může mít i více složek) pak následuje v závorkách. Podívejme se na příklad:

```
type
  TProstr = ( Lod, Vlak, Auto, Letadlo );
  TPohonu = ( Vrtulnik, Vrtulovy, Tryskovy );

  TZazODP = record
    Dopravce: String[ 20 ];           {Záznam o dopravním prostředku}
    Adresa: String[ 20 ];             {Pevná část}
    case TProstr of
      Lod: ( Vytlak:longint );
      Vlak: ( );
      Auto: ( Nosnost:real;
             ObsahValcu:integer );
      Letadlo: ( Pohon:TPohonu );
    end;
```

Velice často je třeba, aby součástí záznamu byla složka, jejíž hodnota by definovala, která varianta je v danou chvíli aktuální. Pascal vám pro tento účel umožňuje zapsat před identifikátor rozlišovacího typu variantní části dvojtečkou oddělený identifikátor složky rozlišovacího typu. Je to však pouze syntaktický obrat, který má vizuálně spojit tuto složku s následující variantní částí a který překladač již nijak dále nepodporuje. Za správný obsah této proměnné, tj. za to, že její hodnota odpovídá rozlišovací hodnotě aktuální varianty, zodpovídá programátor.

```

(* Příklad P2 - 1 *)
type
  TProstr = ( Lod, Vlak, Auto, Letadlo );
  TPohonu = ( Vrtulnik, Vrtulovy, Tryskovy );

  TZazODP = record                                {Záznam o dopravním prostředku}
    Dopravce: String[ 20 ];
    Adresa: String[ 20 ];
    case Prostredek:TProstr of
      Lod: ( Vytlak:longint );
      Vlak: ( );
      Auto: ( Nosnost:real; ObsahValcu:integer );
      Letadlo: ( Pohon:TPohonu );
    end;
end;

procedure (*****) Zadej (***** )
( var DP:TZazODP );
var
  tp:char;
begin
  write( 'Typ prostředku (A=Auto, D=Lod, ', 'L=Letadlo, V=Vlak):' );
  read( tp );
  with DP do
  begin
    case tp of
      'A','a': begin
        Prostredek := Auto;
        write( 'Nosnost: ' );
        read( Nosnost );
        write( 'Obsah válců: ' );
        read( ObsahValcu );
      end;
      'D','d': begin
        Prostredek := Lod;
        write( 'V tlak: ' );
        read( Vytlak );
      end;
      'L', 'l': begin
        Prostredek := Letadlo;
        write( 'Druh pohonu ( T=Trykový, ', 'V=Vrtulový,
                                     H=Helikoptéra): ' );
        read( tp );
        case tp of
          'T','t': Pohon := Tryskovy;
          'V','v': Pohon := Vrtulovy;
          'H','h': Pohon := Vrtulnik;
        end;
      end;
      'V': begin
        Prostredek := Vlak;
      end;
    end;
  end;
end;
(***** Zadej *****)

```

```

var X: TZazODP;
begin
  zadej(X);
  { ... a další zpracování ... }
end.

```

+ : unie

C++ mechanismus struktur s variantní částí nezavádí a místo něj používá tzv. **unie**⁷, které se deklarují obdobně jako struktury, pouze se místo klíčového slova **struct** napíše klíčové slovo **union**. Překladač pak vyhradí pro objekty tohoto datového typu místo odpovídající rozměru nejdelší složky unie.

Na rozdíl od Pascalu, jehož záznamy mohou mít jak pevnou část, jejíž složky jsou v paměti umístěny za sebou, tak variantní část, jejíž složky sdílejí společný paměťový prostor, C++ struktury a unie přísně rozlišuje. Struktura obsahuje pouze složky v pevně daném pořadí, které jsou v paměti uloženy za sebou, kdežto všechny složky unie sdílejí společný paměťový prostor. Nic však nebrání tomu, aby se struktury staly složkami unii a naopak unie složkami struktur.

Vzhledem k odlišné koncepci pascalských záznamů a céčkovských struktur a unii se bude syntaxe jejich definic lišit více, než jak jsme byli doposud zvyklí. Deklarujeme-li unii jako složku struktury, nemusí být posledním členem – může se v ní nacházet kdekoli a navíc struktury jazyka C++ mohou obsahovat i několik unii⁸. Ekvivalent příkladu uvedeného na konci předchozí pascalské části by programátoři odchovaní jazykem C zapsali v C++ asi následovně:

```

/*   Příklad C2 - 1   */
// #define SEPAR // Makro určující, kterou definici typů použijeme
#include <iostream.h>
#include <ctype.h>

enum TProstr {LOD, VLAK, AUTO, LETADLO, _TProstr };
enum TPohonu {VRTULNIK, VRTULOVY, TRYSKOVY, _TPohonu };

#ifdef SEPAR
/***** Varianta se separátně definovanými typy *****/
struct TChA {
  double Nosnost;
  int ObsahValcu;
};

union TChAP {

```

⁷ V literatuře se také setkáte s překladem „svaz“. Používali jsme jej i v časopisecké verzi tohoto textu.

⁸ Ovšem ani v Pascalu nám nic nebrání, abychom jako pevné složky záznamu použili záznamy s variantní částí; takovýchto složek může být i více a také nemusí být poslední.

```

    long Vytlak;
    TChA ChAuta;
    TPohonu Pohon;
};

struct TZazODP {
    char Dopravce[ 21 ];
    char Adresa[ 61 ];
    TProstr Prostredok;
    TChAP CharProstr;
};

#else
/***** Varianta se souhrně definovanými typy *****/
struct TZazODP {
    char Dopravce[ 21 ];
    char Adresa[ 61 ];
    TProstr Prostredok;
    union TChAP {
        long Vytlak;
        struct TChA {
            double Nosnost;
            int ObsahValcu;
        } ChAuta;
        TPohonu Pohon;
    } CharProstr;
};
#endif

void /***/ Zadej /***/
( TZazODP& DP )
{
    char tp;
    cout << "Typ prostředku (A=Auto, D=Lod, L=Letadlo, V=Vlak): ";
    cin >> tp;
    switch( toupper(tp) )
    {
        case 'A':
            DP.Prostredok = AUTO;
            cout << "Nosnost: ";
            cin >> DP.CharProstr.ChAuta.Nosnost;
            cout << "Obsah válců: ";
            cin >> DP.CharProstr.ChAuta.ObsahValcu;
            break;
        case 'D':
            DP.Prostredok = LOD;
            cout << "Výtlač: ";
            cin >> DP.CharProstr.Vytlak;
            break;
        case 'L':
            DP.Prostredok = LETADLO;
            cout << "Druh pohonu (T=Trykový, V=Vrtulový,
H=Helkoptera):";
            cin >> tp;
            tp = toupper(tp);
            DP.CharProstr.Pohon = ( tp=='T' ? TRYSKOVA :

```

```

                                tp=='V' ? VRTULOVOY : VRTULNIK );
        break;
        case 'V':
            DP.Prostredek = VLAK;
    } //switch
}
/***** Zadej *****/
TZazODP X;
int main(){
    Zadej(X);
    return 0;
}

```

Zde jsme si ukázali dvě možnosti deklarace pomocných typů *TChap* a *TCha*: jednak jsme tyto typy definovali samostatně a pak je použili v definici typu *TZazODP* a za druhé jsme je definovali přímo v deklaraci struktury *TZazODP*. První možnost je zpravidla přehlednější, funkční jsou ale obě, jak se snadno přesvědčíme, jestliže tento program přeložíme s makrem *SEPAR* nebo bez něj.

Anonymní unie

S uniemi se setkáme již v jazyku C. Jazyk C++ však navíc zavádí ještě tzv. **anonymní unie**, což jsou unie, které nejsou ani pojmenované ani neslouží k deklaraci žádného konkrétního objektu. Složky takovýchto unií pak vystupují jako samostatné proměnné, které v paměti sdílejí společné místo. Poznamenejme, že globální anonymní unie musí být statické. Podívejme se na příklad:

```

union Jmeno { //Pojmenovaná unie => definuje datový typ Jmeno, který
    int a; //je možno použít v následujících deklaracích;
    long b;
}; //Nedefinujeme zde žádnou proměnnou (mohli bychom)
Jmeno X; //Definice proměnné X typu Jmeno;
union { //Nepojmenovaná unie => nelze ji použít v žádné
    int a; //následující deklaraci. Všechny proměnné tohoto typu
    long b; //se musí definovat
} x, y[3], *z; //<--ZDE
static union { //Unie není pojmenována ani neslouží k deklaraci
    int a; //proměnných => jedná se o
    long b; //ANONYMNÍ UNII, pomocí níž zavádíme
}; //proměnné "a" a "b", které mají stejnou adresu.
//ANONYMNÍ UNIE MUSÍ BÝT DEKLAROVÁNY JAKO LOKÁLNÍ NEBO STATICKÉ!

```

Anonymní unie se používají hlavně v situacích, kdy chceme tentýž obsah paměti interpretovat několika způsoby. Nejčastějším případem asi je práce s jednotlivými bajty slova nebo s jednotlivými slovy dvouslova. Chcete-li toho někdy využít, musíte mít na paměti, že **jednotlivé bajty vícebajtových celočíselných hodnot jsou v paměti uloženy po-zpátku**. Tato specialita mikroprocesorů firmy Intel (mikroprocesory jiných typů to tak mít

nemusí) má pro nás tu výhodu, že nejnižší bajt dvou bajtového čísla má stejnou adresu jako celé číslo (totéž platí pro slova a čísla typulong).

Chcete-li se o tom přesvědčit, zkuste si spustit následující krátký prográmk (nebo vytvořit obdobný), v němž si zároveň ukážeme na rozdíl ve způsobu práce s běžnými a anonymními uniemi.

```

/* Příklad C2 - 2 */
#include <iostream.h>

void /*****/ main /*****/ ()
{
    typedef unsigned char byte;
    typedef unsigned int word;
    typedef unsigned long dword;

    union{          //Anonymní unie, která "položí přes sebe" 3 proměnné
        dword dd;          //Dvojslovo
        word ww[2];        //Dvě slova
        byte bb[4];        //Čtyři bajty
    };

    //Jednotlivé složky anonymní unie používáme bez jakékoli kvalifikace
    dd = 0x12345678L;

    cout << hex << "\n\nDvojslovo: d=" << dd
          << "\nDvě slova: w1=" << *ww << " w2=" << ww[1]
          << "\nČtyři bajty:";
    for( int i=0; i<4; i++ )
        cout << " b[" << i << "]=" << int( bb[i] );
    cout << endl;

    union {
        struct{ byte l,h; } b;    //l = low = nižší (zde bajt)
        word w;
    };

    union WORD {
        struct HL {byte l,h; }b;
        word w;
    } W1, W2;

    w = 0x1234;          //b.h = 0x12 b.l = 0x34
    b.l = 0x78;          //w = 0x1278
    b.h = 0x56;          //w = 0x5678
    W1.w = w++;          //w = 0x5679 W1.w = 0x5678
    W2.b.h = b.l;        //W2.b.h = 0x79 W2.w = 0x79??
    W2.b.l = W1.b.h;     //W2.b.l = 0x56 W2.w = 0x7856

    cout << hex;
    cout << "\nAnonym: w=" << w << " b.h=" << int( b.h )
          << " b.l=" << int( b.l );

    cout << "\nW1: W1.w=" << w << " W1.b.h=" << int( W1.b.h )
          << " W1.b.l=" << int( W1.b.l );
    cout << "\nW2: W2.w=" << w << " W2.b.h=" << int( W2.b.h )
          << " W2.b.l=" << int( W2.b.l );
    cout << endl;
}

```

}

V tomto prográmku jsme pomocí anonymní unie „rozebrali“ dvouslovo *dd* (vlastně proměnnou typu **unsigned long**) jednak na dvě samostatná slova (pole *ww* dvou proměnných typu **unsigned**) a za druhé na jednotlivé bajty (pole *bb* čtyř znaků). Potom jsme podobně „rozebrali“ proměnnou *w* typu **unsigned** na dva samostatné bajty tím, že jsme přes ni „položili“ strukturu, složenou ze dvou bajtů. Nakonec jsme udělali totéž ještě pomocí pojmenované unie.

Anonymní unie mohou být i součástí struktur. S využitím anonymních unií bychom mohli příklad C2 – 01 přepsat do tvaru:

```

/*   Příklad C2 - 3   */
#include <iostream.h>
#include <ctype.h>

//Pomocné výčtové typy
enum TProstr {LOD, VLAK, AUTO, LETADLO, _TProstr };
enum TPohonu {VRTULNIK, VRTULOVY, TRYSKOVY, _TPohonu };

//Záznam o dopravním prostředku
struct TZazODP {
    char Dopravce[ 21 ];
    char Adresa[ 61 ];
    TProstr Prostredok;
    union { //Anonymní unie, která "překládá přes sebe" různé typy údajů
        long Vytlak; //o různých typech dopravních prostředků
        struct /*TChA*/ {
            double Nosnost;
            int ObsahValcu;
        } ChAuta;
        TPohonu Pohon;
    };
};

void /******/ Zadej /******/
( TZazODP& DP ) //Zadávání dat do záznamu
{
    char tp;
    cout << "Typ prostředku (A=Auto, D=Lod, "
        "L=Letadlo, V=Vlak): ";
    cin >> tp;
    switch( toupper(tp) )
    {
    case 'A':
        DP.Prostredok = AUTO;
        cout << "Nosnost: ";
        cin >> DP.ChAuta.Nosnost;
        cout << "Obsah válců: ";
        cin >> DP.ChAuta.ObsahValcu;
        break;
    case 'D':
        DP.Prostredok = LOD;
        cout << "Výtlak: ";
        cin >> DP.Vytlak;
    }
}

```

```

        break;
    case 'L':
        DP.Prostredok = LETADLO;
        cout << "Druh pohonu (T=Trykový, V=Vrtulový, H=Helikoptéra): ";
        cin >> tp;
        tp = toupper(tp);
        DP.Pohon = ( tp=='T' ? TRYSKOVY :
                    tp=='V' ? VRTULOVY : VRTULNIK );
        break;
    case 'V':
        DP.Prostredok = VLAK;
    } //switch
}
/***** Zadej *****/
TZazODP X; //Vyzkoušíme si to na jedné proměnné
int main(){
    Zadej(X);
    return 0;
}

```

2.5 Inicializace strukturovaných objektů

Strukturované objekty, tj. pole, záznamy, struktury a unie, již sice umíme definovat a používat, avšak doposud jsme si neřekli, zda je jim možno přiřadit počáteční hodnotu, a v případě, že ano, jak to udělat.

Pascal nám neumožňuje definovat konstanty strukturovaných typů, a zakrývá to tím, že po vás chce, abychom inicializované proměnné definovali v sekci konstant. O tom jsme již hovořili.

Inicializované vektory definujeme tak, že za rovnítko následující po specifikaci příslušného datového typu uvedeme v kulatých závorkách seznam inicializátorů – čárkami oddělených hodnot jednotlivých prvků definovaného vektoru. Pokud jsou prvky tohoto vektoru opět vektory, zadáváme je stejným způsobem, tedy opět jako seznam inicializátorů uzavřený v kulatých závorkách. Jsou-li prvky inicializovaného vektoru záznamy, zapisujeme je podle pravidel uvedených dále.

Inicializované záznamy definujeme podobně jako vektory, tzn. za rovnítko následující po specifikaci příslušného datového typu uvedeme v kulatých závorkách seznam hodnot jednotlivých složek záznamu, které tentokrát oddělíme středníky. Na rozdíl od vektoru však před hodnotu každé složky napíšeme její identifikátor následovaný dvojtečkou. Je-li některá ze složek záznamu opět strukturovaná, zapisujeme její hodnotu znovu jako v kulatých závorkách uzavřený seznam čárkami či středníky oddělených hodnot zapisovaných podle pravidel uvedených v tomto a předchozím odstavci.

Při definici počátečních hodnot záznamů musíme dodržet pořadí jednotlivých složek, i když musíme v rámci definice inicializační hodnoty uvádět i jejich jména. Nemusíme však

uvádět inicializační hodnoty pro všechny složky, ale můžeme přestat kdykoliv uprostřed struktury (viz následující příklad).

```
(*   Příklad P2 - 2   *)
Program Pascal;
type
  vv = array [1..2] of array [0..1] of integer;
  tvc = record
    a: real;
    z: real;
    case x:integer of
      1: (b:byte);
      2: (i:integer);
      4: (l:longint);
    end;
const
  v:vv = ( (10,20), (100,200) );
  x:tvc = ( a:-123.45678; z:9.87654321; x:1; l:$12345678 );
  x3: array[ -1..3] of tvc =
    ( ( ),
      ( a:0.12345 ),
      ( a:1.23456; z:9.87654321; x:1 ),
      ( a:23.4567; z:9.87654321; x:2; i:-1 ),
      ( a:345.678; z:9.87654321; x:-1024; l:$12345678 ) );
```

Zkuste si hodnoty těchto proměnných zobrazit ve sledovacím okně pomocí příkazů

```
v
v[1,0],4
x,r
x3,rf2x
```

Pozor! V manuálu autoři tvrdí, že řídicí proměnná musí nabývat pouze jedné z hodnot určujících variantu a že inicializovaná varianta musí odpovídat variantě dané rozlišovací proměnnou. Jak jste si však mohli v příkladu všimnout, překladač nám dovolí zapsat při inicializaci záznamu s variantní částí jak inicializaci samotné rozlišovací složky, tak inicializaci varianty, která neodpovídá před tím zadané hodnotě rozlišovací složky. Musíte si tedy ohlídat sami, aby vše bylo správně – na to jste si však, předpokládáme, při práci s Pascallem již zvykli.

V C++ se inicializační hodnoty vektorových proměnných a konstant zadávají jako posloupnost čárkami oddělených hodnot jednotlivých prvků uzavřená ve složených závorkách. Pokud jsou prvky tohoto vektoru opět vektory, budou hodnoty těchto prvků zadávány stejně, tj. jako posloupnost čárkami oddělených hodnot jednotlivých prvků uzavřená ve složených závorkách.

Hodnoty struktur se zapisují také jako posloupnosti čárkami oddělených hodnot jednotlivých složek uzavřené ve složených závorkách. Jsou-li složkami struktur opět vektory či struktury, aplikují se na jejich hodnoty opět výše uvedená pravidla. Na rozdíl od Pascalu se však v C++ neuvádějí identifikátory jednotlivých složek struktury – nejvýše jako komentáře.

V C++ můžeme, stejně jako v Pascalu, zadat pouze hodnoty několika prvních složek struktury. Na rozdíl od Pascalu vám však překladač nedovolí ponechat závorku prázdnou – musíte do ní zapsat alespoň symbolickou nulu (ta je kompatibilní se všemi typy). Na druhou stranu vám však C++ dovolí uvést pouze několik prvních prvků i u polí.

Na rozdíl od pascalských variantních záznamů nemůžete v C++ inicializovat libovolnou složku unie, ale vždy pouze první deklarovanou. Při definici unie si proto musíte pečlivě rozmyslet, kterou složku uvedete v definici jako první. Na pořadí ostatních již prakticky nezáleží.

Všechny uvedené vlastnosti jsme se pokusili shrnout v následujícím příkladku. Doplňte jej o prázdnou funkci `main()` a zkuste jej krokovat.

```
/* Příklad C2 - 4 */
typedef int vv[3][3];

struct tvc {
    double a;
    double z;
    int x;
    union {
        long l;
        int i;
        char b;
    };
};

vv v = {{10,20}, {100,200, 300} };
tvc x = {-123.45678, 9.87654321, 1, 0x12345678L };

tvc x3[5] =
    {{0 },
     {0.12345 },
     {1.23456, 9.87654321, 1 },
     {23.4567, 9.87654321, 2, -1 },
     {345.678, 9.87654321, 1, 0x12345678L }};
```

Při krokování napište do sledovacího okna příkazy

```
v
v[0][0],9
x,r
x3,rf2x
```

Zkuste si pak najet ve sledovacím okně na kteroukoliv položku a vyvolejte pro ni stiskem ALT-F4 inspekční okno. (Podrobnější informace o tomto ladicím nástroji najdete v kapitole 5.)

2.6 Bitová pole

Složkami struktur v C++ mohou být i tzv. bitová pole. Umožňují pracovat s jednotlivými bity nebo jejich skupinami.

Deklarace bitového pole má tvar

typ identifikátor_{opt} : velikost

Typ je typ bitového pole; může to být jakýkoli celočíselný typ (jazyk C dovoluje pouze typy **int** a **unsigned**). Identifikátor je jméno bitového pole a index *opt* naznačuje, že jej můžeme vynechat. *Velikost* udává počet bitů, které má dané bitové pole zabírat. Podívejme se na příklad deklarace struktury, obsahující bitové pole:

```
struct Bp {
    int i: 3;
    unsigned j: 3;
    int k: 1;
    unsigned l: 1;
};
```

Bitová pole se chovají jako celočíselné proměnné s malým rozsahem. Například složka *i*, která je typu **int** a má velikost 3 bity, se chová jako proměnná, která může nabývat hodnot v rozmezí od -4 do 3, a složka *j*, která má stejnou velikost, ale je typu **unsigned**, může nabývat hodnot v rozmezí 0 až 7. Složka *k* typu **int** má velikost 1 bit a může nabývat pouze hodnot 0 a -1, složka *l* typu **unsigned** může nabývat pouze hodnot 0 a 1.

S bitovými poli zacházíme téměř stejně jako s ostatními složkami struktur. Můžeme je inicializovat např. příkazem

```
Bp b = {1, 1, 1, 1};
```

který ale způsobí, že složka *b.k* bude obsahovat hodnotu -1.

Jediné omezení se týká adres bitových polí: bitová pole nemají adresu. Nelze tedy na ně použít operátor **&** a nelze je předávat jako parametry volané odkazem.

Vynecháme-li identifikátor bitového pole, dostaneme bezejmenné pole, které slouží pouze jako výplň.

Bitová pole se často používají k ukládání bitových příznaků nebo k implementaci množin.

Množiny

Jazyk C++ nenabízí samostatný datový typ pro práci s množinami. (Najdeme jej ve standardní knihovně jako objektový typ, ale to je jiná pohádka.) Můžeme je však snadno nahradit strukturami s bitovými poli. Představme si např., že chceme naprogramovat nějakou karetní hru. Pro jednoduchost ponecháme stranou barvy a dáfužeme strukturu

```
struct Karty {
    dny sedm: 1;
    unsigned osm: 1;
```

```

unsigned devet: 1;
unsigned deset: 1;
unsigned spodek: 1;
unsigned svrsek: 1;
unsigned kral: 1;
unsigned eso: 1;
};

```

Určitý balíček karet pak můžeme popsat proměnnou

```
Karty Balicek = {1, 1, 1, 0, 1, 0, 0, 1};
```

Jednotlivá bitová pole zde fungují jako příznaky, které říkají, zda se určitá karta v balíčku nachází. Například náš balíček karet obsahuje na počátku sedmu, osmu, devítku, spodka a eso. Jestliže eso vyneseme a v balíčku už nebude, zaznamenáme to příkazem

```
Balicek.eso = 0;
```

2.7 Množiny v Pascalu

Pascal nezná bitová pole, nabízí ale na druhé straně množinové typy. Definice množinového typu má tvar

set of *typ*

kde *typ* je typ složek – prvků – množiny (tzv. bazový typ). Musí to být ordinální typ, který má nejvýše 256 možných hodnot. Můžeme tedy deklarovat množiny

```

type
  karty = (sedm, osm, devět, deset, spodek, svrsek, kral, eso);
  balicek = set of karty;           {OK - typ karty má 8 hodnot}
  boolset = set of boolean         {OK - typ boolean má 2 hodnoty}
  set of char;                     {OK - typ char má 256 hodnot}

```

Na druhé straně tyto deklarace jsou nesprávné:

```

type
  cela = set of integer           {NELZE - integer má více než 256 hodnot}
  realna = set of real            {NELZE - real není ordinální typ}

```

Hodnoty typu množina

Vezmeme *typ balicek* a definujeme si proměnnou:

```

var
  rozdani: balicek;

```

Tato proměnná představuje např. balíček karet, který máme po rozdání v ruce. Jak mu ale přiřadíme hodnotu? Použijeme tzv. konstruktor množiny. To je seznam hodnot, které chceme do množiny vložit. Například

```
rozdani := [sedm, osm, deset, eso];
```

konstruktor množiny může obsahovat samozřejmě i proměnné bázevého typu nebo specifikaci intervalu:

```
rozdani := [sedm, osm, deset .. eso];
```

Prázdný balíček karet vytvoříme pomocí konstruktoru prázdné množiny:

```
rozdani := [];
```

Operace s množinami

Pro množiny jsou definovány tyto operátory:

- + sjednocení množin ($A + B$ je množina, obsahující prvky, které jsou alespoň v jedné z množin A a B),
- * průnik množin ($A * B$ je množina, obsahující jen ty prvky, které jsou zároveň v obou množinách A a B),
- rozdíl množin ($A - B$ je množina, obsahující prvky, které jsou v A , ale nejsou v B),
- in** test, zda je prvek v množině: x **in** B je **true**, jestliže množina B obsahuje prvek x .

Podívejme se na příklad. Napíšeme program, který přečte ze vstupu řetězec a vypíše velká písmena, která se v něm nevyskytovala.

Nejprve si deklaruujeme typ *znaky*, což bude množina znaků. Pak si vytvoříme několik proměnných, které budou obsahovat malá a velká písmena s diakritickými znaménky a bez nich apod. (Abychom nemuseli opisovat výčet hodnot, inicializujeme některé z nich až v příkazové části bloku programu, neboť Pascal neumožňuje inicializovat jednu typovou konstantu hodnotou jiné typové konstanty.)

Pak přečteme řetězec znaků *cteno*, který chceme analyzovat, zjistíme si pomocí funkce *length* jeho délku a jednotlivé znaky uložíme do množiny znaků *sledovana* příkazem

```
for i := 1 to length(cteno) do sledovana := sledovana + [cteno[i]];0
```

Zde jsme zkonstruovali jednoprvkovou množinu, obsahující *cteno[i]* (*i*-tý znak řetězce *cteno*) a tu jsme sjednotili s množinou *sledovana*. Pak zjistíme, která velká písmena se v textu nevyskytovala: Od množiny všech písmen odečteme množinu malých písmen (neboť ta nás nezajímají) a písmen, která se v textu vyskytla (tj. množinu *sledovana*). Nakonec oznámíme uživateli výsledek, a to tak, že v cyklu probereme všechny znaky, u každého si zjistíme, zda je v množině *sledovana*, a pokud ano vypíšeme jej. Program může vypadat např. takto:

```
(*   Příklad P2 - 3   *)
uses crt;
type
  znaky = set of char;      {Množina znaků}
const
```

```

mala_s_diakritikou: znaky = ['á', 'ä', 'č', 'd', 'é', 'ě', 'í',
                             'í', 'l', 'ň', 'ó', 'ô', 'ř', 'ř', 'š', 't',
                             'ú', 'ů', 'ž'];
velka_s_diakritikou: znaky = [ 'Á', 'Ä', 'Č', 'Ď', 'É', 'Ě', 'Í',
                              'Ĺ', 'Ľ', 'Ň', 'Ó', 'Ô', 'Ř', 'Ř', 'Š', 'Ť',
                              'Ú', 'Ů', 'Ž'];
mala_bez_diakritiky: znaky = ['a'..'z'];
velka_bez_diakritiky: znaky = ['A'..'Z'];

var
  folklor, pismena, pismena_bez_diakritiky: znaky;
  mala, sledovana: znaky;
  cteno: string;                {Přečtený řetězec}
  i: integer;                  {Parametr cyklu}
  c: char;                     {Parametr cyklu}

begin
  clrscr;                       {Smažeme obrazovku}
  {Nejprve si vytvoříme pomocné množiny znaků}
  folklor := mala_s_diakritikou + velka_s_diakritikou;
  pismena_bez_diakritiky := mala_bez_diakritiky +
  velka_bez_diakritiky;
  mala := mala_bez_diakritiky + mala_s_diakritiky;
  pismena := folklor + pismena_bez_diakritiky;
  {Pak si přečteme text, který budeme zkoumat}
  writeln('Zadej text pro rozbor: ');
  read(cteno);
  {Jednotlivá písmena z něj zařadíme do množiny sledovana}
  for i := 1 to length(cteno) do sledovana := sledovana + [cteno[i]];
  writeln(' V přečteném textu se nevyskytovala tato velka pismena:
  ');
  {Zjistíme, která velká písmena se v textu nevyskytovala}
  sledovana := pismena - mala - sledovana;
  {a vypíšeme je}
  for c := #0 to #255 do
    if c in sledovana then write(c);
end.

```

3. Práce se soubory a proudy

O vstupech a výstupech jsme již několikrát hovořili a samozřejmě jsme je také hojně používali. Teď ale přišel čas, abychom své vědomosti utřídili a doplnili.

3.1 Obecně o souborech a proudech

ČSN 36 9001/8-1987 definuje **soubor** (*file*) jako „posloupnost vět nebo záznamů, které lze zpracovávat jako celek“. Některé učebnice programování dokonce hovoří o souborech jako o jedné z možných realizací datového typu „posloupnost“.

Jak jste si mohli v definici přečíst, jednotlivé členy posloupnosti dat tvořících soubor se nazývají **věty** nebo **záznamy** (*record*). Termín „záznam“ je sice doslovným překladem anglického termínu „record“, koliduje však s pascalským názvem jednoho druhu strukturovaných datových typů. Abychom nemuseli vždy vysvětlovat, kdy hovoříme o záznamu jako o datové struktuře a kdy hovoříme o záznamu jako o členu posloupnosti dat v souboru, budeme v dalším textu používat výhradně první z uvedených termínů, **tjěta**.

Všimněte si, že uvedená definice neváže (na rozdíl od definic dřívějších) soubor na žádný nosič dat. Operační systém MS-DOS a systémy s ním kompatibilní definují klasické periférie, jako konzolu (klávesnice + obrazovka), tiskárnu, sériové rozhraní apod., jako tzv. zařízení (*device*), s nimiž lze pracovat jako se soubory. Uživatelé pak umožňují instalaci ovladačů (*device driver*) vlastních zařízení, které mu umožní přistupovat stejným způsobem i k těmto dodatečným zařízením – např. mřícím přístrojům.

Pascal se s těmito možnostmi spokojuje. C++ však jde dále. Knihovní funkce C++ pojem souboru sice znají, avšak kromě toho zavádějí novou datovou strukturu – tzv. **datový proud** (*stream*), které chápou jako posloupnost bajtů směřujících od producenta (zdroje) ke konzumentovi (cíli, spotřebiči). Přitom producentem resp. konzumentem může být prakticky cokoli: klasický soubor, externí zařízení, oblast paměti, náš vlastní, ale i paralelně běžící program atd.

Soubory i datové proudy mají mnoho společných vlastností. Pokud tedy budeme v následujícím textu hovořit o souborech, budou (pokud výslovně neuvědeme opak) vykládané skutečnosti platit i pro datové proudy, přičemž v datových proudech budou větami jednotlivé bajty.

Nyní si zavedeme dva pojmy: **fyzický soubor (proud)** bude pro nás ona výše zmíněná posloupnost vět (bajtů) někde se nacházející či odněkud přicházející a **logický soubor (proud)** pak bude proměnná, která nám v programu daný soubor (proud) reprezentuje a jejímž prostřednictvím se k němu v programu obracíme. Pokud bude z kontextu jasné, o jakém typu souboru (proudu) v daném okamžiku hovoříme, budeme používat pouze termín soubor (proud).

Dříve, než se pustíme do vlastního výkladu práce se soubory, musíme si ozřejmit dvě důležité věci (Pozor! Platí pouze pro soubory a nemusí platit pro proudy!):

- ✧ Fyzický soubor existuje **nezávisle** na programu, který jej vytvořil anebo který jej používá. Mohli bychom o něm říci, že není „majetkem programu“, ale že je „majetkem operačního systému“. Tím se stává velice výhodným prostředkem, s jehož pomocí si mohou programy předávat důležité informace.
- ✧ Soubor má v rámci operačního systému své **jméno**. Toto jméno nemá žádný přímý vztah ke jménu logického souboru, tedy k proměnné, prostřednictvím níž program s daným fyzickým souborem komunikuje. Mechanismus spojení logického souboru s odpovídajícím fyzickým souborem je určen v rámci operačního systému a použitého překladače.

Předchozí dvě tvrzení platí pro datové proudy pouze v případě, že producent či konzument jsou mimo program (druhým z dvojice je vždy náš program) – budeme jim říkat **externí proudy**. Pro datové proudy však může nastat situace, kdy jak zdroj tak příjemce jsou součástí programu (např. čteme-li data z nějaké oblasti paměti nebo předávají-li si je dva paralelně běžící procesy). Takovéto proudy budeme nazývat **interní proudy**, a pro ně samozřejmě předchozí tvrzení neplatí.

Jak jsme již řekli, nezávislost souborů na programech umožňuje využívat souborů vázaných na nějaké nosiče jako prostředků pro předávání dat mezi jednotlivými programy. Typickým příkladem systému programů, které si předávají data prostřednictvím souborů, jsou programy používané při vývoji nových programů. Podívejme se, jak vypadá spolupráce těchto programů v případě, že nepoužíváte integrovaný systém, ale samostatné programy:

1. Po analýze celého problému usedneme k editoru (předpokládáme, že nepoužíváte žádný systém CASE) a s jeho pomocí napíšeme zdrojový text modulu, který uložíme jako soubor (případně skupinu souborů) na disk.
2. Spustíme překladač, kterému oznámíme jména souborů, v nichž se nacházejí zdrojové texty určené k překladu. Překladač tyto zdrojové texty zpracuje a vyrobí tři výstupní soubory: zprávy o chybách (ty směřují do standardního výstupního souboru), protokol o překladu nazývaný „listing“ a výsledný přeložený **relativní modul**.
3. Spustíme sestavovací program, kterému předáme názvy souborů s jednotlivými relativními moduly, z nichž sestává program, spolu s názvy souborů, v nichž se nachází použité knihovny. Sestavovací program vyrobí tři soubory: zprávy o chybách, mapový soubor popisující rozložení jednotlivých objektů (podprogramů, globálních proměnných a konstant) ve výsledném programu a samozřejmě výsledný program.
4. Program musíme odladit. Na to se používají specializované ladící programy – debugery. Spustíme tedy debugger a oznámíme mu jméno souboru, v němž je uložen program, který se chystáme ladit. Pokud při ladění najdeme chybu, vrátíme se k bodu 1 či 2.
5. Odladěný program bývá zvykem „prohnat“ ještě profilátorem (*profiler*), který odhalí, jakou dobu strávil počítač vykonáváním té které části programu. Profilátoru předáme

jméno souboru s testovaným programem a on nám dokáže zjistit, které části programu jsou napsány neúnosně neefektivně.

6. Výsledný soubor nahrajeme na disketu a s doprovodnou dokumentací jej předáme zákazníkovi.

Z programátorského hlediska budeme logické soubory a proudy rozdělovat podle čtyř hledisek:

1. Podle směru přenosu informací:
 - a) vstupní,
 - b) výstupní,
 - c) vstupně-výstupní.
2. Podle způsobu přístupu k jednotlivým položkám:
 - a) sekvenční,
 - b) semisekvenční,
 - c) s přímým přístupem,
 - d) indexsekvenční.
3. Podle využívání vyrovnávací paměti:
 - a) využívající vyrovnávací paměti (*buffered*),
 - b) nevyužívající vyrovnávací paměti (*unbuffered*).
4. Podle typu dat:
 - a) textové,
 - b) binární.

Rozdělení souborů podle směru přenosu informací je vám jistě jasné: ze vstupních souborů můžeme pouze číst, do výstupních souborů můžeme pouze zapisovat a ze vstupně-výstupních souborů můžeme jak číst, tak do nich zapisovat.

Ne tak jasné bude rozdělení podle přístupu k jednotlivým položkám. V zásadě však nejde o nic složitějšího.

Nejjednodušší organizaci mají **sekvenční soubory**, u nichž nemáme jinou možnost, než zapisovat data do souboru pěkně jedno po druhém nebo je stejným způsobem ze souboru číst. Důsledkem tohoto omezení je i to, že **sekvenční soubory mohou být pouze vstupní nebo pouze výstupní**

Typickým příkladem vstupního sekvenčního souboru je soubor dat přicházejících z klávesnice, typickým příkladem výstupního sekvenčního souboru jsou data odcházející na tiskárnu. Typickým médiem (i když dnes již málo používaným), které prakticky nedovoluje ukládání jiných než sekvenčních souborů, je magnetická páska.

Zavedení sekvenčních souborů je v řadě úloh nejjednodušším a nejefektivnějším řešením a většina programovacích jazyků nám pro tuto práci poskytuje dostatek prostředků. Některé jazyky (např. standardní Pascal) dokonce ani práci s jinými typy souborů neumožňují.

Soubory s přímým přístupem umožňují přímou adresaci jednotlivých vět v souboru (tj. jednotlivých členů posloupnosti), a to prostřednictvím nějaké části věty, kterou nazýváme klíč. Práce se soubory s přímým přístupem se proto v mnohém podobá práci s vektory. Nevýhodou souborů s přímým přístupem je naopak to, že je nemůžeme dost dobře číst sekvenčně, protože mohou obsahovat „díry“ s nedefinovaným obsahem. Typickým médiem pro práci se soubory s přímým přístupem je disk.

Oproti sekvenčním souborům jsou soubory s přímým přístupem univerzálnější (sekvenční přístup lze u nich programově realizovat), ale díky své obecnosti také poněkud komplikovanější. Musíme u nich vyřešit např. způsob adresace, zařídit, aby mezi platnými větami (tj. těmi, které jsme tam skutečně zapsali) nebylo vzhledem k nešikovné adresaci příliš mnoho místa, při případném sekvenčním čtení musíme umět tyto díry dbalit atd.

Indexsekvenční soubory jsou spojením předchozích dvou; na počátku je většinou vytváříme sekvenčně, ale zvláštním vnitřním uspořádáním nebo přidavnými údaji v pomocném (indexovém) souboru dosáhneme toho, že k nim můžeme přistupovat jak sekvenčně, tak přímo, přičemž při sekvenčním přístupu ještě můžeme rozlišovat přístup v pořadí daném fyzickým umístěním dat v souboru a přístup v pořadí daném pořadím klíčů (mohli bychom říci v pořadí daném logickým umístěním v souboru).

Na první pohled by se možná zdálo, že by bylo nejlepší používat pouze indexsekvenční soubory. Bohužel tomu tak není. Za prvé si některá média a zařízení přímo diktují, že soubory na nich umístěné musí být sekvenční, a za druhé indexsekvenční soubory mají vzhledem ke své obecnosti daleko větší režii: zaberou více místa v paměti a přístup k nim je pomalejší. Kromě toho, operační systém ani žádný z probíraných jazyků indexsekvenční soubory přímo nepodporuje. S těmito soubory se proto na osobních počítačích setkáte většinou pouze v databázových aplikacích, přičemž ony přidavné údaje jsou uloženy ve zvláštních, tzv. indexových souborech.

Jinou kombinací vlastností sekvenčních souborů a souborů s přímým přístupem jsou soubory, které jsme v našem výčtu označili jako **semisekvenční**. Tyto soubory se chovají jako obyčejné sekvenční soubory, avšak na rozdíl od nich nám umožňují kdykoliv **změnit** svoji **pozici** v rámci souboru a začít číst či zapisovat na libovolně zvoleném místě souboru.

Semisekvenční soubory jsou základním typem souborů podporovaných operačním systémem, a proto jsou také typem souborů, na něž jsou „naladěny“ souborové funkce v obou probíraných programovacích jazycích. Protože sekvenční a semisekvenční soubory jsou v běžných programech daleko nejpoužívanější, bude zbytek této kapitoly věnován pouze jim.

Třetím hlediskem, podle něž jsme v našem úvodním přehledu soubory třídili, bylo, zda dotyčný logický soubor používá či nepoužívá vyrovnávací paměť (*buffer*). Podle to-

hoto hlediska se v programátorské hantýrce dělí logické soubory na „bafrované“ a „nebafrované“. Ať se na nás češtináři nezlobí, ale termín „soubor využívající, resp. nevyávající vyrovnávací paměť“ se nám příliš nelíbí, a tak do doby, než objevíme nějaký hezčí termín, budeme hovořit o bafrovaných a nebafrovaných soubrech.

Vyrovnávací paměť (buffer) je klíčem k pochopení mnoha specifických rysů práce se soubory. U vstupních souborů je to oblast paměti, v níž se nacházejí data od doby, kdy byla fyzicky načtena ze souboru, do doby, kdy si je program převezme. U výstupních souborů se zde vystupující data nacházejí od doby, kdy je program poslal do souboru, do doby, kdy budou do souboru opravdu fyzicky zapsána.

Velikost vyrovnávací paměti většinou nemůžeme volit libovolně. Přenos údajů mezi vyrovnávací paměti a nosičem dat totiž dost často probíhá po blocích a velikost vyrovnávací paměti pak musí být celistvým násobkem velikosti přenášených bloků. To je dobré vědět, i když na to při programování většinou nemusíme myslet. Velikost vyrovnávacích pamětí bývá totiž již předem definována a my ji ve svých programech nebudeme ovlivňovat.

Posledním z uvedených kritérií klasifikace souborů byl typ dat v souboru. Podle tohoto kritéria rozdělujeme soubory na textové a binární. Binární soubory jsou svoji podstatou jednodušší: co do nich pošlete, to tam také najdete.

Poněkud komplikovanější situace je u souborů textových. Textový soubor je vlastně soubor znaků, rozdělený nějakým způsobem na řádky. Prvním problémem je, že délka souboru udávaná operačním systémem nemusí souhlasit s délkou souboru, kterou zjistíte při jeho zpracování. Operační systém totiž na jedné straně považuje za délku souboru celkový počet bajtů, avšak na druhé straně pokud při čtení tohoto souboru narazí na znak s kódem 26 = 1Ah (Ctrl-Z), považuje ho za označení konce souboru.

Druhým problémem je zpracovávání konců řádků. V tomto bodě totiž existuje jistá nejednotnost. Operační systém CP/M, na který první verze systému MS-DOS navazovaly, označoval vzhledem k dálkopisné orientaci svých prvních verzí konec řádku dvojicí znaků *CR* (*carriage return* – návrat vozíku, kód 13 = 0Ch) a *LF* (*line feed* – posun řádku, odřádkování, kód 10 = 0Ah). Naproti tomu operační systém UNIX, k němuž se současné verze DOSu stále více přibližují, označuje konce řádků samotným *LF* (znak '\n' v C++). A aby byl výčet možností úplný, Pascal akceptuje ukončení řádku i samotným znakem *CR*. V jiných operačních systémech mohou být konce řádků vyznačovány ještě jinak.

K problému ukončování řádků se ještě vrátíme v pasážích věnovaných specifikám obou jazyků.

Jak jsme již řekli, standardní prostředky obou jazyků předpokládají práci se sekvenčními a semisekvenčními soubory. Při ní budeme používat následující operace:

Otevření souboru

Připraví vše potřebné pro to, abychom s daným souborem mohli začít pracovat. Součástí otevření souboru je vzájemné propojení fyzického a logického souboru (tj. vlastní posloupnosti zpracovávaných dat a proměnné, prostřednictvím níž se budeme na danou po-

sloupnost odvolávat), definice režimu práce se souborem (podrobněji viz výklad konkrétních funkcí obou jazyků), inicializace vyrovnávacích pamětí a další potřebné inicializační činnosti.

Zavření souboru

U bafrovaných výstupních souborů se přenesou obsah vyrovnávací paměti do sdruženého fyzického souboru a nastaví se hodnoty všech systémových údajů o souboru (např. známů v adresáři) tak, aby odrážely jeho nový stav. Pokud ukončíme program, aniž bychom uzavřeli některý výstupní soubor, je pravděpodobné, že operační systém o novém stavu tohoto souboru nebude vědět. Neuzavření vstupních souborů končí většinou bez následků.

Spláchnutí souboru

Spláchnutí souboru je operace specifická pro bafrované soubory. Jejím výsledkem je u vstupních souborů vyčištění vyrovnávací paměti (příštímu čtení bude předcházet nové naplnění vyrovnávací paměti), u výstupních souborů pak vyčištění spojené s přenosem jejího obsahu do fyzického souboru.

Spláchnutím výstupního souboru zabezpečujeme, aby se vystupující data dostala tam, kam je posíláme v době, kdy je tam chceme mít aniž bychom kvůli tomu museli dotyčný výstupní soubor uzavírat. Pomocí této operace můžeme zabránit např. tomu, aby počítač „zmrzl“ proto, že čeká na nějakou naši reakci, přičemž výzva k této reakci stále ještě čeká ve vyrovnávací paměti.

Test konce souboru

Test konce souboru má smysl pouze u vstupních, resp. vstupně-výstupních souborů. Filozofie testování konce souboru se však v obou probíraných jazycích poněkud liší a ve svých důsledcích výrazně ovlivňuje i chování operace převzetí dat.

Jazyk C++ má vyhrazen speciální kód pro znak označující konec souboru. Tento znak získá volající program pokaždé, když chce číst za koncem souboru.

Naproti tomu jazyk Pascal nabízí speciální funkci, která oznamuje, zda právě načtená věta byla poslední větou v souboru či nikoliv. Operační systém však oznamuje konec souboru až poté, co se pokusil přečíst bajt, který již v souboru není. Chce-li tedy Turbo Pascal implementovat tuto funkci podle regulí, nezbyvá mu nic jiného, než být se čtením neustále o jednu větu napřed. To přináší jisté potíže při programování interaktivní komunikace uživatele s programem, o nichž budeme hovořit v následujícím odstavci.

Převzetí dat

Převzetí dat z vstupního, resp. vstupně-výstupního souboru programem, je poměrně jednoduchá operace, kterou by zde nebylo třeba nijak podrobně rozebírat, kdyby nebylo pascalské speciality s testováním konce souboru. Protože je mezi vámi řada těch, kteří buď

mezi oběma jazyky přecházejí anebo dokonce převádějí programy z jednoho jazyka do druhého, naznačím základní problémy pascalských programů již zde, přestože se jedná o pasáž věnovanou oběma jazykům.

Při dávkovém čtení textů z klávesnice se v Pascalu musíme vyvarovat použití funkce *readln*, která kromě vlastních zadávaných dat přečte i následující označení konce řádku. Vzhledem k nutnému náskoku jednoho přečteného bajtu pak bude program chtít zadání dalšího textu, a to bez jakéhokoliv upozornění. Tím může program přinejmenším přivést uživatele do rozpaků.

Omezení se na používání funkce *read* však také nemusí být někdy dostatečným řešením, protože pak můžete mezi přečtenými znaky nalézt i označení konce řádku, což většinou nebývá žádoucí. Program je proto nutno doplnit o další testy, které tyto nadpočetné znaky vyloučí.

Vložení dat

Vložení do výstupního, resp. vstupně-výstupního souboru je také poměrně jednoduchá operace, kterou není třeba předem nijak podrobně rozebírat. Pouze bychom chtěli upozornit programátory, jejichž programy v C++ by se měly překládat do Pascalu (i takové věci se dějí), aby pokud možno nevyužívali možnosti vrátit zpět do vstupního souboru jednou přečtený znak a pokusili se testovat konec v dané fázi načítaných dat jiným způsobem. Pascal totiž ekvivalentní funkci neposkytuje.

Zjištění a nastavení pozice v souboru

Jak jsme si řekli, semisekvenční soubor se od sekvenčního liší tím, že umožňuje přesun v rámci zpracovávaného souboru. Oba jazyky proto poskytují funkce, pomocí nichž můžeme tento přesun realizovat, a zároveň i funkce, které nám umožňují zjistit aktuální pozici v souboru.

Podívejme se nyní na specifika jednotlivých jazyků.

3.2 Práce se soubory v Pascalu

Pascal rozlišuje tři typy souborů: textové (*text files*), typové (*typed files*) a netypové (*untyped files*). Textové soubory jsou typu *Text*, netypové soubory jsou typu *File*; deklarace typových souborů se skládá z klíčových slov **file of**, za kterými následuje určení typu složek (vět) souboru. Možné deklarace proměnných uvedených tří kategorií ukazuje následující příklad:

```
var
  TextovySoubor :      Text;
  SouborZnaku   :      File of char;
  SouborZaznamu :      File of record
                        a: char;
                        b:word;
```

```

                                c:real;
                                end;
NetyповySoubor : File;

```

Textové soubory mají několik specifíků a v Pascalu pro ně platí následující pravidla:

- ✧ Jsou chápány jako posloupnosti znaků uspořádaných do řádků oddělených znakem *CR*, který může být případně následován i znakem *LF*. Samostatně stojící znak *LF* je považován za běžný znak.
- ✧ Na rozdíl od zbylých dvou typů souborů je můžeme používat pouze jako sekvenční. Jedním z důsledků tohoto omezení je to, že textové soubory mohou být pouze vstupní nebo pouze výstupní.
- ✧ Na rozdíl od typových a netyповých souborů, u nichž se data předávají nekonvertovaná, se při čtení z textových souborů a zápisu do nich provádějí konverze znakových řetězců na hodnotu požadovaného typu tak, jak jsme doposud zvyklí (standardní vstup a výstup považuje Pascal za textové soubory). Tyto konverze sice operace vstupu a výstupu zdržují, avšak na druhou stranu umožňují zadávat a číst data v přijatelném tvaru.
- ✧ Vyrovnávací paměť textových souborů je za normálních okolností velká 128 bajtů (pokud ji potřebujete větší, podívejte se do manuálu, jak se to dělá). Jedním z důsledků této velikosti je i chování procedury *Append* (viz dále).
- ✧ Při spuštění programu je vždy definován (a otevřen) vstupní textový soubor *input* a výstupní textový soubor *output*. Pokud program používá modul *CRT*, jsou tyto soubory sdruženy s přímým vstupem z klávesnice a přímým výstupem na obrazovku. V opačném případě jsou sdruženy se systémovým standardním vstupem a výstupem a lze je tedy např. přesměrovat.

Vedle textových souborů zavádí Turbo Pascal ještě typové a netyповé soubory. Typové soubory se chápou jako posloupnost vět obsahujících údaje stejného (daného) typu, netyповé soubory se chápou jako posloupnosti vět, jež mohou být obecně různých typů.

Abychom mohli v Turbo Pascalu pracovat s nějakým logickým souborem, musíme jej nejprve sdružit („spřáhnout“) s nějakým fyzickým souborem. Teprve pak je možno soubor otevřít a začít z něj číst či do něj zapisovat. Toto sdružení realizujeme zavoláním některé z následujících procedur:

procedure *Assign*(**var** *LogSoubor*; *Jmeno*: *String*);

Tato procedura sdruží logický soubor *LogSoubor* s fyzickým souborem, jehož jméno a případně i cesta k němu jsou předány v druhém parametru. Celková délka jména včetně případného názvu jednotky a cesty nesmí přesáhnout 79 znaků. V parametru *Jmeno* mohou být uváděny i identifikátory zařízení, např. '*LPT1*'. Pokud bude v parametru *Jmeno* prázdný řetězec, sdruží se daný logický soubor se standardním systémovým vstupem, resp. výstupem.

procedure *AssignCrt*(var *LogSoubor*: Text);

Sdruží logický textový soubor *LogSoubor* s přímým vstupem z klávesnice a přímým výstupem na obrazovku.

V průběhu programu můžeme daný logický soubor sdružit postupně s několika fyzickými soubory. (Logický soubor dokonce můžeme sdružit pokaždé s jiným druhem fyzického souboru.) Musíme však dbát na to, abychom nesdružovali soubor, který je ještě otevřený, a abychom s jedním fyzickým souborem nesdružili dva logické soubory.

Pro práci s logickými soubory, které jsou již sdruženy s nějakým fyzickým souborem, nabízí Pascal následující procedury a funkce (hranaté závorky uzavírají volitelné části deklarací, výpustka (...) označuje libovolný nenulový počet parametrů předem nedefinovaných typů):

procedure *Append*(var *Soubor*: Text);

Otevře textový soubor pro zápis a nastaví aktuální pozici na jeho konec definovaný operačním systémem. Nalezne-li v posledním 128 bajtů velkém bloku (přesněji v posledním obsahu vyrovnávací paměti) znaky Ctrl-Z (#26 = #\$1A), nastaví aktuální pozici na první z nich (tj. prvním zápisem se tento znak přepíše).

Byl-li soubor při vyvolání funkce již otevřen, nejprve jej zavře. Sdružený fyzický soubor otevíraného souboru již musí existovat, jinak systém ohlásí chybu.

procedure *BlockRead*(var *Soubor*:File;var *Promenna*; *Pocet*:word [; *Precteno*:word]);

Přečte z netypového souboru *Soubor* *Pocet* vět a uloží je do proměnné *Promenna*. Parametr *Precteno* je volitelný. Pokud jej použijete, vrátí v něm procedura počet skutečně přečtených vět (může se lišit od hodnoty *Pocet*, bylo-li např. nečekaně dosaženo konce souboru), pokud jej nepoužijete, vyvolá předčasné dosažení konce souboru chybu (viz funkce *IOResult*).

procedure *BlockWrite*(var *Soubor*:File; var *Promenna*; *Pocet*:word [; *Zapsano*:word]);

Zapíše do netypového souboru *Soubor* *Pocet* vět z proměnné *Promenna*. Parametr *Zapsano* je volitelný. Pokud jej použijete, vrátí v něm procedura počet skutečně zapsaných vět (může být menší než *Pocet*, byl-li např. nečekaně zaplněn disk).

procedure *Close*(var *Soubor*);

Uzavře otevřený soubor, přičemž u výstupních souborů před tím ještě spláchne obsah vyrovnávací paměti do sdruženého fyzického souboru. Pokud není u výstupních paměťových souborů zaplněna veškerá vyhrazená paměť, přidá za poslední zapsaný znak příznak konce souboru Ctrl-Z.

function *Eof*[(var *Soubor*)] : boolean;

Funkce testující dosažení konce souboru (pokud je uvedena bez parametrů, testuje stav souboru *input*). Vrací hodnotu **true** (*ANO*) v případě, že aktuální pozice je za poslední větou v souboru, a v případě, že soubor je prázdný. V opačném případě vrací **false** (*NE*).

function *EoLn* [(**var** *Soubor*: *Text*)] : *boolean*;

Testuje, zda se na aktuální pozici v souboru nachází příznak konce řádku. Pokud není uveden žádný parametr, testuje stav souboru *input*.

procedure *Erase*(**var** *F*);

Smaže externí fyzický soubor sdružený s logickým souborem *F*. Tato funkce nesmí být použita na otevřené soubory!

function *FilePos*(**var** *NetextovySoubor*) : *longint*;

Jak deklarace naznačuje, tuto funkci nelze použít pro textové soubory. Funkce vrací číslo udávající aktuální pozici v souboru, tj. pořadí věty, kterou se chystáme přečíst nebo zapsat. Počátek souboru má pozici 0 a v situaci, ve které funkce *eof* vrací **true**, je hodnota vrácená funkcí *FilePos* rovna hodnotě vrácené funkcí *FileSize*.

function *FileSize*(**var** *NetextovySoubor*) : *longint*;

Funkce vrací celkový počet vět v souboru. Ani tuto funkci nelze použít pro textové soubory.

procedure *Flush*(**var** *Soubor*: *Text*);

Funkce spláchne vyrovnávací paměť svého parametru do sdruženého fyzického souboru.

function *IOResult* : *word*;

Pokud máte při překladu vypnutou průběžnou kontrolu korektnosti vstupně-výstupních operací (direktivou `{SI-}`), nevedou tyto nekorektní operace k havarijnímu ukončení programu. Systém pouze ignoruje po nekorektní operaci všechny další požadavky na vstupně-výstupní operace, dokud nezavoláte funkci *IOResult*. Voláním funkce *IOResult* zjistí program, zda operace proběhla bezchybně či nikoliv. V případě úspěšné operace vrátí tato funkce 0, v opačném případě vrátí celé číslo označující kód chyby.

V následujícím přehledu jsou uvedena čísla chyb ohlašovaných funkcí *IOResult* spolu s jejich stručným vysvětlením a případně i možnou příčinou.

2 *Soubor nebyl nalezen*

Pravděpodobně bylo zadáno špatné jméno souboru nebo špatná cesta – systém ho hledá jinde. Občas nastane chyba jen proto, že je nastaven jiný aktuální adresář, než program předpokládá.

3 *Cesta nebyla nalezena*

Cesta vede k neexistujícímu adresáři.

4 Příliš mnoho otevřených souborů

Pokud není chyba v programu, musíte zvětšit maximální možný počet současně otevřených souborů v příkazu *FILES* v souboru *CONFIG.SYS*. Protože překladač také otevírá nějaké soubory, můžete se někdy této chyby zbavit i tak, že nebudete spouštět program z integrovaného prostředí, ale samostatně.

5 Přístup k souboru byl odmítnut

Tato chyba může mít více příčin. Většinou se ale jedná o pokus o smazání souboru (*Rewrite*), nebo o pokus otevření výstupního souboru (*Append*, *Reset* při *FileMode=1*, *2*), který má operační systém označen jako *read-only*, tj. jako soubor určený pouze pro čtení. Tato chyba také vzniká při pokusu o čtení (*Read*, *BlockRead*) z výstupního souboru nebo při pokusu o zápis (*Write*, *BlockWrite*) do vstupního souboru.

6 Špatné identifikační číslo souboru

Pokud vám systém oznámí tuto chybu, tak jste nejspíš omylem přepsali část souborové proměnné.

12 Neplatný přístupový režim

Při otevírání souboru byla v proměnné *FileMode* (viz výklad procedury *Reset*) nalezena nesmyslná hodnota.

13 Neplatné číslo diskové jednotky**16 Nelze smazat aktuální adresář****17 Přejmenovávat nelze mezi jednotkami**

Původní i přejmenovávaný soubor musí být na téže jednotce. Není-li tomu tak, musíte soubor na nové místo zkopírovat a pak původní soubor na starém místě smazat.

100 Chyba při čtení

Tuto chybu ohlašuje funkce *Read* ve chvíli, kdy se pokoušíte číst za koncem souboru.

101 Chyba při zápisu

Tato chyba vzniká, není-li možno uložit další data – při splachování vyrovnávací paměti do fyzického souboru na zaplněném disku.

102 Soubor není sdružen

Chcete otevřít logický soubor, který ještě nebyl sdružen s odpovídajícím fyzickým souborem.

103 Soubor není otevřen

Pokusili jste se pracovat se souborem, který ještě nebyl otevřen.

104 Soubor není otevřen jako vstupní

Pokusili jste se zapsat do vstupního souboru.

105 Soubor není otevřen jako výstupní

Pokusili jste se číst z výstupního souboru.

106 Špatný formát vstupních dat

Při konverzi vstupního řetězce z textového souboru byl přečten formát dat, který neodpovídá typu cílové proměnné.

150 Disk je chráněn proti zápisu

U disket stačí většinou odstranit ochranu (u 5,25" sundat přelepku, u 3.5" zakrýt okénko), u pevného disku se většinou jedná o kolizi s protivirovou ochranou.

151 Neznámá jednotka

Překontrolujte název sdružovaného fyzického souboru.

152 Chyba kontrolního součtu

Chyba nosiče dat. Pokud se chyba objeví i po opakování operace, je nejlepší program přerušit a chybu lokalizovat a případně opravit vhodným programem (CHKDSK, NDD (Norton Disk Doctor) apod.).

155 Špatný formát žádosti

Tento formát nemůžete ovlivnit. Pravděpodobně je to sekundární chyba vzniklá v důsledku toho, že jste v programu přepsali něco, co jste přepsat nechtěli a neměli.

156 Nepodařilo se najet na stopu**157 Neznámý typ média****158 Sektor nebyl nalezen**

Postupujte stejně jako v případě chyby 152. Pokud se jedná o chybu při práci s disketou, zkontrolujte, zda je disketa naformátovaná.

159 V tiskárně není papír**160 Chyba při zápisu****161 Chyba při čtení**

Postupujte stejně jako u chyby 152.

162 Blíže nespecifikovaná chyba

Sem patří i chyby způsobené hardwarem.

procedure *Read*([**var** *Soubor*;] ...);

S funkcí procedury *Read* jsme se již seznámili. V našich dosavadních programech jsme ji používali v podobě, kdy jejími parametry byly pouze proměnné, jejichž hodnotu jsme načítali.

Pokud je prvním parametrem logický soubor, pak bude procedura *Read* načítat data z tohoto souboru, jinak čte ze souboru *input*. Při čtení dat z textového souboru se vstupující text konvertuje na hodnotu typu odpovídajícího parametru. Je-li prvním parametrem typový soubor, žádná konverze se neprovádí a data z tohoto souboru se přímo kopírují do

skutečných parametrů, které musejí být všechny stejného typu jako jednotlivé věty souboru.

Pro netypové soubory nelze proceduru *Read* použít; místo ní nám poslouží procedura *BlockRead*.

procedure *ReadLn*([**var** *Soubor*: *Text*;] ...);

Procedura *ReadLn* je definována pouze pro textové soubory. Její funkce je skoro totožná s funkcí procedury *Read*, avšak *ReadLn* navíc po načtení hodnot všech parametrů nastaví aktuální pozici na nejbližší následující příznak konce řádku. Pokud v souboru již další příznak konce řádku není, nastaví aktuální pozici za poslední znak souboru, takže funkce *eof* bude vracet **true**.

procedure *Rename*(**var** *Soubor*; *NoveJmeno*: *String*);

Přejmenuje externí soubor sdružený s bgickým souborem *Soubor* na *NoveJmeno*.

procedure *Reset*(**var** *Soubor* [*:File*; *DelkaVety*: *word*]);

Procedura *Reset* je definována pro všechny typy souborů, i když se pro každý typ parametru chová trochu jinak. Její základní funkcí je otevřít existující soubor *Soubor*. Pokud je soubor již otevřen, nejprve jej zavře a pak jej znovu otevře. Pokud otevíraný soubor neexistuje, vyvolá chybový stav.

Pokud je prvním parametrem **textový soubor**, otevře jej procedura jako vstupní. Pokud je otevřený soubor prázdný, vrátí následné volání *eof*(*Soubor*) hodnotu **true** (*ANO*), není-li prázdný, vrátí **false** (*NE*).

Pokud prvním parametrem není textový soubor, tj. pokud je jím typový nebo netypový soubor, otevře jej procedura *Reset* v režimu definovaném obsahem globální proměnné *FileMode*, která je typu *byte*. Hodnoty této proměnné určují režim práce se souborem takto:

Hodnota	Význam
0	Pouze vstupní
1	Pouze výstupní
2	Vstupně-výstupní

Na počátku programu má tato proměnná hodnotu 2, takže netextové soubory otevírané procedurou *Reset* jsou považovány za vstupně-výstupní.

Pokud je prvním parametrem procedury *Reset* netypový soubor, je možné uvést ještě druhý parametr specifikující délku věty používanou při přenosu dat. Jelikož by skutečná délka souboru měla být násobkem této délky věty, bývají netypové soubory nejčastěji otevírány s délkou věty 1. Počet přenášených vět v procedurách *BlockRead* a *BlockWrite* je pak roven počtu přenášených bajtů. Neuvedeme-li u netypových souborů délku věty, přiřadí se jim délka věty 128 bajtů.

procedure Rewrite(var Soubor [; File; DelkaVety: word]);

Procedura *Rewrite* je definována pro všechny typy souborů, i když se pro každý typ parametru chová trochu jinak. Její základní funkcí je otevřít soubor *Soubor*. Pokud je soubor již otevřen, nejprve jej zavře a pak jej znovu otevře. Pokud otevíraný soubor existuje, je smazán a přepsán souborem otevíraným, který je otevřen jako prázdný s aktuální pozicí nastavenou na počátek souboru.

Pokud je prvním parametrem **textový soubor**, otevře jej procedura *Rewrite* jako výstupní. Netextové soubory otevírá nezávisle na hodnotě proměnné *FileMode* (viz výklad procedury *Reset*) jako vstupně-výstupní.

Pokud je prvním parametrem netypový soubor, je možné uvést ještě druhý parametr specifikující délku věty používanou při přenosu dat. Podrobněji se o významu tohoto parametru dočtete v popisu funkce *Reset*.

Chcete-li soubor otevřít tak, aby se sdružený fyzický soubor vytvořil pouze v případě, že dosud neexistuje, tj. aby se sdružený fyzický soubor nepřepsal, musíte pro typový a netypový použít sekvenci:

```
{ $I- }
Reset( Soubor );
if( IOResult <> 0 )then Rewrite( Soubor );
```

a pro textový soubor sekvenci:

```
{ $I- }
Append( Soubor );
if( IOResult <> 0 )then Rewrite( Soubor );
```

procedure Seek(var NetextovySoubor; NovaPozice: longint);

Přesune aktuální pozici v typovém nebo netypovém souboru na zadanou větu.

function SeekEOF [(var Soubor: Text)];

Tato funkce přesune aktuální pozici za všechny případné následné bílé znaky (mezery, tabulátory a příznaky konce řádku) a vrátí hodnotu funkce *eof*. Volání funkce bez parametrů je ekvivalentní volání s parametrem *input*.

function SeekEoLn [(var Soubor: Text)];

Tato funkce přesune aktuální pozici za všechny případné následné mezery a tabulátory a vrátí hodnotu funkce *EoLn*. Volání této funkce bez parametrů je ekvivalentní volání s parametrem *input*.

procedure Truncate(var Soubor);

Smaže všechny věty za aktuální pozicí, takže aktuální pozice bude za posledním záznamem v souboru a následné volání *eof(Soubor)* by vrátilo hodnotu **true** (*ANO*). Na textové soubory však nemá volání této funkce žádný efekt.

procedure Write([var Soubor: Text;] ...);

S funkcí procedury *Write* jsme se již seznámili. V našich dosavadních programech jsme ji používali v podobě, kdy jejími parametry byly pouze proměnné, jejichž hodnotu jsme tiskli na obrazovku, resp. posílali do přeměňovaného standardního výstupního souboru.

Pokud je prvním parametrem logický soubor, bude procedura *Write* posílat data do tohoto souboru, v opačném případě je posílá do souboru *output* (to byl náš dosavadní případ). Jsou-li data posílána do textového souboru, proběhne při výstupu konverze vystupujících hodnot na textový řetězec. Je-li prvním parametrem typový soubor, žádná konverze neprobíhá a data se do tohoto souboru přímo kopírují ze skutečných parametrů, které musí být všechny stejného typu jako jednotlivé věty souboru.

Pro netyповé soubory nelze proceduru *Write* použít, místo ní použijeme proceduru *BlockWrite*.

procedure *WriteLn*([**var** *Soubor*: *Text*;] ...);

Procedura *WriteLn* je definována pouze pro textové soubory. Její funkce je podobná jako u procedury *Write*, avšak *WriteLn* navíc po vypsání hodnot všech parametrů pošle na výstup ještě příznak konce řádku.

Pro úplnost si na závěr povíme ještě o procedurách, které umožňují některé základní manipulace s celými adresáři.

procedure *ChDir*(*Cesta*: *String*);

Nastaví aktuální adresář podle cesty v parametru *Cesta*. Pokud je součástí cesty i nové označení diskové jednotky, změní i ji.

procedure *GetDir*(*Jednotka*: *Byte*; **var** *Cesta*: *String*);

V parametru *Jednotka* očekává číslo jednotky, na jejíž aktuální adresář se ptáme. Hodnota 0 označuje aktuální jednotku, 1 označuje jednotku A:, 2 jednotku B: atd. V parametru *Cesta* vrátí cestu k aktuálnímu adresáři na označené jednotce.

Pozor! Procedura nekontroluje korektnost čísla jednotky. Pokud zadáte neexistující jednotku, vrátí v parametru *Cesta* hodnotu '\.

procedure *MkDir*(*Cesta*: *String*);

Vytvoří adresář specifikovaný parametrem *Cesta*. Poslední položkou cesty však nesmí být jméno nějakého existujícího souboru nebo adresáře.

procedure *Rmdir*(*Cesta*: *String*);

Smaže prázdný adresář specifikovaný parametrem *Cesta*. Pokud zadaný adresář nenajde, pokud zjistí, že není prázdný, anebo pokud se jedná o aktuální adresář, nastane chyba.

Příklad

Po spouště suchého povídání se podíváme na jednoduchý příklad. Napišeme program, který vytvoří v aktuálním adresáři soubor celých čísel *DATA.DTA* a zapíše do něj deset náhodných čísel z intervalu 0 .. 999. Poté tento soubor uzavře, otevře ho znovu pro čtení, přečte hodnoty v něm a vypíše je do textového souboru *DATA.TXT*. Při zakládání souboru *DATA.DTA* si náš program nejprve zjistí, zda v aktuálním adresáři již soubor se stejným jménem není, a pokud tam je, vypíše upozornění a skončí.

Při čtení dat ze souboru *DATA.TXT* se náš program bude tvářit, že neví, kolik čísel v něm je.

```
(* Příklad P3 - 1 *)
{Jednoduchá ukázka práce se soubory v Pascalu}

program Pascal;

const
    textovy = 'data.txt';
    cely = 'data.dta';
    {Názvy souborů}

var Fint: file of integer;
    ftext: text;
    i, j: integer;

begin
    {Sdružíme logické a fyzické soubory}
    assign(ftext, textovy);
    assign(fint, cely);

    {$I-}
    reset(fint);
    i := IOResult;
    {$I+}
    {Zjistíme, zda soubor celých čísel existuje}
    case i of
        0: begin
            writeln('Soubor ', cely, ' existuje. Sorry. ');
            Halt(1);
        end;
        2:;
        else begin
            writeln(' Chyba při práci se souborem ', cely);
            Halt(1);
        end;
    end;

    {Otevřeme soubor celých čísel pro zápis, vypíšeme do něj deset
    náhodných čísel a pak jej zase uzavřeme}
    rewrite(fint);
    for i := 1 to 10 do begin
        j := random(1000);
        write(fint, j);
        {Procedura write vyžaduje jako
        parametr
        proměnnou}
    end;
    close (fint);
```

```

{Nyní tento soubor otevřeme pro čtení, přečteme z něj čísla
(tváříme se, že nevíme, kolik jich je) a prepíšeme je
do textového souboru}

reset(fint);
rewrite(ftext);

j := 0;                                {Uspořádáme výstup do sloupců po 5}
while not eof(fint) do begin
  read(fint, i);
  write(ftext, i:6, ' ');
  inc(j);
  if j = 5 then begin writeln(ftext); j:=0 end
end;

{Nakonec oba soubory uzavřeme}
close(fint);
close(ftext);
end.

```

Nyní se pokusíme přečíst čísla, zapsaná do souboru *DATA.TXT*, a vypsat je (spolu s pořadovými čísly) na obrazovku. Přitom se budeme opět tvářit, že nevíme, kolik čísel v souboru je. Pro práci se souborem použijeme tentokrát proměnnou

```
var cisla: text;
```

Budeme-li číselná data číst z textového souboru procedurou

```

(* Příklad P3 - 2 *)
procedure CtiSpatne;
var
  i, j: integer;
begin
  {Tento cyklus přečte ještě neexistující nulu na konci souboru}
  j := 0;
  while not eof(cisla) do begin
    inc(j);
    read(cisla, i);
    writeln(j:2, ' ', i:10);
  end;
end;

```

dočkáme se nemilého překvapení: na obrazovku se vypíše 11 údajů, i když v souboru jich je jen 10. Problém je v tom, že po přečtení znakové reprezentace posledního čísla ze souboru *DATA.TXT* nebude *eof(Cisla) = true*, neboť za ní ještě následují znaky konce řádku (a možná i nějaké mezery). Cyklus proto neskončí a pokusí se o další čtení, které se již nepovede.

Řešení je jednoduché: stačí v podmínce cyklu použít místo funkce *eof* funkci *SeekEof*, která přeskočí všechny bílé znaky na vstupu a teprve pak vrátí hodnotu funkce *eof*. Správná podoba cyklu pro čtení tedy je

```

while not SeekEof(cisla) do begin
  inc(j);

```

```

read(cisla,i);
writeln(j:2, ' ', i:10);
end;

```

3.3 Práce s datovými proudy v C++

V C++ je možno pracovat se proudy, jejichž vlastnosti jsou obdobné vlastnostem pascalových souborů – tuto možnost nabízí funkce ze standardní knihovny převzaté z jazyka C (z jazyka C je převzat i termín „proud“, i když byl v C++ ještě dále zobecněn). Kromě klasické koncepce proudů jazyka C však C++ nabízí nový, daleko univerzálnější a mocnější koncept datových proudů, který je již definován s využitím možností objektově orientovaného programování.

Skutečnost, že datové proudy jazyka C++ jsou naprogramovány pomocí prostředků objektově orientovaného programování, výklad poněkud komplikuje, protože spoustu věcí ještě neznáme. Avšak čekat s výkladem datových proudů až na dobu, kdy budete již potřebně teoreticky vyzbrojeni, také není příliš moudré, protože práce s proudy (tedy vstupy a výstupy) tvoří páteř velké části programů.

Máme samozřejmě ještě třetí možnost – vykládat klasické metody práce s proudy zděděné z jazyka C. (Proudy jazyka C byly v podstatě ekvivalentní souborům jazyka Pascal.) Ta se nám také příliš nelíbí, protože ač prostředky jazyka C nabízejí mnohem více než jejich pascalské protějšky, nové prostředky jazyka C++ jsou ještě daleko tvárnější a mocnější, takže byste je nakonec stejně chtěli používat a zbytečně byste se museli učit práci se dvěma sadami prostředků. (Opravdový profesionál však stejně ovládá obě, protože existují některé situace, kdy je použití některé z původních funkcí výhodnější. My o klasických datových proudech jazyka C mluvíme v dodatku knihy *Práce s daty I*)

Vraťme se ale k datovým proudům. Jak jsme si již řekli v úvodu kapitoly, datový proud je označení pro libovolný tok dat směřujících od producenta (zdroje) ke konzumentovi (cíli, spotřebiči), přičemž producentem, resp. konzumentem může být prakticky cokoliv: klasický soubor, externí zařízení, oblast paměti, náš vlastní, ale i paralelně běžící program atd.

Koncepce datových proudů umocněná možnostmi jazyka C++ přináší oproti pascalovým souborům několik velkých výhod. Mezi nejdůležitější patří možnost rozšíření množiny datových typů, pro něž jsou definovány vstupně-výstupní konverze, a možnost zavedení vlastních specializovaných typů proudů, které automaticky zdědí všechny možnosti práce s proudy, aniž bychom je museli dodefinovávat.

Za univerzalitu a mocnost datových proudů se ale musí něco zaplatit: touto cenou je velikost programů definujících operace s datovými proudy (asi 40 KB oproti původním asi 6 KB v jazyce C) a zároveň i jejich mírné zpomalení. Toto zpomalení je tím zřetelnější, čím rychlejší je vlastní fyzický proud. Nejzřetelnější bude u přímého výstupu znaků na

obrazovku nebo do paměti. Pokud však budete místo znaků posílat na obrazovku či do paměti čísla, která se musí na textový řetězec nejprve konvertovat a nebo pokud dokonce budete tato data posílat na disk, bude zpomalení prakticky zanedbatelné.

Komplexní a úplný výklad systému datových proudů by vydal na samostatnou knihu. Proto se omezíme pouze na podmnožinu nejzákladnějších funkcí, která vám nenabídne o mnoho více než sada funkcí a procedur jazyka Pascal, jejichž použití jsme vysvětlovali v minulé podkapitole. S dalšími možnostmi datových proudů jazyka C++ se seznámíme v povídání o objektově orientovaném programování.

Poznámka:

Jak jistě víte, ve firemních manuálech je řada nepřesností a místy i chyb a informace v nich nejsou vždy úplné (koneckonců, nejsou to učebnice). Pokud tedy zjistíte, že v prototypch uvádíme jiné typy parametrů než příručka programátora, a nebudete vědět, komu máte věřit, najděte si daný prototyp v jednom ze souborů iostream.h, fstream.h a strstream.h, a podívejte se, jak je deklarován doopravdy. Pokud nebudete věřit informacím, které v příručce nenajdete, musíte si vše buď odzkoušet nebo se přesvědčit nahlednutím do zdrojových textů knihovnických funkcí. (Zdrojové texty se pro některé z verzí borlandských překladačů dají koupit a u jiných jsou dokonce součástí produktu.)

Podívejme se nejprve na nejdůležitější odchylky datových proudů jazyka C++ od souborů jazyka Pascal:

1. Základní a nejdůležitější rozdíl je způsoben tím, že Pascal neposkytuje prostředky pro definici některých operací se soubory (např. v Pascalu nelze naprogramovat procedury *read* a *write*) a souborové datové typy a operace s nimi jsou proto v Pascalu součástí definice jazyka. Naproti tomu v C++ se definice jazyka otázkami vstupu a výstupu vůbec nezabývá a definice proudových datových typů i příslušných operací jsou součástí standardní knihovny, takže si je každý může kdykoliv upravit nebo zcela předefinovat.

Skutečnost, že definice potřebných datových typů a funkcí jsou součástí knihovny, přenositelnost programů nijak nesnižuje, protože návrh koncepce datových proudů byl vypracován autory jazyka C++ a autoři překladačů tento nepsaný standard dodržují.

2. Turbo Pascal jasně odděluje tři akce: deklaraci logického souboru, jeho sdružení s příslušným fyzickým souborem a jeho otevření. C++ slučuje sdružení logického a fyzického proudu s jejich otevřením. Navíc umožňuje deklarovat logický proud tak, aby se při zahájení jeho doby života (u automatické proměnné při aktivaci deklarace v toku programu, u statických proměnných hned po spuštění programu) datový proud zároveň otevřel.

Na rozdíl od Pascalu nemusíte v C++ myslet ani na zavírání datových proudů, protože ve chvíli, kdy končí doba života daného logického proudu (u automatických proměnných při opouštění bloku s deklarací dané proměnné, u statických proměnných při řádném ukončení programu) zařídí překladač jeho automatické zavření.

3. Operace s datovými proudy jsou v jazyku C++ definovány pomocí objektově orientovaných rysů jazyka. Pamatujte si, že v objektově orientovaném programování se konstanty a proměnné objektových typů, tzv. tříd (**třída** je objektovým zobecněním pojmu struktura, resp. záznam), nepovažují za pasivní entity, na něž aplikujeme nějaké procedury a funkce, ale za aktivní objekty, které pro nás dělají různé služby. Tyto služby plní voláním funkcí, tzv. **metod**, které se považují za složky příslušné struktury. Z toho také přirozeným způsobem vyplývá i syntax těchto volání, která se liší od syntaxe, na niž jste byli doposud zvyklí. Z uvedených důvodů se totiž funkce proudů volají takovým způsobem, jako by se jednalo o složky datové struktury odpovídajícího logického proudů (proudové proměnné) – tj. s využitím operátorů přímého (.) nebo nepřímého (->) přístupu.

Položme nyní pokličku zpět na své místo a podívejme se na vlastní realizaci celé koncepce. V našich dosavadních programech jsme se setkávali se třemi proudy: s proudem umožňujícím přímý výstup na obrazovku (obvykle jsme mu dávali jméno *crt*), se standardním vstupním proudem *cin* a standardním výstupním proudem *cout*, které jsou ekvivalenty pascalských proudů *input* a *output*. Oproti Pascalu však vedle těchto dvou standardních proudů C++ zavádí ještě dva tzv. standardní chybové proudy: nebafovaný proud (tj. nevyužívající vyrovnávací paměť) proud *cerr* a bafrovaný proud *clog*.

Standardní chybové proudy jsme ve svých programech doposud nepoužili. Není však na nich nic zvláštního: chovají se úplně stejně jako standardní výstup, pouze nejdu v DOSu přesměrovat z příkazového řádku⁹. Pokud tedy standardní textový výstup přesměrujete, texty posílané do standardního chybového výstupu budou i nadále směřovat na obrazovku.

Logické proudy, které budeme v našich programech využívat, budou jednoho z následujících šesti typů:

<i>fstream</i>	– externí proud,
<i>ifstream</i>	– externí vstupní proud,
<i>ofstream</i>	– externí výstupní proud,
<i>stringstream</i>	– paměťový proud,
<i>istringstream</i>	– paměťový vstupní proud,
<i>ostringstream</i>	– paměťový výstupní proud.

Kromě toho se v následujících deklaracích objeví ještě dva další typy:

<i>istream</i>	– obecný vstupní proud,
<i>ostream</i>	– obecný výstupní proud.

⁹ Z programu však přesměrovat jdou; v některých jiných prostředích (např. pod UNIXem) mohou být i tyto proudy přesměrovatelné.

Přijměte prosím bez vysvětlování tvrzení, že všechny vstupní a vstupně-výstupní proudy jsou **také** typu *istream* a všechny výstupní a vstupně-výstupní proudy jsou **také** typu *ostream*.

Abychom mohli s objekty uvedených typů pracovat, musíme dovést (**#include**) odpovídající hlavičkové soubory. Pro práci s externími proudy musíme dovést soubor *fstream.h* a pro práci s paměťovými proudy soubor *fstream.h*. Řada dalších potřebných informací se nachází ještě v souboru *iostream.h*, avšak pokud budete dovážet jeden z předchozích dvou souborů, tak se v rámci jejich dovozu doveze i soubor *iostream.h*.

Režimy otevření proudu

Pokud jste četli výklad o práci s externími soubory v Pascalu, jistě si pamatujete, že režim další práce se souborem byl dán za prvé deklarací (dělení na textové, typové a netypové soubory) a za druhé funkcí, pomocí níž jsme soubor otevřeli. Naproti tomu v C++ je režim práce se souborem dán většinou hodnotou parametru předaného otevírací funkci. Hodnotu tohoto parametru vytváříme bitovým součtem (bitovým *OR* – operátor `|`) z následujících hodnot, které jsou konstantami výčtového typu `ios::open_mode`:

ios::app – Po otevření proudu se nastaví aktuální pozice na jeho konec. Nové věty jsou do proudu zapisovány vždy na jeho konec, byť by byla aktuální pozice nastavena ~~ka~~koliv.

ios::ate – Po otevření proudu se nastaví aktuální pozice na jeho konec. První zapisovaný bajt je přidán na konec, avšak další zapisované bajty již respektují právě nastavenou aktuální pozici.

ios::in – Soubor bude otevřen jako vstupní. Pokud existuje, nebude jeho obsah smazán.

ios::out – Soubor bude otevřen jako výstupní.

ios::trunc – Pokud proud již existuje, bude jeho obsah smazán.

ios::nocreate – Používá se při otevírání fyzických proudů. Při otevírání proudu se kontroluje, zda daný fyzický proud (soubor) již neexistuje. Pokud proud existuje, otevře se. Pokud neexistuje, nastane chyba.

ios::noreplace – Používá se při otevírání fyzických proudů. Při otevírání proudu se kontroluje, zda daný fyzický proud (soubor) již neexistuje. Pokud proud neexistuje, vytvoří se. Pokud existuje, nastane chyba.

ios::binary – Otevře proud jako binární. Není-li tento příznak nastaven, bude proud otevřen jako textový, což znamená, že při vstupu budou posloupnosti `"\r\n"` (CR, LF) převáděny na znak `'\n'` a při výstupu bude každý znak `'\n'` převeden naopak na posloupnost znaků `"\r\n"`.

Chybové stavy

Stav, v němž se může proud nacházet, je bitovým součtem hodnot výčtového typu `ios::io_state`, který je definován následovně:

```
enum ios::io_state
{
    goodbit = 0x00,    //Nic není nastaveno - vše OK
    eofbit = 0x01,    //Byl dosažen konec souboru
    failbit = 0x02,   //Chyba při poslední operaci
    badbit = 0x04,    //Pokus o zakázanou operaci
    hardfail = 0x80   //Neodstranitelná chyba
}
```

Na aktuální stav proudu *Proud* se můžeme zeptat několika způsoby – záleží na tom, co se chceme dozvědět. Slouží k tomu následující funkce:

int *rdstate*();

Vrací současný stav proudu.

int *good*()

Vrací *ANO* (nenulovou hodnotu) v případě, že není nastaven žádný chybový příznak.

int *eof*();

Vrací *ANO* (nenulovou hodnotu) v případě, že je nastaven příznak *eofbit*.

int *fail*();

Vrací *ANO* (nenulovou hodnotu) v případě, že je nastaven aspoň jeden z bitů *failbit*, *badbit* a *hardfail*.

int *bad*();

Vrací *ANO* (nenulovou hodnotu) v případě, že je nastaven aspoň jeden z bitů *badbit* a *hardfail*.

void *clear*(**int** *Stav* = 0);

Nastaví nový chybový stav na hodnotu svého parametru. Nebyl-li parametr zadán, vynuluje všechny chybové příznaky.

int *operator !*()

Má stejný význam jako metoda *fail*(). Na bezchybnost operací, které vracejí jako svoji funkční hodnotu odkaz na proud, se tedy můžeme ptát i pomocí homonyma operátoru logické negace – např. chybný stav při čtení znaku z proudu *Vstup* můžeme ošetřit příkazy

```
if( !Vstup.get( Znak ) ) Chyba();
```

Specifika externích datových proudů

Deklarace

V C++ můžeme konstanty a proměnné (časem se dozvíme, že se jim souhrnně říká instance) řady datových typů deklarovat více způsoby: buď klasicky bez jakýchkoliv dodatečných parametrů, nebo naopak s parametry, jejichž seznam umístíme v kulatých závorkách následujících za identifikátorem deklarované instance (konstanty či proměnné).

```
fstream fff;
ifstream fff;
ofstream fff;
```

Deklaruje logický externí proud, který bude v průběhu programu sdružen s nějakým externím proudem. Výsledkem deklarace není nic víc, než zavedení nové proměnné, kterou ještě nemůžeme ke vstupním a výstupním operacím používat.

```
fstream fff( const char *Jmeno, int Rezim );
ifstream fff( const char *Jmeno, int Rezim=ios::in );
ofstream fff( const char *Jmeno, int Rezim=ios::out );
```

Deklaruje logický proud *fff*, sdruží jej s fyzickým proudem se jménem *Jmeno* a zároveň jej otevře v režimu specifikovaném parametrem *Rezim*. Takovéto proudy jsou již připraveny k použití.

Pamatujte, že pokud chcete vstupní soubor otevřít tak, abyste byli upozorněni na případnou neexistenci sdruženého fyzického souboru, musíte jej otevřít v režimu

```
ios::in | ios::nocreate
```

nebo

```
ios::in | ios::nocreate | ios::binary.
```

Otevření, zavření, spláchnutí

Jak jsme si řekli, proud je možno sdružit se souborem a otevřít již v rámci deklarace. Někdy to však není nejvýhodnější postup, a proto C++ nabízí i funkce na samostatné sdružení a otevření, spláchnutí a zavření externího datového proudu.

Než si o těchto funkcích začneme vyprávět podrobněji, chtěli bychom vás upozornit na dvě zvláštnosti:

1. Pokud je platnost některého z dále uváděných prototypů omezena na některý typ externích proudů, je identifikátor tohoto typu uveden před identifikátorem deklarované funkce a oddělen čtyřtečkou.
2. Připomínám, že uváděné funkce je nutno volat „objektově“, tj. použitím operátoru selekce – např.:

```
ifstream Vstup;
Vstup.open( "c:..\DATA\vstupy.txt" );
```

A nyní tedy slíbený přehled nejdůležitějších prostředků specifických pro práci s externími datovými proudy:

```
void fstream::open( const char *Jmeno, int Rezim );
void ifstream::open( const char *Jmeno, int Rezim = ios::in );
void ofstream::open( const char *Jmeno, int Rezim = ios::out );
```

Sdruží externí fyzický proud *Jmeno* s daným logickým proudem a otevře jej v režimu definovaném hodnotou parametru *Rezim*.

Pokud je daný logický proud již s nějakým fyzickým proudem sdružen, případně pokud během sdružování nastanou jiné komplikace, nastaví se chybový příznak *ios::failbit*. Tento příznak bude nastaven také v případě, že fyzický proud se jménem *Jmeno* neexistuje a v parametru *Rezim* je nastaven příznak *ios::nocreate*.

```
void close();
```

Zavře daný proud. Pokud při zavírání souboru nenastane výjimečná situace, budou po zavření všechny chybové příznaky vynulovány.

flush

S identifikátorem *flush* jsme se seznámili již dříve, takže si jistě pamatujete, že je to manipulátor a že s jeho pomocí spláchneme např. proud *Vystup* příkazem

```
Vystup << flush;
```

Specifika paměťových datových proudů

Jak víme, paměťové proudy jsou proudy, které slouží pro práci se znakovými řetězci

```
istream sss;
```

Nepřípustná deklarace! Vstupní paměťové proudy musí být vždy deklarovány s parametrem!

```
ostream sss;
stringstream sss;
```

Deklaruje logický (a nepřímo i fyzický) dynamický paměťový proud, sdruží jej s interně vybudovaným fyzickým proudem (oblastí paměti) a otevře jej. Termín *dynamický* označuje skutečnost, že v době deklarace ještě není známo, jak velkou část paměti bude program pro uložení zapisovaných informací potřebovat. Proud si proto nějakou paměť vyhradí a v případě potřeby ji bude operativně zvětšovat.

```
istream sss( char *Text );
```

Deklaruje vstupní binární (`\n` se nepřevádí) logický paměťový proud *sss*, sdruží jej s textovým řetězcem *Text* (sdružený fyzický proud), z něž budou přebírána načítaná data,

a otevře jej. Délku řetězce není nutno uvádět, protože každý textový řetězec je, jak víme, ukončen prázdným znakem.

```
istream sss( char *Pamet, int Delka );
```

Deklaruje vstupní binární logický paměťový proud *sss*, sdruží jej s oblastí paměti velkou *Delka* bajtů, která začíná na adrese, na niž ukazuje parametr *Pamet*. Tato oblast paměti tedy vystupuje jako fyzický proud, ze kterého budou přebírána data.

Pokud je parametr *Delka* roven nule, funkce předpokládá, že na adrese *Pamet* nalezne řetězec ukončený prázdným znakem. Pokud je *Delka* menší než nula, předpokládá, že má k dispozici paměť nekonečné délky.

```
ostream sss( char *Pamet, int Delka,
int Rezim=ios::out );
stringstream sss( char *Pamet, int Delka, int Rezim );
```

Deklaruje výstupní binární logický paměťový proud *sss* a sdruží jej s fyzickým proudem, kterým je oblast paměti velká *Delka* bajtů a začínající na adrese, na niž ukazuje parametr *Pamet*.

Pokud je parametr *Delka* roven nule, funkce předpokládá, že na adrese *Pamet* nalezne řetězec ukončený prázdným znakem. Pokud je *Delka* menší než nula, předpokládá, že má k dispozici paměť nekonečné délky.

Pokud je v parametru *Rezim* nastaven příznak *ios::app* nebo *ios::ate*, předpokládá se, že ve vyhrazené paměti se nachází řetězec zakončený prázdným znakem, na nějž se nastaví aktuální pozice, tzn. že první vystupující znak přepíše tento ukončující prázdný znak.

Vstup dat

V C++ není na rozdíl od Pascalu definován způsob přenosu dat typem použitého proudu, ale pouze volanou funkcí. Doposud jsme používali pouze operátory formátovaného vstupu (>>) a výstupu (<<). Nyní si povíme něco i o funkcích pro nekonvertovaný přenos.

Funkce, které vracejí hodnotu typu *istream&*, resp. *ostream&*, vrací odkaz na objekt, který je zavolal.

```
int istream::gcount();
```

Vrátí počet znaků přečtený poslední funkcí pro neformátovaný vstup, tj. jednou z funkcí *get()*, *getline()*, *ignore()*, *read()*. Při jejím použití však nesmíte zapomenout na to, že hodnotu této funkce změní každý další, tj. i formátovaný vstup – součástí formátovaného vstupu je totiž úvodní neformátované čtení konvertovaného textu.

```
int istream::get();
```

Přečte jeden znak ze vstupního proudu a vrátí jeho kód.

```
istream& istream::get( char& Znak );
```

Přečte jeden znak ze vstupního proudu a vrátí jej v parametru *Znak*.

```
istream& istream::get( char *Znaky, int Pocet, char Konec = '\n' );
```

Načte znaky ze vstupního proudu do pole *Znaky* velkého *Pocet* bajtů. Pokud je v průběhu tohoto čtení načten znak *Konec*, skončí načítání před zaplněním tohoto pole. Znak *Konec* se ještě do pole *Znaky* přidá. Na konec načtených dat se zapíše prázdný znak – aby se do pole vešel, může se načíst nejvýše *Pocet-1* znaků.

```
istream& istream::getline( char *Znaky, int Pocet, char Konec = '\n' );
```

Pracuje stejně jako funkce *get()*, avšak pokud načte znak *Konec*, neuloží ho do pole *Znaky*, ale „zahodí“ ho.

```
istream& istream::ignore( int Počet = 1, int Konec = EOF );
```

Přeskočí následující znaky až po „znak“ *Konec* včetně – nejvýše však *Pocet* znaků.

```
int istream::peek();
```

Vrátí kód znaku, který je na řadě, aniž by jej přečetla (tedy znak zůstane v proudu).

```
istream& istream::putback( char Znak );
```

Vrátí přečtený znak zpět do vstupního proudu. Pokud se vrácený znak liší od znaku naposledy přečteného, není výsledek deňován.

```
istream& istream::read( char* Kam, int Kolik );
```

Přečte ze vstupního proudu *Kolik* znaků a uloží je do pole *Kam*.

Výstup dat

```
ostream& ostream::put( char Znak );
```

Zapíše do proudu znak *Znak*.

```
ostream& ostream::write( const char *Data, int Pocet );
```

Zapíše do proudu *Pocet* bajtů z pole *Data*.

Zjišťování a nastavování pozice

Protože C++ považuje všechny (tj. i textové) soubory za semisekvenční, musí umožňovat zjistit aktuální čtecí a zapisovací pozici v souboru a nastavit pozici novou.

Aby bylo možno knihovnu proudů snáze přenášet mezi různými počítači, jsou pro funkce pro zjišťování a nastavování aktuální pozice v souboru definovány typy *streampos* a *streamoff*; v Borland C++ vypadají jejich deklarace takto:

```
typedef long streampos
typedef long streamoff
```

Pro zjišťování a nastavování aktuální pozice v souboru nabízejí proudy jazyka C++ následující metody:

```
istream& istream::seekg( streampos Pozice );
```

```
ostream& ostream::seekp( streampos Pozice );
```

Nastaví aktuální pozici pro čtení (*seekg()*), resp. zápis (*seekp()*) na zadanou hodnotu. Počátek souboru má pozici 0.

```
istream& istream::seekg( streamoff Posun, ios::seek_dir VztaznyBod );
```

```
ostream& ostream::seekp( streamoff Posun, ios::seek_dir VztaznyBod );
```

Nastaví aktuální pozici pro čtení (*seekg()*), resp. zápis (*seekp()*) tak, že bude posunutá o *Posun* bajtů vůči vztažnému bodu definovanému parametrem *VztaznyBod*. Tento parametr může nabýt jedné za tří hodnot:

ios::beg – počátek souboru

ios::cur – aktuální čtecí pozice

ios::end – konec soubor

```
streampos istream::tellg();
```

```
streampos ostream::tellp();
```

Vrátí aktuální čtecí (*tellg()*), resp. zapisovací (*tellp()*) pozici.

Poznámka:

Jak jste si mohli všimnout, všechny uvedené funkce jsou řešeny zvlášť pro čtecí a zvlášť pro zapisovací pozici. U externích vstupně-výstupních souborů však tyto dvě pozice nejsou nezávislé. Nastavením jedné se zároveň nastavuje i druhá. Toto omezení je důsledkem omezení operačního systému DOS.

Ostatní operace

Než přejdeme k vlastnímu výkladu poslední operace, nahlédněme nejprve tvůrcům datových proudů zase jednou trochu pod pokličku. Z programátorského hlediska se můžeme na datové proudy dívat jako na úzce spolupracující tandem. Datový proud je deklarován jako struktura (přesněji třída), jejíž jednou složkou (jednou, ale ne jedinou) je ukazatel na buffer. Tento buffer je definován také jako třída a vykonává pro svůj datový proud všechny služby spojené s komunikací se sdruženým fyzickým proudem: sdružování, otevírání a zavírání souborů, vlastní přenos dat, umístování aktuální pozice i splachování. Proud (rozumějte logický proud) si pak bere na starost pouze komunikaci s volajícím programem včetně případné konverze dat a všechny žádosti programu o služby, které má na starosti buffer, mu prostě předává.

Poznámka:

Nesměšujte buffer odkazovaný z datového proudu s obyčejnou vyrovnávací pamětí. Ta samozřejmě může být součástí bufferu, ale jinak je buffer komplexní datová struktura, kterou musí používat i proudy, které s žádnou vyrovnávací pamětí nepracují.

```
int rdbuf()->is_open();
```

Vrátí nenulovou hodnotu v případě, že proud je již otevřen, a nulu v případě opačném. Neděste se složité definice – je taková proto, že funkce není součástí proudu, ale pouze jeho bufferu, takže musíme nejprve pomocí funkce `rdbuf()` získat adresu bufferu a teprve ten můžeme požádat o hledanou informaci. Chcete-li např. zjistit, zda je otevřen proud *Proud*, musíte se zeptat příkazem

```
if( Proud.rdbuf()->is_open() ) ...
```

Příklad

Nyní se konečně podíváme na jednoduchý příklad. Napíšeme program, který vytvoří v aktuálním adresáři binární soubor *DATA.DTA* a zapíše do něj deset náhodných celých čísel z intervalu 0 .. 999. Poté tento soubor uzavře, otevře ho znovu pro čtení, přečte hodnoty v něm a vypíše je do textového souboru *DATA.TXT* (vždy 5 čísel na řádek). Při zakládání souboru *DATA.DTA* si náš program nejprve zjistí, zda v aktuálním adresáři již soubor se stejným jménem není, a pokud tam je, vypíše upozornění a skončí.

Při čtení dat ze souboru *DATA.TXT* se náš program bude tvářit, že neví, kolik čísel v něm je.

```
/* Příklad C3 - 1 */
//Jednoduchá ukázka práce se soubory

#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>

//Jména souborů
const char *textovy = "data.txt";
const char *cely = "data.dta";

//Proudy
fstream Fint, Ftext;

int main(){
    int i, j;
    //Sdružíme soubor (fyzický proud) a logický proud v programu
    //a zároveň otestujeme, zda soubor existuje
    Fint.open(cely, ios::out|ios::noreplace|ios::binary);
    if(!Fint) { //Pokud existuje, skončíme
        cerr << "Soubor " << cely << " existuje. Sorry." << endl;
        exit(1);
    }

    //Vypíšeme binárně do souboru 10 náhodných celých čísel a zavřeme jej
    for(i = 0; i < 10; i++) {
        j = random(1000);
```

```

    Fint.write((char*)&j, sizeof(j));
}
Fint.close();

//Soubor znovu otevřeme, tentokrát pro čtení, a zároveň otevřeme
//textový soubor, do kterého budeme psát přečtená čísla
Fint.open(cely, ios::in|ios::binary);
Ftext.open(textovy, ios::out);

//V cyklu čteme (binárně) data ze souboru DATA.DTA a formátovaná
//je zapisujeme (po 5 na řádek) do souboru DATA.TXT
j = 0;
while(!Fint.read((char*)&i, sizeof(i))){
    Ftext << setw(6) << i << " ";
    j++;
    if(j == 5) {
        Ftext << '\n';
        j = 0;
    }
}

//Nakonec soubory uzavřeme
Ftext.close();
Fint.close();

return 0;
}

```

Cyklus **while** trvá, dokud se čtení daří. Protože operátor „!“ vrací hodnotu *ANO*, jestliže se operace nepodařila, použili jsme ho dvakrát. Ve skutečnosti ho ale nemusíme použít vůbec – můžeme napsat

```
while(Fint.read((char*)&i, sizeof(i))){ ... }
```

V knize *Objektové programování 2* mluvíme o tom, proč. Zatím to vezměte jako fakt.

Nyní přečteme (jako celá čísla) data, uložená v textovém souboru *DATA.TXT*, a vypíšeme je na obrazovku.

```

/*   Příklad C3 - 2   */
#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>

//Program přečte čísla, uložená v textovém souboru data.txt,
//a vypíše je na obrazovku
ifstream Ftext("data.txt");

int main(){
    //Nejprve test, zda soubor existuje a podařilo se jej otevřít:
    if(!Ftext){
        cerr << "Soubor data.dta se nepodařilo otevřít. " << endl;
        exit(1);
    }

    int i, j = 1;

```

```

while(!Fint.read((char*)&i, sizeof(i))){
//Všimněte si, že vstupní operaci můžeme použít jako
//podmínku v příkazu while; je to tím, že výraz
//Ftext >> i
//má hodnotu proudu Ftext po operaci, takže na ni můžeme aplikovat
//operátor !
while(!(Ftext >> i)){
    cout << setw(2) << j++ << setw(6) << i << endl;
}
return 0;
}

```

Všimněte si podmínky opakování v cyklu **while**. Protože hodnotou výrazu

```
Ftext >> i
```

je odkaz na datový proud *Ftext* po provedení výstupní operace, můžeme na něj použít operátor „!“ . I zde je ovšem dvojnásobné použití tohoto operátoru zbytečné, místo toho můžeme napsat

```
while(Ftext >> i){ ... }
```

Vyzkoušejte si to.

3.4 Formátovaný vstup a výstup v C++

Řekli jsme si, že výstup do proudů můžeme formátovat dvěma základními způsoby: voláním formátovacích funkcí a manipulátory. Víme již také, že použití manipulátorů je běžnější. Manipulátory se většinou používají všude, kde je to jen trochu možné, kdežto formátovací funkce se používají většinou pouze v situacích, kdy je použití manipulátorů nevhodné nebo dokonce nemožné.

Probereme si nyní formátovací možnosti pěkně jednu po druhé a u každé si řekneme, jakými způsoby je možné požadovaných výsledků dosáhnout.

Než se pustíme do rozebírání jednotlivých možností, musíme vás upozornit na jednu nepříjemnou skutečnost. Přestože učené knihy praví, že knihovna datových proudů je vytvořena samotnými autory jazyka a je proto téměř shodná ve všech implementacích, ukazuje se, že v případě překladačů firem Borland a Microsoft musíme klást důraz na slovo **téměř**. V každém případě doporučujeme dát při případném přenosu programů mezi těmito překladači pozor i na formátování vstupu a výstupu, i když o něm všichni tvrdí, že je standardizované. (Ostatně původní návrh se poněkud liší od toho, co pravděpodobně bude v konečné verzi normy ANSI C++, takže doba zmatků ještě chvíli potrvá.)

Rozšíření operátorů << a >> pro typy definované uživatelem

V C++ můžeme rozšířit definici většiny operátorů. To znamená, že můžeme definovat jejich význam pro strukturové a výčtové¹⁰ typy. (V literatuře se o tom také hovoří jako o přetěžování operátorů.) Podrobný výklad najdete v knize *Objektové programování 2*. Zde se naučíme, jak si definovat vlastní verze vstupních a výstupních operátorů << a >> pro struktury a unie.

Začneme příkladem. Nejprve definujeme strukturu *bod*, která bude popisovat bod v rovině. Pro typ *bod* pak rozšíříme vstupní a výstupní operátor. Deklarace operátoru je podobná deklaraci funkce, ovšem jméno této funkce se skládá z klíčového slova **operator** následovaného symbolem operátoru (<< nebo >>). Za ním pak následují v závorkách parametry – tedy operandy. Prvním parametrem je levý operand, tedy proud; pro vstupní proud musí být typu *istream&*, pro výstupní proud typu *ostream&*.

Výsledek působení operátoru bude hodnota, vrácená takto deklarovanou funkcí. Vstupní i výstupní operátor musí vracet odkaz na proud, se kterým pracoval. To znamená, že vstupní operátor >> vrací hodnotu typu *istream&*, výstupní operátor << vrací hodnotu typu *ostream&*. Pozorně si prohlédněte následující příklad a zkuste si ho kókovat.

```

/*   Příklad C3 - 3   */
#include <iostream.h>

struct bod {                               //Bod v rovině
    double x, y;
};

//Výstupní operátor pro typ bod
ostream& operator<< ( ostream& Proud, bod &b ){
    Proud << '(' << b.x << ", " << b.y << ')';
    return Proud;
}

//Vstupní operátor pro typ bod
istream& operator>> ( istream& Proud, bod &b ){
    Proud >> b.x >> b.y;
    return Proud;
}

bod a;

int main(){
    cout << "zadej bod a: ";
    cin >> a;
    cout << "bod a má souřadnice " << a;
    return 0;
}

```

¹⁰ Borland C++ umožňuje přetěžovat operátory pro výčtové typy až od verze 4.0.

Šířka výstupního pole

Pod šířkou výstupního pole rozumíme minimální počet vytištěných znaků. Pokud má konvertovaná hodnota více znaků, „vyteče“ z vyhrazeného prostoru doprava.

Šířku pole můžete definovat buď pomocí metody (tj. funkce, kterou chápeme jako složku daného proudu) *width()*, nebo prostřednictvím manipulátoru *setw()*. Hlavní použití metody *width()* při zjišťování aktuální nastavené velikosti vyhrazeného prostoru; pokud nechceme zároveň nastavovat novou šířku, můžeme ji zavolat bez parametru. Jinak dáme metodě *width()* přednost asi tehdy, když je vlastní výstup součástí nějakého složitějšího výrazu a my potřebujeme nastavit velikost prostoru před jeho vyhodnocováním. Ve všech ostatních případech asi použijeme manipulátor *setw()*.

Při nastavování šířky pole nesmíme zapomenout, že toto nastavení (na rozdíl ode všech ostatních formátovacích příkazů) platí pouze pro výstup nejbližší příští hodnoty a pak se opět automaticky nastaví na nulu. Chceme-li tedy tisknout několik stejně dimenzovaných hodnot za sebou, musíme nastavit šířku pole před tiskem každé z nich.

Následující příklad ukazuje velice jednoduché rozšíření operátoru výstupu pro tisk zlomků. Tento operátor předpokládá, že právě nastavená šířka se vztahuje na zlomek jako celek. Proto spočítá, kolik znaků potřebuje na jmenovatele a na lomítko, a zbytek přidělí čitateli. Pokud uživatel nastaví větší šířku než je nezbytně třeba, doplní se zleva mezery.

```

/*   Příklad C3 - 4   */
//Tisk zlomků
#include <iomanip.h>
#include <conio.h>

struct sZlomek {
    int Citatel;
    int Jmenovatel;
};

int /******/ Znaků /******/
( int Cislo )           //Počet znaků potřebných pro dané číslo
{
    int Znamenko=0;
    if( Cislo < 0 )
    {
        Cislo = -Cislo;
        Znamenko = 1;
    }
    if( Cislo >= 10000 ) return Znamenko+5;
    else if( Cislo >= 1000 ) return Znamenko+4;
    else if( Cislo >= 100 ) return Znamenko+3;
    else if( Cislo >= 10 ) return Znamenko+2;
    else return Znamenko+1;
}
/******/ Znaků /******/

ostream& /******/ operator << /******/
( ostream& o, sZlomek& Z )
{
    //Nejprve nastavíme šířku pole pro čitatele
    o.width( o.width() - 3 - Znaků(Z.Jmenovatel) );
}

```

```

    o << Z.Citatel << " / " << Z.Jmenovatel;
    return o;
}
/***** operator << *****/
void main(){
    clrscr();
    sZlomek z = {1323, 15369};
    cout << setw(30) << z;
}

```

Funkce řady *printf()*, které se pro formátovaný výstup používaly v jazyce C a které C++ od tohoto jazyka zdědilo, umožňují dokonce omezit i maximální počet tištěných znaků. Tuto možnost však standardní formátovací funkce datových proudů jazyka C++ nenabízejí. Pokud možnost omezení maximálního počtu vystupujících znaků potřebujete a nechcete měnit přímo funkce ze systémové knihovny, můžete problém vyřešit tak, že danou hodnotu nejprve vytisknete do nějakého řetězce (o paměťových proudech jsme si již řekli) a z něj pak vyberete vhodně velký podřetězec, který teprve pošlete do cílového výstupního proudu.

Zobrazení znaménka u kladných čísel

Při běžném výstupu čísel bývá zvykem uvádět u čísel pouze záporné znaménko a kladné znaménko nezapisovat. Při čtení čísel se pak všechna čísla, u nichž není explicitně uvedeno záporné znaménko, považují za kladná.

V některých případech je však užitečné uvádět znaménko i u kladných čísel. Toho dosáhneme nastavením formátovacího příznaku *ios::showpos*, což můžeme učinit několika způsoby: buď pomocí manipulátoru *setiosflags()*, nebo pomocí metody *setf()*. Pokud se změní situace a nebudeme chtít kladné znaménko tisknout, shodíme tento příznak manipulátorem *resetiosflags()* nebo metodou *unsetf()*. Obě možnosti předvádí následující ukáзка:

```

/*   Příklad C3 - 5   */
#include <iomanip.h>

void main(){
    int Cislo = 12345, CisloSeZn = 321, CisloBezZn = 547;

    //Pomocí manipulátorů:
    cout << setiosflags( ios::showpos )
         << Cislo << endl
         << resetiosflags( ios::showpos );

    //Pomocí metod
    long f = cout.setf( ios::showpos );
    cout << CisloSeZn << endl;
    cout.unsetf( ios::showpos );
    cout << CisloBezZn << endl;
    cout.flags( f );           //Návrat do původního stavu
}

```

Z ukázky je vidět, že při použití samotných manipulátorů nemůžeme vrátit formátovací příznaky do stavu před našimi úpravami. Proto musíme použít metodu, která vrátí původní hodnotu formátovacích příznaků, kterou pak můžeme po skončení akce obnovit.

Způsob zarovnávání

Pokud je šířka pole větší než prostor nezbytně potřebný, vzniká otázka, kam do tohoto prostoru vystupující hodnotu umístit. Pascal to řeší jednoduše: vystupující hodnotu se snaží přirazit k pravému okraji vyhrazeného prostoru. To vyhovuje při tisku čísel, ne však při tisku řetězců, které jsme naopak zvyklí zarovnávat vlevo. C++ nabízí tyto dvě možnosti a kromě toho pro tisk čísel ještě třetí způsob zarovnávání: znaménko dorazí k levému kraji a vlastní hodnotu k pravému. Případný prostor mezi nimi pak zaplní definovanými výplňovými znaky.

Protože volíme jednu ze tří možností, nemůžeme vystačit s nastavením nebo shozením jednoho příznaku. C++ má proto pro způsob zarovnávání vyhrazeny tři příznaky, z nichž právě jeden je vždy nastaven. Vzhledem k tomu, že je třeba zaručit, že bude nastaven právě jeden z nich, nepoužívá se pro tento účel manipulátorů (bylo by to teoreticky možné, ale zbytečně složité), ale metody *setf()*, a to její verze se dvěma parametry. Za první parametr dosadíme příznak, který chceme nastavit, a za druhý parametr pak konstantu *ios::adjustfield*, podle níž funkce pozná, že nastavujeme způsob zarovnávání.

```
/*   Příklad C3 - 6   */
#include <iomanip.h>

//...

long l = cout.setf( ios::left, ios::adjustfield );
cout << "Vlevo: »" << setw( 10 ) << -3.14 << "«\n"; //K levému kraji:
cout.setf( ios::right, ios::adjustfield );
cout << "Vpravo: »" << setw( 10 ) << -3.14 << "«\n"; //K pravému
kraji:
cout.setf( ios::internal, ios::adjustfield );
cout << "Uvnitř: »" << setw( 10 ) << -3.14 << "«\n"; //K oběma krajům:
cout.flags( l );
```

Výplňový znak

Výplňovým znakem, tj. znakem, kterým se zaplňuje vyhrazený, avšak nepotíštěný prostor, je implicitně mezera. Toto implicitní nastavení však můžeme kdykoliv změnit. Opět máme k dispozici manipulátor *setfill()* a metodu *fill()* ve dvou verzích – bez parametru a s jedním parametrem.

Jak jsme již řekli, implicitním výplňovým znakem je mezera. Velmi často však uživatel požaduje, aby před vlastním číslem byly nějaké znaky (např. nuly nebo hvězdičky), aby nebylo možno číslo dodatečně upravit. Jiná situace, kdy můžeme změnu výplňového znaku využít, je tisk různých přehledů, v nichž jsou texty doráženy vlevo a doplňovány

k prvnímu okraji tečkami či jinými znaky. Tisk takového řádku s nadpisem a příslušnou položkou provádí následující procedura:

```

/*   Příklad C3 - 7   */
#include <iomanip.h>

void /*****/ TiskPolozky /*****/
( ostream& proud, const char* Popis, int Hodnota )
{
    const SireTextu = 50;
    const SireCisla = 10;

    //Uchováme si původní hodnotu formátovacích příznaků
    long f = proud.flags();

    //Nastavíme nový výplňový znak '.' a původní si uložíme
    char c = proud.fill( '.' );
    proud.setf( ios::left, ios::adjustfield );
    proud << setw( SireTextu ) << Popis;

    //Pro číslo nastavíme jako výplňový znak nulu
    proud.fill( '0' );
    proud.setf( ios::right, ios::adjustfield );
    proud << setw( SireCisla ) << Hodnota << endl;

    //Obnovíme původní výplňový znak a příznaky
    proud.flags( f );
    proud << setfill(c);
}
/*****/ TiskPolozky *****/

void main() {
    TiskPolozky(cout, "Nějaká položka", 12365);
}

```

Výstup tohoto programu bude

```
Nějaká položka.....0000012365
```

Základ číselné soustavy

Každý profesionální programátor, který se vzmohl na více než prosté databázové aplikace, vám potvrdí, že s desítkovou soustavou při práci s počítačem nevystačí, a pokud mu jeho jazyk nenabízí možnost pracovat také v soustavě osmičkové nebo šestnáctkové (preference jedné z nich záleží na architektuře počítače a koncepci programového vybavení – na počítačích PC je mnohem výhodnější používání šestnáctkové soustavy), nezbude mu nic jiného, než si potřebné funkce co nrychleji vytvořit sám.

Jak víme, jazyk C++ vznikl z jazyka C, což byl jazyk vytvořený zejména pro systémové programování, kde se ve dvojkových soustavách pracuje častěji, než v soustavě desítkové. Proto je přirozené, že jazyk C měl možnost vstupu a výstupu celých čísel v těchto soustavách zabudovanou ve standardní knihovně a nejinak je tomu i v jazyku C++.

Standardní knihovna datových proudů jazyka C++ nabízí možnost vstupu a výstupu celých čísel v desítkové, osmičkové a šestnáctkové soustavě. Koncepce je podobná jako u

způsobu zarovnávání: číselná soustava, z níž (a do níž) systém čísla převádí, je definována tím, který z příznaků `ios::dec`, `ios::oct` a `ios::hex` je nastaven. Pokud není nastaven žádný z nich, jsou vystupující čísla převáděna do desítkové soustavy a pro vstupující čísla platí stejná pravidla jako pro celočíselné konstanty jazyka C++. (Čísla začínající znaky `0x` se považují za šestnáctková, čísla začínající nulou bez následujícího `x` za osmičková a zbylá čísla za desítková.)

Pro nastavení číselné soustavy se většinou používá neparametrických manipulátorů `dec`, `oct` a `hex` nebo manipulátoru `setbase()`, který má jeden celočíselný parametr, jenž může nabývat pouze hodnot 0, 8, 10 a 16, přičemž hodnota nula shazuje všechny tři příznaky a nastavuje tak režim, o němž jsme hovořili na konci minulého odstavce. Pokud je to někomu sympatičtější, může tento „nulový režim“ nastavit i pomocí manipulátoru `resetioflags()` s parametrem `ios::basefield`.

Pokud byste z nějakých důvodů dávali před manipulátory přednost použití některé metody, doporučuji použít metodu `setf()`, jejímž prvním parametrem bude jeden z příznaků `ios::dec`, `ios::oct` nebo `ios::hex` nebo nula a druhým parametrem bude konstanta `ios::basefield`.

```
/* Příklad C3 - 8 */
//Vstup a výstup v různých číselných soustavách
#include <iomanip.h>

void main ()
{
    int i;
    long fi = cin.flags(), fo = cout.flags();
    cout << "Zadej číslo v libovolné soustavě: ";
    cin >> setbase( 0 ) >> i;
    cout << "\nHex=" << hex << i
        << ", Dec=" << dec << i
        << ", Oct=" << oct << i << endl;
    cin .flags( fi ); //Obnovíme původní stav proudů
    cout.flags( fo );
}
```

Velikost znaku při hexadecimálním výstupu a zobrazení základu číselné soustavy

Pokud posíláme na výstup čísla v šestnáctkové soustavě, můžeme si vybrat, zda chceme „číslice“ od 10 do 15 psát jako malá nebo velká písmena, tj. a, b, c, d, e, f nebo A, B, C, D, E, F. Velikost těchto písmen ovládá nastavení příznaku `ios::uppercase`. Je-li nastaven, použijí se velká písmena, je-li shozen, použijí se písmena malá.

Příznak nastavíme tím, že jej pošleme jako parametr manipulátoru `setioflags()` nebo jednoparametrické metodě `setf()`, a shodíme tím, že jej pošleme jako parametr manipulátoru `resetioflags()` nebo metodě `unsetf()`. Domníváme se, že vše je natolik jednoduché, že nemusíme uvádět žádný ilustrační příklad.

Stejně jednoduché je i určení, zda se budou čísla tištěná v osmičkové a šestnáctkové soustavě tisknout samotná, nebo zda se s nimi budou tisknout předpony, které se používají

při zadávání těchto čísel v programech. Pokud tedy chceme zobrazovat osmičková čísla s vedoucí nulou a šestnáctková čísla s předponou *0x*, musíme nastavit příznak *ios::showbase*, a až je budeme chtít tisknout bez těchto předpon, musíme výše zmíněný příznak opět shodit.

Formáty tisku reálných čísel

Při tisku reálných čísel máme možnost určit tři charakteristiky výstupu:

- ✧ Zda se použije přímý nebo semilogaritmický tvar,
- ✧ počet míst za desetinnou tečkou,
- ✧ tisk závěrečných nul.

Podívejme se nyní na jednu možnost po druhé.

Přímý tvar (někdy se mu říká zobrazení v pevné desetinné čárce, resp. tečce) volíme nastavením příznaku *ios::fixed* a semilogaritmický (někdy můžete slyšet vědecký) tvar nastavením příznaku *ios::scientific*. Protože je třeba zajistit, aby byl nastaven právě jeden z těchto příznaků, používá se pro jejich nastavení dvouparametrové verze metody *setf()*, přičemž za druhý parametr dosazujeme konstantu *ios::floatfield*.

Počet míst za desetinnou tečkou můžeme nastavovat buď pomocí parametrického manipulátoru *setprecision()*, nebo pomocí metody *precision()*. Takto nastavený počet desetinných míst bude dodržen i v případě, kdy výsledné číslo „vyteče“ z vyhrazeného prostoru, nebo v případě, kdy se budou tisknout číslice, jejichž hodnotu systém nezaručuje (víme, že čísla ve formátu **double** jsou uložena s přesností na 15 platných cifer – další místa si již program „vymýšlí“). Kromě toho se může při příliš velké požadované přesnosti stát, že v průběhu výpočtu dojde k chybě podtečení či přetečení. Pokud chcete tisknout čísla bez desetinné části, nedosáhnete toho nastavením nulového počtu desetinných míst. Toto nastavení pokládá systém za příkaz tisknout čísla s implicitními šesti desetinnými místy.

K řízení tisku koncových nul nám slouží příznak *ios::showpoint*. Pokud je nastaven, bude se tisknout tolik desetinných míst, kolik je nastaveno (připomínáme: 0 = 6). Pokud je tento příznak shozen, netisknou se desetinná místa, jejichž hodnota je nulová a za nimiž následují samé nuly. U celých čísel se dokonce netiskne ani desetinná čárka. To však může přestat platit ve chvíli, kdy chcete tisknout číslo s větší přesností, než s jakou je systém uchovává – zkuste např. vytisknout číslo 12345.125, které je typu **double**, s přesností na 15 desetinných míst. Systém vám nezávisle na hodnotě příznaku *ios::showpoint* vytiskne 12345.125000000000001.

Chcete-li si ověřit vše, co jsme vám dosud řekli, spusťte si následující programek.

```
/* Příklad C3 - 9 */
#include <iomanip.h>
void /*****/ FloatTest /*****/ ()
{
```

```

long fo = cout.flags();
int po = cout.precision();
double c[ 3 ] = {1234567890.12345, 12345.125, 123 };
cout.setf( ios::showpoint );
for( int t=1; t >= 0; t--, cout.unsetf( ios::showpoint ) )
{
    cout.setf( ios::scientific, ios::floatfield );
    for( int v=1; v >= 0;
        v--, cout.setf( ios::fixed, ios::floatfield ) )
    {
        for( int ip=3, p[4]={ 9, 5, 1, 0 }; ip >= 0; ip-- )
        {
            cout.precision( p[ip] );
            cout << "\nTečka = " << t
                << " Vědecky = " << v
                << " Přesnost = " << p[ip] << endl;
            for( int ic=2; ic >= 0; ic-- )
                cout << "»" << setw( 15 ) << c[ic] << "«*»";
            cout << endl;
        }
    }
}
cout.precision( po );
cout.flags( fo );
}
/***** FloatTest *****/
void main(){
    FloatTest();
}

```

Po spuštění by měl program vypustit následující výstup:

```

Tečka = 1 Vědecky = 1 Přesnost = 0
» 1.230000e+02«*»» 1.234512e+04«*»» 1.234568e+09«*»

Tečka = 1 Vědecky = 1 Přesnost = 1
» 1.2e+02«*»» 1.2e+04«*»» 1.2e+09«*»

Tečka = 1 Vědecky = 1 Přesnost = 5
» 1.23000e+02«*»» 1.23451e+04«*»» 1.23457e+09«*»

Tečka = 1 Vědecky = 1 Přesnost = 9
»1.230000000e+02«*»»1.234512500e+04«*»»1.234567890e+09«*»

Tečka = 1 Vědecky = 0 Přesnost = 0
» 123.000000«*»» 12345.125000«*»»1234567890.123450«*»

Tečka = 1 Vědecky = 0 Přesnost = 1
» 123.0«*»» 12345.1«*»» 1234567890.1«*»

Tečka = 1 Vědecky = 0 Přesnost = 5
» 123.00000«*»» 12345.12500«*»»1234567890.12345«*»

Tečka = 1 Vědecky = 0 Přesnost = 9
» 123.000000000«*»»12345.125000000«*»»1234567890.123450041«*»

```

```

Tečka = 0 Vědecky = 1 Přesnost = 0
»      1.23e+02««*»»  1.234512e+04««*»»  1.234568e+09««*»

Tečka = 0 Vědecky = 1 Přesnost = 1
»      1.2e+02««*»»   1.2e+04««*»»       1.2e+09««*»

Tečka = 0 Vědecky = 1 Přesnost = 5
»      1.23e+02««*»»  1.23451e+04««*»»  1.23457e+09««*»

Tečka = 0 Vědecky = 1 Přesnost = 9
»      1.23e+02««*»»  1.2345125e+04««*»» 1.23456789e+09««*»

Tečka = 0 Vědecky = 0 Přesnost = 0
»      123««*»»       12345.125««*»»1234567890.12345««*»

Tečka = 0 Vědecky = 0 Přesnost = 1
»      123««*»»       12345.1««*»»   1234567890.1««*»

Tečka = 0 Vědecky = 0 Přesnost = 5
»      123««*»»       12345.125««*»»1234567890.12345««*»

Tečka = 0 Vědecky = 0 Přesnost = 9
»      123««*»»       12345.125««*»»1234567890.123450041««*»

```

Takto tištěná čísla se však těžko zarovnávají v tabulkách do sloupců podle desetinné tečky – jediná možnost, která nás napadá, je, tisknout zvlášť celou část a zvlášť desetinou část.

Přeskakování bílých znaků na vstupu

Přeskakování bílých znaků ovládá příznak `ios::skipws`. Je-li nastaven, jsou před každým vstupem nejprve přeskočeny všechny bílé znaky a čtení začíná teprve prvním nebílým znakem. Je-li shozen, čte se hned. Myslíme si, že jde o natolik jednoduchou věc, že nepotřebuje žádný demonstrační příklad a že si každý může vše vyzkoušet sám.

Splachování

Zbývají ještě dva příznaky, o nichž jsme dosud nemluvili: `ios::unitbuf` a `ios::stdio`. Oba popisují způsob splachování proudů. Nastavení příznaku `ios::unitbuf` způsobí spláchnutí všech proudů po každé operaci výstupu. Nastavení příznaku `ios::stdio` způsobí spláchnutí proudů `cout` a `cerr`. Nazveme-li tento stav nuceným splachováním (není to nejlepší termín, ale nic lepšího nás nenapadlo), pak stav při shození těchto příznaků bychom mohli nazvat řízeným splachováním.

Není to úplně přesné, protože ve chvíli, kdy se vyrovnávací paměť zaplní a my budeme chtít poslat na výstup další data, se vyrovnávací paměť automaticky nejprve spláchne. Pokud však můžeme zaručit, že nás systém se splachováním nepředběhne, můžeme si při řízeném splachování postupně připravovat data, která ve vhodný okamžik spláchneme do fyzického výstupního proudu.

Příprava výstupu a řízení jeho přenosu do fyzického proudu předpokládající dostatečnou kapacitu vyrovnávací paměti je sice možné, ale přece jen to není řešení optimální. Rozebíráme je zde spíše proto, abyste získali lepší představu o možných podzdech bafrovaného výstupu splachujícího v nepravou chvíli, protože chyby synchronizace splachování mohou někdy vyvolat nepochopitelně záhadné chování programu.

V následující ukázce se vykonává tělo cyklu dvakrát: poprvé zůstává příznak `ios::unitbuf` v proudu `cerr` nastaven (implicitní stav), před spuštěním druhého běhu tento příznak shodíme. Pro tento druhý běh jsou také určeny komentáře, které hovoří o splachování. Nelekněte se deklarací proměnných `bi` a `bo` – jsou určeny pro výklad v následující části.

```

/*   Příklad C3-10   */
#include <fstream.h>
#include <iomanip.h>
#include <conio.h>

streambuf *bo = cerr.rdbuf(); //Příprava pro následující výklad
streambuf *bi = cin.rdbuf();

void /*****/ main /*****/ ()
{
    clrscr();
    for( int p=2;      p--; cerr.unsetf( ios::unitbuf ) )
        //Poprvé se splachováním nuceným,
        //podruhé se splachováním řízeným
        {
            cerr << "Toto ";           //Jednotlivé výstupy jsou
            cerr << "je test ";        //zadány samostatně, aby je
            cerr << "splachování ";    //bylo možno krokovat
            cerr << "proudů" << endl; //Splachujeme a jedeme znovu
            cerr << "Zadej nějaká čísla (vstup končí nulou): ";
            int i;
            do {                       //Zde zadejte: 123 456 789 0
                cin >> i;
                cerr << setw(4) << i ;
            } while( i != 0 );
            cerr << "\nTohle jsi zadal" << endl;           //Teď teprve
            spláchneme cerr
            char c;
            cin.unsetf(ios::skipws); //Protože jsme potlačili
            //přeskakování úvodních bílých znaků, bude při
            //následujícím čtení přečten nejprve závěrečný
            //'\\n' z minulého vstupu.
            cerr << "\\nA teď zadej znaky - '*' znamená konec: " << endl;
            do {
                cin >> c;           //Zadejte: N a z -
                cerr << c;           // d a r !*
            } while( c != '*' );    //Dokud nestiskneme *
            cin.setf( ios::skipws ); //Zpátky nastavíme přeskakování
            //úvodních bílých znaků
            cerr << "\\nTohle jsi zadal\\n"; //Odřádkování bez
            splachování
        }
}

```

```

    cerr << flush;                //Spláchneme výstupní proud
}
}

```

Zkuste si tento program odkrokovat. Pokud zadáte vstupní data uvedená v příslušných komentářích, obdržíte následující výstup:

```

Toto je test splachování proudů
Zadej nějaká čísla (vstup končí nulou): 123 456 789 0
123 456 789 0
Tohle jsi zadal

A teď znaky - '*' znamená konec:
N a z -
N a z -
d a r !*
d a r !*
Tohle jsi zadal
Toto je test splachování proudů
123 456 789 0
Zadej nějaká čísla (vstup končí nulou): 123 456 789 0
Tohle jsi zadal
A teď znaky - '*' znamená konec:
N a z -
d a r !*

N a z -
d a r !*
Tohle jsi zadal

```

Při krokování tohoto příkladu si všimněte, že při druhém průchodu cyklem se vždy přečetly vstupy a teprve pak se vypsaly výstupy, takže při zadávání čísel jsme vlastně nevěděli, co po nás počítač chce.

Poznamenejme, že pokud bychom místo proudu *cerr* použili proud *cout*, dopadlo by to jinak. Proud *cin* a *cout* jsou totiž svázané v tom smyslu, že před použitím proudu *cin* se vždy spláchnou proud *cout*. Vyzkoušejte si to.

3.5 Ladění komunikace s datovými proudy

Na závěr této kapitoly bychom se s vámi chtěli podělit o několik zkušeností týkajících se ladění komunikace s datovými proudy. Největším kamenem úrazu většinou bývá, že programátor neumí průběžně sledovat, které znaky v daný okamžik do výstupního proudu poslal (přesněji které do proudu opravdu došly) a naopak, které znaky na něj ve vstupním proudu čekají. Jinými slovy: programátor neumí zjistit, jak to vypadá v proudu kolem aktuální pozice. Tato neschopnost ovšem nevyplývá z podstaty datových proudů, ale pouze z neznalosti samotného programátora. Ukážeme si proto několik triků, jak se o situaci kolem aktuální pozice něco dozvědět.

Vše se bude točit kolem vyrovnávací paměti, do níž jsou načítána data z fyzických vstupních proudů a odkud si je program přebírá, resp. kam jsou data posílána programem, aby byla ve vhodnou chvíli poslána do fyzického výstupního proudu. Budeme se tedy bavit o tzv. bafrovaných proudech – potřebujete-li ladit operace s nebafrovanými proudy, musíte si poradit sami.

Jak víme, obsahuje každý logický datový proud ukazatel na buffer, který získáme voláním metody *rdbuf()*. Pokud je daný proud bafrovaný, obsahuje buffer ukazatel na počátek vyrovnávací paměti, na její konec a také na aktuální pozici. Abychom nemuseli být ve sledovacím okně moc „ukecaní“, definujeme si v modulu globální proměnnou typu *streambuf**, které přiřadíme adresu bufferu sledovaného proudu, např.

```
streambuf *bo = cerr.rdbuf();
streambuf *bi = cin.rdbuf();
```

a ve sledovacím okně pak můžeme zadat např. následující výrazy:

```
bo->pbase_
```

Tento výraz představuje adresu počátku vyrovnávací paměti (zde pro výstupní proud *cerr*). Obsah paměti je zobrazován jako textový řetězec. Nulový bajt tento řetězec ukončí.

```
bo->pptr_ -5
```

Představuje adresu aktuální výstupní pozice proudu (zde *cerr*) minus 5 znaků. Pokud nejsme na počátku vyrovnávací paměti, ukáže nám posledních 5 vyslaných znaků a další znaky ve vyrovnávací paměti.

```
bi->gptr
```

Představuje adresu aktuální vstupní pozice v proudu (zde *cin*). Při další operaci čtení budou čteny následující bajty.

```
bi->egptr
```

Představuje adresu prvního bajtu za řetězcem připraveným ve vstupní vyrovnávací paměti, tj. adresu prvního bajtu, který se již číst nebude.

Zkuste si doplnit tyto příkazy do sledovacího okna a příklad, ukazující nucené a řízené splachování, odkrokuje ještě jednou.

4. Podmíněný překlad a makra

V této kapitole si budeme vyprávět o prostředcích, které nám umožňují ovlivňovat předzpracování zdrojového textu do podoby, kterou teprve bude zpracovávat překladač, a zároveň ovlivňovat i následnou práci překladače. Nejprve si povíme o základních charakteristikách prostředků, které nám oba jazyky pro toto předzpracování zdrojového textu nabízejí. Pak si ukážeme, jak je možné ovlivnit některé rysy překladu, a ukážeme si, že vkládat soubory do zdrojového textu můžeme i v pascalských programech. Poté se budeme chvíli věnovat pouze jazyku C++ a vysvětlíme si, co jsou to makra a jak je můžeme v našich programech použít. Nakonec se naučíme určovat, které části našeho textu budou a které nebudou překládány.

S řadou věcí jsme se už v této knize setkali, ale mnohé ještě musíme doplnit a upřesnit.

4.1 Dolarové poznámky a příkazy preprocesoru

Prostředky pro ovlivňování překladu, které nám oba jazyky nabízejí, se dost podstatně liší, a proto si je probereme v každém z nich zvlášť.

Pascalským prostředkem pro řízení práce překladače jsou tzv. **direktivy překladače**, kterým se často říká **dolarové poznámky**, a to proto, že mají podobu komentáře, jehož prvním znakem je „\$“ (dolar) bezprostředně následovaný identifikátorem dané direktivy.

Dolarové poznámky rozdělujeme do tří skupin, a to na **přepínače**, **direktivy s parametrem** a **direktivy pro podmíněný překlad**. Pro direktivy překladače platí následující pravidla:

- ✧ Identifikátor přepínačů je jednoznakový a musí za ním bezprostředně následovat znak + (plus) nebo - (minus) podle toho, zda chceme volbu ovládanou daným přepínačem nastavit či potlačit.
- ✧ V jedné dolarové poznámce může být několik přepínačů. Jednotlivé přepínače jsou pak navzájem odděleny čárkami a nesmí mezi nimi být žádný bílý znak.
- ✧ Mezi identifikátorem direktivy s parametrem a následujícím parametrem musí být alespoň jeden bílý znak.
- ✧ Součástí dolarové poznámky může být i komentář, který překladač ignoruje. Tento komentář píšeme za direktivy, resp. za jejich parametry, a oddělujeme jej od nich alespoň jedním bílým znakem.

V našich dosavadních programech jsme se prozatím setkali pouze s přepínačem `$I` ovlivňujícím reakci systému na chyby při vstupu a výstupu. V této kapitole si povíme ještě o řadě dalších.

Jazyk C++ zpracovává zdrojový text ve dvou fázích. Nejprve jej předzpracuje tzv. **preprocesor** a teprve preprocesorem předzpracovaný text dostane k dispozici vlastní překladač.

Činnost preprocesoru můžeme ovlivňovat speciálními příkazy (direktivami), které poznáme podle toho, že **prvým nebílým znakem** na řádce je znak # (mříž). Za tímto znakem, resp. za případnými jej následujícími bílými znaky, pak napíšeme identifikátor požadovaného příkazu následovaný případně dalšími potřebnými parametry.

Z direktiv preprocesoru jsme dosud poznali pouze příkaz **#include**. V této kapitole se seznámíme s některými dalšími.

4.2 Řízení překladače v Pascalu

Činnost překladače ovlivňujeme v Pascalu prostřednictvím přepínačů, pomocí nichž nastavujeme (+) nebo potlačujeme (-) některé aktivity. Tyto aktivity můžeme samozřejmě nastavit či potlačit i prostřednictvím nabídek IDE, avšak umístíme-li do programu přepínače, máme zaručen požadovaný režim kompilace nezávisle na původním nastavení IDE.

Přepínače dělíme na **globální**, které ovlivňují překlad celého modulu, a **lokální**, které ovlivňují překlad pouze části modulu, která začíná danou direktivou a končí dalším výskytem téže direktivy.

Globální přepínače musíme umístit před deklarační část modulu, tj. před výskyt kteréhokoliv z klíčových slov **uses**, **label**, **const**, **type**, **var**, **procedure**, **function** a **begin**. Lokální přepínače se mohou vyskytnout na kterémkoliv místě programu.

V následujícím přehledu jsou uvedeny všechny přepínače, které nám Turbo Pascal nabízí ve verzi 6.0. U každého přepínače je uvedena jak dolarová poznámka, kterou se nastavuje a potlačuje odpovídající aktivita přepínače, tak i odpovídající posloupnost voleb potřebných k řízení dané aktivity prostřednictvím stromu nabídek v IDE.

Globální přepínače

{ \$A+ }, { \$A- }

Options | Compiler | Word Align Data

Nastavením tohoto přepínače přikážeme překladači, aby všechny datové struktury delší než jeden bajt ukládal na sudé adresy. Na rozdíl od podobné direktivy v C++ však tento přepínač v Pascalu neovlivní ukládání jednotlivých složek záznamů. Aby byla daná složka záznamu zaručeně na sudé adrese, musí být tento přepínač nastaven a navíc musí být součet délek všech předchozích složek sudý. Obdobně prvky pole budou zaručeně na sudých adresách pouze v případě, že jejich délka bude sudá.

Nastavením tohoto přepínače zrychlujeme na počítačích řady AT (přesněji na počítačích s 16 nebo 32 bitovými procesory) přístup k datům. Cenou za toto zrychlení jsou přeskocené liché bajty, tedy větší spotřeba paměti.

`{SD+}`, `{SD-}`*Options | Compiler | Debug Information*

Tímto přepínačem nastavujeme a potlačujeme vkládání ladicích informací do generovaného programu. Pokud chceme program krokovat nebo pokud chceme, aby nám překladač ukázal místo, které bylo při běhu programu příčinou chyby, musíme mít tento přepínač nastaven. Cenou za toto pohodlné ladění je nárůst délky výsledného přeloženého programu – někdy i na dvojnásobek. Proto je vhodné každý modul po odladění přeložit s potlačeným generováním ladicích informací.

`{SE+}`, `{SE-}`*Options | Compiler | Emulation*

Tímto přepínačem ovlivňujeme připojení knihovny s funkcemi emulujícími činnost numerického koprocessoru. Tuto knihovnu musíme připojit v případě, že chceme, aby náš program uměl využívat služeb koprocessoru (viz přepínač `$N`), ale zároveň budeme chtít, aby byl spustitelný i na počítačích, které koprocessorem vybaveny nejsou. Pokud počítač bude vybaven koprocessorem, program jej bude používat, pokud jím vybaven nebude, program bude pro jeho funkci automaticky využívat služeb emulační knihovny. (Vzhledem k tomu, že procesory Intel 486 a novější již matematický koprocessor automaticky obsahují, ztrácí tato volba zvolna na aktuálnosti. Nicméně v překladači je, a proto si o ní musíme povědět.)

Tento přepínač má význam pouze tehdy, použijete-li jej v hlavním programu. Jeho použití v jednotkách nemá žádný efekt. Jeho použití nemá žádný efekt ani v případě, že jak hlavní program, tak jednotky přeložíme s direktivou `$N-`.

`{SL+}`, `{SL-}`*Options | Compiler | Local Symbols*

Tímto přepínačem nastavujeme a potlačujeme vkládání ladicích informací o lokálních proměnných a funkcích jednotky do výsledného programu. Tento přepínač blíže specifikuje rozsah informací, jejichž přidání či nepřidání do výsledného programu má „na starosti“ přepínač `$D`. Pokud je nastaveno `$D-`, přepínač `$L` se ignoruje.

`{SN+}`, `{SN-}`*Options | Compiler | 8087/80287*

Tímto přepínačem nastavujeme a potlačujeme práci s reálnými čísly v numerickém koprocessoru. Pokud je nastaven, můžeme v programu používat všechny reálné datové typy, avšak musíme mít na počítači k dispozici numerický koprocessor 80x87 nebo musíme k programu připojit emulační knihovnu (viz přepínač `$E`). Nevýhodou nastavení tohoto přepínače je, že na počítačích bez koprocessoru se pak operace s objekty typu *real* vykonávají více než dvakrát pomaleji než při nastavení `$N-`.

{ $\$O+$ }, { $\$O-$ }

Options | Compiler | Overlays Allowed

Tímto přepínačem nastavujeme a potlačujeme generování překryvných modulů.¹¹

{ $\$X+$ }, { $\$X-$ }

Options | Compiler | Extended syntax

Tímto přepínačem nastavujeme a potlačujeme překlad podle rozšířených syntaktických pravidel, která umožňují, abychom ignorovali hodnotu vrácenou po volání funkce. (Typickým příkladem je volání funkce *readkey* při čekání na stisk libovolné klávesy.) Použití tohoto přepínače je ovšem dvojsečné, protože je-li nastaven($\$X+$), nebude překladač pokládat za chybu, když v definici funkce nepřiradíme identifikátoru funkce žádnou vrácenou hodnotu (funkce pak vrátí nějaké smetí).

Pozor! Nastavení přepínače $\$X$ nemá vliv na interpretaci zabudovaných funkcí, tj. funkcí, které jsou k dispozici i v případě, že nedovázíte (příkazem *uses*) žádný modul.

Lokální přepínače

{ $\$B+$ }, { $\$B-$ }

Options | Compiler | Complete Boolean Eval

Tímto přepínačem nastavujeme a potlačujeme úplné vyhodnocování logických výrazů. Pokud je úplné vyhodnocování nastaveno, jsou vždy vyhodnoceny všechny členy výrazu. Je-li úplné vyhodnocování logických výrazů potlačeno, vyhodnocují se jejich členy pouze do té doby, dokud není zřejmá výsledná hodnota výrazu. Pokud je levý operand logického součtu (**or**) roven **true** (*ANO*) nebo pokud je levý operand logického součinu (**and**) roven **false** (*NE*), pravý operand se již nevyhovuje, protože je zřejmé, že nezávisle na jeho hodnotě bude hodnota celého výrazu rovna hodnotě levého operandů.¹²

¹¹ Jak víme, lze pro dosovský program využít maximálně 640 KB (ve skutečnosti je to vždy méně).

Pokud se do tohoto limitu nevejdeme, nabízí Turbo Pascal možnost rozdělit program na několik částí – překryvů (*overlay*). Za běhu programu je pak v paměti vždy jen řídicí část (hlavní program) a některé z překryvných modulů. Zavoláme-li funkci z modulu, který právě není v paměti, zavede se tento modul nejprve z disku do paměti a teprve pak se provede volaná funkce. Přitom se nepotřebné překryvné moduly odloží z paměti na disk. Podrobnější informace najdete ve firemní dokumentaci.

¹² Neúplné vyhodnocování umožňuje – kromě toho, že je rychlejší – používat i takové podmínky, které by jinak mohly způsobit chybu. Podívejme se na příklad:

```
if(x > 0) and (sqrt(x) < 3.5) then writeln(x);
```

Je-li $x < 0$, nebude se při neúplném vyhodnocování vyhodnocovat druhá část podmínky, takže se nestane, že bychom počítali odmocninu ze záporného čísla. Pokud bychom trvali na úplném vyhodnocování, museli bychom napsat

```
if(x > 0) then if (sqrt(x) < 3.5) then writeln(x);
```

`{ $F+ }, { $F- }`*Options | Compiler | Force Far Calls*

Tímto přepínačem ovlivňujeme překlad následně definovaných procedur a funkcí. Pokud je přepínač nastaven, jsou i lokální procedury a funkce (tj. ty, které **nejsou** deklarovány v části **interface**) překládány jako vzdálené, tzn. používá se při jejich volání čtyřbajtových adres. V opačném případě jsou překládány jako blízké, tj. při jejich volání se používá dvoubajtových adres (pouze ofsetové části). Vyvážené funkce (tj. funkce deklarované v části **interface**) jsou vždy překládány jako vzdálené.

Volání blízkých procedur a funkcí zabere trochu méně místa v paměti a je trochu rychlejší. Blízké procedury a funkce však není možno používat jako procedurální parametry (vzpomeňte si na „karlovskou“ proceduru *Opakuj*) ani je není možno přiřazovat procedurálním proměnným. Chceme-li danou proceduru nebo funkci používat v kterékoliv z výše zmiňovaných konstrukcí, musíme ji nechat přeložit jako vzdálenou¹³.

`{ $G+ }, { $G- }`*Options | Compiler | 286 instruction*

Tímto přepínačem nastavujeme a potlačujeme generování kódu využívajícího všechny možnosti množiny instrukcí procesoru 80286. Pokud budete mít tento přepínač při překladu nastaven, nebudou přeložené programy spustitelné na PC/XT (to už stejně dnes nikomu nevádí – zkuste najít počítač s procesorem 8086), ale na druhé straně alespoň trochu lépe využijete možností počítače. (Poznamenejme, že Turbo Pascal neumí generovat instrukce pro novější procesory, to umí až Delphi.)

`{ $I+ }, { $I- }`*Options | Compiler | I/O Checking*

Tímto přepínačem nastavujeme a potlačujeme implicitně definovanou reakci programu na chyby při vstupu a výstupu. Je-li nastaven, vedou chyby vstupu a výstupu k předčasnému ukončení programu. V opačném případě program pouze ignoruje další požadavky na vstup a výstup až do volání funkce *IOResult*, která vrátí kód chyby poslední provedené akce. Podrobnosti najdete v kapitole o práci se soubory.

`{ $R+ }, { $R- }`*Options | Compiler | Range Checking*

Tímto přepínačem nastavujeme a potlačujeme kontrolu rozsahu indexů (tj. kontrolu dodržení deklarovaných mezí polí) a kontrolu správnosti přiřazovaných hodnot při běhu programu. Pokud hodnota indexu či proměnné intervalového typu opustí vyhrazené meze, přeruší program předčasně svoji činnost a vypíše chybovou zprávu.

¹³ V Turbo Pascalu 7.0 můžeme také použít direktivu *far*, připsanou za hlavičku procedury nebo funkce v její definiční deklaraci. Např. funkce s hlavičkou

```
function f(var s: integer): integer; far;
```

se bude překládat jako vzdálená bez ohledu na nastavení přepínačů překladače.

Možnost kontroly rozsahu indexů při běhu programu je při ladění velice vítaná. Může však výrazným způsobem prodloužit a zpomalit výsledný program, a proto nezapomeňte odladěný program přeložit s nastavením `SS-`.

`{SS+}`, `{SS-}`

Options | Compiler | Stack Checking

Tímto přepínačem nastavujeme a potlačujeme kontrolu možného přetečení zásobníku při běhu programu. Pomocí nastavení této volby můžete snadno odhalit chyby, které jsou způsobeny velkým rozsahem parametrů volaných hodnotou, dlouhou posloupností rekurzivních volání apod., a které byste jinak velmi špatně odhalovali.

Pro kontrolu přetečení zásobníku platí totéž jako pro kontrolu rozsahu indexů – pomáhá, ale prodlužuje a zpomaluje program. Proto nezapomeňte odladěný program přeložit s nastavením `SS-`.

`{SV+}`, `{SV-}`

Options | Compiler | Strict Var-Strings

Tímto přepínačem nastavujeme a potlačujeme přísnou typovou kontrolu řetězcových parametrů předávaných odkazem. Pokud je tento přepínač nastaven, požaduje překladač, aby typy formálního a skutečného řetězcového parametru byly shodné. Pokud je přísná typová kontrola potlačena, smíří se překladač s tím, když formální i skutečný parametr jsou řetězce, a nekontroluje jejich délku – ekvivalent přístupu C++.

4.3 Řízení překladače v C++

Pro řízení práce překladače je v C++ vyhrazen preprocesorový příkaz `#pragma`. Tento příkaz byl zaveden do definice jazyka proto, aby mohli tvůrci každé implementace definovat svá vlastní implementačně závislá rozšíření, a přitom aby byl zdrojový text mezi jednotlivými implementacemi jazyka co nejnázorněji přenositelný. Pokud tedy překladač objeví v příkazu `#pragma` direktivu, kterou nezná, nepovažuje to za důvod k přerušení překladače a pouze vás na to zdvořile upozorní. Syntaxe příkazu `#pragma` je

```
#pragma <Jméno direktivy> [ <Parametr> ...]
```

(Hranaté závorky označují volitelnost dané položky, výpustka její opakovanost.) Podívejme se nyní na ty direktivy Borland C++, které pro nás mohou být při naší úrovni znalostí užitečné.

Speciální direktivy

```
#pragma argsused
```

Tuto direktivu můžete použít pouze mezi definicemi funkcí, tj. nikoliv uvnitř funkce. Překladač nebude vydávat žádné varovné hlášení v případě, když v definici nejbližší následující funkce nepoužijete některý z jejích parametrů.

Tuto direktivu není v C++ nutno používat. Pokud potřebujeme některý z parametrů uvést v hlavičce, ale v těle funkce jej nepotřebujeme (obecně to zavání chybou, ale existují situace, kdy to jinak nejde), stačí uvést v seznamu parametrů pouze typ daného parametru a neuvádět jeho identifikátor. Pokud je však takovýchto parametrů více za sebou, je rozumné je pojmenovat, abyste nějaký nepřidali nebo nevypustili, a pak je použití výše uvedené direktivy výhodnější.

```
#pragma startup <Jméno funkce> [<priorita>]
#pragma exit <Jméno funkce> [<priorita>]
```

S těmito direktivami jsme se již setkali. Proto si pouze připomeneme, že první z nich nařizuje překladači, aby funkci *<Jméno funkce>* vykonal po inicializaci všech statických proměnných, avšak před voláním funkce *main()*.

Druhá mu pak nařizuje, aby funkci *<Jméno funkce>* zavolal při řádném ukončení programu, tj. po opuštění funkce *main()* nebo po volání funkce *exit()*. Volitelný parametr *<priorita>* smí nabývat hodnot od 64 do 255, přičemž funkce s vyšší prioritou budou volány po funkcích s nižší prioritou. Nejvyšší priorita má číslo 0, avšak hodnoty od 0 do 63 jsou vyhrazeny pro překladač. Pokud tento parametr nevedeme, přiřadí se funkci priorita 100.

```
#pragma hrdfile "<jméno_souboru>"
#pragma hdrstop
```

Tyto dvě direktivy souvisí s možností předkompilování hlavičkových souborů, kterou zavedla firma Borland od verze 2.0 svých překladačů. Celý problém vyžaduje trochu podrobnější vysvětlení.

V praktických programech se velice často stává, že doba překladu jednotlivých vkládaných hlavičkových souborů je i několikrát delší než doba překladu vlastního programu. Proto nabízí Borland C++ od verze 2.0 možnost uložit výsledky překladu hlavičkových souborů do zvláštního souboru (volba *Options | Compiler | Code Generation | Precompiled headers*) a při příštím překladu modulu již tyto dovážené soubory nepřekládat a místo toho použít tabulky připravené v průběhu minulého překladu.

Tato možnost se ukázala jako velice užitečná (Microsoft ji do svého nového překladače honem také zařadil) v případě opakovaného překladu téhož modulu, avšak přece jenom trochu problematická při postupném překladu několika modulů, protože všechny moduly sdílely společný soubor předkompilovaných záhlaví a aby pro ně byla jeho existence opravdu přínosem, musela posloupnost jejich dovážených hlavičkových souborů začínat stejně. (Ve skutečnosti je to ještě trochu složitější.)

Direktiva **hdrstop** slouží k označení konce společné posloupnosti vkládaných souborů. Soubory, které se vkládají po zpracování této direktivy, se zpracovávají klasickým způsobem, tj. čtou se a překládají při každém překladu modulu.

Direktiva **hdrfile** se objevila v borlandských překladačích jazyka C++ až od verze 3.0. Slouží k určení souboru, do něž se budou ukládat předkompilovaná záhlaví modulů.

Stále platí, že několik modulů může sdílet společný soubor předkompilovaných záhlaví (budeme mu dále říkat *headerfile*), ale již není nutno, aby posloupnost dovážených souborů začínala stejně u všech modulů v projektu, ale stačí, když to bude platit pouze pro soubory, které sdílejí společný headerfile.

Pozor! Nemyslete si, že nejlepší co můžete udělat je přiřadit každému modulu vlastní headerfile. Při větším množství dovážených souborů totiž délka headerfilů dosahuje až několika set kilobajtů a mohli byste velice rychle vyčerpat veškerý prostor na disku.

Povolení a potlačení generování chybových hlášení

Jak víte, v Borland C++ můžeme povolit či potlačit testování kterékoliv situace vedoucí k vydání varovné zprávy.

Pokud tyto testy povolíte, „ohlídá“ vás překladač před takovými chybami, jako je práce s automatickou proměnnou, které ještě nebyla přiřazena počáteční hodnota, záměna operátorů `=` a `==` v podmínce apod. Pokud je potlačíte, nebude vás překladač s varovnými zprávami obtěžovat, avšak musíte se před takovými chybami ohlídat sami. Povolení či potlačení těchto testů můžete ovlivnit pomocí direktivy **warn**, jejíž syntax je následující:

```
#pragma warn +<idz>
#pragma warn -<idz>
#pragma warn .<idz>
```

Tato direktiva ovlivňuje vydávání varovných zpráv označených tříznakovým identifikátorem zprávy `<idz>`. Tři podoby parametru označují tři možné akce:

- + (plus) – daná varovná zpráva bude vydávána,
- (minus) – daná varovná zpráva bude potlačena,
- . (tečka) – nastaví se stav platný při začátku kompilace daného modulu.

Direktiva **warn** umožňuje na krátkou chvíli potlačit kontroly, které jsou normálně povoleny, případně povolit kontroly, které jsou potlačeny. To umožňuje programovat stylem, který často používáme: standardně máme všechny kontroly povoleny a pouze v případech, kdy se porušení některých omezení nelze vyhnout, potlačíme na chvíli příslušné varovné zprávy, avšak hned za tímto místem je zase povolíme. Volby překladače máme při tom nastaveny tak, že důvodem k zastavení překladu je i vydání varovné zprávy.

Podívejme se nyní na nejdůležitější varovné zprávy. V následujícím seznamu je u každé zprávy uvedena její tříznaková zkratka následovaná anglickým heslem, z něhož je odvozena. Na dalším řádku je pak český překlad zprávy. (Podle identifikátorů si jej můžete s pomocí manuálu sdužit s odpovídajícím anglickými vzorem.) U zpráv, jejichž text není dostatečně samovysvětlující, je stručně vysvětlen její význam. U každé zprávy pak najdete doporučení, kdy je vhodné danou varovnou zprávu potlačit.

amb – ambiguous

Pořadí vyhodnocování není potvrzeno závorkami. Protože se priority posunových, relačních a bitových operací programátorům občas pletou, požaduje překladač potvrzení předpokládaného pořadí vyhodnocování závorkami. Doporučujeme tuto zprávu **nepotlačovat** a raději doplnit označené výrazy závorkami.

amp – ampersand

Nadbytečný znak & u funkcí. Identifikátory funkcí vystupují v programu implicitně jako ukazatele, a proto je použití operátoru & zbytečné. Potlačením této varovné zprávy sice nic neztratíte a umožníte si naopak zdůraznit tímto způsobem pravou povahu oněch identifikátorů, ale je to zbytečné. My ji necháváme povolenu.

aus – assigned and not used

Identifikátoru <Ident> je přiřazena hodnota, která není použita. Tato situace by se v normálním programu neměla vyskytnout. Při ladění však může být užitečné zavést si pomocnou proměnnou, kterou přidáme do sledovacího okna a do níž budeme ukládat hodnotu nějakého výrazu, aniž bychom tuto hodnotu chtěli v programu přímo použít. V takové situaci je vhodné při ladění tuto zprávu potlačit – nejlépe v podmíněné sekci. (O podmíněných sekcích budeme hovořit za chvíli.)

bei – bad enumeration initialization

Objekt výčtového typu je inicializován typem <Typ>. Místo potlačování zprávy je většinou výhodnější přetypovat přiřazovaný výraz.

big – big

Šestnáctkový kód má více než tři cifry. Tuto zprávu určitě *nepotlačujte*. Používáním vícemístných čísel v řídicích posloupnostech nic nezískáte.

ccc – constant compare condition

Podmínky je vždy splněna nebo není nikdy splněna. Překladač objevil v programu konstantní logický výraz v podmínce. V běžných programech je to většinou příznak chyby, avšak při ladění tak někdy můžeme výhodně ovlivnit běh programu. V takové situaci je vhodné pro fázi ladění tuto zprávu potlačit – nejlépe v podmíněné sekci.

cln – constant is long

Celočíselný literál je typu **long**. Tuto zprávu určitě *nepotlačujte*! Místo toho doplňte všechny označené literály „prodlužujícím“ **L** (např. 0x1234567L).

cpt – comparison portability

Nepřenositelné porovnání ukazatelů. Tuto zprávu vydává překladač ve chvíli, kdy v programu porovnáváte hodnotu ukazatele s nějakou „neukazatelovou“ nenulovou hodnotou.

Vzhledem k tomu, že se velmi často jedná o chybu, je lepší zprávu nepotlačovat, protože její příčinu lze vždy odstranit vhodným přetypováním.

def – use before definition

Pravděpodobně užití proměnné `<Ident>` před přiřazením hodnoty. Tato zpráva upozorňuje na velice častou příčinu chyb. Nedovedeme si představit situaci, kdy by bylo její potlačení užitečné.

dsz – delete size

Velikost pole je operátorem **delete** ignorována. Jazyk C++ změnil od verze Cfront 2.1 způsob používání operátoru **delete**. (Tato verze jazyka je v borlandských překladačích implementována od verze 3.0.) Podrobnější výklad rozdílů najdete v první kapitole.

dup – duplicate

Redefinice makra není identická. Tuto varovnou zprávu vydává překladač ve chvíli, kdy je znovu definováno již dříve definované makro a tato nová definice není identická s definicí původní. Překladač pak přepíše původní definici definicí novou.

Tuto zprávu není moudré potlačovat ve chvíli, kdy máme dvě stejnojmenná makra a domníváme se, že ve chvíli definice druhého již nebudu první potřebovat. Tuto situaci je mnohem výhodnější řešit přejmenováním jednoho z konfliktních maker. Má ji smysl potlačit pouze v případech, kdy opravdu chci předefinovat makro. I tuto situaci však lze řešit bez potlačování varovné zprávy, a to tak, že nejprve zrušíme jeho definici a vzápětí ho definujeme znovu.

eas – enumeration assigning

Přiřazení hodnoty typu `<Typ>` proměnné typu `<Enum>`. Pokud to jde, je vhodnější řešit tyto konflikty přetypováním přiřazované hodnoty. Jsou však situace, kdy se přetypováním z konfliktu „nevyvléknete“, a pak je vhodné pro daný příkaz či skupinu příkazů tuto varovnou zprávu potlačit.

eff – effect

Kód nemá účinek. Tuto zprávu nepotlačujte, protože většinou indikuje chybu – nejčastěji asi zapomenutí funkčních závorek při volání funkce bez parametrů, ale může se jednat i o jiné druhy chyb. Ani u této zprávy se neumíme představit situaci, kdy by její potlačení mohlo být k čemukoli dobré.

ext – extern

`<Ident>` je deklarován jako externí i jako statický. Tuto zprávu překladač vydá, když zjistí, že daný identifikátor je někde deklarován z kontextu jako externí a jinde je deklarován s atributem **static**. Překladač jej pak bude považovat za externí. Domníváme se, že lepší, než potlačení zprávy, je sjednocení všech deklarací.

ill – ill-formed pragma

Neznámá direktiva v příkazu **#pragma**. Direktivu v příkazu **#pragma** překladač nezná. Důvody mohou být dvoji: buď jde o překlep nebo jiné nedodržení syntaktických pravidel, anebo se jedná o příkaz určený pro jiný překladač. V prvním případě je třeba chybu opravit, v druhém může být užitečné varovnou zprávu potlačit.

inl – inline

Funkce obsahující *<ident>* nebude přeložena jako vložená. Pokud chcete funkci přeložit jako vloženou, nesmíte v ní používat klíčová slova **do**, **for**, **while**, **goto**, **switch**, **break**, **continue** a **case** a některé další konstrukce – toto omezení není dané definicí jazyka, ale je specifické pro překladače firmy Borland.

Tuto zprávu nepotlačujte – jinak se nedozvíte, že vámi označenou funkci překladač neakceptoval jako vloženou.

lin – local variable for initialization

K inicializaci *<Ident>* byla použita dočasná proměnná.

lvc – local variable for call

K předání parametru *<Ident>* byla použita dočasná proměnná.

Tyto dva druhy zpráv se používají u parametrů předávaných odkazem (referencí), přičemž typ skutečného parametru neodpovídá typu formálního parametru. Pokud nepotřebujete použít označený parametr jako výstupní, můžete pro dané volání tuto zprávu potlačit. Pokud jej však potřebujete použít i jako výstupní parametr, musíte si zřídit pomocnou proměnnou odpovídajícího typu, této pomocné proměnné přiřadit předávanou hodnotu (je-li nějaká) a po vykonání vyvolané funkce si pak výstupní hodnotu v této proměnné „vzvednout“.

nod – not declared

Funkce *<Fce>* není deklarována. Má význam, překládáte-li program v jazyku C. V C++ vede použití nedeklarované funkce vždy k chybě. Doporučujeme nepotlačovat, pracujete-li v C.

par – parameter

Parametr *<Ident>* není použit. Potlačení této varovné zprávy lze pro jednotlivou funkci dosáhnout i direktivou **#pragma argsused**.

pia – possibly incorrect assignment

Pravděpodobně nechtěné přiřazení. Záměna operátorů `=` a `==` v podmínce je velice častou chybou. Pokud je kontrola *pia* nastavena, upozorní vás překladač na všechny podmínky, v nichž je poslední prováděnou operací přiřazení (`=`), protože se domnívá, že jste asi chtěli použít operátor porovnání (`==`). Pokud jste se nespolehlí a opravdu jste chtěli v rámci tes-

tování podmínky přiřadit hodnotu, nepotlačujte tuto kontrolu, místo toho raději testujte nenulovost přiřazovaného výrazu, tj. místo

```
if( a = x/2 )...
```

použijte

```
if( (a = x/2) != 0 ) ...
```

Nezapomeňte však na závorky, protože operátor přiřazení má menší prioritu než operátor nerovnosti.

pro – prototype

Volání funkce, která nemá prototyp. Má význam, překládáte-li program v jazyku C. V C++ vede použití nedeklarované funkce vždy k chybě. Doporučujeme nepotlačovat, pracujete-li v C.

ret – return

Ve funkci je použit jak příkaz **return** bez hodnoty, tak **return** s předáním hodnoty. Má význam, překládáte-li program v jazyku C. V C++ vyvolá tato situace vždy chybu. Doporučujeme nepotlačovat, pracujete-li v C.

rch – unreachable code

Nedosažitelný kód. Překladač objevil část programu, které nikdy nemůže být předáno řízení. V běžných programech je to většinou příznakem chyby, avšak při ladění někdy můžeme chtít záměrně blokovat části programu – ať již původního či některých ladicích dodatků. V takové situaci může být vhodné pro fázi ladění tuto zprávu potlačit – nejlépe v podmíněné sekci.

rng – range

Konstanta překročila povolený rozsah. Toto varování obdržíme, pokusíme-li se např. porovnávat neznaménkovou proměnnou s hodnotou -1. To znamená prakticky vždy chybu, takže doporučujeme upravit program do podoby, v níž již překladač toto varování nevydává, nepotlačovat.

rpt – conversion portability

Nepřenositelná konverze ukazatelů. Tuto zprávu překladač vydá, použijete-li v programu hodnotu ukazatele na místě, kde je očekáváno celé číslo, nebo naopak. Vzhledem k tomu, že se velmi často jedná o chybu, je lepší zprávu nepotlačovat, protože její příčinu lze vždy odstranit vhodným přetypováním.

rvt – return value

Funkce musí vracet hodnotu. Funkce s deklarovaným typem (jiným než **void**) musí vracet hodnotu. Kvůli kompatibilitě s jazykem C (zejména se staršími verzemi), kde se místo

funkcí typu **void** používaly funkce typu **int**, se toleruje, jestliže funkce hodnotu nevrátí. Většinou to ale znamená chybu, takže doporučujeme nepotlačovat.

sig – significant digit

Konverze může vést ke ztrátě významných číslic. Před tímto varováním se někdy špatně „utíká“ a jedinou možností je potlačit ho pro kritickou část programu. Měli bychom si ale být jisti, že ke ztrátě významných číslic nemůže ve skutečnosti dojít nebo že nám nebude vadit.

stv – structure passed by value

Struktura je předávána hodnotou. C++ umožňuje předávat strukturové parametry hodnotou. Ne vždy je to však naším záměrem – možná jsme pouze zapomněli uvést v deklaraci příznak ukazatele (*) nebo příznak předávání odkazem (&). Proto nás překladač pro jistotu na všechny definice funkcí se strukturovými parametry předávanými hodnotou upozorní. Pokud je předávání parametru hodnotou naším úmyslem a přitom chceme mít překlad bez rušivých varovných zpráv, nezbývá, než toto varovné hlášení pro danou funkci potlačit.

sus – suspicious pointer conversion

Podezřelá konverze ukazatelů. Toto varování překladač vydává v situaci, kdy navzájem přiřazujete hodnoty ukazatelů, které ukazují na různé typy dat. Pokud jste přesvědčeni o správnosti přiřazení, nepotlačujte varovnou zprávu, ale „zbavte“ se jí vhodným přetyčováním.

ucp – unsigned character pointers

Současné použití ukazatelů na znaménkové a neznaménkové znaky. Tato situace většinou k chybě nevede. Pokud byste měli přetyčovat častěji, bude možná výhodnější generování této varovné zprávy potlačit.

use – not used

<Ident> je deklarován, není však použit. Tato zpráva může mít několik příčin:

- ✧ v důsledku úprav v programu jsme daný objekt přestali potřebovat – pak je nejlepší deklaraci odstranit,
- ✧ deklarovaný objekt je použit v části programu, která se tentokrát nepřekládá – pak bude nejlepší umístit do sekce podmiňujících překlad i deklaraci nebo direktivu potlačující generování této zprávy,
- ✧ jedná se o pomocnou proměnnou, kterou jsme zavedli pro účely ladění – naše reakce pak bude odpovídat našim dalším záměrům s touto proměnnou,
- ✧ důvod je jiný – pak vám neporadíme, reakce závisí na konkrétní situaci.

zdi – zero division

Dělení nulou. Tuto varovnou zprávu překladač vydává v situaci, kdy najde ve výrazu operaci dělení (/) nebo operaci modulo (získání zbytku po dělení, %), která má v děliteli konstantu rovnou nule. Většinou je to chyba, ale někdy to bývá záměr autora, který chce vyvolat evidentní chybu ve větvi, do níž se program nemá nikdy dostat. Existují ale lepší způsoby řešení tohoto problému – např. použití makra *assert*.¹⁴ Proto uvedené varování nepotlačujte a poohlédněte se raději po korektnějších způsobech řešení problému.

Vlastní modifikace práce překladače

Pomocí příkazu **#pragma** můžete ovlivnit i další parametry překladu a podobu výsledného kódu. K tomu nám slouží direktiva **option**, jejíž syntax je:

```
#pragma option <Volba> ...
```

Volby, které jsou parametry této direktivy, začínají vždy znakem - (minus) následovaným identifikátorem volby.¹⁵ Podle jejich syntaxe je můžeme rozdělit do dvou skupin: na přepínače a na volby s parametrem.

Za identifikátorem přepínače může následovat bílý znak, minus nebo tečka. Následuje-li za ním bílý znak, přepínač danou volbu nastavuje, následuje-li minus, přepínač danou volbu potlačuje, a následuje-li tečka, vrací nastavení volby do stavu nastaveného při zahájení překladu – přesněji do stavu nastaveného volbami v IDE.

U voleb s parametrem musíme dát pozor, abychom mezi identifikátor volby a následující parametr nevložili žádný bílý znak.

Volby nastavitelné direktivou **option** dělíme ještě podle jiného hlediska – podle místa povoleného výskytu, a to na:

1. volby, které se mohou vyskytovat pouze na začátku zdrojového textu modulu,
2. volby, které se mohou vyskytovat pouze mezi definicemi funkcí a dalších globálních objektů modulu,
3. volby, které se mohou vyskytovat kdekoliv.

Podívejme se nyní na některé nejdůležitější volby:

Volby přípustné pouze na počátku modulu

```
-f f- -f87 -f287
```

Touto volbou řídíme práci s reálnými čísly. Implicitně je tato volba nastavena, což znamená, že na počítačích s koprocesorem bude program koprocesor využívat, kdežto na počítačích bez koprocesoru se budou operace koprocesoru emulovat.

¹⁴ O makru *assert* mluvíme podrobněji v knize *Objektové programování 1*.

¹⁵ V direktivě **#pragma option** uvádíme volby z příkazového řádku samostatného překladače.

Při potlačení volby nepředpokládáme připojení knihovny pro práci s reálnými čísly (výsledný program je pak o nějaké to kilo kratší), takže pak v programu smíme používat pouze operace s celými čísly. Potlačení volby můžeme chápat i jako žádost překladači, aby nás upozornil na případné „zatoulané“ reálné operace.

Nastavením volby **-f87** nebo **-f287** žádáme překladač, aby nevyužíval možnosti emulace a operace s reálnou aritmetikou generoval rovnou pro koprocesor. Programy přeložené při tomto nastavení nebudou spustitelné na počítačích bez koprocesoru.

-n<Cesta>

Touto volbou můžeme nastavit adresář, do nějž se bude ukládat přeložený modul.

o<Jméno souboru>

Touto volbou můžeme ovlivnit přímo jméno přeloženého souboru. Jméno, ne však příponu. Ta bude vždy OBJ, takže přeložený soubor bude mít jméno *<Jméno souboru>.OBJ*.

-1- -1 -2 -3 -4 -5

Těmito volbami ovlivňujeme instrukční soubor použitý překladačem. Volbou **-1-** nastavujeme instrukční soubor procesorů 8086 a 8088 (program pak bude spustitelný i na PC XT), volbou **-1** nastavujeme instrukční soubor procesorů 80186 a 80188 a volbou **-2** instrukční soubor procesoru 80286 (program pak bude spustitelný pouze na PC AT). Další možnosti jsou použitelné jen v pozdějších verzích překladačů (4.5) a příkazují generovat kód pro procesory 80386, 80486 a Pentium (poslední pouze u 32bitových překladačů).

Volby přípustné pouze mezi definicemi globálních objektů

-a -a-

Nastavením tohoto přepínače přikážeme překladači, aby všechny datové struktury delší než jeden bajt ukládal na sudé adresy. Na rozdíl od podobné direktivy v Pascalu ovlivní tento přepínač v C++ i ukládání jednotlivých složek záznamů. Nastavením tohoto přepínače zrychlujeme na počítačích řady AT (přesněji na počítačích s 16 nebo 32 bitovými procesory) přístup k datům. Cenou za toto zrychlení jsou přeskočené liché bajty.

-G -G-

Tento přepínač ovlivňuje celkovou strategii generování kódu. Je-li nastaven, dáváme přednost kratšímu přeloženému kódu, je-li potlačen, dáváme přednost rychlejšímu přeloženému kódu.

-N

Tímto přepínačem nastavujeme a potlačujeme kontrolu možného přetečení zásobníku při běhu programu. Nastavením této volby můžete snadno odhalit chyby, které jsou způsobeny velkým rozsahem parametrů volaných hodnotou, dlouhou posloupností rekurzivních volání apod., které bychom jinak velmi špatně odhalovali.

Možnost kontroly přetečení zásobníku při běhu programu je velice vítaná při ladění. Může však výrazným způsobem prodloužit a zpomalit výsledný program, a proto nezapomeňte odladěný program přeložit bez průběžné kontroly přetečení zásobníku.

-O1 -O2 -Od

Tuto volbu najdete až od verze 3.0 výše. Nabízí rozsáhlé možnosti ovlivnění optimalizace kódu, které jsou podrobně popsány v manuálu. Zde si povíme pouze o třech z nich: Volbou **-O1** žádáme překladač o nejkratší možný kód, volbou **-O2** jej žádáme o nejrychlejší možný kód a volbou **-Od** zakazujeme veškeré optimalizace.

Nepleťte si volbu **-O1** s volbou **-G** a volbu **-O2** s volbou **-G-**. Nastavením přepínače **-G** říkáme překladači, kterou ze dvou možností překladu některých konstrukcí má zvolit – zda tu rychlejší či tu kratší. Naproti tomu volbami **-O** jej žádáme o dodatečnou analýzu programu a z ní vycházející úpravy výsledného kódu, které by mohly vést k jeho zkrácení či zrychlení. Nebezpečí těchto optimalizací však tkví v tom, že ve chvíli, kdy v programu „divočíme“ s ukazateli nebo provádíme některé ne zcela korektní operace, může optimalizovaný program přestat fungovat.

-r -r- -rd

Touto volbou ovlivňujeme práci s registrovými proměnnými. Volbou **-r** žádáme překladač, aby použil registrové proměnné všude, kde to považuje za vhodné. Volbou **-r-** jej napopak žádáme, aby registrové proměnné nepoužíval, a to ani v případech, kdy jsou proměnné deklarovány s modifikátorem **register**. Volbou **-rd** volíme zlatou střední cestu – překladač použije registrové proměnné pouze v případě, kdy jej o to modifikátorem **register** explicitně požádáme.

-v -v- -vi -vi-

Těmito přepínači ovlivňujeme možnosti ladění. Přepínačem **-v** nastavujeme, resp. potlačujeme vkládání ladicích informací do generovaného programu a přepínačem **-vi** nastavujeme, resp. potlačujeme rozvoj vložených funkcí (je-li rozvoj vložených funkcí potlačen, překládají se i vložené funkce, tj. funkce označené modifikátorem **inline**, jako „normální“).

Počítejte s tím, že při nastavení vkládání ladicích informací se zároveň potlačí rozvoj vložených funkcí, protože vložené funkce nejdou krokovat. Pokud jste ochotni možnost jejich krokování oželeť a chcete, aby se vložené funkce přeložily opravdu jako vložené, musíte použít přepínače **-vi**.

-Z -Z-

Nastavením této volby žádáme procesor, aby si v průběhu překladu pamatoval obsah jednotlivých registrů (tj. hodnotu konstanty či proměnné, kterou do registru načítal) a v případě, že by potřeboval do registru načíst obsah konstanty či proměnné, kterou již v registru má, aby od nového načítání upustil.

Ačkoliv to tak na první pohled nevypadá, může nastavení této volby vést někdy ke špatné funkci programu.

Volby přípustné kdekoliv mezi příkazy

-b -b-

Nastavením tohoto přepínače žádáme překladač, aby pro hodnoty výčtových typů vyhradil v paměti vždy celé slovo (2 bajty), jejím potlačením jej naopak žádáme, aby pro ně vyhradil pouze jeden bajt (je-li to vzhledem k rozsahu hodnot možné).

-d -d-

Nastavením této volby žádáme překladač, aby stejné řetězce (řetězce, jejichž shodnost může posoudit již v době překladu) ukládal v paměti pouze jednou. Tím sice zkrátíme program, avšak na druhou stranu si znemožníme s těmito řetězci pracovat, protože změnou jednoho z nich bychom změnili všechny řetězce na tomto místě uložené.

-K -K-

Nastavením tohoto přepínače žádáme překladač, aby typ **char** považoval za **unsigned char**, jeho potlačením jej žádáme, aby typ **char** považoval za **signed char**. Od verze 4.0 tím předepisujeme pouze implementaci typu **char**, tzn. překladač bude považovat typ **char** i nadále za něco jiného než typy **signed char** a **unsigned char**. Chceme-li i pozdější překladače donutit, aby považovaly typ **char** za jeden ze zbývajících dvou, tedy aby rozlišovaly pouze dva znakové typy, musíme použít přepínač **K2**.

4.4 Makra

Makra jsou specialitou jazyka C++ (zděděnou z jazyka C), která nemá v Pascalu obdobu. Proto mohou uživatelé Pascalu, kteří se o jazyk C++ nezajímají, tuto a následující podkapitulu přeskočit.

Jednoduchá makra

Makro je jazyková konstrukce, kterou převzaly některé vyšší jazyky od assembleru. Makra mají hodně společného s funkcemi; mají svá záhlaví obsahující identifikátor makra a případné parametry a těla obsahující vlastní výkonný text. Od funkcí se liší především tím, že jejich tělo se na každé místo použití celé „opíše“, zatímco funkce se volají.

Makra definujeme preprocesorovým příkazem **#define**. Jeho syntax je následující:

```
#define <Záhlaví> <Tělo makra>
```

V tomto oddílu si probereme makra bez parametrů, jejichž záhlaví je tvořeno pouze identifikátorem makra, pro nějž platí stejná pravidla jako pro identifikátor jakéhokoliv jiného objektu jazyka.

Záhlaví je od těla makra odděleno nejméně jednou mezerou nebo tabulátorem, případně kombinací obou. Tělo makra je ukončeno koncem řádku. Pokud by se nám text těla makra nevešel na jeden řádek nebo pokud bychom jej chtěli z důvodů větší přehlednosti rozepsat na více řádků, museli bychom zapsat na konci každého neukončeného řádku znak \ (zpětné lomítko), který se v případě, že je posledním znakem na řádku, do vlastního textu makra nezapočítává.

Jak víme, zdrojový text prochází před vlastním překladem preprocesor. Kdykoliv při procházení zdrojovým textem narazí na identifikátor makra, nahradí ho odpovídajícím textem – tělem makra. Toto nahrazení nazýváme **rozvinutí (expanze) makra**. Preprocesor však rozvine (expanduje) pouze ty identifikátory, které se vyskytují **mimo komentáře a textové řetězce**, protože komentáře preprocesor nepročítá, ale rovnou je ze zdrojového textu vypouští, a textové řetězce pouze otrocky kopíruje.

Makra můžeme i vnořovat do sebe, protože preprocesor rozvinuté tělo makra znovu prochází a objeví-li v něm identifikátor nějakého dalšího makra, rozvine jej. Pokud však narazí na identifikátor právě rozvinutého makra, přejde jej bez povšimnutí – rekurzivní definice maker tedy možné nejsou.

Po ukončení všech expanzí preprocesor výsledný text již nijak nezpracovává. Tělem makra proto nemohou být příkazy preprocesoru, protože by zůstaly ve výsledném textu a překladač by u nich ohlásil chybu.

Tělo makra může být i prázdné. V takovém případě se tělo makra rozvine v prázdný řetězec. Makra s prázdným tělem se používají ze dvou důvodů:

- ✧ Když chceme, aby se v programu nějaký text jednou vyskytoval a jindy ne. Pak definujeme makro, jehož tělem bude v prvním případě tento text a v druhém případě bude tělo prázdné. Podle situace pak necháme aktivovat tu definici, kterou zrovna potřebujeme, a nemusíme pokaždé procházet celý zdrojový text a ve všech potřebných místech tento text vypouštět či vkládat.
- ✧ Když potřebujeme pouze zavést nový identifikátor, o němž víme, že jej nebudeme nikdy chtít rozvinout. V podkapitole o podmíněném překladu uvidíte, k čemu to může být dobré.

Preprocesor zpracovává vstupní text sekvenčně. Každé makro je definováno (a můžeme je používat) od místa, kde je jeho definice zapsána, až do konce zdrojového textu překládaného modulu. Makro můžeme definovat i vícekrát. Pokud však jeho nová definice není přesnou kopií definice původní, vydá překladač varovnou zprávu (samozřejmě není-li potlačena) a pracuje od této chvíle s definicí novou.

Této varovné zprávě se lze vyhnout tím, že makro „oddefinujeme“ (zrušíme jeho definici) a poté ho definujeme znovu. K oddefinování maker slouží preprocesorový příkaz **#undef**, jehož syntax je

#undef <Identifikátor>

Tento příkaz můžeme však použít i tehdy, když makro nemíníme znovu definovat, ale pouze potřebujeme, aby nebylo nadále definováno. O situacích, kdy potřebujeme makro oddefinovat, se opět dozvíte více v podkapitole o podmíněném překladu.

V následující ukázce najdete několik příkladů definice maker a jejich použití.

```

/* Příklad C4 - 1 */
#include <iostream.h>
#define ef else if
#define FFs "\xFF"
const char *cFFs = "\xFF";
/*
Výhoda definice makra ve srovnání s definicí konstanty spočívá v
tom,
že takto definované makro můžete vložit do řetězce, např.
"Další " FFs "položka"
který preprocesor rozvine v řetězec
"Další " "\xFF" "položka"
Na druhé straně vložení konstanty by zde způsobilo chybu.
*/
#define FOR for( i = 0; \
    Polozka[i]; \
    i++ )
long /*****/ Zaklicuj /*****/
( char *Polozka, int Klic )
{
    int i;
    if( Klic > 1 )
    {
        long p = 0;
        FOR
        p += Polozka[ i ];
        return p;
    }
    ef( Klic > 0 )
    {
        long p = 1;
        FOR
        p *= Polozka[ i ];
        return p;
    }
    else
    {
        double p = 0;
        FOR
        p -= Klic / Polozka[ i ];
        return int( p );
    }
}

```

```
/***** Zaključ *****/
```

```
#undef FOR
```

Makra s parametry

Stejně jako u procedur a funkcí, i u maker začíná opravdové „makroprogramování“ až s příchodem parametrů. Při definici makra s parametrem nenásleduje za identifikátorem makra bílý znak, ale otevírací kulatá závorka. Za ní pak patří seznam formálních parametrů, oddělených čárkami, a ukončený uzavírací kulatou závorkou. Teprve pak můžeme definovat vlastní tělo makra. Formálně bychom mohli jeho syntax popsat takto:

```
#define <Ident>( <seznam parametrů> ) <Tělo makra>
```

Volání maker s parametry je pak syntakticky shodné s voláním funkcí, včetně toho, že mezi identifikátorem makra a otevírací závorkou následovanou seznamem skutečných parametrů mohou být i bílé znaky (to už preprocesor ví, že se jedná o makro s parametry, takže ho bílé znaky nespletou).

Makra s parametry se rozvíjejí ve dvou fázích: nejprve se makro rozvine tak, jako by nemělo parametry, a pak preprocesor v tomto rozvinutém těle nahradí formální parametry skutečnými. Z toho vyplývají i některá specifika používání parametrů maker.

Protože jsou parametry makra chápány jako texty, které se v daném místě dosadí do těla makra, neprovádí se žádná předběžná vyhodnocení. Je-li tedy parametrem makra výraz, bude se tento výraz vyhodnocovat tolikrát, kolikrát se v těle makra objeví. Typickým příkladem několika chyb, kterých se můžeme dopustit, je definice

```
#define Delka( x, y ) sqrt( x*x + y*y )
```

následovaná použitím

```
D = Delka( a+=3, b+=4, a+b), c + d );
```

Po rozvinutí makra totiž obdržíme výraz

```
D = sqrt( (a+=3, b+=4, a+b)*(a+=3, b+=4, a+b) + c + d*c + d );
```

Předpokládáme-li, že proměnné a , b , c , d mají po řadě hodnoty 1, 2, 3, 4, nebude proměnné D přiřazena hodnota

```
Delka( 10, 7 ) = sqrt( 149 )
```

a proměnné a a b nebudou mít hodnotu 4, jak by tomu bylo v případě, kdyby bylo *Delka* definováno jako funkce; proměnná D bude mít hodnotu

```
sqrt( (4+6)(7+10) + 3 + 4*3 + 4 ) = sqrt( 189 )
```

Abychom se takovýmto chybám vyhnuli, musíme v definici makra využít závorek,

```
#define Delka( x, y ) sqrt( (x)*(x) + (y)*(y) )
```

a nesmíme ve skutečných parametrech maker používat výrazy s vedlejšími efekty, jako byly

```
a += 3, b += 4;
```

V případě, že bychom chtěli program ještě dále zefektivnit, odstraníme z parametrů i výrazy, které se jinak budou vyhodnocovat tolikrát, kolikrát se daný parametr v těle makra objeví. Místo

```
D = Delka( (a+=3, b+=4, a+b), c + d );
```

bychom tedy měli napsat

```
a += 3;
b += 4;
double d1 = a+b;
double d2 = c+d;
D = Delka( d1, d2 );
```

(Odstranění opakovaného vyhodnocení by však měl dobře optimalizující překladač udělat za nás.)

Preprocesor nabízí dvě operace s parametry maker: převedení na řetězec (operátor #) a spojení (operátor ##).

– Při převedení parametru na řetězec se prostě text, který je hodnotou daného parametru, vloží mezi uvozovky. Pokud jsou součástí textu uvozovky, vloží před ně preprocesor zpětné lomítko. S žádným jiným znakem to však neudělá, ani se zpětným lomítkem. Pokud se tedy v parametru vyskytuje zpětné lomítko, můžete se dožít zajímavých překvapeň.

Tato operace se nejčastěji používá při definici maker pro kontrolní tisky, např.

```
#define kt( a ) cout << #a << " = " << (a) << endl;
```

– Pomocí tohoto operátoru můžete sestavovat identifikátory zpracovávaných objektů z několika oddělených částí – např. při definicích

```
#define s( a, b ) (a ## b)
#define a 1
#define x X
#define y
```

se příkaz

```
s(i,a) = s(i,x) + s(i,y);
```

rozvine do tvaru

```
i1 = iX + i;
```

Pokud sestavením hodnot parametrů vznikne identifikátor definovaného makra, preprocesor neopomene toto makro rozvinout.

Následující příklad ukazuje některé z možností maker s parametry.

```

/* Příklad C4 - 2 */
#include <iostream.h>

#define abs( a ) ( (a) > 0 ? (a) : (-a) )
#define Na3( b ) ((b)*(b)*(b))
#define KT( x ) cerr << #x << " = " << x << endl;

double /*****/ fce /*****/
( int i, double d )
{
    KT( i ); KT( d ); //Kontrolní tisky parametrů
    return( Na3( abs( d*d / i ) ) );
    //Velmi neefektivní zápis - preprocesor jej převede do tvaru
    // ((d*d/i)>0?(d*d/i):-(d*d/i)) *
    // ((d*d/i)>0?(d*d/i):-(d*d/i)) *
    // ((d*d/i)>0?(d*d/i):-(d*d/i))
    //
    //Vhodnější je zřídit pomocnou proměnnou a upravit program do tvaru
    // double pom = d*d/i;
    // pom = abs( pom );
    // return( Na3( Pom ) );
}

int main(){
    cout << fce(3, 2.1);
    return 0;
}

```

4.5 Podmíněný překlad

V praxi se velmi často stává, že vytváříme několik nepatrně odlišných modifikací téhož programu. V takovém případě bývá často výhodné mít možnost označit, které části zdrojového textu se při překladu dané modifikace programu mají použít a které nikoliv. Oba dva jazyky k tomu poskytují prostředky, které jsou obdobou podmíněného příkazu **if – then – else**.

Podmíněný překlad v Pascalu

Pascal k tomuto účelu používá (jak jinak) dolarových poznámek, které nám ohraničují jednotlivé sekce programu. Poznamenejme ještě, že sekce podmíněného překladu mohou být do sebe zanořovány stejně jako klasické příkazy **if – then – else**.

Podívejme se nejprve na význam jednotlivých direktiv:

```
{DEFINE <Jméno> <Případný komentář>}
```

Definuje nový symbol <Jméno>, který je možno použít v direktivách pro řízení podmíněného překladu. <Jméno> je známo (definováno) do té doby, než je zrušíme direktivou **{UNDEF}**. Bylo-li <Jméno> již dříve definováno, nemá direktiva žádný účinek.

{SUNDEF <Jméno> <Případný komentář>}

Pro zbytek překladu je <Jméno> zapomenuto – leda bychom je znovu definovali poznámkou `{$DEFINE}`. Pokud <Jméno> není definováno, nemá tato direktiva žádný účinek.

{SIFDEF <Jméno> <Případný komentář>}

Uvádí sekci podmíněného překladu. Část programu mezi touto direktivou a sdruženou direktivou `{$ELSE}` nebo `{$ENDIF}` bude překládána právě tehdy, je-li <Jméno> definováno.

{SIFNDEF <Jméno> <Případný komentář>}

Uvádí sekci podmíněného překladu. Část programu mezi touto direktivou a sdruženou direktivou `{$ELSE}` nebo `{$ENDIF}` bude překládána právě tehdy, není-li <Jméno> definováno.

{SIFOPT <Volba> <Případný komentář>}

Uvádí sekci podmíněného překladu, přičemž parametr <Volba> obsahuje jednoznačný identifikátor volby následovaný znakem + nebo -. Část programu mezi touto direktivou a sdruženou direktivou `{$ELSE}` nebo `{$ENDIF}` bude překládána právě tehdy, je-li daná direktiva nastavena do daného stavu.

{\$ELSE <Případný komentář>}

Odděluje sekce podmíněného překladu, které se budou překládat při splnění (část před `$ELSE`) či nesplnění (část za `$ELSE`) podmínky v příslušné direktivě `$IF`.

{\$ENDIF <Případný komentář>}

Ukončuje sekci podmíněného překladu.

Ukážeme si použití podmíněného překladu na příkladu modulu, který při ladění definujeme jako hlavní program a po odladění jako běžný modul.

```
(* Příklad P4 - 1 *)
{$DEFINE LADIM - pokud neladíme, vsuneme před dolar mezeru}

{$IFDEF LADIM - během ladění překládáme modul jako hlavní}
Program Ukazka;
{$ELSE - NELADÍM => modul se překládá jako řadový}
Unit Ukazka;
interface
{$ENDIF}

procedure Proc( S:String ); forward;
function Fce( S:String ):integer; forward;

{$IFNDEF LADIM} implementation {$ENDIF}

function (*****) Fce (*****)
```

```

( S:String ):integer;
var ret:integer;
begin
  write( 'Kolik opakování řetězce "', s, '" : ' );
  read( ret );
  Fce := ret;
end;

procedure (*****) Proc (*****)
( S:String );
var i:integer;
begin
  for i:=1 to Fce(s) do
    writeln( i, ' : ', s );
  end;
begin
{$IFDEF LADIM}
{Sem by přišla inicializace, kdyby nějaká byla}
{$ELSE - NELADÍM - do této sekce dám testovací program}

  Proc( 'První' );
  Proc( 'Druhy' );
{$ENDIF}
end.

```

Podmíněný překlad v C++

V C++ ohraničují jednotlivé části programu, které se mají a nemají překládat, opět příkazy preprocesoru. Tyto podmíněně překládané sekce mohou být do sebe vnořovány stejně, jako klasické příkazy **if – else**. Podívejme se na jednotlivé příkazy a operátory, které přitom budeme používat.

defined(<Jméno>)

Operátor preprocesoru, vracející 1 v případě, že <Jméno> je již definováno, a 0 v případě, že definováno není.

#if <Podmínka>

Uvádí sekci podmíněného překladu, která bude překládána právě tehdy, je-li <Podmínka> splněna. Podmínkou může být jakýkoliv celočíselný výraz, který je možno vyhodnotit v době zpracování textu preprocesorem. Např. skutečnost, že budou definovány alespoň dva z identifikátorů **A1**, **A2** a **A3**, můžeme testovat příkazem

```
#if defined(A1)+defined(A2)+defined(A3) >= 2
```

#else

Odděluje sekce podmíněného překladu, které se budou překládat při splnění (část před **#else**) či nesplnění (část za **#else**) podmínky v příslušném příkazu **#if**.

#endif

Označuje konec sekce podmíněného překladu.

```
#ifndef <Jméno>
```

Zkrácená verze příkazu **#if defined**(<Jméno>).

```
#ifndef <Jméno>
```

Zkrácená verze příkazu **#if !defined**(<Jméno>).

```
#elif <Podmínka>
```

Tento příkaz je zkrácenou verzí posloupnosti příkazů

```
#else  
# if <Podmínka>
```

Výhodou použití příkazu **#elif** – kromě zkrácení zápisu – také je, že odpadne jedno závěrečné **#endif**.

```
/*   Příklad C4 - 3   */  
#include <iostream.h>  
  
#define LADIM           //Pokud neladím, udělám z definice komentář  
#ifndef LADIM  
    #include <fstream.h>  
    static int KI=0;  
    static ofstream kontr( "KONTROLA." );  
    #define PRB( P ) { kontr << endl; \  
                      for( int i=KI++; i-- > 0; \  
                      kontr << "+++++" ); \  
                      kontr << " " << #P << endl; \  
                    }  
    #define PRE( P ) { for( int i=--KI; i-- > 0; \  
                      kontr << "-----" ); \  
                      kontr << " " << #P << endl; \  
                    }  
    #define KT( x ) kontr << #x << "= " << x << endl;  
#else  
    #define PRB( P );  
    #define PRE( P );  
    #define KT( x )  
#endif  
  
//Následující prototypy by v normálním programu byly  
//v hlavičkovém souboru - zde jsou pro zjednodušení  
void p( char *S );  
int f( char *S );  
  
void /*****/ p /*****/  
( char *s )  
{  
    PRB( p );    KT( s );  
    int Max=f(s);  
    for( int i=1; i <= Max; i++ ) cout << i << ": " << s ;
```

```

    cout << endl;
    PRE( p );
}
/***** p *****/
int /*****/ f /*****/
( char *s )
{
    PRB( f ); KT( s );
    cout << "Kolik opakování řetězce \" << s << "\": ";
    int ret;
    cin >> ret;
    KT( ret ); PRE( f );
    return ret;
}
/*****/ f *****/
#ifdef LADIM //Do této sekce dám testovací program
void /*****/ main /*****/ ( )
{
    PRB( main );
    p( "Prvni" );
    p( "Druhy" );
    PRE( main );
}
/*****/ main *****/
#endif

```

4.6 Vkládání souborů

Jako vkládání souborů označujeme operaci, při níž v jednom zdrojovém souboru (řekněme mu *S1*) označíme místo, po jehož dosažení v průběhu zpracovávání zdrojového textu se pokračuje zpracováním textu z jiného souboru (dejme tomu *S2*). Po vyčerpání souboru *S2* se pokračuje ve zpracovávání textu původního souboru, tj. souboru *S1*, od označeného místa. Výsledný efekt je tedy stejný, jako kdybychom do souboru *S1* opravdu vložili na označeném místě celý soubor *S2*.

Vkládání souborů v Pascalu

V Pascalu nám pro vkládání souborů slouží direktiva s parametrem, jejíž syntax je:

```
{SI <JménoSouboru> <Případný komentář>}
```

Připomínáme, že mezi znakem *I* a jménem souboru musí být alespoň jedna mezera (nebo jiný bílý znak), stejně jako mezi jménem souboru a případným dalším komentářem.

I když byla možnost vkládání souboru zavedena do Turbo Pascalu v jeho pradávných verzích, které jinou možnost rozdělení zdrojového textu do více souborů neposkytovaly,

(neobsahovaly možnost deklarovat jednotku – **unit**), i dnes se občas najdou situace, kdy je tato dolarová poznámka užitečná.

Vkládání souborů v C++

V C++ používáme vkládání souborů pomocí příkazu **#include** prakticky již od knihy *Základy algoritmizace*. Slouží nám k dovozu hlavičkových souborů, v nichž jsou definována jednotlivá mezimodulová rozhraní. Parametrem příkazu **#include** je název vkládaného souboru. Tímto názvem musí být jméno existujícího souboru, přičemž jeho součástí může být jak označení logického disku, tak cesta, přičemž v cestě můžeme používat i symbol `..` (dvě tečky) označující nadřazený adresář. Příkaz akceptuje tři podoby svého parametru:

- ✧ v úhlových závorkách,
- ✧ v uvozovkách,
- ✧ makro, jehož tělem je název souboru v uvozovkách nebo úhlových závorkách.

Třetí případ je jediným případem, kdy preprocesor expanduje makro v příkazu **#include**. O tom, že neexpanduje makra v textových řetězcích (případ 2) již víme. Pamatujte si však, že preprocesor nerozvine ani případná makra ve složených závorkách direktivy **#include**.

Pokud je součástí názvu vkládaného souboru cesta, hledá preprocesor vkládaný soubor pouze v adresáři určeném touto cestou. Jinak bude záležet na „uzávorkování“ parametru.

Je-li název souboru v úhlových závorkách, považuje jej preprocesor za název standardního hlavičkového souboru a hledá jej postupně ve všech adresářích, které jste zadali ve vstupním okně *Options | Directories | Include Directories* (jednotlivé cesty se oddělují středníky). Pokud zde zadaný soubor nenajde, vydá chybové hlášení.

Je-li název souboru v uvozovkách, považuje ho preprocesor za název souboru definovaného uživatelem a hledá ho nejprve v aktuálním adresáři a v případě, že jej zde nenajde, začne jej hledat jako systémový, tj. v adresářích zadaných ve vstupním okně *Options | Directories | Include Directories*.

Pozor! Název vkládaného souboru v uvozovkách není zpracováván jako řetězec. Uvozovky jsou zde chápány pouze jako zvláštní druh závorek. Z toho vyplývá, že v případě, kdy součástí tohoto názvu je i cesta, nebudeme zdvojit obrácená lomítka.

Doposud jsme stále předpokládali, že pomocí direktivy **#include** vkládáme hlavičkový soubor definující mezimodulové rozhraní. Samozřejmě, že je možné, abychom pomocí příkazu **#include** vkládali i normální zdrojové soubory, avšak v praxi se to téměř nepoužívá.

5. Podrobnosti o ladění programů

Při ladění programů si můžeme nechat ve sledovacím okně průběžně vypisovat hodnoty proměnných a konstant. To platí i pro konstanty a proměnné strukturovaných datových typů, tj. pro pole, záznamy, struktury a unie. V případě strukturovaných datových typů nám systém vypíše postupně hodnoty jednotlivých prvků nebo složek, dokud stačí řádek. V okně je pak zobrazen vždy pouze počátek řádku, a pokud chceme vidět i jeho pokračování, musíme aktivovat sledovací okno (*Watch*), najet na řádek s hodnotou analyzované strukturované proměnné kurzorem a pak s tímto řádkem pomocí vodorovných šipek pohybovat (samotná šipka vpravo a vlevo pohybuje s řádkem o 1 znak, šipka při stisknutí přefadovači CTRL posune řádek o šíři okna, klávesy HOME a END nás přesunou na počátek a konec řádku).

Je-li ve sledovacím okně zobrazována unie nebo variantní část záznamu, zobrazí počítač také hodnoty všech složek, avšak je zřejmé, že změna hodnoty kterékoliv z nich může zároveň ovlivnit hodnotu všech ostatních.

Jsou-li prvky pole či složky záznamů (struktur) dále strukturované, vypisuje program seznam těchto hodnot uzavřený v závorkách – kulatých pro Pascal a složených pro C++.

V kapitole o ladění jsme si říkali o formátovacích pokynech pro zobrazování hodnot sledovaných konstant a proměnných. Tyto formátovací pokyny můžete využít i při formátování výstupu prvků vektorů, avšak nelze je dost dobře použít pro záznamy, struktury a svazy, u nichž může být obecně každá složka jiného typu. Přesněji řečeno, nelze je použít pro jednotlivé prvky nebo složky daného objektu, ale pouze hromadně pro všechny najednou.

Při práci s vektory můžeme využít další formátovací možnosti: pokud za názvem sledovaného objektu a následující čárkou napíšeme nejprve číslo, požádáme tím ladicí program, aby ve stejném formátu vypsal na daný řádek ještě daný počet hodnot stejného typu, které jsou na následujících adresách. Chceme-li tedy např. sledovat činnost programu na vektoru *Vec* reálných čísel v průběhu cyklu s parametrem *i*, můžeme např. požádat o sledování příkazem

```
Vec[ i-2 ], 5f4
```

Přítom bude ladicí program průběžně zobrazovat s přesností na 4 platné číslice 5 hodnot počínaje hodnotou s indexem *i-2*.

Opakování lze použít nejen pro prvky pole. V závěrečném příkladu (C++) v kapitole 14 jsou např. proměnné *W1* a *W2* uloženy v paměti evidentně za sebou. Můžeme si proto nechat vypsat hodnoty obou dvou na jeden řádek, a to příkazem

```
w1,2
```

Pro snadnější sledování hodnot složek strukturovaných datových typů máme kromě toho možnost použít formátovacího pokynu *r*, kterým žádáme překladač, aby před každou

složkou strukturového datového typu uvedl její identifikátor. Abyste si funkci tohoto popisu ověřili, zapište při krokování příkladu z kapitoly 14 do sledovacího okna příkaz

```
W1,r
```

Jak jsme si již řekli, formát zobrazení strukturovaných objektů nemůžeme definovat pro každou jejich složku zvlášť. Vezměme např. strukturový datový typ definovaný v Pascalu

```
type
  SDT = record
    Dec: integer;
    Hex: long;
    Presne: real;
    Zhruba: real;
end;
```

a v C++

```
struct SDT {
  int Dec;
  long Hex;
  double Presne;
  float Zhruba;
}
```

a vektor SV objektů tohoto typu. Při zobrazování prvků tohoto vektoru nemůžeme chtít zobrazovat složku *Dec* v desítkové soustavě a složku *Hex* v soustavě šestnáctkové, stejně jako nemůžeme chtít zobrazovat složku *Zhruba* s přesností na 3 platné cifry a složku *Presne* s přesností na 8 platných cifer. Buďto musíme zvolit nějaký kompromis, např.

```
SV[ i ],2rf4x
```

nebo si budeme muset nechat dané údaje vypsát na několikrát. Uvědomte si však, že přidáním příkazu

```
SV[ i ].Presne,2f8
```

nezískáme hodnoty složek s různými indexy, $SV[i].Presne$ a $SV[i+1].Presne$, ale hodnoty $SV[i].Presne$ a $SV[i].Zhruba$, protože systém při zadání opakování vypisuje hodnoty, které za první hodnotou v paměti bezprostředně následují.

Při zobrazování reálných čísel platí nastavená přesnost pro všechna zobrazovaná reálná čísla až do té doby, než explicitně nastavíme přesnost jinou. Toto pravidlo jde dokonce tak daleko, že poslední nastavená přesnost ve sledovacím okně platí i pro okno vyhodnocování výrazů.

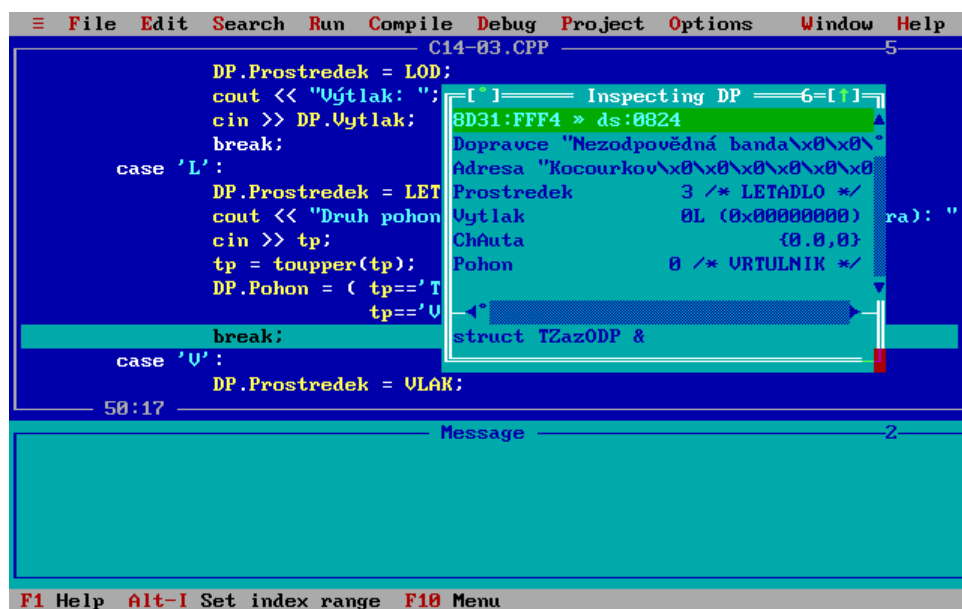
Při práci s většími datovými strukturami nám velice často jeden řádek ve sledovacím okně nestačí. Pokud pracujeme v Pascalu, máme jedinou možnost: rozdělit své požadavky, tj. nechat si u vektorů vypsát jejich prvky na několikrát a u struktur požádáme o sledování každé složky zvlášť. Pracujeme-li v C++, máme pak k dispozici ještě druhou možnost: použít inspekčních oken.

Inspekční okno v prostředí Borland C++

Inspekční okno můžeme otevřít buďto pomocí klávesové zkratky ALT-F4, nebo pomocí volby *Debug | Inspect*. Pokud se v editovacím okně vyskytuje konstanta či proměnná, jejíž inspekce vás zajímá, stačí na ni před otevřením okna najet kurzorem. Pokud nemáme nikde „po ruce“ specifikaci objektu, o jehož inspekci chceme požádat, musíme před otevřením inspekčního okna najet kurzorem někam do volného prostoru, aby nám systém nabídl dialogové okno, v němž bychom specifikaci daného objektu zadali.

Na prvním řádku inspekčního okna najdeme adresu analyzovaného objektu, na dalších řádcích pak rozepsanou hodnotu. Dole pod oddělovací čarou je pak uveden typ objektu. Při inspekci vektoru se nám v okně objeví na každém řádku výpis hodnoty jednoho jeho prvku, při inspekci záznamu, struktury či svazu se na každém řádku objeví výpis hodnoty jedné složky. Pokud je daný prvek či složka opět strukturovaná, můžeme na ni najet kurzorem a stiskem klávesy ENTER otevřít inspekční okno i pro ni.

Na rozdíl od sledovacího okna si inspekční okna neuchovávají po opětovém spuštění programu svůj obsah, takže vám nezbuďte než znovu specifikovat objekty, pro něž chcete inspekční okna otevřít.



5.1 Inspekční okno v Borland C++ 3.1

Formát zobrazení celých čísel v inspekčních oknech ovlivníme z dialogového okna *Options | Debugger*, kde můžeme nastavit, zda se celá čísla v inspekčních oknech budou zobrazovat v desítkové nebo v šestnáctkové soustavě nebo v obou (tj. zobrazí se obě hodnoty). Formát reálných čísel, tj. počet zobrazovaných cifér, je dán posledním nastavením. Příklad inspekčního okna vidíte na obr. 5.1.

6. Používání funkcí z jazyka C v C++

Jak víme, vychází jazyk C++ z jazyka C a rozšiřuje jej o řadu rysů týkajících se především „bezpečného programování“ (např. zavedení výčtových typů, důsledná typová kontrola, nahrazení maker vloženými funkcemi apod. – mnohé z nich jazyk C zpětně převzal) a především o objektivě orientovanou nadstavbu.

Jednou z novinek jazyka C++ je tzv. **zdobení jmen** funkcí a dalších objektů, které umožňuje typovou kontrolu jejich parametrů i ve fázi sestavování. Zdobení jmen spočívá v tom, že překladač doplní (ozdobí) vámi definovaný identifikátor funkce předponami a příponami, které popisují typy jejich jednotlivých parametrů. Na podkladě takto ozdobených jmen pak sestavovací program dokáže na jedné straně kontrolovat při sestavování programu kompatibilitu definice podprogramu v jednom modulu a jeho volání v jiném modulu. (Výběr správného homonyma – jedné z přetížených funkcí – je věcí překladače a ozdobená jména se přitom nepoužívají.)

Poznámka:

*Termín **zdobení jmen** (name decorating) jsme převzali z manuálů firmy Microsoft. Myslíme si, že lépe vystihuje prováděné akce než termín **name mangling** (komolení jmen), který najdete ve manuálech firmy Borland.*

Překladače jazyka C ovšem s identifikátory funkcí zacházejí jinak – zpravidla před ně připojují podtržítko. Zákonitým důsledkem je nekompatibilita programů přeložených kompilátory jazyka C a C++. Vzhledem k tomu, že v jazyku C jsou naprogramovány rozsáhlé knihovny, které je užitečné mít k dispozici i v C++, zavedli autoři jazyka C++ konstrukci, která zakazuje překladači komolení jmen a některé další nekompatibilní praktiky pro označené deklarace.

Jazyk C++ proto rozšířil význam deklarace **extern** tak, že pokud za ní následuje textový řetězec "C" nebo "C++", tak ji překladač pochopí jako direktivu příkazující interpretovat následující deklaraci podle konvencí jazyka, jehož jméno je uvnitř uvozovek. Pokud tedy objevíte v programu definici

```
extern "C" int FunkceJazykaC( void );
```

znamená to, že daná funkce je přeložena podle konvencí jazyka C (tj. např. bez zdobení jména).

Protože jazyky C a C++ často sdílí rozsáhlé knihovny, kterým odpovídají rozsáhlé hlavičkové soubory obsahující velká množství nejrůznějších deklarací, byla do jazyka zavedena i možnost zadat tuto direktivu pro celý blok programu. Toho dosáhneme tak, že příslušnou část programu uzavřeme do složených závorek, před které napíšeme deklaraci **extern "C"**. Objevíte-li tedy v hlavičkových souborech konstrukci

```
#ifdef __cplusplus
extern "C" {
#endif
```

```
//Zde jsou deklarace kompatibilní s jazykem C
#ifdef __cplusplus
}
#endif
```

znamená to, že část programu uvnitř podmíněných sekcí přikazuje překladači C++, aby se jmény, deklarovanými uvnitř této sekce, zacházel podle konvencí jazyka C. Po zpracování preprocesorem pro překladač jazyka C zůstane z programu pouze část mezi oběma podmíněnými sekcemi (neboť při překladu kompilátorem jazyka C není definováno makro `__cplusplus`), kdežto po zpracování překladačem C++ se konstrukce převede do `ř` doby:

```
extern "C" {
//Deklarace kompatibilní s jazykem C
}
```

Pamatujte tedy na to, až budete chtít dovážet funkce z nějaké cizí knihovny v C, která nebyla původně určena pro jazyk C++. V takovém případě můžete do hlavičkového souboru obsahujícího prototypy všech používaných funkcí a externích proměnných tuto konstrukci vložit; jednodušší je ale napsat

```
extern "C" /
#include <cizí_soubor.h>
}
```

Výskyt některých deklarací uvnitř výše zmíněné konstrukce ovlivňuje i jejich používání v dalším programu. Nejčastěji vás to asi potká u výčtových a strukturových datových typů (struktur a uníí). Pokud totiž bude deklarace výčtového nebo strukturového datového typu, který chcete používat, uvedena v „kompatibilní části“ hlavičkového souboru, a pokud nebude zavedena specifikátorem **typedef**, budete muset v deklaracích datových objektů tohoto typu uvádět i klíčová slova **enum**, **struct** či **union**. Například:

```
extern "C" {
    struct TPoloha {
        int x;
        int y;
    };
    enum eObrat{ NECHAT, PREKLOPIT, OTOCIT, _eObrat };
}
struct TPoloha Pozice0 = {0, 0}, Pozice1{1, 0};
void Posun( struct TPoloha Start,
            struct TPoloha Cil,
            enum eObrat Zmena );
```

7. Základy práce s grafikou

Základní kurs programování nelze ukončit, aniž bychom si pověděli o základech práce s grafikou. Borlandské překladače obou jazyků přicházejí s knihovnamí grafických funkcí, které si jsou velmi podobné. Liší se sice trochu rozsahem, ale základní funkce jsou v obou knihovnách stejné, takže nic nebrání tomu, abychom si je vybrali společně.

Pokud chceme pracovat s grafickou knihovnou, musíme v Turbo Pascalu dovést modul *Graph* a v C++ musíme do souboru vložit hlavičkový soubor *graphics.h*. V C++ musíme navíc požádat sestavovací program o to, aby zařadil grafickou knihovnu mezi knihovny prohledávané. Dosáhneme toho nastavením volby *Graphics library*, kterou najdeme v nejstarších verzích překladačů v dialogovém okně *Options | Linker*, v novějších v dialogovém okně *Options | Linker | Libraries*; v nejnovějších překladačích nastavujeme použití grafické knihovny v okně *Target Expert* zaškrtnutím políčka *BGI* ve skupině *Libraries*. (Tato volba je ovšem dostupná pouze pro aplikace, určené pro DOS.) Pokud tak neučiníme, bude nám sestavovací program tvrdit, že funkce z grafické knihovny nemůže najít.

Poznámka 1:

Omlouváme se, že většina ukázkových příkladů bude napsána pouze v jednom z jazyků, ale vede nás k tomu několik důvodů. Za prvé si myslíme, že obě knihovny jsou si natolik podobné, že by pro pascalisty nemělo nemělo být problémem převést si uvedené příklady do Pascalu a naopak, pro céčkaře by nemělo být tak velkým problémem převést si uvedené příklady do C++. Za druhé si myslíme, že obě knihovny jsou si natolik podobné, že by nás při uvádění příkladů v obou jazycích někteří začali podezřívát z průhledné snahy o vyšší honorář. Za třetí si myslíme, že obě knihovny jsou si natolik podobné, že by pro vás bylo výhodnější, kdybychom ušetřili papír na duplicitní příklady a věnovali jej raději podrobnějšímu výkladu.

Poznámka 2:

Všechny příklady v této kapitole jsou koncipovány tak, aby jejich poskládáním vznikl chodící program, který potřebuje dodat pouze proceduru `main()` resp. hlavní program, jež uvedené příklady odpovídajícím způsobem zavolá.

7.1 Zpracování chyb

Než se pustíme do výkladu procedur a funkcí realizujících různé grafické operace, měli bychom si nejprve něco říci o způsobu, jakým se borlandské grafické knihovně zpracovávají chyby.

Obecně se v programech používají tři přístupy ke zpracování chyb:

- ✧ okamžité volání podprogramu ošetřujícího vzniklý výjimečný stav,
- ✧ vrácení předem definované hodnoty jednoznačně oznamující vznik chybového stavu,

✧ nastavení hodnoty nějaké globální proměnné, kterou může uživatelský program vhodným způsobem otestovat. Tato (tj. třetí) metoda je použita i v borlandské grafické knihovně, přičemž v některých chvílích je navíc kombinována s metodou druhou.

Všechny grafické podprogramy, při jejichž plnění může dojít k chybě, nastaví kód vzniklé chyby do nějaké vnitřní proměnné, jejíž hodnotu nám přečte a vrátí funkce *graphresult*. Musíme ovšem počítat s tím, že voláním funkce *graphresult* se vnitřní proměnná s kódem chyby vynuluje, takže pokud budeme chtít kód chyby ještě později použít, musíme si jej někde uložit (obdobnou situaci jsme zažili s pascalskou funkcí *ioresult*).

Obě knihovny navíc definují konstanty, jejichž identifikátory označují příčinu chyby s daným kódem. V C++ jsou tyto konstanty shrnuty ve výčtovém typu *graph_errors*, který je definován následovně:

```
enum graphics_errors
{
    grOk = 0,                //Bez chyby
    grNoInitGraph = -1,     //Grafika není inicializována,
                           //tj. nebyl volán initgraph
    grNotDetected = -2,    //Nebyla nalezena odpovídající
                           // grafická karta
    grFileNotFound = -3,   //Nebyl nalezen ovladač daného
                           //zařízení (karty)
    grInvalidDriver = -4,  //Neplatný soubor s ovladačem
                           //zařízení (karty)
    grNoLoadMem = -5,      //Nedostatek paměti pro načtení
                           //ovladače grafického zařízení
    grNoScanMem = -6,      //Vyčerpání paměti při vybarvování
                           //polygonu procedurou fillpoly
    grNoFloodMem = -7,     //Vyčerpání paměti při vybarvování obecné
                           //plochy procedurou floodfill
    grFontNotFound = -8,   //Soubor s fontem nenalezen
    grNoFontMem = -9,      //Nedostatek paměti pro načtení fontu
    grInvalidMode = -10,   //Neplatný režim pro daný
                           //ovladač grafického zařízení
    grError = -11,         //Blíže nespecifikovaná chyba
    grIOerror = -12,       //Chyba grafického vstupu a výstupu
    grInvalidFont = -13,   //Neplatný soubor s fontem
    grInvalidFontNum= -14, //Neplatné číslo fontu
    grInvalidVersion= -18  //Špatné číslo verze
};
```

V Pascalu jsou definovány stejnojmenné konstanty odpovídající (s výjimkou kódu -18, který Pascal nepoužívá).

7.2 Inicializace grafického systému

Borlandský grafický balík obsahuje kromě vlastní knihovny (a v C++ příslušného hlavičkového souboru) i sadu ovladačů nejpoužívanějších grafických karet. Díky tomu můžete téměř všechny grafické operace programovat jednotně, nezávisle na tom, na jaké konkrétní grafické kartě daný program běží.

Vy se sice v programu můžete tvářit, že vůbec nic nevíte o hardwaru, na němž váš program v danou chvíli kreslí (ono to zase tak jednoduché není, ale o tom až za chvíli), avšak systém charakteristiky tohoto hardwaru znát musí. Před vyvoláním jakékoliv funkce z grafické knihovny musíte proto grafický systém nejprve inicializovat. To zařídíte vyvoláním procedury *initgraph* (deklarace všech vysvětlovaných procedur funkcí najdete v tabulce).

Jejím prvním parametrem je v Pascalu proměnná (v C++ adresa proměnné), jejíž hodnota definuje typ grafické karty, na níž program běží. Protože předpokládáme, že většina z vás bude chtít, aby byl program co nejuniverzálnější, aby si zjistil sám, na jaké kartě běží, a aby si i sám nastavil odpovídající ovladač grafického zařízení, doporučujeme nulovou vstupní hodnotu, která spustí mechanismus autodetekce. Pokud nás poslechnete, najdete po návratu z této procedury v dané proměnné číslo definující kartu, kterou procedura našla.

Druhým parametrem je proměnná (v C++ opět adresa proměnné), jejíž hodnota definuje grafický režim, který se má v průběhu inicializace nastavit. Pokud je však hodnota prvního parametru nulová (a to je náš případ), procedura druhý parametr ignoruje a automaticky nastaví grafický režim s nejvyšším možným rozlišením. To nám také většinou vyhovuje (o tom, co dělat v případě, že bychom to chtěli jinak, si povíme za chvíli).

Třetím parametrem je textový řetězec definující cestu do adresáře, v němž program najde soubory s ovladači jednotlivých karet. Pokud jste si při instalaci svého překladače moc nevymýšleli, měl by se jmenovat BGI a být jedním z podadresářů zřízených při instalaci. Poznáte jej podle toho, že obsahuje (mimo jiné) několik souborů s příponou .BGI – to jsou právě ty zmiňované ovladače.

Obrazovka

Jednou z prvních věcí, kterou si musíme při práci s grafikou zapamatovat, je, že grafická obrazovka má společně s textovou obrazovkou počátek v levém horním rohu, avšak na rozdíl od ní nepočítá souřadnice od jedné, ale od nuly. Tak jako jsme v textovém režimu pracovali se souřadnicemi znaků, budeme v grafickém režimu pracovat se souřadnicemi obrazových bodů – tzv. *pixelů*¹⁶.

„Pixelový“ rozměr obrazovky se liší podle použité grafické karty. Jak vyplývá z předchozího odstavce, levý horní roh bude mít na všech kartách souřadnice (0;0). Při

¹⁶ Slovo pixel vzniklo z anglického picture element – česky obrazový prvek. Podle definice se jedná o nejmenší prvek zobrazovací plochy, jemuž lze nezávisle přiřadit barvu nebo intenzitu.

maximální rozlišovací schopnosti bude mít pravý dolní roh na kartách CGA souřadnice (639;199), na kartách EGA souřadnice (639;349), na kartách VGA souřadnice (640;479) a na kartách Hercules souřadnice (719;347). Poznamenejme, že standardní borlandská knihovna BGI neumí využít možností, které nabízejí karty SVGA, a zachází s nimi jako s kartami VGA, resp. EGA. Existují ale různá rozšíření knihovny BGI, která to umějí.

Abychom mohli v programu kdykoliv zjistit rozlišovací schopnost právě nastaveného grafického režimu, nabízí nám grafická knihovna funkce *getmaxx* a *getmaxy*, které vrací nejvyšší možnou hodnotu souřadnice v daném směru – viz následující ukázka.

Poznámka:

Na tomto místě je třeba upozornit na to, že skutečnost, že chceme na obrazovku něco napsat, nemusí být ještě důvodem k opuštění grafického režimu. Pokud budeme k našim vstupům a výstupům používat standardní systémový vstup a výstup, je úspěšná komunikace záležitostí operačního systému, který ví, jaký obrazovkový režim je právě nastaven (protože všechny nastavovací funkce nakonec vedou k volání odpovídající funkce operačního systému), a podle toho se zařídí.

V následujícím příkladu inicializujeme grafický režim, zjistíme rozměr obrazovky a vypíšeme jej pomocí standardních výstupů, a pak grafický režim ukončíme. Tento příklad si ukážeme v obou jazycích. Poznamenejme, že pokud si tento příklad budete chtít spustit, musíte v příkazu, označeném v komentáři několika vykřičníky, zapsat skutečnou cestu ke grafickému ovladači, platnou na vašem počítači.

```

/*   Příklad C7 - 1   */
//Inicializace grafického režimu
//Při zkoušení je třeba změnit cestu ke grafickému ovladači
//v řádce označené !!!!

#include <iomanip.h>
#include <graphics.h>
#include <conio.h>

int mx; //Maximální souřadnice ve vodorovném směru
int my; //Maximální souřadnice ve svislém směru

int /******/ GrInit /******/ ()
//Inicializuje grafický systém a vrátí číslo chyby
{
    int mode; //Pomocná, nepoužitá proměnná
    int driver = DETECT; //Autodetekční režim
    int gr;
    initgraph( &driver, &mode, "c:\\bc31\\bgi\\" ); // !!!!
    if( (gr=graphresult()) == 0 )
        cout << "Grafický režim nastaven" << endl;
    else
    {
        cout << "Chyba - kód " << gr << endl;
        return gr;
    }
}
mx = getmaxx();

```

```

my = getmaxy();
cout << "\nBodový rozměr obrazovky: "
      << mx << " x " << my << endl;
cout << "Stiskni klávesu ..." ;
getch();
closegraph();
return( 0 );
}

```

V Pascalu předáváme proceduře *InitGraph* proměnné, nikoli jejich adresy (předávají se odkazem, takže to opravdu musí být proměnné). Poznamenejme, že chceme-li použít k výstupu v grafickém režimu funkce *write* nebo *writeln*, nesmíme použít knihovnu *Crt* (takže nemáme k dispozici funkci *readkey*).

```

(* Příklad P7 - 1 *)
uses graph;

var mx, my: integer; {Maximální souřadnice ve vodorovném směru
                     a maximální souřadnice ve svislém směru }

function (****) GrInit (****): integer;
{ Inicializuje grafický systém a vrátí číslo chyby }
var
  mode, driver, gr: integer;
  c: char;
begin
  driver := DETECT; {Autodetekční režim}
  initgraph( driver, mode, 'c:\aplikace\prekl\bp\bgi\' ); { !!!! }
  gr := graphresult;
  if (gr = 0 ) then
    writeln( 'Grafický režim nastaven' )
  else
    begin
      writeln( 'Chyba - kód ', gr );
      GrInit := gr;
    end;
  mx := getmaxx;
  my := getmaxy;
  writeln(#13'Bodový rozměr obrazovky: ', mx, ' x ', my);
  writeln('Stiskni klávesu + Enter...');
  read(c);
  closegraph;
  GrInit := 0;
end;

begin
  GrInit;
end.

```

Ovládání celého grafického systému

Nyní už umíme inicializovat grafický systém a nastavit grafický režim. Pokud budeme chtít při dalším průběhu programu přepínat mezi textovým a grafickým režimem, musíme si před opuštěním grafického režimu zapamatovat nastavený režim, abychom se do něj mohli vrátit. K tomuto účelu poslouží celočíselná funkce *getgraphmode*, která vrací hodnotu nastaveného grafického režimu.

K dočasnému opuštění grafického režimu slouží procedura *restorecrtmode*, která nastaví zpět ten textový režim, který byl nastaven při přepnutí do režimu grafického. K opětovnému nastavení grafického režimu se pak používá procedura *setgraphmode*, které v parametru předáme číslo režimu, jež chceme nastavit – nejspíše režimu získaného voláním funkce *getgraphmode*, ale klidně i jiného.

Pokud chceme opustit grafický režim definitivně a uvolnit i veškerou paměť, kterou grafický systém zabral, zavoláme proceduru *closegraph*. Kdychom si to později rozmysleli a chtěli grafický režim znovu nastavit, musíme se vrátit zcela na počátek a inicializovat celý grafický systém procedurou *initgraph*.

Nevyhovuje-li nám z nejrůznějších důvodů autodetekce připojeného grafického zařízení, nabízí vám knihovna několik podprogramů, které vám pomohou zjistit některé informace.

Poznámka:

Spokojíte-li se s automatickým nastavením grafického zařízení a odpovídacího režimu, můžete zbytek této podkapitoly a celou další přeskočit a začít číst až od začátku podkapitoly Grafický kurzor.

První z těchto pomocných procedur pro získání potřebných informací je procedura *detectgraph* (tuto proceduru volá *initgraph* při nastaveném zařízení *DETECT*), která má dva výstupní parametry.¹⁷ V prvním parametru vrací číslo definující doporučený ovladač nalezeného zařízení a v druhém pak nejvyšší přípustné číslo grafického režimu, které je pro daný ovladač zároveň číslem režimu s největším rozlišením.

Pro hodnoty vracené procedurou *detectgraph* jsou opět připraveny předdefinované konstanty, které jsou v C++ navíc sdruženy do dvou výčtových typů. Jejich definice je následující (pascalské konstanty mají stejné identifikátory):

```
enum graphics_drivers
{
    DETECT,           //0 - Autodetekce
    CGA,              //1 - Color/Graphics Adapter
    MCGA,             //2 - Multi-Color Graphics Adapter
    EGA,              //3 - Enhanced Graphics Adapter (256KB RAM)
    EGA64,            //4 - Enhanced Graphics Adapter (64KB RAM)
    EGAMONO,         //5 - EGA s monochromatickým monitorem
    IBM8514,          //6 - IBM 8514 Graphics Adapter
}
```

¹⁷ Tedy: V Pascalu jsou to dva parametry typu *integer*, předávané odkazem, v C a v C++ to jsou adresy dvou proměnných typu *int*.

```

HERCMONO,          //7 - Hercules Graphics Adapter
ATT400,            //8 - AT&T 400řádkový grafický adaptér
VGA,               //9 - Video Grphics Array
PC3270,           //10- 3270 PC Graphics Adapter
CURRENT_DRIVER = -1 //Aktuální ovladač
};

enum graphics_modes {
  CGAC0 = 0,       // 320x200 - 1 stránka - paleta 0
  CGAC1 = 1,       // 320x200 - 1 stránka - paleta 1
  CGAC2 = 2,       // 320x200 - 1 stránka - paleta 2
  CGAC3 = 3,       // 320x200 - 1 stránka - paleta 3
  CGAHI = 4,       // 640x200 - 1 stránka - 2 barvy
  MCGAC0 = 0,     // 320x200 - 1 stránka - paleta 0
  MCGAC1 = 1,     // 320x200 - 1 stránka - paleta 1
  MCGAC2 = 2,     // 320x200 - 1 stránka - paleta 2
  MCGAC3 = 3,     // 320x200 - 1 stránka - paleta 3
  MCGAMED = 4,    // 640x200 - 1 stránka - 2 barvy
  MCGAHI = 5,     // 640x480 - 1 stránka - 2 barvy
  EGALO = 0,      // 640x200 - 4 stránky - 16 barev
  EGAHI = 1,      // 640x350 - 2 stránky - 16 barev
  EGA64LO = 0,    // 640x200 - 1 stránka - 16 barev
  EGA64HI = 1,    // 640x350 - 1 stránka - 4 barvy
  EGAMONHI = 0,   // 640x350 - 1 stránka ( 64K paměti)
                  // - 4 stránky (256K paměti)
  HERCMONHI = 0,  // 720x348 - 2 stránky - 2 barvy
  ATT400C0 = 0,   // 320x200 - 1 stránka - paleta 0
  ATT400C1 = 1,   // 320x200 - 1 stránka - paleta 1
  ATT400C2 = 2,   // 320x200 - 1 stránka - paleta 2
  ATT400C3 = 3,   // 320x200 - 1 stránka - paleta 3
  ATT400MED = 4,  // 640x200 - 1 stránka - 2 barvy
  ATT400HI = 5,   // 640x400 - 1 stránka - 2 barvy
  VGALO = 0,     // 640x200 - 4 stránky - 16 barev
  VGAMED = 1,    // 640x350 - 2 stránky - 16 barev
  VGAHI = 2,     // 640x480 - 1 stránka - 16 barev
  PC3270HI = 0,  // 720x350 - 1 stránka - 2 barvy
  IBM8514LO = 0, // 640x480 - 1 stránka - 256 barev
  IBM8514HI = 1  //1024x768 - 1 stránka - 256 barev
};

```

Na základě údajů získaných procedurou *detectgraph* pak můžeme zadáním vhodných parametrů ovlivnit činnost procedury *initgraph*. V prvním parametru zadáme místo nuly číslo zařízení, na něž chcete grafický systém nastavit, a procedura *initgraph* rovnou inicializuje systém pro toto zařízení.

Jakmile je však první parametr procedury *initgraph* nenulový, začne se procedura starat o parametr druhý, kde očekává číslo grafického režimu, který má nastavit. Jak jste si všimli v předchozí tabulce, režimy jednotlivých zařízení jsou číslovány tak, že režim s vyšším číslem má i vyšší (v nejhorším případě stejnou) rozlišovací schopnost, která však většinou bývá vyvážena menší použitelnou paletou barev.

Aby bylo možno snadno nabídnout uživateli programu alternativy na výběr, nabízí knihovna několik procedur. Bohužel, knihovna neobsahuje funkci, která by vracela číslo použitého zařízení. Místo ní obsahuje funkci, která vrací název použitého ovladače, avšak

vzhledem k tomu, že jeden ovladač může pokrývat několik zařízení, nezbude vám než si číslo nastaveného zařízení pamatovat sami.

Jednodušší je situace s nastaveným režimem. Lze jej kdykoliv zjistit vyvoláním funkce *getgraphmode*, která vrací číslo režimu jako svoji funkční hodnotu. Kromě toho obsahuje knihovna ještě funkci *getmaxmode*, která vrací nejvyšší možné číslo režimu pro inicializované zařízení.

Chcete-li dát uživateli vybrat mezi režimy, které jsou na daném zařízení možné, můžete využít funkce *getmodename*, která ve svém jediném parametru očekává číslo režimu, načež vrátí jako funkční hodnotu textový řetězec obsahující rozlišovací schopnost v daném režimu a identifikaci zařízení. Nový režim pak nastavíte procedurou *setgraphmode*.

Nyní se podíváme na maličký prográmeček, ve kterém je třeba opět upravit cestu ke grafickému ovladači podle skutečného stavu na vašem počítači.

```

/* Příklad C7 - 2 */
#include <graphics.h>
#include <iostream.h>
int mx, my;

extern "C" int getch( void ); //Prototyp knihovní funkce
//umožňující nedovážet (#include) soubor CONIO.H

void /*****/ Rucne /*****/ ( )
{
    int mode = 0; //Pomocná, nepoužitá proměnná
    int driver = CGA;
    int gr;
    detectgraph( &driver, &mode );
    cout << "Detekované zařízení č. " << driver << endl
         << "Doporučený režim č. " << mode << endl;
    getch();
    do {
        initgraph( &driver, &mode, "c:\\aplikace\\prekl\\bc31\\bgi" );
        if( (gr=graphresult()) != 0 )
            cout << "\n\nChyba - kód " << gr << endl;
        else
        {
            cout << "Grafický režim nastaven (" << gr << ")\n";
            do {
                cout << "Nastavené zařízení č. " << driver
                     << "\nOvladač: " << getdrivername() << endl;
                cout << "Nastavený režim: " << (mode=getgraphmode())
                     << " - " << getmodename( mode ) << endl;
                mx = getmaxx();
                my = getmaxy();
                cout << "\nBodový rozměr obrazovky: "
                     << mx << " x " << my << endl;
                cout << "\nK dispozici jsou režimy: " << endl;
                for( int i=0; i <= getmaxmode(); i++ )
                    cout << i << " - " << getmodename(i) << endl;
                //Přeškrtni celou obrazovku, aby bylo zřejmé,

```

```

        //že jsme opravdu v grafice
        line( 0, 0, mx, my ); //O této funkci si
        line( 0, my, mx, 0 ); // povíme za chvílku
        cout << "Chybový kód: " << graphresult();
        cout << "\nNastavíš režim č. ";
        cin >> mode;
        if( mode >= 0 )
            setgraphmode( mode );
    } while( mode >= 0 );
}
cout << "Nastavíš zařízení č. ";
cin >> driver;
if( driver == -2 ) //Nastavením zařízení -2 ukončím
    break; //program, nechám grafický režim
mode = 0; //Režim s nejmenším rozlišením
//je k dispozici vždy
closegraph(); //Grafický systém se musí odinstalovat
//i v případě, že chci pouze nastavit nové zařízení
} while( driver >= 0 );
//Nastavením záporného zařízení ukončím program.
//Bude-li mít zařízení jiné číslo, než -2,
//vrátím se do textového režimu.
}

```

7.3 Barvy

I když by se možná mohlo zdát, že nastavování barev by měla být věc poměrně jednoduchá, není to tak úplně pravda. Situaci totiž komplikuje skutečnost, že barevné možnosti používaných grafických karet jsou značně různorodé. Funkce borlandské grafické knihovny se sice snaží ovládnání všech zařízení maximálně sjednotit, avšak přece jenom zůstávají některá specifika, které se jim zcela odstínit nepodařilo.

Karty a barvy

Než si začneme vykládat o jednotlivých procedurách a funkcích sloužících k nastavování barev, dovolte nám nejprve malou technickou odbočku, ve které si povíme některé důležité informace o grafických kartách.

Nejprve však dva termíny: **barvou pozadí** budeme nazývat barvu smazané obrazovky – mohli bychom o ní hovořit jako o barvě „papíru, na který kreslíme“. Jako **barvu popředí** označíme barvu kreslených objektů. O této barvě budeme také hovořit jako o barvě „pera“, abychom tak zdůraznili, že se jedná o barvu, kterou budeme kreslit pouze chvíli a kterou můžeme kdykoliv vyměnit za barvu jinou.

V předchozím povídání jsme se dozvěděli, že borlandská grafika umí pracovat s řadou grafických karet. S většinou z nich se však dnes už neseťkáte, takže se omezíme na karty EGA a VGA. Nikoli snad proto, že by se stále ještě používaly, ale proto, že jsou tím nej-

dokonalejším, s čím umí standardní borlandská grafická knihovna zacházet (a karty SVGA je umějí emulovat, tj. mohou se v jednom z režimů chovat jako EGA nebo VGA).

Grafická karta EGA (*Enhanced Graphic Adapter* – zdokonalený grafický adaptér) nabízí 64 barev. Programátor si může vybrat, kterých 16 barev může každý jednotlivý bod nabýt. (U karty s menší pamětí na desce jsou to v jemnějším režimu pouze 4 barvy.) Této vybrané šestnáctce (čtveřici) barev budeme říkat **paleta**.

Grafická karta VGA (*Video Graphics Array* – grafické obrazové pole) nabízí výběr z 256 tisíc (přesně 256 x 1024) barev a v závislosti na nastaveném režimu může každý bod obrazu nabývat až jednu z 256 předem nastavených barev.

Karta VGA umožňuje ovšem i nastavování barev kompatibilní s kartou EGA. Toho využívá i borlandská grafická knihovna a s oběma výše zmiňovanými kartami pracuje stejně. (To znamená, že v borlandské knihovně nemůžeme využít všech možností ani u karty VGA, natož pak u SVGA.)

Nastavování barev

Podívejme se nyní postupně na jednotlivé možnosti ovládání barev.

První věcí, kterou musíme při nastavování barev vzít v úvahu, je, že barvy se většinou nastavují dvoufázově. Jak jsme si řekli v technické odbočce, grafické karty mohou v danou chvíli zobrazit pouze předem definovanou podmnožinu množiny všech zobrazitelných barev. V první fázi je proto třeba definovat tuto podmnožinu – tzv. **paletu**, a teprve potom můžeme z této podmnožiny vybírat potřebné barvy. Kódem barvy je pak její umístění v paletě.

V případě grafických karet EGA nebo VGA definujeme nastavením zobrazovacího režimu pouze rozlišovací schopnost a velikost palety, ale nastavení jednotlivých barev palety se provádí zvlášť.

Pro práci s paletami barev je v obou knihovnách definován datový typ *palettetype*, jehož pascalská definice má tvar

```
const
  MaxColors = 15;           {Maximální počet barev v paletě}
type
  PaletteType = record
    Size: byte;             {Počet barev v nastaveném režimu}
    Colors: array[ 0 ..MaxColors ] of Shortint;
                           {Charakteristika jednotl. barev}
  end;
```

a definice v C++ má tvar

```
#define MAXCOLORS 15        //Maximální počet barev v paletě
//Definice vychází z konvencí jazyka C – proto je v ní
//místo definice konstanty použito definice makra
struct palettetype
{
  unsigned char size;       //Počet barev v nastaveném režimu
  signed char colors[ MAXCOLORS+1 ];
```

```
}; //Charakteristika jednotlivých barev
```

Jak vidíte, objekty typu *palettetype* obsahují vektor 16 barev. Je to proto, že v zájmu kompatibility a hlavně jednoduchosti podporuje borlandská grafická knihovna na kartě VGA pouze režimy umožňující zobrazit v daném okamžiku na displeji maximálně 16 barev (viz tabulka o nastavování grafických režimů). Pro jednoduchost se však tento datový typ používá (a 16 bajtů vyhrazuje) i v případě, že v daném režimu bude z tohoto vektoru možno použít pouze první čtyři (režim *EGA64HI*) nebo dokonce pouze první dvě (*EGAMONOHI*) barvy.

Paletu barev můžeme pro karty EGA a VGA nastavovat buď v celku procedurou *setallpalette*, jejímž jediným parametrem je objekt (v C++ adresa objektu) typu *palettetype*, nebo jednotlivě procedurou *setpalette*, jejímž prvním parametrem je pořadí nastavované barvy v paletě (0 .. 15, přesněji 0 .. *size*) a druhým číslo dané barvy mezi 64 barvami poskytovanými kartou EGA.

Na barvy palety se také můžeme ptát. Ukazatel na paletu, kterou na počátku práce nastavila procedura *initgraph*, získáme voláním podprogramu *getdefaultpalette*, který je v C++ deklarován jako funkce vracející ukazatel na **struct** *palettetype*, kdežto v Pascalu jako procedura (i když manuál i nápověda tvrdí opak) s výstupním parametrem typu *palettetype*.

Pokud se zajímáme o aktuální nastavení, použijeme proceduru *getpalette*, jejímž výstupním parametrem je právě hledaná paleta, resp. ukazatel na ni. Nezajímá-li nás paleta celá, ale prahneme-li pouze po počtu barev, které je možno v aktuálním grafickém režimu zobrazit najednou, vystačíme s funkcí *getpalettesize*, jejíž celočíselná návratová hodnota tento počet vyjadřuje.

Jak víme, jsou čísla barev vlastně pořadová čísla nastavované barvy v paletě. K ovládní barvy „pera“ (tj. barvy, kterou kreslíme) nám slouží tři podprogramy: funkce *getmaxcolor*, která nám vrátí nejvyšší číslo barvy, které můžeme v daném grafickém režimu zadat. (Dáváte-li pozor, víte, že toto číslo je o jedničku menší než velikost palety.) Aktuální barvu pera zjišťujeme funkcí *getcolor* a nastavujeme procedurou *setcolor*. Při zadávání barev se zpravidla používají hodnoty výčtového typu *COLORS*, definovaného takto (v Pascalu jsou k dispozici stějnomené konstanty):

```
enum COLORS {
    BLACK,           // 0 - Černá
    BLUE,            // 1 - Modrá
    GREEN,           // 2 - Zelená
    CYAN,            // 3 - Azurová
    RED,             // 4 - Červená
    MAGENTA,        // 5 - Purpurová
    BROWN,          // 6 - Hnědá
    LIGHTGRAY,     // 7 - Světle šedá
    DARKGRAY,      // 8 - Tmavě šedá
    LIGHTBLUE,     // 9 - Světle modrá
    LIGHTGREEN,    //10 - Světle zelená
    LIGHTCYAN,     //11 - Světle azurová
```

```

LIGHTRED,           //12 - Světle červená
LIGHTMAGENTA,      //13 - Světle purpurová
YELLOW,            //14 - Žlutá
WHITE              //15 - Bílá
};

```

Podobně jako barvu pera můžeme zjišťovat a nastavovat i barvu pozadí – tj. barvu, kterou bude mít obrazovka poté, co ji smažeme. Pro zjišťování barvy pozadí nám slouží funkce *getbkcolor* a pro její nastavování procedura *setbkcolor*.

Vraťme se ještě k paletám karet EGA a VGA. Procedura *initgraph* přiřadí jednotlivým registrům palety počáteční hodnoty, které jsou souhrnně definovány ve výčtovém typu *EGA_COLORS*.

```

enum EGA_COLORS {
    EGA_BLACK = 0,           //Černá
    EGA_BLUE = 1,           //Modrá
    EGA_GREEN = 2,          //Zelená
    EGA_CYAN = 3,           //Azurová
    EGA_RED = 4,            //Červená
    EGA_MAGENTA = 5,        //Purpurová
    EGA_BROWN = 20,         //Hnědá
    EGA_LIGHTGRAY = 7,      //Světle šedá
    EGA_DARKGRAY = 56,      //Tmavě šedá
    EGA_LIGHTBLUE = 57,     //Světle modrá
    EGA_LIGHTGREEN = 58,    //Světle zelená
    EGA_LIGHTCYAN = 59,     //Světle azurová
    EGA_LIGHTRED = 60,      //Světle červená
    EGA_LIGHTMAGENTA = 61,  //Světle purpurová
    EGA_YELLOW = 62,        //Žlutá
    EGA_WHITE = 63          //Bílá
};

```

Všimněte si, že v počátečním stavu odpovídá význam konstant datového typu *COLORS* nastaveným barvám odpovídajících registrů. Pokud však při nastavování barev palety nebudete pouze upravovat odstíny, ale budete zásadně měnit barvy palety, mnemotechnický význam těchto konstant přestane platit a může se pak stát, že příkazem

```
setcolor( YELLOW );
```

kterým jste chtěli pro pero nastavit žlutou barvu, nastavíte barvu brčálově zelenou nebo naopak starorůžovou.

Vraťme se ještě k nastavování barev pozadí. Řekli jsme, že příkazem

```
setbkcolor( X );
```

nastavíme u EGA a VGA karty barvu pozadí na barvu v *X*-tém registru palety. To znamená, že se barva nastavená v *X*-tém registru zkopíruje do nultého registru, protože barvou pozadí je vždy barva nultého registru. To platí s jedinou výjimkou. Pokud je *X* = 0, nastaví se hodnota nultého registru na nulu – čímž se barva pozadí nastaví na černou.

Abychom si mohli vyzkoušet, co jsme se o barvách a jejich nastavování dozvěděli, rozšíříme své znalosti ještě o tři podprogramy. První z nich bude procedura *cleardevice*,

kteřá slouží ke smazání obrazovky. Na počátku programu ji však mazat nemusíme – to za nás udělá procedura *initgraph*. Pokud nám však záleží na tom, jakou bude mít prázdná obrazovka barvu, nezbude nám, než proceduru *cleardevice* použít i na počátku programu.

Druhá bude procedura *putpixel*, kterou zobrazíme nejjednodušší útvar, jaký můžeme nakreslit: bod. V prvních dvou parametrech předáme této proceduře souřadnice *x* a *y* vykreslovaného bodu a ve třetím parametru informace o barvě, kterou se má bod zobrazit.

Na bod se můžeme – na rozdíl od ostatních grafických objektů, o nichž zde budeme hovořit – i zeptat. K tomu nám slouží funkce *getpixel*, která přečte z videopaměti informaci o barvě bodu, jehož souřadnice jsou jejím prvním a druhým parametrem, a vrátí ji jako funkční hodnotu.

Následující jednoduchý prográmek nám umožní pohybovat pomocí kurzorových šipek čtverečkem, který za sebou bude zanechávat barevnou stopu. Kdykoliv v průběhu pohybu můžeme zadáním čísla změnit velikost tohoto čtverečku a zadáním velkého písmene změnit barvu popředí (nulťá barva se zadává jako šnek – @). Běh programu ukončíte stiskem klávesy ESC. Vyzkoušejte si, jak se jednotlivé funkce chovají na vašem počítači. (Před překladem je opět nutno změnit cestu k adresáři s ovladčem.)

```

/* Příklad C7 - 3 */
#include <graphics.h>
#include <iostream.h>
#include <conio.h>

int mx, my;

void /*****/ Bod /*****/
( int x, int y, int s )
//Nakreslí čtverec o síle hrany s kolem bodu [x;y]
{
    int c = getcolor();
    for( int ix = x-s; ix <= x+s; ix++ )
        for( int iy = y-s; iy <= y+s; iy++ )
            putpixel( ix, iy, c );
}
/***** Bod *****/

const ESC = 0x1B;
const NAHORU = 0x48;
const DOLU = 0x50;
const DOLEVA = 0x4B;
const DOPRAVA= 0x4D;

void /*****/ SunBod /*****/ ( )
//Stěhuje čtvercovým kurzorem podle pokynů z klávesnice
{
    int x = mx / 2; //Střed obrazového pole
    int y = my / 2;
    int v = 0; //Velikost rámu kolem bodu
    int mc = getmaxcolor();
    char c;
    cout << "Barva může nabývat hodnot od 0 do "
        << mc << endl;
}

```

```

Bod( x, y, 0 ); //Nakresli samotný bod
while( ( c = getch() ) != ESC ) //Končíme stiskem Esc
{
    if( c ) //Jedná se o běžný, nenulový kód
    {
        if( c >= '@' ) //Barvu zadáváme velkými písmeny:
            { //@=0, A=1, B=2, ..., N=14, O=15
                c -= '@';
                if( (0 <= c) && (c <= mc) )
                    setcolor( c );
            }
        else if( c >= '0' ) //Velikost čtverce zadáváme
            { //číslicemi 0..9
                c -= '0';
                if( (0 <= c) && (c <= 9) )
                    v = c;
            }
    }
    else //Rozšířený kód s 1. bajtem nulovým
    {
        switch( getch() ) //Čteme 2. část rozšířeného kódu
        {
            case NAHORU: y--; break;
            case DOLU: y++; break;
            case DOLEVA: x--; break;
            case DOPRAVA: x++; break;
        }
    }
    Bod( x, y, v ); //Namalovat na nové pozici
} //while
}
/***** SuňBod *****/

void main(){
    initgraph(&(mx=0),&my, "\\bc31\\bgi");
    mx = getmaxx();
    my = getmaxy();
    SunBod();
    closegraph();
}

```

Pascalskou verzi tohoto programu najdete na doplňkové disketě v souboru *P7-03.PAS*.

7.4 Grafický kurzor

Grafický kurzor funguje naprosto stejně jako textový: označuje místo, kde skončila poslední akce a kde tedy bude pokračovat akce následující. Na rozdíl od textového kurzoru však není grafický kurzor na obrazovce znázorněn. Pokud byste ho chtěli vidět, museli byste se o to postarat sami programově (např. procedurou *Bod* z posledního programu).

Stejně jako v textovém režimu potřebujeme i v grafickém režimu občas zjistit, kde se zrovna grafický kurzor nachází, nebo jej naopak někam umístit. Polohu grafického kurzoru zjišťujeme prostřednictvím funkcí *getx* a *gety*, které nemají žádné parametry a vracejí celočíselnou hodnotu udávající souřadnici grafického kurzoru v daném směru.

Polohu grafického kurzoru můžeme nastavit dvěma různými způsoby. Buď pomocí procedury *moveto*, které předáme přímo souřadnice x a y požadované polohy (relativně vůči počátku výřezu), nebo pomocí procedury *moverel*, které předáme požadovanou změnu souřadnic grafického kurzoru – mohli bychom říci, že jí předáváme souřadnice relativní vůči aktuální pozici kurzoru.

7.5 Kreslení

Hlavním důvodem práce v grafickém režimu bývá většinou potřeba nakreslit nějaký obrázek. K tomuto účelu poskytují obě grafické knihovny bohatou škálu užitečných procedur a funkcí.

U bodu jsme si mohli vybrat, jakou barvou jej necháme zobrazit. Barvy všech ostatních útvarů, o nichž budu za chvíli hovořit, se kreslí v předem nastavené barvě pera. O jejím nastavování jakož i o některých souvisejících problémech jsme hovořili v oddílu 7.3.

Základním netriviálním zobrazovaným útvarem je úsečka. Pro její vykreslení nám knihovna nabízí tři procedury. Procedura *line* vyžaduje čtyři parametry, v nichž jí předáme souřadnice jejích krajních bodů.

Vedle toho můžeme použít proceduru *lineto*, která vyžaduje pouze dva parametry se souřadnicemi koncového bodu. Za počáteční bod se totiž považuje aktuální pozice grafického kurzoru. Obdobně se chová i procedura *linerel*. I jejími parametry jsou souřadnice koncového bodu úsečky, avšak tentokrát zadané relativně vůči jejímu počátku, tj. vůči aktuální poloze grafického kurzoru.

Tvůrci borlandské grafické knihovny věděli, že většina uživatelů s jednoduchými čarami nevystačí, a snažili se jim umožnit blíže specifikovat své okamžité požadavky. K tomu slouží procedura *setlinestyle*, která očekává v prvních dvou parametrech specifikaci podoby čáry a ve třetím parametru její tloušťku v bodech. Manuál však pro třetí parametr uvádí pouze možnost zadání hodnoty 1 pro tenkou čáru a 3 pro tlustou čáru, a to nejlépe pomocí konstant *NormWidth* a *ThickWidth* (Pascal), resp. *NORM_WIDTH* a *THICK_WIDTH* (C++). Zkoušeli jsem zadat i jiná čísla, avšak čáry měly nakonec vždy pouze jednu z těchto dvou tloušťek.

Podívejme se ale blíže na prvé dva parametry, které specifikují podobu čáry. Hodnoty prvního z nich najdete v tabulce 7.1.

Hodnota	Pascal	C++	Význam
0	SolidLn	SOLID_LINE	Plná čára
1	DottedLn	DOTTED_LINE	Tečkovaná čára
2	CenterLn	CENTER_LINE	Čerchovaná čára
3	DashedLn	DASHED_LINE	Čárkovaná čára
4	UserBitLn	USERBIT_LINE	Uživatелеm definovaná čára

Tab. 7.1 Konstanty, specifikující podobu čáry

Pokud má první parametr hodnotu 0 – 3, hodnota druhého parametru se ignoruje. Druhý parametr vstupuje do hry až ve chvíli, kdy první parametr nabude hodnoty 4 (pascalský manuál tvrdí, že navíc musí mít třetí parametr (tloušťka čáry) hodnotu 1 nebo 3). V tom případě obsahuje 16bitové slovo definující průběh kreslení čáry, kde každá jednička znamená rozsvícení a 0 nerozsvícení bodu na čáře. Čára se pak kreslí cyklickou aplikací tohoto vzoru. Budete-li tedy chtít definovat plnou čáru, zadáte \$FFFF, resp. 0xFFFF. Budete-li chtít definovat přerušovanou čáru, můžete zadat (uvádím jen pascalskou syntax) \$3333, \$7777, \$F0F0 nebo jakoukoliv další hodnotu. Budete-li chtít kreslit „divnou“ čáru, zadáte odpovídající „divnou“ hodnotu.

Stejně jako v předchozích nastaveních, i u typu čar máme k dispozici možnost zeptat se na aktuální nastavení. K dotazu využijeme procedury *getlinesettings*, která má jeden výstupní parametr volaný referencí (Pascal), resp. ukazatelem. Tento parametr je typu *linesettingstype*, který je v Pascalu definován

```
type
  LineSettingsType = record
   LineStyle: word;
    Pattern: word;
    Thickness: word;
  end;
```

a v C++

```
struct linesettingstype {
  int linestyle;
  unsigned upattern;
  int thickness;
}
```

Jak vidíte, pořadí složek v této struktuře je stejné jako pořadí parametrů v proceduře *setlinesettings*.

Na závěr vyprávění o nastavování typu čar již zbývá jen dodat, že nastavený typ čáry bude respektován nejen při kreslení úseček, ale i útvarů z úseček sestávajících. Knihovna vám pro kreslení takovýchto útvarů nabízí dvě procedury: proceduru *rectangle*, která nakreslí obdélník o zadaných souřadnicích stran (parametry jsou souřadnice levé, horní, pravé a dolní strany) a proceduru *drawpoly*, která kreslí obecnou bmenou čáru.

Procedura *drawpoly* má dva parametry: počet vrcholů a ukazatel na datovou strukturu se souřadnicemi těchto vrcholů. V C++ je touto strukturou vektor celých čísel. V Pascalu

její přesný datový typ definován není (jedná se o netypový parametr předávaný odkazem), ale musíte ji definovat tak, aby s ní mohla procedura pracovat jako s vektorem celých čísel.

Chcete-li touto procedurou nakreslit n úseček, musí mít první parametr hodnotu $n+1$, protože n úseček je definováno $n+1$ body. Druhý parametr pak musí ukazovat na vektor s $2*(n+1)$ hodnotami, které postupně představují souřadnice x a y jednotlivých vrcholů. Chcete-li nakreslit uzavřený útvar, např. n -úhelník, musí být souřadnice posledního vrcholu totožné se souřadnicemi vrcholu prvního.

Pokud si chcete na svém počítači vyzkoušet, jak bude systém reagovat na různá nastavení čar, zkuste si následující prográmek:

```

/* Příklad C7 - 4 */
#include <graphics.h>
#include <iostream.h>

void /*****/ Cary /*****/ ()
{
    int dx = getmaxx();
    struct linesettingstype ls;
    cleardevice();
    unsigned m, t;
    do
    {
        linerel( dx, 0 );
        linerel( 0, 5 );
        dx = -dx;
        getlinesettings( &ls );
        cout << "Čára: " << dec << ls.linestyle << ", "
             << hex << ls.upattern << ", "
             << dec << ls.thickness << endl;
        cout << "Nová podoba čáry: ";
        cin >> hex >> m >> dec >> t;
        if( m > 3 )
            setlinestyle( 4, m, t );
        else
            setlinestyle( m, 0, t );
    } while( t != 0 );
}
/***** Cary *****/

void main(){
    int g=0, d;
    initgraph(&g, &d, "\\bc31\\bgi");
    Cary();
    closegraph();
}

```

Pascalskou verzi tohoto programu najdete na doplňkové disketě v souboru *P7-04.PAS*.

Pozor, verze v C++ očekává zadání prvního čísla v šestnáctkové soustavě **bez prefixu 0x** (tedy např. F0A1), zatímco v Pascalu je třeba první číslo zadat s prefixem '\$', tedy např.

\$FOA1. Druhý parametr zadáváme pro změnu v desítkové soustavě. Program skončí, je-li druhý parametr nulový.

Další útvar, který budete chtít pomocí knihovních funkcí umět nakreslit, bude určitě kruh. Ten se kreslí procedurou *circle*, která má tři parametry: souřadnice x a y středu a poloměr. Při kreslení kruhu se nerespektuje nastavená podoba čáry, bere se v úvahu pouze její tloušťka.

Pokud nepoužíváme grafickou kartu VGA nebo SVGA, nebude asi vzhledem k charakteristice obrazovky nakreslený kruh vypadat opravdu jako kruh. Proto nám knihovna nabízí možnost nastavit poměr šířky a výšky obrazovky (stranový poměr, *aspect ratio*) prostřednictvím funkce *setaspectratio*, jejíž dva parametry získáme změřením délek stran obdélníka, který vznikl nakreslením čtverce.

Při inicializaci grafického systému procedura *initgraph* tento poměr nastavuje v závislosti na zvoleném grafickém režimu, ale není vyloučeno, že toto nastavení nebude přesně odpovídat charakteristice právě používaného displeje. Funkce *setaspectratio* nám umožní nastavený poměr doladit.

Pokud chceme zjistit hodnoty, které jsou v daný okamžik pro kruhovou korekci nastaveny, použijeme proceduru *getaspectratio*. Tato procedura má dva výstupní celočíselné parametry, kterým hledané hodnoty přiřadí.

Grafická knihovna využívá nastavený stranový poměr pouze při kreslení kruhu. Pokud však chceme, aby i ostatní útvary (zejména čtverce) dodržovaly tento stranový poměr, musíme jejich ypsilonové souřadnice odpovídajícím způsobem přepočítávat (jak se můžete přesvědčit z následujícího příkladu, pravolevý rozměr kruhu – ve směru souřadnice x – nastavený stranový poměr neovlivňuje). Teprve potom máme zajištěno, že např. kružnice vepsaná do čtverce se na displeji opravdu zobrazí jako kružnice vepsaná do čtverce.

```
(* Příklad P7 - 5 *)
uses graph;

var
  R: integer;
  x1, x2, y1, y2: integer;
  ax, ay: word;
  rr: integer;

procedure (*****) Kruhy (*****);
begin
  R := getmaxy div 4;
  x1 := getmaxx div 4;
  x2 := 3*x1;
  y1 := getmaxy div 3;
  y2 := y1;
  repeat
    cleardevice;
    getaspectratio( ax, ay );
    writeln('Aspect: ', ax, ' x ', ay);
    rr := (ax div ay) * R;
    circle( x1, y1, R );
    rectangle( x1-R, y1-R, x1+R, y1+R );
```

```

circle( x2, y2, R );
rectangle( x2-R, y2-rr, x2+R, y2+rr );
writeln('Nový poměr: (dvě čísla, 0 = konec) ');
read(ax, ay);
if( ax*ay <> 0 ) then
    setaspectratio( ax, ay );
until ( ax = 0 );
end;
(***** Kruhy *****)
var
    g, d: integer;
begin
    g := 0;
    initgraph(g, d, '\aplikace\prekl\bp\bgi');
    Kruhy;
    closegraph;
end.

```

Céčkovskou variantu tohoto programku najdete na doplňkové disketě ve zdrojovém souboru *C7-05.CPP*.

Na kreslení křivých čar nabízí grafická knihovna ještě dvě další procedury. Procedura *arc* nakreslí část kruhového oblouku. Její parametry jsou všechny celočíselné; jsou to souřadnice středu tohoto oblouku, počáteční a koncový úhel oblouku zadávaný ve stupních a poloměr oblouku. Při kreslení kruhového oblouku se respektuje nastavený stranový poměr.

Úhly se měří tak, jak jsme z matematiky zvyklí, tj. proti směru hodinových ručiček. Úhel číslce 3 vůči středu ciferníku je 0° , úhel čísla 12 je 90° , úhel čísla 6 je 270° nebo -90° .

Proti směru hodinových ručiček se postupuje i při kreslení kruhového oblouku. Chcete-li tedy nakreslit oblouk, který by na hodinách procházel čísly 9, 10, 11, 12, musíte zadat počáteční úhel 90° a koncový úhel 180° . Chcete-li naopak, aby oblouk procházel číslicemi 12, 1, 2, ..., 8, 9, musíte zadat počáteční úhel 180° a koncový úhel 90° .

Kromě kruhových oblouků umožňuje knihovna i kreslení eliptických oblouků – ovšem pouze pro elipsy, jejichž osy jsou rovnoběžné se souřadnicovými osami. K tomu slouží procedura *ellipse*, které zadáte souřadnice středu elipsy, počáteční a koncový úhel oblouku a délku vodorovné a svislé poloosy.

Při kreslení eliptických kruhových oblouků se nerespektuje nastavený stranový poměr, takže pokud zadáte stejnou velikost vodorovné a svislé poloosy, nakreslí se na obrazovce nezávisle na nastaveném stranovém poměru takový útvar (kruh nebo elipsa), jaký vám nakreslí procedura *circle* při stranovém poměru 1:1.

Stejně jako při kreslení kruhu, ani při kreslení kruhového a eliptického oblouku se nerespektuje nastavená podoba čáry (tj. plná, čárkovaná, tečkovaná apod.), a jediné, co se bere v úvahu je požadovaná tloušťka čáry.

|| /* Příklad C7 - 06 */

```

//Eliptický a kruhový oblouk
#include <iostream.h>
#include <graphics.h>
#include <conio.h>

void /*****/ Elipsa /*****/ ()
{
    int x = getmaxx() / 4;
    int y = getmaxy() / 2;
    int xx = x;
    int a1, a2, yy;
    double ar;
    do
    {
        cleardevice();
        cout << "Pomer x/y: ";
        cin >> ar;
        if( ar == 0 ) return;
        setaspectratio( 1000*ar, 1000 );
        cout << "xx=" << xx << "; druhá poloosa a úhly: yy, a1, a2: ";
        cin >> yy >> a1 >> a2;
        if( yy == 0 ) yy = xx;
        setlinestyle( 0, 0, NORM_WIDTH );
        putpixel(x,y,WHITE);
        arc( x, y, a1, a2, xx );
        setlinestyle( 0, 0, THICK_WIDTH );
        ellipse( 3*x, y, a1, a2, xx, yy );
        getch();
    } while( 1 );
}
/***** Elipsa *****/

void main(){
    int d = 0,g;
    initgraph(&d, &g, "\\aplikace\\prekl\\bc31\\bgi");
    Elipsa();
    closegraph();
}

```

7.6 Okna

Podobně jako jsme mohli v textovém režimu definovat na obrazovce okno, můžeme v grafickém režimu definovat **okno** – viewport (a stejně jako v textovém režimu na něj standardní výstup nereaguje). Okno slouží ke dvěma účelům: za prvé jako referenční bod, tzn. veškeré adresování je relativní vůči počátku okna, a za druhé jako definice oblasti viditelnosti zobrazovaných struktur. Na rozdíl od textových oken si však u okna v grafickém režimu můžeme nastavit, zda bude výřez oblast viditelnosti opravdu omezo- vat či nikoliv.

Zde je potřeba dodat, že veškeré souřadnice grafického kurzoru (zjišťované i nastavované), o nichž budeme hovořit, jsou vždy **relativní vůči počátku aktuálního okna**

Okno nastavujeme procedurou *setviewport*, jejíž první čtyři parametry definují postupně souřadnice levé, horní, pravé a dolní hrany, a pátý parametr pak nastavuje nebo potlačuje ořezávání zobrazených tvarů na hranici okna. Musíme však upozornit na skutečnost, že **souřadnice grafického kurzoru nejsou oknem omezeny**. Jinými slovy „vytečení“ obrázku z okna není chyba, pouze se při nastaveném ořezávání přeteklá část nezobrazí. Jiný vliv na grafické operace ořezávání nemá.

K tomu, abychom se dozvěděli, jaké okno je v danou chvíli nastaveno, nám slouží procedura *getviewsettings*, jejímž výstupním parametrem je proměnná datového typu *viewporttype*. Tento datový typ je v Pascalu deklarován

```
type
  ViewPortType = record
    x1: integer;      {Souřadnice x levé hrany}
    y1: integer;      {Souřadnice y horní hrany}
    x2: integer;      {Souřadnice x pravé hrany}
    y2: integer;      {Souřadnice y dolní hrany}
    clip: boolean;    {Ořezávat "přetékající" části?}
end;
```

a v C++

```
struct viewporttype {
  int left;           //Souřadnice x levé hrany
  int top;            //Souřadnice y horní hrany
  int right;          //Souřadnice x pravé hrany
  int bottom;         //Souřadnice y dolní hrany
  int clip;           //Ořezávat "přetékající" části?
}
```

Souřadnice hran okna jsou udány absolutně, tj. vůči počátku celé obrazovky. Kromě zjištění charakteristiky a nastavení okna máme k dispozici ještě jednu operaci: smazání okna pomocí procedury *clearviewport*.

7.7 Výřezy

Abychom mohli i v grafickém režimu zabezpečit překrývání obrázků, umožňuje nám grafická knihovna uložit obsah definované části obrazovky (říkejme jí **výřez**) do paměti a zase jej obnovit. K tomuto účelu jsou připraveny dvě procedury a jedna funkce.

Chceme-li uložit výřez do paměti, musíme nejprve vědět, kolik paměti pro něj potřebujeme vyhradit. To nám poví funkce *imagesize*, které předáme po řadě souřadnice levé, horní, pravé a dolní hrany ukládaného výřezu. Funkce nám vrátí velikost potřebné paměti zvětšenou o 6 B (4 bajty vyhrazené k uložení výšky a šířky přenášené oblasti a další dva rezervované bajty). Pokud je však tato potřebná velikost paměti větší než 65535 bajtů, vrátí pascalská verze nulu a céčkovská hodnotu 0xFFFF.

Jakmile známe velikost potřebné paměti, můžeme ji vyhradit a pak do ní pomocí procedury *getimage* uložit obsah požadované oblasti – výřezu. Tato procedura má pět para-

metrů: souřadnice levého, horního, pravého a dolního okraje výřezu a adresu vyhrazené oblasti paměti.

Chceme-li uložený výřez opět vykreslit, nemusíme jej vracet na totéž místo, kde jsme jej předtím přečetli. Parametry procedury *putimage*, která má vykreslení uložených výřezů na starost, jsou souřadnice x a y levého horního rohu budoucí polohy výřezu, adresa místa v paměti, kde je obsah výřezu uložen, a způsob překreslení uloženého výřezu na obrazovku.

Možných způsobů překreslení je celkem 5 a zadávají se pomocí následujících předdefinovaných konstant (nejprve je vždy uvedena hodnota, pak identifikátor konstanty v Pascalu a nakonec identifikátor konstanty v C++):

Hodnota	Pascal	C++	Význam
0	CopyPut	COPY_PUT	Vykreslovaný výřez překryje původní obsah obrazovky.
1	XORput	XOR_PUT	Program sloučí původní a nový obsah pomocí operace XOR, tj. změni v původním obsahu videopaměti hodnotu těch bitů, které jsou ve vykreslovaném výřezu nastaveny (mají hodnotu 1).
2	ORPut	OR_PUT	Program sloučí původní a nový obsah operací OR, tj. nastaví na hodnotu 1 v původním obsahu videopaměti navíc všechny bity, které jsou ve vykreslovaném výřezu nastaveny (mají hodnotu 1).
3	ANDPut	AND_PUT	Program sloučí původní a nový obsah operací AND, tj. vynuluje v původním obsahu videopaměti všechny bity, které nejsou ve vykreslovaném výřezu nastaveny (mají hodnotu 0).
4	NOTPut	NOT_PUT	Původní obrázek bude překryt negací nového obrázku, přesněji obrázkem vzniklým negací vnitřní reprezentace nového obrázku.

Tab. 7.2 Způsob překreslení

7.8 Výstup textu

Výstup textu na obrazovku můžete realizovat několika způsoby:

1. Prostřednictvím standardního výstupu, který sice nabízí možnost formátování všech typů dat (v Pascalu pouze všech pěti standardních typů), ale neposkytuje žádnou možnost ovlivnit podobu tištěného textu a dokonce ani jeho umístění na obrazovce (to se

však dá za jistých podmínek obejít). Standardní výstup navíc nerespektuje nastavená zorná pole.

2. Prostřednictvím grafických funkcí *outtext* a *outtextxy*, které vám sice nabízejí veškeré možnosti ovlivnění podoby vystupujícího textu, ale připraví vás o možnost formátování, protože umějí zase vytisknout pouze předem připravený textový řetězec, který jim zadáte jako parametr (*outtext* tiskne text na aktuální pozici kurzoru, *outtextxy* očekává v druhém a třetím parametru souřadnic *x* a *y* vztažného bodu pro zarovnání textu).
3. Výhody obou předchozích možností můžeme spojit, pokud budeme v grafickém režimu tisknout „s mezidechem“ – přes paměťové soubory a proudy, pomocí nichž si text naformátujeme a výsledný řetězec vytiskneme s využitím všech možností, které grafická knihovna nabízí. Nevýhodou tohoto řešení je ten nutný „mezidech“ a z něj vyplývající zbytečná komplikovanost.

Textový kurzor v grafickém režimu

S textovým kurzorem standardního vstupu a výstupu můžeme pracovat, a to i v grafickém režimu, jestliže v souboru CONFIG.SYS instalujeme při spuštění operačního systému ovladač ANSI.SYS příkazem

```
DEVICE=XXX\ANSI.SYS
```

(XXX je cesta k souboru s ovladačem). Práce s tímto ovladačem je pak jednoduchá: každý výstup, který začíná znaky s kódovými čísly 27 (šestnáctkově 1B, odpovídá stisknutí klávesy ESC) a 91 (tj. 5B, znak '[') se pokládá za příkaz pro tento ovladač. Např. příkaz pro umístění kurzoru na zadanou pozici má tvar

```
Esc[nr;nsH
```

kde *nr* je číslo řádku a *ns* je číslo sloupce. (Podrobný popis tohoto ovladače najdete v dokumentaci k operačnímu systému nebo – od verze 5.0 – v nápovědě k němu.)

Ukážeme si jednoduchý program v Pascalu, který bude v grafickém režimu nastavovat pozici textového kurzoru:

```
(* Příklad P7 - 6 *)
uses graph;
{Přímé adresování textového kurzoru v grafickém režimu pomocí
 ANSI.SYS}
procedure (*****) gotoxy (*****)
{Umístí textový kurzor na absolutní pozici na obrazovce}
(x, y: integer);
begin
  write( #27['', x, ';', y, 'H' ); {Umístí textový kurzor}
end;
(***** gotoxy *****)
procedure (*****) gotorel (*****)
{Posune kurzor relativně vůči aktuální pozici}
```

```

( x, y: integer );
begin
  if( x > 0 )then
    write( #27'[', x, 'C' )      {Posun kurzoru doprava}
  else if( x < 0 )then
    write( #27'[', abs(x), 'D' ); {Posun kurzoru doleva}
  if( y > 0 )then
    write( #27'[', y, 'B' )      {Posun kurzoru dolů}
  else if( y < 0 )then
    write( #27'[', abs(y), 'A' ); {Posun kurzoru nahoru}
end;
(***** gotorel *****)
var a, b: integer;
begin
  a := 0;
  initgraph(a, b, '\aplikace\prekl\bp\bgi\');
  write('qwertyu');
  gotoxy(10,10);
  write('asdfg');
  gotorel(-15, 5);
  write('zxcvb');
  gotorel(25, -5);
  write('lkjhg');
  lineto(100,100);                {Hrátky s textovým kurzorem
                                   neovlivní grafický kurzor}
  closegraph;
end.

```

V C++ si ukážeme jen tvar funkcí *gotoxy()* a *gotorel()*. Analogii příkladu P7 – 06 najdete v souboru *C7–06.CPP* na doplňkové disketě.

```

void /****/ gotoxy /*****/
// Umístí textový kurzor na absolutní pozici na obrazovce
( int x, int y )
{
  cout << "\x1B[" << x << ';' << y << 'H' ; //Umístí textový kurzor
}
/*****/ gotoxy *****/

void /****/ gotorel /*****/
//Posune kurzor relativně vůči aktuální pozici
( int x, int y )
{
  if( x > 0 )
    cout << "\x1B[" << x << 'C';           //Posuň kurzor
  doprava
  else if( x < 0 )
    cout << "\x1B[" << abs(x) << 'D';       //Posuň kurzor doleva
  if( y > 0 )
    cout << "\x1B[" << y << 'B' ;           //Posuň kurzor
  dolů
  else if( y < 0 )
    cout << "\x1B[" << abs(y) << 'A';       //Posuň kurzor nahoru
}

```

```
||/***** gotorel *****/
```

Grafický výstup textu

Tiskneme-li procedurou *outtext* vodorovně s nastaveným zarovnáváním vlevo (*LEFT_TEXT*), posune se v průběhu tisku i grafický kurzor. Tiskneme-li jakkoliv jinak, poloha grafického kurzoru se během tisku nemění.

V manuálu i v nápovědě (Help) se nám snaží namluvit, že procedura *outtextxy* nemá na nastavení polohy grafického kurzoru vliv. Není to pravda! Když po zobrazení textu

```
outtextxy(100,100,"Text.");
```

nakreslíme čáru

```
lineto(200,200);
```

nebude tato čára vycházet z předchozí pozice grafického kurzoru, ale právě z bodu (100,100). Naproti tomu vůči následujícím tiskům procedurou *outtext* se chová naprosto podle předpokladů. Musíme proto kreslení čar a tisky kombinovat opatrně.

Pokud tiskneme vektorovým písmem, budou části sahající mimo zorné pole zaříznuty v závislosti na nastavení ořezávání stejně jako jakékoliv jiné grafické obrazce. Pokud tiskneme bitovým písmem, je situace trochu odlišná. Není-li zařezávání povoleno, text se zobrazí celý, je-li povoleno, zobrazí se pouze znaky, které celé zasahují do zorného pole.

7.9 Práce s videostránkami

Na závěr povídání o grafickém systému se ještě zmíníme o dvou procedurách, které ocení majitelé grafických karet EGA a novějších. Jistě jste si všimli, že v popisu typu *graphics_modes* jsme u některých režimů „lepší“ videokaret uváděli, že nabízejí 2 nebo 4 stránky. To znamená, že máme v paměti k dispozici prostor pro 2 nebo 4 grafické obrazovky, které můžeme libovolně přepínat.

Ovládání je jednoduché. Procedurou *setactivepage* zadáme, která stránka bude aktivní, tj. ke které se budou od tohoto okamžiku vztahovat všechny grafické operace. Procedurou *setvisualpage* pak zadáme, která stránka bude viditelná, tj. která se objeví na monitoru. Můžeme totiž jednu stránku zobrazovat, ale pracovat mezitím s druhou, a teprve po provedení všech grafických operací tuto druhou stránku zobrazit. Často tím zamezíme nepříjemnému blikání obrazu (např. při častém přesouvání bloků pomocí funkcí *getimage/putimage*).

Pokud zadáme číslo neexistující stránky, nebudeme tisknout nikam a obraz nám zcela zmizí.

Některé z možností práce s grafickou knihovnou si můžete ověřit za pomoci programu Bludiště v souborech *BLUDISTE.CPP* resp. *BLUDISTE.PAS* na doplňkové disketě. Tento

program umožňuje definovat jednoduché bludiště při pohledu shora a potom do tohoto bludiště vstoupit a pokusit se z něj najít východ.

7.10 Vyplňování

Uzavřené obrazce mohou být „prázdné“ – tvořené jen obrysovými čarami – nebo vyplněné vzorem v předepsané barvě.

Vyplněné obrazce

Vyplněnou elipsu nakreslíme pomocí procedury *fillellipse*, jež nakreslí elipsu se středem v bodě (x, y) a s poloosami xr a yr . Vyplněnou eliptickou výseč nakreslí procedura *sector* a vyplněnou kruhovou výseč procedura *pieslice*. Jejich parametry mají stejný význam jako parametry procedur *ellipse* a *arc*.

Chceme-li nakreslit vyplněný mnohoúhelník, použijeme proceduru *fillpoly*, která se chová podobně jako *drawpoly*.

Chceme-li vyplnit jiný uzavřený obrazec nebo celou plochu obrazovky, použijeme funkci *floodfill*, které zadáme počáteční bod a barvu, která tvoří hranici vyplňované oblasti. Je-li počáteční bod uvnitř uzavřeného obrazce v barvě *hranice*, vyplní se tento obrazec, jinak se vyplní celá obrazovka s výjimkou uzavřených obrazců v dané barvě

Způsob vyplňování

Dosud jsme si neřekli, čím bude daná oblast vyplněna. Implicitně je to bílá barva; pokud chceme něco jiného, můžeme použít proceduru *setfillstyle*, jež definuje vzor a barvu výplně. Jako vzor můžeme předepsat jednu z konstant, uvedených v tabulce 7.3.

Jméno	Hodnota	Výplň
EMPTY_FILL	0	Barva pozadí
SOLID_FILL	1	Souvislá výplň danou barvou
LINE_FILL	2	Vodorovné čáry
LTSLASH_FILL	3	///
SLASH_FILL	4	///, silné čáry
BKSLASH_FILL	5	\\, silné čáry
LTBKSLASH_FILL	6	\\
HATCH_FILL	7	Mříž z tenkých čar
XHATCH_FILL	8	Mříž ze silných čar
INTERLEAVE_FILL	9	Prokládané čáry

Jméno	Hodnota	Výplň
WIDE_DOT_FILL	10	Body s velkými mezerami
CLOSE_DOT_FILL	11	Body s malými mezerami
USER_FILL	12	Uživatелеm definovaný vzor

Tab. 7.3 Konstanty, definující způsob výplně

Chceme-li použít vlastní vzor, musíme jej popsat pomocí osmi bajtů, které představují čtverec o velikosti 8×8 pixelů (jedničky svítící body, 0 tmavé body; v Pascalu jde o proměnnou typu *fillpatterntype*). Zadáme jej jako pole 8 znaků, resp. položek typu *shortint*, které předáme jako první parametr funkci *setfillpattern*. Druhý parametr této funkce popisuje opět barvu výplně.

Chceme-li naopak získat informaci o aktuálním vzoru pro vyplňování, zavoláme funkci *getfillpattern*, která uloží do daného pole znaků aktuální vzor.

Jako příklad napíšeme program, který nakreslí 11 elips, vyplněných předdefinovanými vzory, a zbytek výkresu vyplní uživatelským vzorem. Uvedeme jej pouze v C++, pascalskou verzi najdete na doplňkové disketě v souboru *P7-07.PAS*.

```

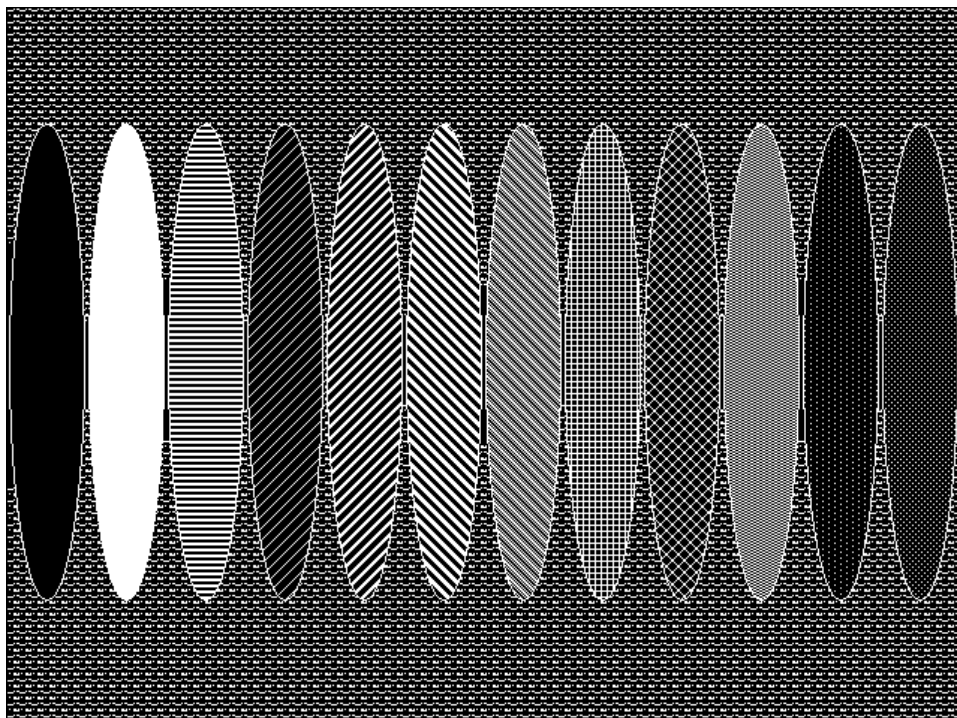
/* Příklad C7 - 7 */
#include <graphics.h>
#include <conio.h>

void main(){
    int a = 0, b;
    int mx, my;
    char vzor[8] = {0xf, 0, 0xff, 11, 0xf0, 0xf, 22, 77,};
    initgraph(&a,&b,"c:\\aplikace\\prekl\\bc31\\bgi");
    mx = getmaxx(); my = getmaxy();

    for(int i = 0; i < 12; i++){
        setfillstyle(i, WHITE);
        fillellipse(mx/12*i + mx/24, my/2, mx/25,my/3);
    }
    setfillpattern(vzor, YELLOW);
    floodfill(0,0, WHITE);
    getch();
    closegraph();
}

```

Výsledek ukazuje obr. 7.1.



7.1 Výsledek programu 7-7

8. Piškorky

Na doplňkové disketě najdete mimo jiné i několik rozsáhlejších programů; v této kapitole se podíváme na jeden z nich – *PISKORKY*. Pokusíme se ukázat vám postup vzniku takového programu od prvotního nápadu až po výslednou realizaci.

8.1 Úvodní kroky

Když se rozhodneme vytvořit nějaký program, první co musíme udělat, je důkladně si rozmyslet jeho koncepci. V našem případě je situace jednoduchá: program by měl hrát podle předem známých pravidel. Náš program vychází z následujících pravidel hry:

1. Hru hrají dva hráči – v našem případě **hráč** a **počítač**.
2. Hra se hraje na **čtvercové šachovnici** o předem zadaných rozměrech.
3. Hráči střídavě pokládají své kameny na políčka šachovnice, přičemž se snaží vytvořit tzv. **piškorku** a zároveň zabránit protihráči ve vytvoření jeho piškorky.
4. **Piškorka je uskupení pěti kamenů** na pěti sousedních polích šachovnice v **jedné řadě**, přičemž tato řada může být situována vodorovně, svisle i šikmo.
5. **Vítězí hráč**, který první sestaví svoji piškorku.
6. Pokud je šachovnice zaplněna kameny tak, že již není možno žádnou piškorku vytvořit, **končí hra nerozhodně**.

Kromě jasné definice pravidel bychom si měli v této fázi důkladně rozmyslet i způsob komunikace s uživatelem. Když jsme kdysi tento program psali, chtěli jsme se hlavně naučit pracovat s ukazateli jazyka C, a proto jsme uživatelské rozhraní trochu odbyli. Naše rozhraní zvýrazňovalo poslední tah počítače tím, že na toto pole umístilo kurzor. Hráči pak umožňovalo pohybovat kurzorem pouze pomocí kurzorových kláves, a to v osmi základních směrech (vodorovně, svisle a šikmo). Navíc jsou tu pouze dvě služby: stiskem některé z kláves F1, ?, h nebo H („help“) může hráč požádat počítač o nápovědu kam by měl další kámen umístit a stiskem klávesy ESC může předčasně ukončit hru.

Poznámka:

Poslední zmiňovanou možnost, tj. možnost předčasného ukončení, bychom chtěli obzvláště zdůraznit. Pamatujte ve svých programech vždy na to, aby bylo možno každou akci (a tím spíše celý program) předčasně přerušit! Sami možná víte z vlastní zkušenosti, jakou averzi získá člověk k programu, který je možno při nešikovném zadání (nebo dokonce vždy) opustit pouze klasickým trojhmatem CTRL-ALT-DEL.

Pokud bychom chtěli takovýto program nabídnout také svým známým, měli bychom komfort uživatelského rozhraní zvýšit přinejmenším o ovládání myši (o tom, jak toho do-

sáhnout, mluvíme v knize *Objektové programování 2*) a o možnost vrátit poslední tah nebo sérii několika posledních tahů, případně ještě o několik dalších drobností.

K základní koncepci programu, při jejímž vytváření odpovídáme především na otázku **co má program dělat**, bychom mohli přidat ještě mnohé, ale zůstaneme u toho, že základním cílem tohoto programu je naučit se pracovat s ukazateli, a přejdeme proto k další etapě, ve které zvolíme klíčové algoritmy a navrheme odpovídající základní datové struktury.

Nejprve se pokusme odvodit onen klíčový algoritmus. Víme, že piškorka se rozkládá na pěti sousedních polích. Každé z těchto polí však může v danou chvíli vystupovat v několika piškorkách. Pokud se zeptám v kolika, odpovíte nejspíše že ve čtyřech: vodorovné, svislé, šikmé ve směru hlavní diagonály a šikmé ve směru vedlejší diagonály. To je však pravda pouze částečně – záleží totiž na tom, jak si definujeme piškorku.

Pokud budeme piškorku chápat jako pěti kamenů na přesně dané pěti sousedních polí, pak je zřejmé, že políčko, které je dostatečně daleko od krajů šachovnice, může v každém směru vystupovat až v pěti piškorkách: v jedné jako první, v další jako druhé a v páté jako poslední políčko dané piškorky. Políčkem mohou procházet piškorky čtyřmi různými směry, dohromady tedy 20 piškorek.

Políčko pro nás bude tedy tím cennější, čím více piškorek jím bude moci procházet. Přiřadíme proto každému políčku takovou počáteční váhu, kolik piškorek jím bude moci procházet. Políčka ve středu šachovnice mohou tedy dosáhnout počáteční váhy až 20, políčka v rozích šachovnice budou mít váhu 3.

Jakmile se na nějakém políčku objeví kámen, ovlivní to váhu všech volných políček, které s ním tvoří jednu piškorku. Položením dalšího kamene na některé z těchto políček bychom totiž ještě zvýšili pravděpodobnost vytvoření piškorky. Volná políčka, která jsou součástí potenciální piškorky, která je již částečně sestavena, budou pro nás mít tím větší váhu, čím více kamenů již daná piškorka obsahuje. Můžeme to například udělat tak, že přidáním kamene zvýšíme váhu zbylých volných polí v piškorce o tolikátou mocninou dvou, kolikátý kámen na danou piškorku přidáváme.

Přidáváním kamenů se však poruší váhová symetrie pro oba hráče. Totéž políčko začne být jinak důležité pro hráče, jenž má již několik kamenů v piškorce, která daným políčkem prochází, a jinak důležité pro hráče, který v žádné z procházejících piškorek svůj kámen nemá. Nebudeme tedy hovořit o obecné váze daného políčka, ale o jeho útočných a obranných vahách pro jednotlivé hráče. (Útočná váha políčka pro hráče je zároveň obrannou vahou pro protihráče.)

Začíná se nám tedy rýsovat strategie, podle které by mohl počítač hrát. Pokud bude počítač na tahu, prohlédne si útočné a obranné váhy jednotlivých políček, a podle toho, zda pak bude hrát spíše útočně či obranně, se rozhodne, na které políčko umístí svůj další kámen. Kritériem pro jeho rozhodnutí by mohl být vážený součet (tj. součet, v němž je každý sčítanec násoben nějakým váhovým koeficientem určujícím, nakolik se daný sčítanec uplatní ve výsledném součtu) obou vah, přičemž vhodnou volbou koeficientů bychom

volili míru útočnosti či obrannosti daného algoritmu. Po každém tahu pak znovu upravíme váhy všech políček.

Pokud jste někdy piškorky hráli, víte, že se občas vyskytnou situace, na které musíte reagovat, nechcete-li vzápětí prohrát. Pokud má např. protihráč položené čtyři kameny v řadě, ke které je možno přiložit i pátý kámen, musíte pátý kámen přiložit vy, jinak bude mít protihráč v příštím tahu piškorku. Pokud náhodou budete mít čtyři kameny v řadě vy, nemusíte se rozmyslet nad vahami jednotlivých polí a můžete k řadě rovnou přiložit pátý kámen a tím vyhrát. Bylo by tedy vhodné, aby počítač při analýze pozice nejprve zjistil, zda náhodou neexistuje nějaký vyhrávající tah a bez dalšího zkoumání pozice jej hned realizoval.

Obdobná situace nastane v případě, kdy má hráč volnou trojici kamenů. Pokud mu tuto trojici protihráč včas z jedné strany neomezí, bude mít v příštím tahu volnou čtveřici a tím dva vyhrávající tahy – vítězství pak nebude možno zabránit. Přiložení čtvrtého kamene bychom v této situaci mohli vyhlásit za nutný tah a opět bychom mohli chtít po počítači, aby tyto tahy realizoval přednostně a nezávisle na vahách políček.

Výše uvedený algoritmus asi není nejlepší možný, ale je dostatečně jednoduchý na to, abychom se mohli soustředit na jiné rysy programu. Zkušenost přitom ukázala, že počítač podle něj nehraje nejhůř.

Vezměme jej tedy jako výchozí algoritmus a pokusme se vymyslet, jaké bychom pro jeho realizaci měli navrhnout optimální datové struktury. Nejprve si definujeme základní konstanty. (Vzhledem k velké podobnosti definic je uvedeme pouze v C++.)

```
const char Nic = '+'; //Znak prázdného políčka
const char ANic = 0x07; //Atribut prázdného políčka
const char ZHR = 'O'; //Znak pro hráče
const char AHR = 0x0C; //Atribut znaku hráče
const char ZPO = 'X'; //Znak pro počítač
const char APO = 0x0E; //Atribut znaku počítače
const RADKU = 20; //Počet řádků šachovnice
const SLOUPCU = 20; //Počet sloupců šachovnice
const PISKORKA = 5; //Počet kamenů tvořících piškorku
const MAX_NUT = 8; //Max. předpoklád. počet nadějných
    tahů
const MAX_VYH = 2; //Max. předpoklád. počet vyhrávaj.
    tahů
const PO_RA = RADKU-1; //Index posledního řádku
const PO_SL = SLOUPCU-1; //Index posledního sloupce
const PPP = PISKORKA-1; //Index posledního pole v piškorce
const POLI = RADKU*SLOUPCU; //Počet polí na šachovnici
const PISKOREK = //Počet všech možných piškorek
((RADKU-PPP)*SLOUPCU + //Svislých +
RADKU*(SLOUPCU-PPP) + //Vodorovných +
2*(RADKU-PPP)*(SLOUPCU-PPP)); //Šikmých / a \
const Nikdo = 0; //Příznak neobsazené piškorky
const BLOK = 0x7FFF; //Příznak neperspektivnosti políčka

//Kosmetická makra (= makra pro kosmetickou úpravu programu)
#define ef else if
#define word unsigned
```

Jediná věc, která by možná mohla vyvolat dohady, je výpočet počtu piškorek na šachovnici. Nejde o výpočet počtu piškorek, které mohou být najednou na šachovnici, ale o výpočet všech možných umístění piškorek. Pokud tedy máme S sloupců, které budeme číslovat od nuly do $S-1$, mohou být v řádku piškorky na pozicích $[0, 1, 2, 3, 4]$, $[1, 2, 3, 4, 5]$, $[2, 3, 4, 5, 6]$ atd. až do $[S-5, S-4, S-3, S-2, S-1]$ – celkem tedy $S-4$ vodorovných piškorek v každém řádku. Víme-li, že na šachovnici je R řádků, víme, že vodorovné piškorky mohou být v $R*(S-4)$ pozicích.

Podobným způsobem můžeme vypočítat i počet svislých piškorek.

Počet šikmých piškorek můžeme snadno odvodit z předchozích dvou. Vzhledem k jejich poloze stačí např. spočítat velikost obdélníku, v němž budou jejich horní počátky. Pro piškorky rovnoběžné s hlavní i vedlejší diagonálou bude platit, že je na šachovnici můžeme umístit v $(R-4)*(S-4)$ pozicích. Všech šikmých piškorek tedy může být právě $2*(R-4)*(S-4)$.

Nyní si definujeme několik pomocných datových typů. Za prvé to bude logický (v C++ výčtový) datový typ *LogH*, který nám umožní zůstat při zápisu logických výrazů v češtině.

Dalším pomocným datovým typem bude logický (v C++ opět výčtový) datový typ *THrác*. Pro jeho hodnoty zavedeme synonyma *POCITAC* a *HRAC*. Zavedení tohoto datového typu nám umožní jednoduchou operací logické negace přepínat mezi libovolným hráčem a jeho protihráčem.

Pak přistoupíme k definici dvou klíčových datových typů: typu *Tpole*, jehož instance budou popisovat charakteristiky jednotlivých polí šachovnice, a typu *TPišk*, jehož instance budou popisovat charakteristiky jednotlivých piškorek. Ukažme si nejprve zdrojové texty definic datových typů:

```
enum LogH {NE, ANO, _LogH };           //Logické hodnoty
enum THrac {POCITAC, HRAC, _THrac };
//Tento typ se používá jako logický, tj. !POCITAC == HRAC a naopak
struct TPisk;                          //Pouze předběžná deklarace
struct TPole                           //Charakteristika jednotlivého pole šachovnice
{
    char Obsah;                         //Nic nebo znak hráče
    int PoPrPi;                          //Počet tudy procházejících piškorek
    TPisk* ProPi[ PISKORKA*4 ];         //Adresy procházejících piškorek
    int Vaha [ _THrac ];                 //Důležitost pole pro oba hráče
};
typedef TPole *TuPole;                  //Ukazatel na TPole
struct TPisk                             //Charakteristika piškorky
{
    int Majitel;
    //0 = Nikdo
    //- = Hráč - hodnotou je záporný počet obsazených polí
    //+ = Počítač - hodnotou je počet obsazených polí piškorky
    //BLOK - piškorka je neperspektivní, protože už ji
```

```

    // nebude možno doplnit
    TPole* Proch[ PISKORKA ]; //Adresy polí, jimiž piškorka prochází
};

const
    NE = FALSE;
    ANO = TRUE;
    POCITAC = FALSE;
    HRAC = TRUE;

(***** Datové typy *****)
type
    TRadek = array[ 0..SLOUPCU-1 ] of TPole;
    UkTRadek = ^TRadek;
    LogH = boolean;
    THrac = boolean;
    UkTPisk = ^TPisk;
    UkUkTPisk = ^UkTPisk;
    UkTPole = ^TPole;
    UkUkTPole = ^UkTPole;

    TPole = record          {Charakteristika jednotlivého pole šachovnice}
        Obsah: char;          {Nic nebo znak hráče}
        PoPrPi: integer;      {Počet tudy procházejících piškorek}
        ProPi: array [ 0 ..PISKORKA*4 - 1 ] of UkTPisk;
                                {Adresy procházejících piškorek}
        Vaha: array [ THrac ] of integer;  {Důležitost pole pro hráče}
    end;

    TPisk = record          {Charakteristika piškorky}
        Majitel: integer;
            {0 = Nikdo}
            {- = Hráč - hodnotou je záporný počet obsazených polí}
            {+ = Počítač - hodnotou je počet obsazených polí piškorky}
            {BLOK - piškorka je neperspektivní, protože už ji}
            {nebude možno doplnit}
        Proch: array [ 0 ..PISKORKA - 1 ] of UkTPole;
            {Adresy polí, jimiž piškorka prochází}
    end;
end;

```

Vtip celého návrhu je v tom, že si každé pole pamatuje adresy všech piškorek, které jím procházejí, a naopak každá piškorka si pamatuje adresy všech polí, které ji tvoří.

Kromě toho musí piškorka vědět, kterému z hráčů patří. To má na starosti složka *Majitel*, která současně přechovává i počet kamenů, které jsou již na políčka piškorky položeny:

- ✧ Pokud na pěti polích dané piškorky není ještě položen žádný kámen, je ve složce majitel uložena nula symbolizující, že se ještě nikdo nepokusil zabrat piškorku pro sebe.
- ✧ Pokud jsou všechny kameny na dané pěti polích počítače, je jejich počet uložen jako kladné číslo.
- ✧ Pokud jsou naopak všechny kameny hráče, je zde uložena záporná hodnota jejich počtu.

✧ Pokud jsou některé z kamenů počítače a některé hráče, je do složky *Majitel* uložena předem zadaná hodnota *BLOK* označující, že v dané pětiici polí již nemůže vytvořit svoji piškovku žádný z hráčů.

Pole si toho musí o sobě pamatovat více. Kromě výše zmíněných adres procházejících piškorek si musí pamatovat i jejich počet, protože ten se pole od pole mění.

Každé pole si navíc pamatuje komu patří (tj. kdo na ně položil svůj kámen) a aby se program zjednodušil, je tato informace uchována jako znak, který se vykreslí na obrazovce na místě daného pole.

Poslední informaci, kterou najdeme ve vnitřní reprezentaci pole, jsou útočné váhy pro jednotlivé hráče. Jak jsme si již řekli, táhne-li kterýkoliv z hráčů na nějaké pole, projde počítač všechny piškovky, které daným polem procházejí. U každé z nich projde všemi jejími poli a zvýší jejich útočnou váhu pro hráče, který kámen položil, o tolikátou mocninu dvou, kolikátý kámen byl do piškovky přiložen. Pokud hráč přiložil kámen do piškovky, která byla do té doby „v cizích rukou“, sníží se naopak váha všech jejích kamenů o podíl, který doposud přidala deaktivovaná piškovka.

Nyní si ukážeme seznam deklarácí globálních proměnných.

```
TPisk Pisk [ PISKOREK ];          //Charakteristiky jednotlivých
  piškorek
TPole* Pozice [ RADKU ];          //Vektor ukazující na řádky šachovnice
TPole NaPoli [ POLI ];           //Charakteristika jednotlivých polí šachovnice
TPole* Nadejny [ _THrac ][ MAX_NUT ]; //Seznam
  nadějných tahů
int Nadejných[ _THrac ];          //Počet nadějných tahů
//Nadějný tah je takový tah, při jehož uskutečnění bude příští tah
//vyhrávající. Konkrétně jde o tahy na volné konce řady tří kamenů
TPole* Vyhra [ _THrac ][ MAX_VYH ]; //Seznam vyhrávajících tahů
int Vyher [ _THrac ];
//Uskutečněním vyhrávajícího tahu hráč dokončí piškovku a zvítězí
const char Znak [ _THrac ] = {ZHR, ZPO }; //Znaky označ.zabrané pole
const char Atr [ _THrac ] = {AHR, APO }; //Atributy zabraných polí
int X0, Y0, //Souřadnice levého horního rohu šachovnice na
  obrazovce
int Mezery; //2 = tisk mezer mezi sloupci; 1 = bez mezer
word PuvodniKur; //Informace o tvaru kurzoru při spuštění programu
int RadekProKur; //Číslo ř., kam se po skončení programu umístí kurzor
int Radek=0; //Řádek, kam se má umístit značka
int Sloupec=0; //Sloupec, kam se má umístit značka
LogH Konec; //Příznak konce hry

var
  Pisk: array [ 0 ..PISKOREK-1 ] of TPisk;
    {Charakteristiky jednotlivých piškorek}
  Pozice: array [ 0..RADKU-1 ] of UkTPole;
    {Vektor ukazatelů na řádky šachovnice}
  NaPoli: array [ 0..POLI-1 ] of TPole;
    {Charakteristiky jednotlivých polí
  šachovnice}
  Nadejny: array [ THrac, 0..MAX_NUT-1 ] of UkTPole;
    {Seznam nadějných tahů obou hráčů}
```

```

Nadejnych: array [ THrac ] of integer;
                                {Počet nadějných tahů obou hráčů}
{Nadějný tah je takový tah, při jehož uskutečnění bude příští tah
vyhrávající. Konkrétně jde o tahy na volné konce řady tří kamenů}
Vyhra: array [ THrac, 0..MAX_VYHER-1 ] of UkTPole;
                                {Seznam vyhrávajících tahů}
Vyher: array [ THrac ] of integer;
                                {Počet vyhrávajících tahů}
{Uskutečněním vyhrávajícího tahu hráč dokončí piškorku a zvítězí}
const
  Znak: array [ THrac ] of char = ( ZHR, ZPO );
                                {Znaky označující zabraná pole}
  Atr: array [ THrac ] of byte = ( AHR, APO );
                                {Atributy (vybarvení) zabraných polí}
  Radek: integer = 0;           {Řádek kam se má umístit značka}
  Sloupec: integer = 0;        {Sloupec, kam se má umístit značka}
var
  X0, Y0,                       {Souřadnice levého horního rohu šachovnice na
obrazovce}
  Mezery: integer;              {2 = tisk mezer mezi sloupci; 1 = bez mezer}
  PuvodniKur: word;            {Informace o tvaru kurzoru při
spušt.programu}
  RadekProKur: integer;        {Řádek, kam se dá po skončení programu
kurzor}
  Konec: LogH;                 {Příznak konce hry}

```

Předpokládáme, že komentáře jsou dostatečné pro to, abychom nemuseli účel jednotlivých proměnných vysvětlovat v dalším textu.

Je jasné, že při běžném programování se nám samozřejmě nepodaří určit hned napoprvé všechny globální proměnné, které budeme v programu potřebovat. My jsme je zde vy-psali všechny proto, abychom se k nim již nemuseli vracet a dodatečně je doplňovat.

Na závěr si ukážeme, jak by mohl vypadat hlavní program (pro stručnost jej uvedeme pouze v C++):

```

void /*****/ main /*****/ ()
{ Priprava();                  //Inicializace globálních proměnných
ZobrazStart();                //Zobrazení počátečního stavu
Hraj();                        //Vlastní hra
Uklid();                      //Uvedení obrazovky do původního stavu
}/***** main *****/

```

Než budete číst dál, zkuste si sami rozmyslet, jak by měla vypadat procedura *Příprava*, která inicializuje všechny potřebné proměnné.

Příprava

Na konci minulého oddílu jsme vás vyzvali, abyste se pokusili naprogramovat proceduru *Příprava*, která by inicializovala všechny globální proměnné. Podívejme se nyní na ni.

Inicializaci proměnných, nesoucích informace potřebné pro zobrazování, ponecháme na proceduře, která bude zobrazovat počáteční pozici (procedura *ZobrazStart*) a bude mít proto k dispozici všechny informace potřebné pro jejich správné počáteční nastavení, a soustředíme se na proměnné, jejichž hodnoty budou mít přímý vliv na vlastní hru.

Nejprve inicializujeme to, co nám dá nejméně práce, a to jsou pole se seznamy nadějných a vyhrávajících tahů. O vlastní seznamy tahů se nemusíme starat, protože nám vlastně stačí vynulovat proměnné, v nichž máme uložen počet tahů v seznamu.

Další jednoduchou akcí je inicializace vektoru ukazatelů na počátky řádků šachovnice. Víme, že první řádek začíná tamtéž co celá šachovnice a každý další řádek začíná o *SLOUPCU* sloupců dále. Programátory v C++ bychom chtěli jen upozornit, že příkaz v těle cyklu provádí tři akce: 1. posune ukazatel *aPole* o *SLOUPCU* polí dále, 2. jeho novou hodnotu přiřadí té složce vektoru *Pozice*, na niž právě ukazuje ukazatel *aPocR*, a 3. posune tento ukazatel o položku dále.

Další akce je pak v celé proceduře nejsložitější: musíme inicializovat záznamy o jednotlivých piškorcích. Abychom se v algoritmu lépe vyznali, připravíme si několik pomocných datových typů, proměnných a konstant (v Pascalu si je sice musíme připravit ještě před tělem procedury, ale to by zase nemělo být tak veliké neštěstí).

Za prvé si definujeme výčtový typ *TSměr*, jehož hodnotami budou čtyři směry, do nichž mohou být piškorky natočeny: východ, jih, jihovýchod a jihozápad. K němu si na definujeme výčtový datový typ *TInPis*, jehož hodnoty nám budou symbolizovat informace, které si o množinách piškorek natočených do jednotlivých směrů chceme předem připravit.

Podívejme se nyní, jaké to budou informace. Víme, že piškorku můžeme jednoznačně charakterizovat směrem natočení a počátkem, o němž se dohodneme, že bude u svislých a šikmých piškorek nahoře a u vodorovné piškorky vlevo. Počátky každé množiny piškorek souvisle vyplňují nějaký obdélník na šachovnici. Budeme-li znát krajní body tohoto obdélníku, můžeme snadno projít všechny piškorky natočené do daného směru. Horní a dolní řádek a levý a pravý sloupec tohoto obdélníka proto budou prvými čtyřmi užitečnými informacemi.

Chceme-li postupně projít všemi poli tvořícími piškorku, musíme umět skočit na šachovnici z daného políčka na políčko, které jej v piškorce následuje nebo které mu předchází. Vzhledem k tomu, že informace o polích šachovnice jsou uloženy ve vektoru, stačí nám znát vzdálenost dvou sousedních polí piškorky v tomto vektoru, protože vzhledem k uložení je tato vzdálenost pro všechna sousední políčka piškorek natočených do daného směru stejná. Tato vzdálenost tedy bude pátou informací, kterou si o dané množině piškorek připravíme.

Všech pět informací o všech čtyřech množinách piškorek jsme v programu uložili do tabulky, kterou jsme nazvali *VeSměru*. V C++ je tato tabulka deklarována jako statická konstanta (statická proto, aby se do ní hodnoty uložily již při překladu, nikoli až při běhu programu), v Pascalu nemáme jinou možnost, než ji deklarovat jako inicializovanou proměnnou a její ředitelnost zabezpečit vlastními silami.

V následující části programu budeme postupně procházet jednotlivými množinami piškorek a jejich připravené charakteristiky ukládat po řadě do polí vektoru *Pisk*. Ukazatel na právě inicializované pole (C++), resp. jeho index (Pascal) bude takovým průběžným parametrem všech čtyř úrovní cyklů, v nichž budeme piškorkami procházet.

Vnější cyklem bude cyklus přes množiny piškorek natočených do jednotlivých směrů. V něm pak budou dva vnořené cykly, které nás provedou oním výše zmiňovaným obdélníkem s počátky piškorek natočených do daného směru. Počáteční a koncové hodnoty těchto dvou cyklů získáme z výše popsané tabulky *VeSměru*.

Jak si můžete ověřit v deklaraci v předchozím oddílu, má záznam o piškorce dvě složky: majitele a seznam políček, jimiž piškorka prochází. (Je pravda, že by nám stačila souřadnice prvního políčka a ostatní bychom si mohli vždy dopočítat, ale v zájmu zvýšení rychlosti raději obětujeme trochu paměti.) S majitelem je to jednoduché – prozatím jím není nikdo. Seznam políček, kterými piškorka prochází, však musíme nejprve připravit. K tomu nám slouží čtvrtý vnořený cyklus.

Podíváte-li se na deklaraci záznamu o políčku, zjistíte, že kromě obsahu a váhy, ke kterým se vrátíme za chvíli, je mezi jeho položkami také seznam adres piškorek, které daným políčkem procházejí. Ve čtvrtém vnořeném cyklu je ta správná chvíle, kdy bychom měli postiženým políčkům sdělit, že jimi právě inicializovaná piškorka také prochází, tj. přidat její adresu na konec seznamu a zvýšit obsah složky s počtem procházejících piškorek.

Tady pozor. Abychom mohli zvyšovat obsah složky s počtem procházejících piškorek, musí být tato složka před první inkrementací bezpečně vynulována. Toto vynulování proto musíme ještě vložit před vnější cyklus přes jednotlivé množiny.

Tím bychom měli být s inicializací piškorek hotovi. Zbývá ještě dokončit inicializaci polí. Dosud jsou v polích nastaveny pouze informace o procházejících piškorkách. Je třeba do nich doplnit ještě informace o jejich obsahu (tam je to jednoduché: zatím na dané políčko ještě nikdo žádný kámen nepoložil) a váhu daného políčka. Jak jsme se však již dohodli, za počáteční váhu políčka položíme počet piškorek, které daným políčkem procházejí.

Na závěr je v daných procedurách ještě nastavena pozice, kam bude položen první kámen, i když tu bylo možno nastavit i v rámci deklarací.

Podívejme se nyní na zdrojový text této procedury v obou jazycích.

```
void /*****/ Priprava /*****/
    //Inicializuje všechny potřebné datové struktury
(void)
{
    Nadejnych[ POCITAC ] = Nadejnych[ HRAC ] = 0;
    Vyher [ POCITAC ] = Vyher [ HRAC ] = 0;
    //Inicializace vektoru ukazatelů na počátky řádků:
    TPole** aPocR = Pozice; //Adresa adresy počátku řádku
    TPole* aPole = (NaPolí - SLOUPCU); //Adresa prvního pole v řádku
    do
        *aPocR++ = (aPole += SLOUPCU);
    while( aPocR < &Pozice[ RADKU ] );
}
```

```

//Inicializace charakteristik piškorek:
//-----
enum TSmer {VYCHOD, JIH, JV, JZ, _TSmer};
//Možné orientace piškorek - JV a JZ označují diagonály \a /
enum TInPis {RA_1, RA_N, SL_1, SL_N, VZD_S, _TInPis};
//Indexy pro informace o piškorce orientované v daném směru:
// RA_1 = Počáteční (nejmenší možný) řádek jejího počátku
// RA_N = Koncový (největší možný) řádek jejího počátku
// SL_1 = Počáteční (nejmenší možný) sloupec jejího počátku
// SL_N = Koncový (největší možný) sloupec jejího počátku
// VZD_S = Vzdálenost sousedních polí dané piškorky ve vektoru
// NaPoli v němž je uložen obsah celé šachovnice

static const int VeSmeru[ _TSmer ][ _TInPis ] =
{ //Charakteristika piškorek natočených daným směrem
  //Směr \Obsah: RA_1 RA_N SL_1 SL_N VZD_S
  //-----
  /* V: */      { 0, RADKU, 0, SLOUPCU-PPP, 1 },
  /* J: */      { 0, RADKU-PPP, 0, SLOUPCU, SLOUPCU },
  /* JV: */     { 0, RADKU-PPP, 0, SLOUPCU-PPP, SLOUPCU+1 },
  /* JZ: */     { 0, RADKU-PPP, PPP, SLOUPCU, SLOUPCU-1 }
};

//Vynulování počtu procházejících piškorek
for( TPole *Pole = NaPoli; Pole < &NaPoli[ POLI ];
     (Pole++)->PoPrPi = 0 );

//Cyklus přes všechny piškorky převádíme na cyklus přes
//skupiny piškorek natočených do jednotlivých směrů
TPisk *aP = &Pisk[ -1 ]; //Zpracování začíná preinkrementem ukazatele
//Tato proměnná je skrytým parametrem následujících cyklů
for( const int (* SInf)[_TInPis] = VeSmeru;
     //SInf = ukazatel na (konstantní) vektor informací
     //o piškorách v daném směru
     SInf < &VeSmeru[ _TSmer ]; //Cyklus přes všechny směry
     SInf++ )
{
  int KDalsi = (*SInf)[ VZD_S ]; //Vzdálenost souseda v piškorce
  //Následující dvojitý cyklus je ve skutečnosti cyklem
  //přes všechny piškorky natočené v daném směru
  for( Radek = (*SInf)[RA_1]; Radek < (*SInf)[RA_N]; Radek++ )
    for( Sloupec = (*SInf)[SL_1]; Sloupec < (*SInf)[SL_N]; Sloupec++ )
    {
      (++aP)->Majitel = Nikdo; //Zatím ji nikdo nenárokuje
      //Cyklus přes všechna pole tvořící piškorku
      TPole **Zabr = aP->Proch; //Vektor polí zabraných piškorkou
      TPole *Pole = &Pozice[ Radek ][ Sloupec ]; //Počátek piškorky
      for( int PoPi=0; //PoPi = pořadí políčka piškorky
           PoPi < PISKORKA; //Cyklus přes všechna políčka
           PoPi++, //Další políčko zkoumané pišk.
           Pole += KDalsi) //Jeho pozice na šachovnici
      {
        *Zabr++ = Pole; //Toto pole piškorka také zabírá
        Pole->ProPi[ (Pole->PoPrPi)++ ] = aP;
        //Přidej ji k piškorkám,
        //které procházejí tímto polem
      }
    }
}

```

```

    } //Přes všechny řádky a sloupce
  } //Přes všechny směry

//Inicializace základních charakteristik polí šachovnice:
for( TPole *Pole = NaPoli;
    (Pole < &NaPoli[ POLI ] );
    Pole++ )
{
  Pole->Obsah = Nic; //Nikdo ještě nezabral pole pro sebe
  //Váhu (= důležitost) pole pro začátek nastavíme na počet
  //piškokrek, které daným polem procházejí
  Pole->Vaha[ POCITAC ] = Pole->Vaha[ HRAC ] = Pole->PoPrPi;
}
Radek = PO_RA / 2; //Pozice, kam bude na počátku hry
Sloupec = PO_SL / 2; //umístěna první piškorka
}/****** Příprava *****/

procedure (*****) Priprava (*****)
{Inicializuje všechny potřebné datové struktury}
;
type
  TSmer = ( VYCHOD, JIH, JV, JZ );
  {Možné orientace piškokrek - JV a JZ označují diagonály \a /}
  TInPis = ( RA_1, RA_N, SL_1, SL_N, VZD_S );
  {Indexy pro informace o piškokorce orientované v daném směru:
  RA_1 = Počáteční (nejmenší možný) řádek jejího počátku
  RA_N = Koncový (největší možný) řádek jejího počátku
  SL_1 = Počáteční (nejmenší možný) sloupec jejího počátku
  SL_N = Koncový (největší možný) sloupec jejího počátku
  VZD_S = Vzdálenost sousedních polí dané piškokorky ve vektoru
  NaPoli, v němž je uložen obsah celé šachovnice}
  TPInPis = array [ TInPis ] of integer;
  UkTPIP = ^TPInPis;

const
  VeSmeru: array [ TSmer, TInPis ] of integer = (
    {Charakteristika piškokrek natočených daným směrem}
    {Směr \Obsah: RA_1 RA_N SL_1 SL_N VZD_S}
    {-----}
    {V: } ( 0, RADKU, 0, SLOUPCU-PPP, 1 ),
    {J: } ( 0, RADKU-PPP, 0, SLOUPCU, SLOUPCU ),
    {JV: } ( 0, RADKU-PPP, 0, SLOUPCU-PPP, SLOUPCU+1 ),
    {JZ: } ( 0, RADKU-PPP, PPP, SLOUPCU, SLOUPCU-1 )
  );

var
  aPocR: UkUkTPole; {Adresa počátku řádku v poli NaPoli}
  aPole: UkTPole; {Adresa zpracovávaného pole}
  KDalsi: integer; {Vzdálenost k dalšímu poli dané piškokorky}
  PoPi: integer; {Pořadí zpracovávaného políčka v piškokorce}
  Zabr: UkUkTPole; {Uk.na vekt.ukazatelů na pole zabraná piškokorkou}
  ip: integer; {Parametr cyklu - index pole nebo piškokorky}
  ir: integer; {Parametr cyklu - index řádku}
  is: TSmer; {Parametr cyklu - index směru natočení piš}
  aP: UkTPisk; {Adresa piškokorky}
  SInf: UkTPIP; {Ukazatel na vektor se souborem informací}

```

```

                                o piškorkách natočených daným směrem}
begin
  Nadejnych[ POCITAC ] := 0;
  Nadejnych[ HRAC ] := 0;
  Vyher [ POCITAC ] := 0;
  Vyher [ HRAC ] := 0;
  {Inicializace vektoru ukazatelů na počátky řádků:}
  ip := 0; {Index počátečního pole v řádku}
  for ir:=0 to RADKU-1 do
    begin
      Pozice[ ir ] := @NaPoli[ ip ];
      Inc( ip, SLOUPCU );
    end;
  {Vynulování počtu procházejících piškorek}
  for ip:= 0 to POLI-1 do
    NaPoli[ ip ].PoPrPi := 0;
  {Inicializace charakteristik piškorek - cyklus přes všechny piškorky
  převádíme na cyklus přes skupiny piškorek natoč. do jednotl. směrů}
  ip := 0; {Skrytý parametr následujících cyklů}
  for is:= VYCHOD to JZ do
    begin
      SInf := @VeSmeru[ is ]; {Vektor informací o pišk. v daném směru}
      KDalsi := SInf^[ VZD_S ]; {Vzdálenost souseda v piškorce}
    {Následující dvojitý cyklus je ve skutečnosti cyklem
    přes všechny piškorky natočené v daném směru}
      for Radek:=SInf^[ RA_1 ] to SInf^[RA_N]-1 do
        begin
          for Sloupec:=SInf^[ SL_1 ] to SInf^[SL_N]-1 do
            begin
              aP := @Pisk[ ip ];
              ap^.Majitel := Nikdo; {Zatím ji nikdo nenárokuje}
              PoPi := 0; {PoPi= pořadí políčka v piškorce}
              Zabr := @aP^.Proch; {Adresa vektoru zabraných polí}
              aPole := @Pozice[ Radek ]^[ Sloupec ];
              {Adresa počátku piškorky}
              repeat {Cyklus přes všechna políčka v dané piškorce}
                Zabr^ := aPole; {Toto pole piškorka také zabírá}
                {Ukaž na místo pro další pole}
                PInc( Zabr, sizeof(pointer) );
                with( aPole^ )do {Přidej zpracovávanou piškorku}
                  begin {k piškorkám, které tímto polem}
                    ProPi[ PoPrPi ] := aP; {procházejí}
                    Inc( PoPrPi );
                  end;
                Inc( PoPi ); {Další políčko zkoumané piškorky}
                IncP( aPole, KDalsi*sizeof(TPole) );
                {Jeho pozice na šachovnici}
              until( PoPi >= PISKORKA );
              {Dokud neprojdeme všechna pole piškorek}
              Inc( ip ); {Index další piškorky k zpracování}
            end;
          {Přes všechny sloupce}
        end;
      {Přes všechny řádky}
    end;
  {Přes všechny směry}
  {Inicializace základních charakteristik polí šachovnice}
  for ip:= 0 to POLI-1 do

```

```

with( NaPoli[ ip ] )do
begin
  Obsah := Nic;      {Nikdo ještě nezabral pole pro sebe}
                    {Váhu (= důležitost) všech polí pro začátek nastavíme
                     na počet piškorek, které daným polem procházejí}
  Vaha[ POCITAC ] := PoPrPi;
  Vaha[ HRAC ] := PoPrPi;
end;
Radek := PO_RA div 2;      {Pozice, kam bude na počátku hry}
Sloupec := PO_SL div 2;   {umístěna první piškorka}
end;

```

Zobraz Start

Další procedurou z hlavního programu je procedura *ZobrazStart*, která má za úkol zobrazit počáteční stav šachovnice. To vypadá na první pohled jednoduše, ale my jsme si to opět zkomplikovali. Protože jsme zpočátku s počítačem pořád prohrávali, dospěli jsme k závěru, že je to tím, že je šachovnice málo přehledná.

Rozhodli jsme se proto, že pokud to půjde, budeme na dobu hry přepínat režim na čtyřicetisloupcový a kurzor uděláme co největší. Toho se však dá dosáhnout pouze přímým voláním služeb BIOSu. K tomu je však třeba vědět něco o registrech procesoru, možnostech volání těchto služeb nabízených překladačem a svým způsobem i o podobě přeloženého programu, tj. o tom, jak vypadá přeložený program v assembleru. Řekneme si alespoň ve stručnosti, o co jde.

Služby BIOSu jsou (podobně jako služby DOSu) dostupné prostřednictvím přerušení; konkrétně videoslužby jsou spravovány přerušením č. 0x10. Toto přerušení můžeme programově vyvolat v C++ voláním funkce *geninterrupt()*, v Pascalu voláním procedury *intr*. Řada služeb je „zapouzdřena“ do knihovnic funkcí, některé si ale musíme naprogramovat sami – například zjištění tvaru a polohy kurzoru.

Při volání tohoto přerušení je třeba do registru *AH* uložit číslo, které bude říkat, o jakou službu („funkci“) žádáme; zjištění tvaru a polohy kurzoru je funkce č. 3. Doplnující informace ukládáme zpravidla do registru *AL* nebo *BX*. Výsledek, číslo, popisující původní tvar kurzoru, se nám vrátí v registru *CX*.

Borland C++ umožňuje pracovat přímo s registry procesoru 8086 pomocí pseudoproměnných *_AX*, *_BX*, ... (typu **unsigned**) a *_AH*, *_AL*, ... (typu **unsigned char**). V prvních dvou řádcích tedy uložíme do registrů *AH* a *BH* potřebné hodnoty a pak ve třetím řádku vyvoláme přerušení 0x10. Ve čtvrtém řádku uložíme tvar kurzoru do globální proměnné, abychom jej mohli použít na konci programu k obnovení původního stavu.

Turbo Pascal nenabízí přímý přístup k registrům. Místo toho se používá proměnná typu *registers*, což je záznam, který obsahuje složky *AX*, *BX* ..., *AH*, *AL* atd. Proměnná typu *registers* se předává jako druhý parametr procedury *intr*. V ní pak také najdeme výsledky.

Význam dalších funkcí by již měl být zřejmý z komentářů a případně z nápovědy.

```

|| void /*****/ ZobrazStart /*****/

```

```

//Inicializuje obrazovku a generátor náhodných čísel,
//nakreslí počáteční situaci na obrazovku,
//provede za oba hráče první tah do středu šachovnice.
()
{
    _AH = 3;                //Funkce: Čti pozici a velikost kurzoru
    _BH = 0;                //Ptáme se na základní grafickou stránku
    geninterrupt( 0x10 );  //Přerušeni: videoslužby ROM-BIOS
    PuvodniKur = _CX;      //Uložíme si původní tvar kurzoru
    textattr( ANic );      //Barevný atribut prázdného pole
    textmode( C40 );       //Pokus o přepnutí na režim 40 sloupců
    curtype( 0, 8 );       //Nastavím kurzor jako plný blok
    struct text_info t_i;
    gettextinfo( &t_i );
    Mezery = (t_i.currmode == C40) //Žádaný textový režim nastaven?
              ? 1                //Ano => nepotřebujeme mezery mezi
    sloupci
              : 2;                //80 sloupců => chceme mezery mezi sloupci
    X0 = (t_i.screenwidth - SLOUPCU*Mezery) / 2 + 1;
    Y0 = (t_i.screenheight - RADKU) / 2 + 1;
    RadekProKur = t_i.screenheight - 1;
    clrscr();              //Sem nastavíme na konci kurzor
    randomize();           //Vyčištění obrazovky
    TPole *Pole = NaPoli;  //Inicializace generátoru náhodných čísel
    for( int r=0; r < RADKU; r++)
    {
        gotoxy( X0, Y0+r );
        for( int s=0; s < SLOUPCU; s++)
        {
            putchar( Pole++->Obsah );
            if( Mezery == 2 )
                putchar( ' ' );
        }
    }
    //Relizace prvních dvou tahů:
    (Pole = &Pozice[ Radek ][ Sloupec ])->Obsah = Znak[ HRAC ];
    Tiskni( Radek, Sloupec, HRAC );
    Analyza( Pole, HRAC );
    (Pole = &Pozice[ ++Radek ][ ++Sloupec ])->Obsah = Znak[POCITAC];
    Tiskni( Radek, Sloupec, POCITAC );
    Analyza( Pole, POCITAC );
}/****** ZobrazStart *****/

```

```

procedure (****) ZobrazStart (****)
{Inicializuje obrazovku a generátor náhodných čísel,
 nakreslí počáteční situaci na obrazovku,
 provede za oba hráče první tah do středu šachovnice}
;
var
    r, s: integer;
    reg: registers;
    ap: UkTPole;

```

```

begin
  reg.AH := 3;           {Funkce: Čti polohu a velikost kurzoru}
  reg.BH := 0;           {Ptáme se na základní grafickou stránku}
  intr( $10, reg );     {Přerušení: videoslužby ROM-BIOS}
  PuvodniKur := reg.CX;  {Uložíme si původní tvar kurzoru}
  textattr := byte( ANic ); {Nastavíme barevný atribut prázdného
  pole}
  textmode( CO40 );     {Pokus o přepnutí na režim 40 sloupců}
  curtype( 0, 8 );     {Nastavím kurzor jako plný blok}
  if( lo(windmax) <= 40 )then {Žádaný textový režim nastaven?}
    Mezery := 1         {Ano => Nepotřebujeme mezery mezi sloupci}
  else
    Mezery := 2;       {80 sloupců => chceme mezery mezi sloupci}
  X0 := (lo(windmax) - SLOUPCU*Mezery) div 2 + 1;
                                {Souřadnice levého horního}
                                {rohu šachovnice na obrazovce}
  Y0 := (hi(windmax) - RADKU) div 2 + 1;
  RadekProKur := hi(windmax); {Sem přesuneme na konci programu kurzor}
  randomize;                   {Inicializace generátoru náhodných čísel}
  clrscr;                       {Vyčištění obrazovky}
  for r := 0 to RADKU-1 do      {Vlastní vykreslení počátečního stavu}
  begin
    gotoxy( X0, Y0+r );
    for s := 0 to SLOUPCU-1 do
    begin
      write( Pozice[r]^[s].Obsah );
      if Mezery = 2 then write(' ');
    end
  end;
  {Realizace prvních dvou tahů:}
  ap := @Pozice[Radek]^[Sloupec];
  ap^.Obsah := Znak[ HRAC ];
  Tiskni( Radek, Sloupec, HRAC );
  Analyza( ap, HRAC );
  inc( Radek );
  inc( Sloupec );
  ap := @Pozice[Radek]^[Sloupec];
  ap^.Obsah := Znak[ POCITAC ];
  Tiskni( Radek, Sloupec, POCITAC );
  Analyza( ap, POCITAC );
end;
(***** ZobrazStart *****)

```

8.2 Hra

Když jsme si vše připravili, můžeme začít hrát. Algoritmus procedury *Hraj* je poměrně jednoduchý, a proto jej uvedeme pouze v C++.

```

void /*****/ Hraj /*****/
//Řídí vlastní hru
()
{

```

```

do{
    Analyza( Vstup(), HRAC );
    if( !Konec )
        Analyza( Tahni(), POCITAC );
    } while( !Konec );
getch(); //Počkej, než si prohlédne výledek
}/***** Hraj *****/

```

Objevily se nám zde dvě nové funkce a jedna procedura. Funkce *Vstup* má na starosti komunikaci s uživatelem a vrací ukazatel na pole, na něž uživatel položil svůj kámen. Funkce *Tahni* má naopak zjistit optimální pole pro tah počítače. Výstup předchozích funkcí je pak jedním ze dvou parametrů procedury *Analýza*, jejímž druhým parametrem je označení hráče, jehož tah má za úkol zpracovat. Procedura *Analýza* přehodnotí váhy polí pro oba hráče po tomto tahu.

Funkce *Vstup* je sice možná trochu delší, domníváme se ale, že je dostatečně jednoduchá na to, abychom mohli uvést pouze její pascalskou verzi:

```

function (*****) Vstup (*****)
: UkTPole;
{Ošetřuje pohyb kurzoru po šachovnici a vstup zadání.
 Lze zadávat i šikmé tahy - např. 7/Home = šikmo vlevo nahoru.
 Na stisk Esc se program ihned ukončí.
 Na stisk Fl, '?' nebo 'h' se kurzor nastaví na políčko,
 které se hráči doporučuje pro následující tah.
 Funkce vrací adresu pole, na které hráč táhl.
}
var
    c: char;                {Znak čtený z klávesnice}
    Tah: UkTPole;          {Adresa pole, kam se předpokládá tah}
begin
    repeat {Radek a Sloupec na počátku označují poslední tah počítače}
        gotoxy( X0+Sloupec*Mezery, Y0+Radek );
    {Umístí kurzor na šachovnici na pozici [Radek, Sloupec]}
    c := readkey;
    if c = #0 then {Pokud nebyla přečtena znaková klávesa}
        begin
            c := readkey; {Převezmeme polohový kód znaku}
            if (c = #$47) or (c = #$48) or (c = #$49) then {Nahoru}
                begin
                    dec( Radek );
                    if Radek < 0 then Radek := PO_RA;
                end
            else if (c = #$4F) or (c = #$50) or (c = #$51) then {Dolu}
                begin
                    inc( Radek );
                    if Radek > PO_RA then Radek := 0;
                end;
            if (c = #$47) or (c = #$4B) or (c = #$4F) then {Doleva}
                begin
                    dec( Sloupec );
                    if Sloupec < 0 then Sloupec := PO_SL;
                end
        end

```

```

else if (c = #$49) or (c = #$4D) or (c = #$51) then {Doprava}
begin
    inc( Sloupec );
    if Sloupec > PO_SL then Sloupec := 0;
end
else if (c = #$3b) then {F1 - žádá nápovědu}
    Nabidni( HRAC ); {Nastaví doporučený řádek a sloupec}
end {if c = #0}
else if (c = '?' ) or (c = 'h' ) or (c = 'H' ) then {žádá nápovědu}
    Tah := Nabidni( HRAC ) {Nastaví doporučený řádek a sloupec}
else if c = #$1B then {Klávesa Ecs předčasně ukončí hru}
begin
    Uklid;
    halt(0);
end;
Tah := @Pozice[Radek]^[Sloupec];
until( (c = #$0D) and (Tah^.Obsah = Nic)); {Cyklus ukončí stisk}
{klávesy ENTER na neobsazeném poli}
Vstup := Tah; {Bude vracet adresu zadaného
pole}
textattr := Atr[ HRAC ]; {Nastav barvu hráče}
Tah^.Obsah := Znak[ HRAC ]; {Zakresli jeho tah}
write( Znak[ HRAC ] );
end;
(***** Vstup *****)

```

Funkce *Tahni* je pak již naprosto triviální: využívá pouze služeb funkce *Nabidni*, která je koncipována tak, aby mohla své služby poskytovat oběma hráčům: uživateli ve formě nápovědy a počítači vyhledáním řešení optimalizujícího předem zadané kritérium. Ukážeme si ji opět v Pascalu:

```

function (*****) Tahni (*****)
: UkTPole;
{Realizuje tah počítače. Vrací adresu pole, na které táhl.}
begin
    Tahni := Nabidni( POCITAC );
    Tahni^.Obsah := Znak[ POCITAC ];
    if Konec = NE then
        Tiskni( Radek, Sloupec, POCITAC );
end;
(***** Tahni *****)

```

Funkce *Nabidni* využívá informací, které ji před tím připravila procedura *Analýza*. Podívejme se proto nejprve na ni.

Jak jsme si již řekli, tato procedura má dva parametry: identifikaci hráče a ukazatel na pole, na které hráč táhl. Jejím úkolem je zjistit, jak tento tah ovlivnil váhy jednotlivých polí hráče i jeho protihráče.

Nejprve zrušíme všechny záznamy o nadějných a vyhrávajících tazích, protože se posledním tahem mohla situace natolik změnit (a většinou i změnila), že je jednodušší naplnit tyto vektory znovu.

U protihráče nemusíme mazat vše, protože tam by se situace změnila pouze v případě, že by hráč táhl na některé z protihráčových nadějných nebo vyhrávajících polí. Pokud k tomu došlo, je třeba toto pole ze seznamu odebrat. U nadějných polí je třeba navíc odebrat i párová pole, protože pokud se tři volné kameny z jedné strany ohraničí, přestává být i druhé pole nadějným.

Protože dlouhé a nepřehledné procedury zvětšují pravděpodobnost chyby, naprogramovali jsme výše uvedenou činnost do samostatné procedury *SmažNadějně*, kterou procedura *Analýza* na svém počátku vyvolá.

Po smazání svých a aktualizaci protivníkových nadějných a vyhrávajících polí prochází procedura *Analýza* postupně všechny piškorcky, které zasahují na pole, na něž hráč právě táhl. Každá z těchto piškorek může být v jednom ze čtyř stavů:

1. Piškorcka je nepoužitelná, protože na jejích polích jsou kameny obou hráčů. Taková piškorcka má ve složce *Majitel* uloženu hodnotu *BLOK*. Tento případ je ze všech nejjednodušší, protože položení dalšího kamene nijak neovlivní váhu polí piškorcky: piškorcka zůstane i nadále nepoužitelná.
2. Piškorcka nepatří nikomu (poznáme to podle toho, že ve složce *Majitel* je nula – Nikdo). Hráč, který na dané pole táhl, si ji tedy nárokuje pro sebe (do složky *majitel* dá hodnotu +1 nebo -1). Protože je tím v piškorce položen první kámen, zvedne se váha všech ostatních políček piškorcky o $2^1 = 2$. Je proto třeba projít všechna pole patřící k piškorce (jejich adresy jsou ve vektoru *Proch*, který je složkou záznamu o piškorce) a jejich váhy zvýšit o 2.
3. Piškorcka patří hráči, důsledky jehož tahu analyzujeme. (Poznáme to podle toho, že znaménko hodnoty ve složce *Majitel* je stejné jako znaménko hráče.) Je proto třeba zvednout absolutní hodnotu obsahu složky *Majitel* o jedničku (znaménko zůstane, protože majitel se nemění) a pak projít všechna pole patřící k piškorce a váhy dosud neobsazených polí zvýšit o 2^n , kde n je počet kamenů v dané piškorce včetně kamene právě položeného.

V tomto případě je třeba zároveň zjistit, zda nebyl položen kámen na poslední volné pole piškorcky a zda tedy hráč nedovedl svým tahem hru k vítěznému konci. V případě, že ano, rozbliká se na obrazovce vítězná piškorcka a vypíše se ukončující zpráva.

4. Piškorcka patří protihráči hráče, důsledky jehož tahu analyzujeme (poznáme to podle toho, že znaménko hodnoty ve složce *Majitel* je jiné než znaménko hráče). Protože jsou nyní na políčkách této piškorcky kameny obou hráčů, je třeba piškorcku označit za nepoužitelnou (do složky *Majitel* se zapíše hodnota *BLOK*). Kromě toho je třeba protihráči snížit váhy všech volných polí piškorcky o hodnotu, kterou k nim daná piškorcka přispěla. Je-li v piškorce n kamenů protihráče, klesne váha volných polí piškorcky o

$$(1 + 2^1 + 2^2 + \dots + 2^n) = 2^{n+1} - 1$$

V předposledním případě, tj. v případě, kdy hráč posiluje svoji vlastní piškorcku, je vhodné myslet ještě na dvě věci. Především bychom se měli podívat, zda v dané piškorce není

volné poslední pole, a pokud ano, tak si je zapamatovat, aby nás nějaká souhra většího počtu menších vah neodlákala od tohoto vítězného tahu.

Další případ, kterým je vhodné se zabývat zvlášť, je situace s třemi volnými kameny. Pokud bychom k nim totiž přiložili čtvrtý, získali bychom dva vyhrávající tahy a vítězství by již nebylo možno zabránit. Algoritmus, kterým se tato skutečnost v programu zjišťuje, však není přesný. Pídi se pouze po tom, zda jsou daná tři pole v piškorce uprostřed. Abychom však mohli tři volné kameny rozšířit na čtyři volné kameny, musí být alespoň po jedné straně této trojice nejméně dvě sousední pole volná. To však již náš program netestuje. V praxi se totiž ukázalo, že se i bez tohoto doplňujícího testu rozhoduje rozumně, a proto jsme algoritmus zbytečně dále nekomplikovali.

Vedlejším efektem procedury *Analýza* je nastavení hodnoty globální proměnné *Konec*. Hodnotou *ANO* sděluje procedura volajícím programům, že hra je u konce.

V programu pro C++ bychom ještě chtěli upozornit na makro *CPVJPP*, které definuje hlavičku cyklu přes všechna pole, jimiž piškorka prochází. Makro je několikařádkové.

```
void /****/ Analyza /****/
//Analyzuje situaci po tahu hráče Hrac na pole Pole.
//Aktualizuje váhy jednotlivých polí a nadějných a vyhrávajících tahů.
//Vrací hodnotu proměnné Konec, kterou také patřičně nastavuje.
( TPole * Pole, THrac Hrac )
{
    int q = Hrac == HRAC ? -1 : 1;
    /* Udává, zda je počet obsazených políček piškorky uložený v poloze
       TPisk.Majitel udán jako kladné nebo jako záporné číslo
       (HRAC nebo POCITAC).
    */
    TPisk **pi;      //Parametr cyklu přes všechny ovlivněné piškorky
    TPisk **pik;     //Hodnota parametru ukončující cyklus
    TPole **po;     //Parametr cyklu přes všechna ovlivněná pole
    TPole **pok;     //Hodnota parametru ukončující cyklus
    //Cyklus Přes Všechna Pole, Jimiž Piškorka Prochází
    #define CPVJPP \
        for( po = (*pi)->Proch, pok = &(*pi)->Proch[ PISKORKA ]; \
            po < pok; po++ )

        SmazNadejne( Pole, Hrac );
    /* Každý kámen položený do piškorky, která prochází daným polem,
       zvedne váhu tohoto pole o 2^N, kde N je celkový počet kamenů
       v piškorce po položení daného kamene.
    */

    //Cyklus přes všechny piškorky, které procházejí zadaným polem
    for( pi = Pole->ProPi, pik = &Pole->ProPi[ Pole->PoPrPi ];
        pi < pik; pi++ )
    {
        int m = (*pi)->Majitel;
        int kp = abs( m ); //Počet položených kamenů v dané piškorce
        if( m == BLOK ) //Piškorku nelze doplnit => žádná akce
            ;
        ef( m == Nikdo )
    }
}
```

```

/* Piškorka zatím nemá majitele - bude to Hrac. Jinými slovy:
   byl položen první kámen budoucí piškorky
*/
(*pi)->Majitel = q; //Zapíšeme značku majitele
CPVPJPP //Cyklus přes všechna pole, jimiž piškorka prochází
{
(*po)->Vaha[ Hrac ] += 2;
//Přibylo aktivní piškorka (2^1 == 2)
(*po)->Vaha[ !Hrac ]--;
//Protihráči ubylo použitelných piškorek
}
}
ef( (m > 0) == (q > 0) ) //Majitel byl hráč - posílí se váha
polí
{
m = (*pi)->Majitel += q; //Aktualizuj počet kamenů v piškorce
kp = abs( m ); //a obsah pomocných proměnných
if( kp == PISKORKA) //Byl-li položen poslední kámen
{
textattr( Atr[ Hrac ]+BLINK ); //Rozblikej vítěznou piškorku
CPVPJPP //Cyklus přes všechna pole, jimiž piškorka prochází
{
gotoxy( X0+( *po-NaPoli)%SLOUPCU )*Mezery,
Y0+( *po-NaPoli)/SLOUPCU );
putch ( Znak[ Hrac ] );
}
gotoxy( 10, Y0 + RADKU );
cputs( Hrac == HRAC ? "G R A T U L U J I !"
: "Čest poraženým ..." );
Konec = ANO; //Nastav příznak konce hry
return; //a skonči
}
}
int v = 1 << kp; //Váha = 2^(Počet položených kamenů v pišk.)
CPVPJPP //Cyklus přes všechna pole jimiž piškorka prochází
{
if( (*po)->Obsah == Nic) //Pokud na poli nic neleží
{
(*po)->Vaha[ Hrac ] += v; //Zvedni patřičně jeho váhu
if( kp == PISKORKA-1 )
//Pokud zbývá položit poslední kámen...
{
//závěrečný vyhrávající tah - zaznamenej ho
int *vh = &Vyher[ Hrac ];
if( *vh < MAX_VYH )
Vyhra[ Hrac ][ (*vh)++ ] = *po;
}
}
}
}
if( (kp == PISKORKA-2) && //Jsou polženy tři kameny
(Nadejnych[ Hrac ] < MAX_NUT) &&
//Je místo na zapamatování další
( (*pi)->Proch[ 0 ]->Obsah == Nic) && //Prostřední tři
( (*pi)->Proch[ PPP ]->Obsah == Nic) ) //kameny v pišk.
{
//... je tah na zbylá volná pole prohlášen za nadějný.
Nadejny[ Hrac ][ (Nadejnych[ Hrac ])+ ] = (*pi)->Proch[ 0 ];
Nadejny[ Hrac ][ (Nadejnych[ Hrac ])+ ] = (*pi)->Proch[ PPP];
}
}

```


zjednoduší, protože Pascal nepodporuje adresovou aritmetiku a museli bychom proto používat umělých konstrukcí (viz původní pascalská verze programu na doplňkové stránce).

```

procedure (*****) Analyza (*****)
{Analyzuje situaci po tahu hráče Hrac_ na pole Pole.
 Aktualizuje váhy jednotlivých polí a nadějných a vyhrávajících tahů.
 Vrací hodnotu proměnné Konec, kterou také patřičně nastavuje.
}
( aPole: UkTPole; Hrac_: THrac );

procedure (*222*) SmazNadejne (*222*)
{Reaguje na tah hráče Hrac_ na pole Pole. Pokud byl tento tah jedním
 z nadějných nebo vyhrávajících tahů protihráče (tj. byla-li jím
 zažehnána hrozba), umaže se tento tah protihráči z příslušného
 vektoru
}
;
type
  VUPo = array[ 0..MAX_NUT ]of UkTPole;

var
  p: ^VUPo;           {Ukazatel na vektor ukazatelů na pole}
  n: ^integer;       {Ukazatel na počet prvků odkazovaného vektoru}
  i: integer;        {Parametr cyklu přes všechny složky vektoru}
  ii: integer;       {Pomocná proměnná}

begin
  Vyher [ Hrac_ ] := 0;           {Nadějně a vyhrávající tahy hráče}
  Nadejných[ Hrac_ ] := 0;       {se budou vypočítávat znovu.}
  {Nyní zjistíme, jestli se jednalo o protihráčův nadějný tah}
  i := 0;
  p := @Nadejny[ not Hrac_ ][ 0 ];
  n := @Nadejných[ not Hrac_ ];
  {S touto buňkou se často pracuje, tak jsme zavedli ukazatel,
  abychom nemuseli často indexovat}
  while i < n^ do
    if p^[i] = aPole
    then begin
  {Přemažeme tah, o který šlo posledním prvkem vektoru Nadejny,
  Zároveň přestává být nadějný i tah příslušný "do dvojice"
}
      dec( n^ );
      p^[i] := Nadejny[ not Hrac_ ][ n^ ];
      dec( n^ );
  {Zjistíme, zda měl tah p sudý nebo lichý index
  ...a přemažeme "kolegu z dvojice".
}
      if (i and 1) = 1
      then ii := i-1
      else ii := i+1;
      p^[ii] := Nadejny[ not Hrac_ ][ n^ ];
    end
  else
    {Tah nesměřoval na testované pole}
    Inc( i );
  {Následuje test na protihráčovy vítězné tahy}
  i := 0;
}

```

```

n := @Vyher[ not Hrac_ ];
p := @Vyhra[ not Hrac_ ][ 0 ];
while i < n^ do
  if p^[i] = aPole
  then begin
    dec( n^ );
    p^[i] := Vyhra[ not Hrac_ ][ n^ ];
  end
  else
    Inc( i );
end;
end;
(*222222222 SmazNadejne 222222222*)

var
  ipi: integer;           {Parametr cyklu přes všechny ovlivněné
  piškorky}
  ipo: integer;          {Parametr cyklu přes všechna ovlivněná pole}
  q: integer;
  {Udává, zda je počet obsazených políček piškorky
  uložený v položce TPisk.Majitel udán jako záporné nebo jako
  kladné číslo ( HRAC resp. POCITAC )}
  m: integer;           {Hodnota ve složce majitel}
  kp: integer;          {Počet kamenů v piškorce = abs(m)}
  v: integer;           {Předem spočítaný přírůstek či úbytek váhy}
  por: integer;         {Pořadí testovaného pole ve vektoru NaPoli}

begin
  if Hrac_ = HRAC then q := -1           {Majitelem je hráč}
  else q := 1;                           {Majitelem je počítač}
  SmazNadejne;
  {Každý kámen položený do piškorky, která prochází daným polem,
  zvedne váhu tohoto pole o 2^N, kde N je celkový počet kamenů
  v piškorce po položení daného kamene.}
}
{Cyklus přes všechny piškorky, které procházejí zadaným polem}
for ipi:=0 to aPole^.PoPrPi-1 do
begin
  with aPole^.ProPi[ ipi ]^ do
  begin
    m := Majitel;
    if m = BLOK then
      {Piškorku nelze doplnit => žádná akce}
    else if m = Nikdo then
      {Piškorka nemá majitele - bude to Hrac_.}
    begin
      {Jinými slovy: byl položen první kámen.}
      Majitel := q; {Zapíšeme značku majitele}
      for ipo:=0 to PPP do {Cyklus přes všechna pole,}
      begin
        {jimiž piškorka prochází}
        with Proch[ ipo ]^ do
        begin
          Inc( Vaha[ Hrac_ ], 2 );
          {Přibyla aktivní piškorka (2^1 == 2)}
          Dec( Vaha[ not Hrac_ ] );
          {Protihr.ubylo použitelných pišk.}
        end;
      end;
    end;
  end;
end;

```

```

        end;
    end
else if ( m > 0 ) = ( q > 0 ) then
begin
    {Majitel byl hráč => posílí se váha polí}
    Inc( Majitel, q ); {Aktualizuj počet kamenů v piškorce}
    m := Majitel;     {a obsah pomocných proměnných}
    kp := abs( m );
    if kp = PISKORKA then {Byl-li položen poslední kámen}
begin
    textattr := Atr[ Hrac_ ]+blink;
    {Rozblikej vítěznou piškorku}
    for ipo:=0 to PPP do {Cyklus přes všechna pole,}
begin
    {jimiž piškorka prochází}
    por := integer( longint(Proch[ ipo ] ) -
                    longint(@NaPoli ) )
            div sizeof(TPole);
    {Pořadí pole ve vektoru NaPoli}
    gotoxy( X0 + ( por mod SLOUPCU ) * Mezery,
            Y0 + por div SLOUPCU );
    write( Znak[ Hrac_ ] );
end;
    if Hrac_ = HRAC
    then begin
        gotoxy( 10, Y0 + RADKU );
        write( 'G R A T U L U J I !' );
    end
    else begin
        gotoxy( 11, Y0 + RADKU );
        write( 'Čest poraženým ...' );
    end;
    Konec := ANO; {Nastav příznak konce hry a skonči}
    exit;         {Opouštíme tutu funkci}
end;
    v := 1 shl m; {Váha = 2^(Počet polož.kamenů v piškorkách)}
    if (kp = PISKORKA-2) and {Jsou-li položeny
prostřední}
        ( Proch[ 0 ]^.Obsah = Nic ) and {tři kameny... }
        ( Proch[ PPP ]^.Obsah = Nic ) then
begin
    {...je tah na zbylá volná pole prohlášen za nadějný.}
    Nadejny[ Hrac_ ][ Nadejnych[ Hrac_ ] ] := Proch[ 0 ];
    Inc( Nadejnych[ Hrac_ ] );
    Nadejny[ Hrac_ ][ Nadejnych[ Hrac_ ] ] := Proch[ PPP
];

    Inc( Nadejnych[ Hrac_ ] );
end;
    for ipo:=0 to PPP do {Cyklus přes všechna pole,}
begin
    {jimiž piškorka prochází}
    with Proch[ ipo ]^ do
begin
    if Obsah = Nic then {Pokud na poli nic neleží}
begin
    Inc( Vaha[ Hrac_ ], v );
    {Zvedni patřičně jeho váhu}
    if kp >= PISKORKA-1 then

```

```

                {Pokud zbývá položit poslední kámen}
                begin
{Jedná se o závěrečný vyhrávající tah - zaznamenej jej}
                if Vyher[ Hrac_ ] < MAX_VYHER then
                begin
                    Vyhra[ Hrac_ ][ Vyher[ Hrac_ ] ] := Proch[ ipo
];
                    Inc( Vyher[ Hrac_ ] );
                end;
                end;
                end;
                end;
            end;
        else {Majitelem piškorky byl protihráč -}
        begin {piškorka se stává neperspektivní}
            v := (2 shl abs(m)) - 1;
            {Součet dosud nasbíraných vah od piškorek}
            for ipo:=0 to PPP do
            {Cyklus přes všechna pole piškorky}
                Dec( Proch[ ipo ]^.Vaha[ not Hrac_ ], v ); {Uber váhu}
                Majitel := BLOK;
                {Smíšení majitelé => nepoužitelná piškorka}
            end;
        end;
    end;
    Konec := NE;
end;
(***** Analyza *****)

```

Poslední funkcí, která nám zbývá, je *Nabídní*. Jejím úkolem je projít všechny podklady připravené funkcí *Analýza* a vybrat optimální tah. Kritérium optimality je zakódováno v podprogramu (v C++ ve funkci, v Pascalu v proceduře) *Porovnej*, který porovnává výhodnost pole, jehož adresa je mu předána jako parametr, s výsledky dosavadního zkoumání.

Konstrukce podprogramu *Porovnej* se liší podle jazyka, v němž je úloha naprogramována. V Pascalu je to procedura definovaná jako vnořená ve funkci *Nabídní*. Toho můžeme využít k tomu, že si bude své mezivýsledky pamatovat v proměnných funkce *Nabídní*, která zároveň obstará i inicializaci těchto proměnných.

Jazyk C++ nám možnost vnořování procedur nenabízí a musíme proto definovat výhodnocování kritéria jako samostatný podprogram. Pomocné proměnné pro mezivýsledky pak můžeme definovat třemi způsoby: jako další parametry, jako globální proměnné nebo jako lokální statické proměnné. Parametrické řešení nám však připadalo neefektivní a na řešení s globálními proměnnými nám zase vadilo, že k těmto proměnným by pak bylo možno přistoupit i z jiných částí programu prostou záměnou identifikátorů.

Zvolili jsme proto třetí možnost a podprogram *Porovnej* definovali jako funkci, která si potřebné mezivýsledky pamatuje ve svých lokálních statických proměnných a adresu nejlepšího z dosud předložených polí vrací jako svoji funkční hodnotu. Toto řešení je sice o něco rychlejší než parametrické, ale oproti předchozím dvěma možnostem vyžaduje

ještě dodatečně vyřešit otázku inicializace oněch pomocných statických proměnných. V našem programu inicializujeme funkci tak, že jí místo adresy testovaného pole předáme prázdný ukazatel.

Kritérium, které jsme zvolili, je jednoduché: vážený součet útočných vah obou hráčů. Podle toho, zda chceme, aby se počítač více soustředil na útok nebo na obranu, volíme velikost celočíselných konstant *KU* (váha útoku) a *KO* (váha obrany). Těmito koeficienty zároveň i nepřímo nastavujeme obtížnost hry. Zkušenost ukázala, že čím více budeme protěžovat útok, tím snadněji dosáhneme toho, aby počítač přehlédl nějakou naši kombinaci a prohrál. Čím větší budeme klást naopak důraz na obranu, tím hůře se nám bude dařit rozvíjet jakékoliv kombinace a tím více nám hrozí prohra z prostého přehlnutí.

```
TPole* /*****/ Porovnej /*****/
//Zjistí, zda pole Pole je pro tah hráče Hrac výhodnější, než nejlepší
//dosud nalezené. Pokud je, předá jeho adresu jako svoji funkční
//hodnotu
( THrac Hrac, TPole * Pole )
{
    static const KU = 1;      //Koeficient útočnosti hry
    static const KO = 2;      //Koeficient obrannosti hry
    static int Max;
    static TPole *Nejlepsi;
    if( !Pole )                //Inicializace před dalším cyklem volání
        return Max=0, Nejlepsi=0;
    if( Pole->Obsah == Nic) //Vlastní vyhodnocování kritéria
    {
        //Kritérium testujeme pouze pro prázdná pole
        int Vetsi = KU * Pole->Vaha[ Hrac ] +
                    KO * Pole->Vaha[ !Hrac ];
        if( ( Vetsi > Max ) ||
            ( (Vetsi == Max) && (rand() > RAND_MAX/2) ) )
        {
            Nejlepsi = Pole;
            Max = Vetsi;
        }
    }
    return Nejlepsi;
}/***** Porovnej *****/
```

Jediným závažnějším podprogramem, který jsme ještě nerozebrali, je funkce *Nabidni*. Její činnost je jednoduchá:

1. Nejprve otestuje, zda hráč nemá možnost nějakého vyhrávajícího tahu. Pokud ano, je to bezesporu ten nejlepší, který může provést.
2. Pokud žádný vyhrávající tah není k dispozici, podívá se, zda náhodou nemá obdobnou možnost jeho soupeř. Pokud takový tah existuje, je nutno příslušné pole ihned obsadit, protože v opačném případě by soupeř v příštím tahu vyhrál.
3. Pokud ani soupeř nemá šanci hned vyhrát, podíváme se na jeho nadějně tahy, protože víme, že kdybychom na ně hned nereagovali, soupeř by mohl jeden z nich zvolit a partner tohoto tahu by se pak automaticky stal tahem vyhrávajícím. Protože nadějně

tahy jsou vždy nejméně dva, vybereme z nich ten, který nám podprogram *Porovnej* označí jako nejvýhodnější.

4. Pokud soupeř nemá žádné nadějně tahy, podíváme se na vlastní nadějně tahy a jsou-li nějaké, vybereme z nich ten nejvýhodnější.
5. Nejsou-li k dispozici žádné zvlášť výhodné tahy, nezbude nám, než projít celou šachovnicí a vybrat za pomoci podprogramu *Porovnej* nejvýhodnější pole, tj. pole maximalizující naše kritérium.

```
TPole * /*****/ Nabidni /*****/
/*Vrátí adresu pole doporučeného pro tah
 a nastaví na něj proměnné Radek a Sloupec.
*/
( THrac Ja )
{
  THrac On = THrac( !Ja );
  TPole *Nejlepsi;          //Pole, na něž funkce doporučuje táhnout
                          //Pokud je některý tah vyhrávající, je volba
  jasná
  if( Vyher[ Ja ] )        //1) Vyhrávám
  Nejlepsi = *Vyhra[ Ja ]; //Bez přemýšlení беру 0. položku
  ef( Vyher[ On ] )        //2) Snažím se zabránit jeho výhře
  Nejlepsi = *Vyhra[ On ]; //Bez přemýšlení беру 0. položku
  else                      //Jasně na výhru to není - bude se muset porovnávat
  {
    Porovnej( Ja, 0 ); //Inicializace statický prom. funkce
    Porovnej
                          //Nejprve zkusíme vybrat nejlepší z
  nadějných
    if( Nadejnych[ On ] )
    { /* 3) Zachytávám jeho nadějně tahy. Kdybych nezachytil jeho
      nadějný tah, získal by protihráč v příštím kole vítězný tah.
      */
      TPole **n = Nadejny[ On ]; //Parametr cyklu
      TPole **nk = &n[ Nadejnych[ On ] ];
      //Ukončovací hodnota cyklu
      while( n < nk )
        Nejlepsi = Porovnej( Ja, *(n++) );
    }
    ef( Nadejnych[ Ja ] ) //4) Vyberu-li jeden z nadějných tahů,
    { //stane se jeho kolega vyhrávajícím
      TPole **n = Nadejny[ Ja ]; //Parametr cyklu
      TPole **nk = &n[ Nadejnych[ Ja ] ];
      //Ukončovací hodnota cyklu
      while( n < nk )
        Nejlepsi = Porovnej( Ja, *(n++) );
    }
    else //Nejsou-li ani nadějně tahy, hledám pole s nejvyšší vahou
    {
      for( TPole *Pole = NaPoli; //Cyklus přes
      všechny pole
      Pole < &NaPoli[ POLI ]; Pole++ )
        Nejlepsi = Porovnej( Ja, Pole );
    }
  }
}
```

```

        if( ( Konec = LogH(Nejlepsi == 0) ) == ANO )
        { //Nenašlo se pole, do nějž má smysl investovat =>
remíza
            textattr( Atr[ Ja ] + BLINK ); //Rozblikej nápis
            gotoxy( 10, YO + RADKU );
            cputs( "Remíza - GRATULUJI!" );
            getch(); //Počkej, než si prohlédne
výledek
            exit( 0 );
        }
    }
}
int Pozic = Nejlepsi - NaPoli; //Pozice nalezeného pole ve
vektoru
Radek = Pozic / SLOUPCU; //přepočítaná na řádek
Sloupec = Pozic % SLOUPCU; //a sloupec
return( Nejlepsi );
} /***** Nabidni *****/

```

Pascalská verze této funkce obsahuje navíc i zdrojový text procedury *Porovnej*, protože se nám zdálo výhodné ji definovat jako vnořenou proceduru do funkce *Nabidni*.

```

function (****) Nabidni (****)
{Vrátí adresu pole doporučovaného pro tah
 a nastaví na něj proměnné Radek a Sloupec.}
( Ja:THrac ): UkTPole;
type
    VUkTPole = array[ 0..POLI-1 ] of UkTPole;
    UkVUkTPole = ^VUkTPole;
var
    On: THrac;
    Nejlepsi: UkTPole; {Pole, na něž funkce doporučuje táhnout}
    Max: integer; {Největší dosažená hodnota kritéria}
    Pozice: integer; {Pozice pole ve vektoru NaPoli}
    i: integer;
    vn: UkVUkTPole; {Ukazatel na vektor ukazatelů na pole}
    aPole: UkTPole;

procedure (*222*) Porovnej (*222*)
{Zjistí, zda pole aPole je pro tah hráče výhodnější, než nejlepší
 dosud nalezené pole. Pokud je, zapamatuje si jeho adresu v proměnné
 Nejlepsi. Snažíme se najít tah, který hráči nejvíce "přidá"
 a protihráči nejvíce "ubere".}
( aPole: UkTPole );
const
    U = 1; {Koeficient útočnosti hry}
    KO = 3; {Koeficient obrannosti hry}
var
    Vetsi, Mensi, i: integer;
begin
    with aPole^ do
    begin
        if Obsah = Nic then {Kritérium testujeme pouze pro volná pole}

```



```

                Uklid;
                halt( 0 );
            end
        end
    end;
    Pozice := integer(longint(Nejlepsi) - longint(@NaPoli)) div
                                                sizeof(TPole);
    Radek := Pozice div SLOUPCU;                {Pozice nalezeného pole ve}
    Sloupec := Pozice mod SLOUPCU;             {vektoru přepočítaná na řádek}
    Nabidni := Nejlepsi;                       {a sloupec }
end;
(***** Nabidni *****)

```

Tolik tedy k programu *Piškorky*. Na závěr ještě jednu poznámku: *Piškorky* napsal R. Pečinovský kdysi jako program, na němž se učil pracovat s ukazateli v jazyku C. Základní algoritmy nevymyslel ani neoptimalizoval. Dostal jej před delším časem od J. Pivoňky jako program v Basicu, který uměl pracovat pouze s celými čísly a vektory. Program potom přepsal do C++, trochu zrychlil zavedením ukazatelů a přidal do něj možnost nápovědy. Jedinou závažnější změnou oproti původnímu algoritmu bylo zavedení počátečních vah políček na prázdné šachovnici. V původní verzi totiž vstupovala do hry všechna políčka s nulovou vahou, nyní je vstupní váha každého políčka rovna počtu piškorek, v nichž může dané políčko vystupovat.

Rejstřík

#	hdrfile, 114	\$IFOPT, 130
#, 109; 128	hdrstop, 114	\$UNDEF, 129
##, 128	option, 121	
#define, 124; 127	startup, 114	
#elif, 132	warn, 115	_
#else, 131	#undef, 125	__cplusplus, 139
#endif, 132		__cs, 32
#if, 131	\$	__ds, 32
#ifndef, 132	\$DEFINE, 129	__es, 32
#include, 134	\$ELIF, 130	__far, 30; 39
#pragma, 113	\$ELSE, 130	__huge, 30; 39
argsused, 114	\$ENDIF, 130	__near, 30; 39
exit, 114	\$IFDEF, 130	__seg, 33
	\$IFNDEF, 130	__ss, 32

A

adresa, 14
 normalizovaná, 30
 ANSLSYS, 162
 Append, 74
 aritmetika
 adresová, 24
 v Pascalu, 26
 Assign, 73
 AssignCrt, 73

B

barva
 popředí, 148
 pozadí, 148
 v grafickém režimu, 148
 bitové pole, 62
 BlockRead, 74
 BlockWrite, 74
 buffer, 69; 92

C

case, 52
 cerr, 85
 cin, 85
 clearviewport, 160
 cleradevice, 151
 clog, 85
 Close, 74
 close(), 89
 closegraph, 145
 const, 45
 coreleft(), 44
 crt (proud), 85

Č

číslo
 reálné
 formát, 102

D

datový proud. viz proud
 dec, 101
 define. viz #define
 defined, 131
 direktiva

\$I <jméno souboru>, 133
 direktiva překladače, 108
 dispose, 40
 drawpoly, 155

E

EGA_COLORS, 151
 else, 131
 end, 46
 Eof, 74
 Eoln, 75
 Erase, 75
 extern "C", 138

F

far, 30; 39
 farcoreleft(), 44
 farfree(), 44
 farmalloc(), 44
 farrealloc(), 44
 file of, 72
 FileMode, 78
 FilePos, 75
 FileSize, 75
 fill(), 99
 fillellipse, 165
 fillpoly, 165
 floodfill, 165
 Flush, 75; 89
 free(), 44
 FreeMem, 41
 funkce
 blížká, 39
 coreleft(), 44
 Eof, 74
 Eoln, 75
 farcoreleft(), 44
 farfree(), 44
 farmalloc(), 44
 farrealloc(), 44
 fiktivní, 12
 FilePos, 75
 FileSize, 75
 free(), 44
 getdefaultpalette, 150
 getgraphmode, 145
 getmaxmode, 147
 getmaxx, 143

getmaxy, 143
 getmodename, 147
 getpalette, 150
 getpalettesize, 150
 getx, 154
 grafické funkce v C++.
 viz procedura
 graphresult, 141
 imagesize, 160
 IOResult, 75
 malloc(), 44
 MaxAvail, 41
 MemAvail, 41
 realloc(), 44
 robustní, 39
 SeekEof, 79
 SeekEoLn, 79
 setx, 154
 vzdálená, 39

G

gcount(), 90
 get(), 90
 getaspectratio, 157
 getbkcolor, 151
 getcolor, 150
 getdefaultpalette, 150
 GetDir, 80
 getfillpattern, 166
 getgraphmode, 145
 getimage, 160
 getline(), 91
 getlinesettings, 155
 getmaxmode, 147
 getmaxx, 143
 getmaxy, 143
 GetMem, 40
 getmodename, 147
 getpalette, 150
 getpalettesize, 150
 getpixel, 152
 getviewsettings, 160
 getx, 154
 gety, 154
 grafický režim, 140
 inicializace, 142
 textový kurzor, 162
 grafický režim

výstup textu, 161
 grafický režim
 opuštění, 145
 přepínání, 145
 graph_errors, 141
 graphics_drivers, 145
 graphics_modes, 146
 graphresult, 141

H

halda, 40
 heap. viz halda
 hex, 101
 huge, 30; 39

C

ChDir, 80

I

if. viz #if
 imagesize, 160
 in, 64
 include. viz #include
 IncP, 27
 IncX, 27
 initgraph, 142
 input, 73
 IOResult, 75
 ios::open_mode, 86
 ios::state, 87

K

kvalifikace, 48

L

line, 154
 lineto, 154

M

makro, 124
 bez parametrů, 124
 s parametry, 127
 zrušení, 125
 malloc(), 44
 manipulátor

dec, 101
 flush, 89
 hex, 101
 oct, 101
 resetiosflags(), 98
 setbase(), 101
 setfill(), 99
 setprecision(), 102
 setw(), 97

Mark, 41

MaxAvail, 41

MemAvail, 41

metoda

 close(), 89
 fill(), 99
 gcount(), 90
 get(), 90
 getline(), 91
 open(), 89
 peek(), 91
 precision(), 102
 put(), 91
 putback(), 91
 rdbuf(), 93
 read(), 91
 seekg(), 92
 seekp(), 92
 setf(), 98; 99
 tellg(), 92
 unsetf(), 98
 width(), 97
 write(), 91
 Mkdir, 80
 množina, 62
 operace, 64
 model
 drobný (tiny), 33
 kompaktní (compact), 34
 malý (small), 34
 paměťový, 33
 rozsáhlý (huge), 34
 smíšené programování,
 34; 40
 střední (medium), 34
 velký (large), 34
 moverel, 154
 moveto, 154

N

near, 30; 39
 new (operátor), 42
 new (procedura), 40
 NULL, 16
 v C++, 17

O

objekt procedurální, 35
 objekty
 dynamické, 40
 oct, 101
 ofset, 29
 okno
 inspekční, 137
 sledovací, 135
 v grafickém režimu, 159
 Watch, 135
 open(), 89
 operator, 96
 operátor
 !, 43
 a proudy, 94
 #, 128
 ##, 128
 &, 16
 &&, 43
 *, 64
 ., 48
 @, 16
 ||, 43
 +, 64
 ++, 24
 +=", 24
 <<
 uživatelská definice,
 96
 -=", 24
 >>
 uživatelská definice,
 96
 defined, 131
 delete, 42
 a pole, 42
 in, 64
 new, 42
 získání adresy, 16
 output, 73

outtext, 162
outtextxy, 162

P

paleta, 149
palettetype, 149
peek(), 91
pieslice, 165
PInc, 27
pixel, 142
pointer, 14
pole, 19

bitové, 62
konverze na ukazatel, 23
prvek za, 25
ukazatelů, 19
vztah k ukazatelům, 23
poměr výšky a šířky, 157
poznámka

dolarová. viz direktiva
pragma, 113
precision(), 102
preprocessor, 109

procedura

Append, 74
Assign, 73
AssignCrt, 73
BlockRead, 74
BlockWrite, 74
clearviewport, 160
cleradvice, 151
Close, 74
closegraph, 145
dispose, 40
drawpoly, 155
Erase, 75
fillellipse, 165
fillpoly, 165
floodfill, 165
Flush, 75
FreeMem, 41
getaspectratio, 157
getbkcolor, 151
getcolor, 150
GetDir, 80
getfillpattern, 166
getimage, 160
getlinesettings, 155

GetMem, 40
getpixel, 152
getviewsettings, 160
ChDir, 80
IncP, 27
IncX, 27
initgraph, 142
line, 154
lineto, 154
Mark, 41
MkDir, 80
moverel, 154
moveto, 154
new, 40
outtext, 162
outtextxy, 162
pieslice, 165
PInc, 27
putimage, 161
putpixel, 152
Read, 77
ReadLn, 78
Release, 41
Rename, 78
Reset, 78
restorecrtmode, 145
Rewrite, 79
Rmdir, 80
sector, 165
Seek, 79
setactivepage, 164
setaspectratio, 157
setbkcolor, 151
setcolor, 150
setfillstyle, 165
setgraphmode, 145
setlinestyle, 154
setviewport, 160
Truncate, 79
Write, 80
WriteLn, 80
XInc, 27

procesor 80x86, 28
proměnná
FileMode, 78
proud, 67; 83
externí, 67
deklarace, 88

formát reálných čísel,
102
hlavičkové soubory, 85
chybový stav, 87
interní, 67
ladění, 106
otevření, 89
paměťový, 89
deklarace, 89
vstup dat, 90
přeskočení bílých znaků,
104
režim otevření, 86
spláchnutí, 104
šířka pole, 97
velikost písmen, 101
výplňový znak, 99
základ číselné soustavy,
100
zarovnání, 99
zobrazení znaménka, 98

překlad

podmíněný, 129

přepínač, 108

A, 109

D, 110

E, 110

F, 112

G, 112

globální, 109

I, 112

L, 110

lokální, 111

N, 110

O, 111

R, 112

S, 113

V, 113

v Pascalu, 109

X, 111

příkaz

with, 49

put(), 91

putback(), 91

putimage, 161

putpixel, 152

R

rdbuf(), 93
 Read, 77
 read(), 91
 ReadLn, 78
 realloc(), 44
 record, 46
 registr
 a ukazatel, 32
 Release, 41
 Rename, 78
 Reset, 78
 resetiosflags(), 98
 restorecrtmode, 145
 Rewrite, 79
 Rmdir, 80

S

sector, 165
 Seek, 79
 SeekEof, 79
 SeekEoLn, 79
 seekg(), 92
 seekp(), 92
 segment, 29
 datový, 29
 kódový, 29
 zásobníkový, 29
 selektor složky, 48
 set, 63
 setactivepage, 164
 setaspectratio, 157
 setbase(), 101
 setbkcolor, 151
 setcolor, 150
 setf(), 98; 99
 setfill(), 99
 setfillstyle, 165
 setgraphmode, 145
 setlinestyle, 154
 setprecision(), 102
 setviewport, 160
 setw(), 97
 siba (příklad), 22
 simulovaná bakterie. *viz*
 siba
 směrník. *viz* ukazatel

smíšené programování, 34;
 40

soubor, 66
 deklarace v Pascalu, 72
 druhy, 68
 fyzický, 66
 indexsekvencní, 69
 input, 73
 konec, 71
 logický, 66
 otevření, 70
 output, 73
 pozice v něm, 72
 převzetí dat, 71
 s přímým přístupem, 69
 sekvencní, 68
 semisekvencní, 69
 spláchnutí, 71
 textový, 70
 vkládání, 133
 vložení dat, 72
 zavření, 71
 struct, 47
 struktura, 46
 a funkce, 47
 deklarace, 47
 inicializace, 60
 složka, 46

T

tellg(), 92
 tellp(), 92
 Truncate, 79
 typ
 bázový, 63
 doménový, 14
 EGA_COLORS, 151
 graph_errors, 141
 graphics_drivers, 145
 graphics_modes, 146
 ios::open_mode, 86
 ios::state, 87
 palettetype, 149
 pointer, 14
 procedurální, 35
 strukturovaný, 46
 strukturový, 46
 základní ukazatele, 14

typedef, 20

U

ukazatel, 14
 adresová aritmetika, 24
 aritmetika
 v C++, 22
 v Pascalu, 26
 bezdoménový, 14
 blízký, 29
 doménový typ, 14
 konstantní, 44
 konverze, 31
 na konstantu, 44
 na pole, 19
 na proceduru (funkci),
 35
 na strukturu, 50
 na záznam, 50
 normalizovaný, 30
 prázdný, 16
 segmentová část, 33
 v programu pro reálný
 režim 8086, 28
 volný, 14
 vzdálený, 30
 vztah k registrům, 32
 undef. *viz* #undef
 unie, 54
 anonymní, 56
 inicializace, 61
 union, 54
 unsetf(), 98

V

variantní část záznamu, 51
 věta, 66
 videostránka, 164
 viewport, 159
 vyplňování, 165
 výřez, 160

W

width(), 97
 with, 49
 Write, 80
 write(), 91

WriteLn, 80

X

XInc, 27

Z

záznam, 46; 66
deklarace, 46
inicializace, 59

s variantní částí, 51
složka, 46
zdobení jmen, 138
zobrazení znaménka, 98