Obsah

Předmluva

Ve druhém dílu knihy o objektově orientovaném programování vystoupíte na zbývající dva schody – seznámíte se s dědičností a s polymorfismem v C++ a v Turbo Pascalu. Vedle toho se dočtete o pokročilejších vlastnostech jazyka C++, které nemají analogii v Pascalu: Kapitola 7 se zabývá šablonami, kapitola 8 objektovými datovými proudy. Výjimky, o kterých hovoříme v kapitole 9, sice Turbo Pascal nenabízí, najdeme je ale v Object Pascalu v Delphi. V následujících kapitolách se dočtete o dynamické identifikaci typů v C++ a v Delphi, o nových operátorech přetypování zavedených normou jazyka C++ a o prostorech jmen.

Náš výklad v úvodních kapitolách tohoto dílu je založen především na překladačích Borland C++ 3.1 a Turbo Pascal 6.0, které mohou běžet na velké většině počítačů, běžně dostupných nejširší čtenářské obci. Náš výklad o výjimkách, dynamické identifikaci typů a o nových přetypovacích operátorech je založen na překladačích Borland C++ 4.x, výklad o prostorech jmen na Borland C++ 5.0. Výklad o výjimkách v Delphi vychází ze zkušeností s Delphi 1.0; o Delphi se zmiňujeme ovšem pouze okrajově, neboť jde o nástroj pro vývoj aplikací pro Windows 3.1, resp. Windows 95.

První kapitola této knihy navazuje na seriál „Cesta k profesionalitě“, který vycházel v letech 1992 – 1994 v časopisu ComputerWorld a jehož autorem byl R. Pecinovský. Další kapitoly se opírají o publikované i nepublikované díly „Kursu C/C++“, který vycházel v letech 1994 – 1995 v časopisu Bajt a jehož autorem byl M. Virius.

Předpoklady

Od čtenářů opět očekáváme, že jejich znalosti zhruba odpovídají obsahu předchozích dílů. To znamená, že

- ✧ umějí používat běžné programové konstrukce,
- ✧ umějí rozložit úlohu na dílčí algoritmy,
- ✧ umějí dobře zacházet s procedurami a funkcemi a znají jednotlivé způsoby předávání parametrů a vracení vypočtené hodnoty (např. funkce, které v C++ vracejí reference),
- ✧ dobře znají standardní datové typy a umějí definovat vlastní (neobjektové) datové typy,
- ✧ znají základy práce s ukazateli a umějí používat dynamické proměnné,
- ✧ umějí deklarovat a používat objektové typy v obou probíraných jazycích (bez dědičnosti a polymorfismu),
- ✧ umějí zacházet s některým z vývojových prostředí pro tyto jazyky dodávaných firmou Borland, Microsoft, Watcom nebo jinou.

Terminologie

Čtenáři, kteří sledovali časopisecké verze některého z uvedených kursů, 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 „Cesta k profesionalitě“, také označení *přetížené funkce* resp. *operátory*. Jde o doslovný (a často používaný) překlad původních termínů *overloaded function*, resp. *overloaded operator*.

V knize také občas používáme termín *řadová funkce*. Označujeme tak funkce, které nejsou metodami objektových typů (v situacích, kdy je podobné rozlišení potřebné).

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“, „pluskový“, „borlandský“, „pascalista“, „céčkař“ apod., přesto, že proti nim lze mít výhrady – přinejmenším podle mínění jazykových korektorů.

Typografické konvence

V textu knihy používáme následující konvence:

while	Tučně píšeme klíčová slova.
třída	Tučně píšeme nově zaváděné termíny a také pasáže, které chceme z jakýchkoli důvodů zdůraznit.
main()	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.
encapsulation	Kurzivou také píšeme anglické názvy.
ALT+F4	Kapitálky používáme pro vyznačení kláves a klávesových kombinací.
`break;`	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. Najdete na ní úplné zdrojové texty příkladů z jednotlivých kapitol.

2. OOP pro mírně pokročilé

V úvodu knihy Objektové programování I jsme si řekli, že k objektově orientovanému programování (OOP) vedou tři schody: zapouzdření, dědičnost a polymorfismus (mnohotvárnost). V minulé knize jsme vystoupili na první schod a soustředili jsme se na implementaci zapouzdření v obou jazycích. Ukázali jsme si, jaké výhody nám zapouzdření přináší a jak je můžeme ve svých programech využít.

Nyní vystoupíme na další schod na cestě k OOP a začneme si povídat o dědičnosti (*inheritance*) a pak plynule přejdeme k mnohotvárnosti (*polymorphism*) a prostředkům k jejímu naplnění.

Díky širokým možnostem přetěžování mohou programátoři v C++ již nyní považovat OOP za užitečné rozšíření možností jazyka. Nedivili bychom se však, kdyby pascalisté byli prozatím možnostmi OOP zklamáni a vyslovovali nahlas pochybnosti, zda je ten humbuk kolem OOP úměrný přínosu, který jim mohou nové rysy jazyka poskytnout.

Můžeme vás však potěšit sdělením, že v oblasti dědičnosti již nejsou rozdíly tak markantní (z velké části jsou spíše důsledkem odbyté implementace zapouzdření v Pascalu) a na třetím schodě, tj. v implementaci polymorfismu, jsou již možnosti obou jazyků téměř rovnocenné.

Zapouzdření nám umožňovalo zpřehlednit program definicí vhodných datových tříd a snížením nutného počtu identifikátorů (např. v Pascalu se většinou všechny destruktory jmenují *Done*). Naproti tomu využití dědičnosti nám ve spojení s mnohotvárností ušetří mnohé přeprogramovávání dříve vytvořených knihoven a samostatných modulů, protože nám umožní dosáhnout veškerých změn, které potřebujeme, aniž bychom jakýmkoliv způsobem modifikovali jednou napsané a odladěné části programu (samozřejmě za předpokladu, že byly vhodně navrženy).

3. Dědičnost

Dědičnost je způsob, jakým se od jedné třídy odvodí jiná. V podstatě se nejedná o nic složitého: Stejně jako lidé, i datové třídy mohou mít své potomky[1]. Tito potomci dědí vlastnosti svých rodičů a mohou k nim navíc přidat i některé další.

Podobně jako lidé, ani objekty nedědí vše. Nedědí např. přátele a vnořené typy. Z toho, že je někdo přítelem rodiče, nelze nic usuzovat o jeho přátelství k potomkům. Pokud má být daný objekt (podprogram nebo třída) přítelem potomka, musí jej potomek jako svého přítele deklarovat. Jiné cesty není.

Jednou z význačných vlastností dědičnosti je, že všichni potomci jsou považováni nejen za objekty toho datového typu, který jsme uvedli v jejich deklaraci, ale zároveň i za objekty všech jeho rodičovských datových typů. Jinými slovy: Tam, kde potřebujeme instanci předka, můžeme vždy použít instanci potomka. Tam, kde potřebujeme ukazatel, resp. referenci na předka, můžeme použít ukazatel, resp. referenci na potomka. Tato vlastnost je pro OOP klíčová.

Vzájemné vztahy tříd se obvykle graficky znázorňují pomocí stromů; vlastnosti tříd ve stromu dědičnosti se v učebnicích OOP často přirovnávají k vlastnostem klasifikačních stromů. Rodičovská třída definuje vlastnosti obecnější a jednotlivé dceřinné třídy pak některé její vlastnosti blíže specifikují a jiné doplňují.

Zkusme si to ukázat na příkladu. Představte si třídu všech objektů tohoto světa. V ní bychom mohli definovat dvě podtřídy: jednu pro objekty živé a druhou pro neživé. Každá z těchto podtříd by mohla mít opět svoje podtřídy. Víme např., že živé objekty můžeme zařadit mezi živočichy, rostliny nebo houby – přirozeně bychom tedy mohli definovat tři podtřídy. Pro neživé objekty bychom také mohli definovat podtřídy – např. podtřídu přírodních objektů a objektů vzniklých přičiněním člověka.

Předpokládáme, že byste tuto klasifikaci dokázali dlouho úspěšně rozvíjet. Odpoutejme se však na chvíli od vlastní klasifikace a podívejme se, jak bychom mohli tyto třídy reprezentovat v programu.

Začněme třídou obecných objektů, kterou si nazveme prostě *Objekt*. Pro tuto třídu definujeme statický atribut *Vzniklo*, ve kterém bude uložen počet doposud vzniklých instancí. Pro každý objekt pak definujeme textový atribut *Jméno*, ve kterém si bude uchovávat svůj název, a celočíselný konstantní atribut *RodCis* (rodné číslo), v němž bude uloženo pořadí jeho vzniku mezi ostatními objekty.

Pro datovou třídu *Objekt* můžeme dále definovat i dvě metody: statickou metodu *JménoTřídy*, která bude vracet textový řetězec s názvem třídy, a nestatickou metodu *Tiskni*, která vytiskne veškeré dostupné údaje o objektu. Kromě toho definujeme pro tu-

[1] Třídu, **od** které odvozujeme, označujeme jako rodiče, rodičovskou třídu, nadtřídu nebo také jako bázovou třídu. Odvozenou třídu pak označujeme jako potomka, podtřídu nebo dceřinnou (případně synovskou) třídu. U termínů „podtřída“ a „nadtřída“ je ale třeba dávat pozor, neboť někteří programátoři používají obrácené názvy – jako podtřídu označují předka. (Instance potomka v sobě totiž obsahuje vždy instanci předka.)

to třídu destruktor a jednoparametrický konstruktor, kterému předáme jméno konstruovaného objektu. Po obou budeme chtít, aby nám podaly zprávu o své činnosti.

Pokusíme-li se takto koncipovanou třídu definovat prostředky jazyka, obdržíme následující deklarace:

```
/*  Příklad C2 - 1  */
typedef unsigned word; //Abychom měli s Pascalem stejné identifikátory
class /*****/ Objekt /*****/
{
 public:
 Objekt( const char *Jm="???" );
 ~Objekt();
 void Tiskni( const char *Text = "===" );
 static char *JmenoTridy() {return "Objekt"; }
private:
 const char *Jmeno;
 const word RodCis;
 static word Vzniklo;
};
/********** Object **********/
word Objekt::Vzniklo = 0;

/*****/ Objekt::Objekt /*****/
( const char *Jm )
: Jmeno( Jm ), RodCis( ++Vzniklo )
{ Tiskni( "KONSTR" );
}/********** Objekt::Objekt **********/

/*****/ Objekt::~Objekt /*****/
()
{ Tiskni( "DESTR" );
}/********** Objekt::~Objekt **********/

#define left setiosflags( ios::left )
#define right setiosflags( ios::right )

void /*****/ Objekt::Tiskni /*****/
( const char *Text )
{ cout << '\n' << left << setw(20) << Text
       << " Rod. č: " << right << setw(3) << RodCis
       << " Objekt: " << left << setw(15) << Jmeno
       << " Třída: " << setw(10) << JmenoTridy() << right;
}/********** Objekt::Tiskni **********/

Objekt o1 = "První objekt";
Objekt o2 = "Druhý objekt";

void /*****/ Test_1 /*****/ ()
{ o1.Tiskni( "Tisk" );
 o2.Tiskni( "Tisk" );
}/********** Test_1 **********/
```

Než přistoupíme k vlastní deklaraci třídy v Pascalu, dovolte nám ještě malou poznámku o grafické úpravě. Jedna z věcí, které nás na Pascalu mrzí, je, že zařazení deklarace do

sekce **private** je zároveň příkazem pro debugger, aby zde deklarované atributy zatajil i před programátorem. Aby před námi debugger nezamlčoval vnitřní strukturu jednotlivých instancí, přestaneme klíčové slovo **private** používat.

Abychom se ale snadno dohodli, které složky je vhodné deklarovat jako veřejně přístupné a které jako soukromé, budeme toto klíčové slovo sice v programu uvádět, ale vložíme je do komentářových závorek, takže překladač o něm nebude vědět a debugger pak před námi nebude nic zatajovat.

Aby byly naše programy ještě dokonalejší, měli bychom v našich programech dodržovat zásadu, že věci potřebné mají být snadno k nalezení a věci nepotřebné nemají naopak zaclánět a bránit v hledání věcí potřebných. Měli bychom proto řadit deklarace datových složek tak, aby veřejné složky byly uvedeny na počátku definice, kde bychom je mohli v případě potřeby rychle najít, a aby soukromé složky, které by stejně neměl nikdo cizí používat, byly umístěny raději někde na konci definice, kde nebudou příliš rušit.

Tato chvályhodná zásada však trochu koliduje s předchozí dohodou o zakrytí klíčového slova **private**, protože Pascal požaduje, aby v každé sekci byly nejprve deklarovány atributy a teprve pak metody. V programech však většinou bývají metody veřejné a atributy soukromé. V Pascalu 7.0 a v Delphi si můžeme velice snadno poradit tím, že využijeme nově zavedeného klíčového slova **public**, které uvedeme za komentářovými závorkami s ukrytým **private**. Tím otevřeme další sekci a v ní můžeme deklarovat potřebné atributy bez problémů.

Uživatelé Pascalu 6.0 tuto fintu používat nemohou. Mohou si však vždy uspořádat deklarace tak, aby byl překladač s jejich pořadím spokojen.

V souvislosti s „vykomentováním“ klíčového slova **private** bychom vás chtěli upozornit ještě na jednu fintu. Všimněte si, jak jsou v následujícím prográmku kolem tohoto klíčového slova uspořádány komentářové závorky. Interpretace celého řádku nyní významně závisí na tom, jakým znakem řádek začíná.

Pokud bude začínat tak, jako v následující ukázce, tj. otevírací komentářovou závorkou, bude počátek řádku až do odpovídající zavírací komentářové závorky (tj. do zavírací složené závorky) považován za komentář následovaný klíčovým slovem **public** a prázdným komentářem.

Pokud počáteční komentářovou závorku smažeme, bude řádek začínat klíčovým slovem **public** následovaným komentářem. Protože otevírací komentářovou závorkou je tentokrát dvojznak z otevírací kulaté závorky a hvězdičky, bude následující zavírací složená závorka považována za znak komentáře a komentář skončí až na konci řádku zavírací komentářovou závorkou tvořenou hvězdičkou a zavírací kulatou závorkou. (Využíváme tak vlastně skutečnosti, že Turbo Pascal nedovoluje vnořování komentářů.)

Na základě tohoto triku můžeme změnou jediného znaku ovlivnit, zda bude následující sekce přeložena tak, jak nám to vyhovuje při krokování, tj. jako sekce **public**, nebo naopak tak, abychom mohli snáze vyhledat všechna případná porušení přístupových práv, tj. jako sekce **private**.

```
(*  Příklad P2 - 1  *)
type
 (*****) Objekt (******) = object
```

```
 public
 constructor Init( const Jm : PString );
 destructor Done;
 procedure Tiskni( const Txt : PString );
 function JmenoTridy: PString;
 {private (*{} public (**)
 Jmeno : PString;
 RodCis: word;
 end;
 (********** Object **********)

const
 Vzniklo : word = 0;

function Objekt.JmenoTridy:PString;
begin
 JmenoTridy := 'Object';
end;

constructor (*****) Objekt.Init (******)
( const Jm : PString );
begin
 Jmeno := Jm;
 Inc( Vzniklo );
 RodCis := Vzniklo;
 Tiskni( 'KONSTR' );
end;
(********** Objekt.Init **********)

destructor (*****) Objekt.Done (******)
;
begin
 Tiskni( 'DESTR' );
end;
(********** Objekt.Done **********)

procedure (*****) writs (******)
( S:PString; d:word );
{Pomocná procedura pro tisk řetězců S do sloupce širokého d znaků
 tak, aby byly pod sebou vytištěné řetězce zarovnány vlevo}
var
 l:integer;
begin
 {$ifdef Pascal_7_0}
 l := strlen( S );
 {$else}
 l := length( S );
 {$endif}
 write( S );
 if( (d-l) > 0 )then
 write( ' ':(d-l) );
end;
(********** writs **********)
procedure (*****) Objekt.Tiskni (******)
( const Txt : PString );
begin
 writeln;
```

```
 write( Txt, 15 );
 write( ' Rod. č:', RodCis:3 );
 write( ' - Objekt: ' ); writs( Jmeno, 15 );
 write( ' Třída: ' ); writs( JmenoTridy, 10 );
end;
(********** Objekt.Tiskni **********)
var
 o1 : Objekt;
 o2 : Objekt;

procedure (*****) Test_1 (******)
;
begin
 o1.Init( 'První objekt' );
 o2.Init( 'Druhý objekt' );
 o1.Tiskni( 'Tisk' );
 o2.Tiskni( 'Tisk' );
 {Objekty zde nedestruujeme, protože je ještě budeme využívat}
end;
(********** Test_1 **********)
```

Tolik tedy ke třídě *Objekt*. Než se pustíme do jejích potomků, vrátíme se na chvíli k povídání o dědičnosti jako takové.

Již víme, že všichni potomci jsou považováni nejen za objekty toho objektového typu, který jsme uvedli v jejich deklaraci, ale zároveň i za objekty všech jeho rodičovských typů. Toho využijeme zejména ve volání různých funkcí. Musíme ovšem mít na paměti, že pokud s danou instancí zacházíme tak, jako kdyby byla instancí některého z rodičovských typů (předků), můžeme používat také pouze odpovídající podmnožinu jejích atributů a metod.

Než postoupíme dále, musíme si něco povědět o přístupových právech.

3.1 Přístupová práva

V C++ jsme se zatím setkali se dvěma typy přístupových práv ke složkám třídy: složky jsme rozdělovali na veřejné a soukromé. S veřejnými složkami může pracovat kdokoliv, se soukromými složkami mohou pracovat pouze metody dané třídy, metody spřátelených tříd a spřátelené funkce.

V praktickém programování se často vyskytnou situace, ve kterých byste potřebovali zpřístupnit některé složky nejen přátelům, ale i potomkům, avšak na druhou stranu byste je chtěli před zbytkem programu zatajit. Pro tyto účely byla do C++ zavedena třetí možnost ochrany, deklarovaná klíčovým slovem **protected** (chráněný). Chráněné složky jsou přístupné nejen pro metody třídy a její přátele, ale také pro všechny její potomky a jejich přátele.

Pascalisté mají život na jednu stranu jednodušší a na druhou složitější. Jednodušší v tom, že v Pascalu nejsou přístupová práva vázána na třídu, ale na modul – jednotku (*unit*). Umožňuje nám tedy rozdělit atributy a metody na soukromé (**private**), které budou známy pouze v daném modulu, a na veřejně přístupné (**public**), které budou známy

i v jiných modulech. (Jde vlastně trochu o koncepci ochrany, založenou na představě, že každá třída by měla mít svůj vlastní modul.)

Nevýhodou tohoto přístupu je, že Pascal zde ustupuje od své zásady hlídat programátora všude, kde je to možné, a ponechává dodržování konvencí zcela na programátorovi.

Turbo Pascal nezná analogii pluskové specifikace **protected**; s tou se setkáme až v Object Pascalu v Delphi. Podobně jako v C++ deklarujeme v chráněné (**protected**) sekci složky, které budou přístupné metodám daného typu a jeho potomků, ale nikomu jinému.

Vraťme se nyní k naší třídě *Object***.** Zamyslíme-li se nad ní trochu, zjistíme, že všechny její atributy by asi bylo vhodné deklarovat jako chráněné. Definujme si proto třídu *cObjekt_0* (číslujeme ji, protože do své definitivní verze bude potřebovat ještě několik modifikací), do jejíž definice zaneseme výše uvedenou modifikaci. Definice třídy by tedy mohla vypadat následovně (definice jejích metod budou shodné s definicemi třídy *Objekt*, a proto je vypustíme):

```
/*  Příklad C2 - 2  */
typedef unsigned word; //Abychom měli s Pascalem stejné identifikátory
class /*****/ cObjekt_0 /*****/
{ public:
 cObjekt_0( const char *Jm="???" );    //Konstruktor definující jméno
 ~cObjekt_0();
 void Tiskni( const char *Text         //Pomocné tisky - parametr
 = "===" );                            //definuje nadpis pomoc. tisku
 static char *JmenoTridy()
 {return "Objekt"; }
protected:                  //Atributy jsou deklarovány jako chráněné
 const char *Jmeno;                    //Jméno instance
 const word RodCis;                    //Pořadí vzniku objektu
 static word Vzniklo;                  //Počet dosud zkonstr. objektů
};
/********** cObjekt_0 **********/
```

Ani v pascalské definici nebudeme uvádět těla metod:

```
(*  Příklad P2 - 2  *)
type
 (*****) cObjekt_0 (******) = object
 public
 constructor Init( const Jm : PString );    {Konstr. definující jméno}
 destructor Done;
 procedure Tiskni( const Txt : PString );   {Pomocné tisky }
function JmenoTridy: PString;
 {private (*{} public (**)
 Jmeno : PString;                           {Jméno dané instance }
RodCis: word;                               {Pořadí vzniku instance }
end;
 (********** cObjekt_0 **********)
const
```

```
Vzniklo : word = 0;                          {Počet dosud zkonstr.inst.}
```

Třída *cObjekt_0* bude mít dvě **dceřinné třídy**: třídu živých objektů *cŽivý* a třídu neživých objektů *cNeživý*. Neživé objekty přidají ke zděděným atributům údaj o převažujícím skupenství, kdežto živé objekty přidají informaci o počtu chromozomů.

Z definice je zřejmé, že bude třeba znovu definovat metodu *JménoTřídy*, avšak při bližším pohledu zjistíme, že vzhledem k přibyvším atributům bude třeba pro obě dceřinné třídy znovu definovat i ostatní metody.

V C++ deklarujeme dceřinnou třídu tak, že za identifikátor deklarované třídy napíšeme dvojtečku, za ní uvedeme seznam identifikátorů jejích rodičovských tříd spolu s případným popisem rodičovského vztahu a za ním pak ve složených závorkách seznam složek dané třídy, které daná dceřinná třída přidává k zděděným složkám nebo kterými tyto složky modifikuje. Syntaktická definice této deklarace by mohla vypadat následovně:

Deklarace třídy:
<TypTřídy> <Identifikátor> [: <SpecifRodiče>]
+ {<Seznam deklarací složek> }

TypTřídy:
1 z: class struct union

SpecifRodiče: *//Specifikace rodiče*
[<SpecifPřístupu>] <Identifikátor>

SpecifPřístupu: *//Specifikace přístupu*
1 z: public protected private

Poznámky:

- ✧ *Předchozí syntaktická definice není ještě úplná, protože v C++ mohou mít třídy (na rozdíl od lidí) teoreticky libovolný počet přímých rodičů (prarodiče nejsou přímými rodiči). My se však prozatím omezíme pouze na situace, kdy bude mít třída jediného přímého rodiče. K třídám s více rodiči, tedy k tzv. vícenásobné dědičnosti, se vrátíme až později (Pascal je totiž neumí).*
- ✧ ***Unie nesmějí vystupovat v dědické hierarchii, tj. nesmějí být rodičem ani potomkem**.*
- ✧ *Specifikace chráněného rodičovství (rodič deklarovaný jako* **protected***) byla do borlandských překladačů zavedena až od verze 3.0.*

Specifikací přístupu definujeme nejvolnější možný přístup k zděděným složkám. Pokud třída deklaruje svého rodiče se specifikací **public**, budou přístupová práva ke zděděným složkám v dceřinné třídě naprosto stejná, jako byla ve třídě rodičovské. Deklarujeme-li rodiče se specifikací **protected**, omezí se přístupová práva k veřejným zděděným složkám. Složky, které byly v předkovi veřejné nebo chráněné, budou v potomkovi vystupovat jako chráněné, a složky, které byly v předkovi soukromé, zůstanou soukromé i v potomkovi. Deklarujeme-li rodiče se specifikací **private**, budou všechny zděděné

složky považovány za soukromé nezávisle na tom, jaká byla jejich přístupová práva v rodičovské třídě.

Možná, že vše bude pochopitelnější, když posuny specifikací uspořádáme do tabulky. V záhlaví řádků uvedeme deklarovanou specifikaci přístupu k rodiči a v záhlaví sloupců pak specifikaci přístupu k dané složce deklarovanou v definici rodičovské třídy. V příslušné kolonce pak bude výsledná charakteristika přístupu k zděděné složce v dceřinné třídě.

Rodič \ Složka	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Pokud u rodiče nedeklarujeme specifikaci přístupu, platí implicitní specifikace, která je stejná jako implicitní specifikace přístupu k jednotlivým složkám: u třídy typu **struct** je to specifikace **public** a u tříd typu **class** specifikace **private**.

Podívejme se nyní na definici tříd *cŽivý* a *cNeživý*:

```
/*  Příklad C2 - 3  */
class /*****/ cZivy /*****/
: public cObjekt_0
{ public:
 cZivy( int Chrom, const char *Jm="???" );
 ~cZivy();
 void Tiskni( const char *Text = "===" );
 static char *JmenoTridy()  {return "cŽivý"; }
protected:
 int Chromozomu;
};
/********** cZivy **********/

enum eSkup {NEDEF_SKUPENSTVI, PEVNE, TEKUTE, TUHE, PLAZMA, _eSkup };

class /*****/ cNezivy /*****/
: public cObjekt_0
{ public:
 cNezivy( eSkup Skup, const char *Jm="???" );
 ~cNezivy();
 void Tiskni( const char *Text = "===" );
 static char *JmenoTridy() {return "cŽivý"; }
protected:
 eSkup Skupenstvi;
};
/********** cNezivy **********/
```

Pascal řeší otázku deklarace rodičovství mnohem jednodušeji, avšak tato jednoduchost je vykoupena také zmenšenými možnostmi. Dceřinou třídu deklarujeme v Pascalu tak, že za klíčové slovo **object** uvedeme v kulatých závorkách jméno rodičovské třídy. Syntaktickou definici deklarace třídy v Pascalu bychom tedy mohli zapsat:

DeklaraceTřídy:
<IdentTřídy> = **object (** *<IdentRodiče>* **)**
+ *<DaklaraceSložek>* **end;**

Deklarace výše uvedených tříd by pak mohla mít tvar:

```
(*  Příklad P2 - 3  *)
type
 (*****) Zivy (******) = object( cObjekt_0 )
 public
 constructor Init( Chrom:integer; const Jm:PString );
 destructor Done;
 procedure Tiskni( const Txt:PString );
 function JmenoTridy:PString;
 {private (*{} public (**)
 Chromozomu: integer;
 end;
 (********** Zivy **********)

 eSkup = ( NEDEF_SKUPENSTVI, PEVNE, TEKUTE, TUHE, PLAZMA );

 (*****) Nezivy (******) = object( cObjekt_0 )
 public
 constructor Init( Skup:eSkup; const Jm:PString );
 destructor Done;
 procedure Tiskni( const Txt:PString );
 function JmenoTridy:PString;
 {private (*{} public (**)
 Skupenstvi: eSkup;
 end;
 (********** Nezivy **********)
```

Vlastní deklarace žádné komplikace nepřináší. Problémy se objeví až ve chvíli, kdy budeme chtít definovat těla jednotlivých metod. Pokud nebudeme mít na paměti některá specifika dědičnosti, určitě nebudou dělat to, co bychom chtěli. První, nač si posvítíme, budou konstruktory.

3.2 Konstruktory

Konstruktory dceřinných tříd mají za úkol zkonstruovat objekt, jehož datové složky bychom mohli rozdělit na dvě části: na část zděděnou a na část nově přidanou. V C++ na to myslí překladač, který před tělem konstruktoru dané třídy zavolá nejprve konstruktor jejího rodiče a zkonstruuje zděděnou část datových složek. V Pascalu na to musí myslet programátor a konstruktor zděděné části zavolat sám.

Pokud programátor nenaznačí něco jiného, vyvolá překladač C++ nejprve bezparametrický konstruktor rodičovské třídy a poté začne provádět příkazy vlastního těla kon-

struktoru. Pokud chcete inicializovat zděděné složky jiným způsobem, musíte v hlavičce konstruktoru uvést za dvojtečkou odpovídající volání rodičovského konstruktoru.

```
/*   Příklad C2 - 4   */
void /*****/ cZivy::cZivy /*****/
( int Chrom, const char *Jm )
: Chromozomu( Chrom ),
 cObjekt( Jm )
{ cout << "\n --- Chromozomů: " << Chromozomu;
}/********** cZivy::cZivy **********/

void /*****/ cNezivy::cNezivy /*****/
( Skup:eSkup, const char *Jm )
: Skupenstvi( Chrom ),
 cObjekt( Jm )
{ static char* Nazev[] =
 {"NEDEF ", "PEVNE ", "TEKUTE", "TUHE ", "PLAZMA" };
 cout << "\n --- Skupenství: " << Nazev[ Skup ];
}/********** cNezivy::cNezivy **********/
```

Předchozí prográmky si odkrokujte. Všimněte si přitom, že pořadí volání jednotlivých položek inicializačního seznamu není dáno pořadím uvedení těchto položek v seznamu, ale pořadím deklarace inicializované složky. Je tedy zákonité, že bez ohledu na to, že v obou případech je konstruktor rodičovské třídy uveden jako druhý, bude vyvolán jako první.

Kromě toho při krokování asi zjistíte, že konstruktor nedělá přesně to, co má. O tom si ale něco povíme až po následující části věnované Pascalu.

Chcete-li zavolat kteroukoli zděděnou proceduru nebo funkci, musíte ji kvalifikovat jménem odpovídající rodičovské třídy, aby překladač věděl, kde ji má hledat. Pokud používáte sedmou nebo pozdější verzi borlandského Pascalu, můžete navíc využít i klíčového slova **inherited**, jehož prostřednictvím se odvoláte na odpovídající metodu bezprostředního rodiče. V následující ukázce vám předvedeme obě verze:

```
(*  Příklad P2 - 4  *)
constructor (*****) cZivy.Init (******)
( Chrom:integer; const Jm:PString );
begin
 cObjekt_0.Init( Jm ); //Řešení pro Pascal 6.0 i 7.0
 Chromozomu := Chrom;
 writeln;
 write( ' --- Chromozómů: ', Chrom:2 );
end;
(********** cZivy.Init **********)

constructor (*****) cNezivy.Init (******)
( Skup:eSkup; const Jm:PString );
const
 Nazev: array[ eSkup ] of String[6] =
 ( 'NEDEF ', 'PEVNE ', 'TEKUTE', 'TUHE ', 'PLAZMA' );
begin
```

```
 inherited Init( Jm ); //Řešení pouze pro Pascal 7.0 a pozdější
 Skupenství := Skup;
 writeln;
 write( ' --- Skupenstvi: ', Nazev[Skup] );
end;
(********** cNezivy.Init **********)
```

Zavoláme-li takto definovaný konstruktor, asi se nám výsledek nebude líbit. Konstruktor totiž vytiskne zprávu o tom, že konstruuje objekt třídy *cObjekt_0*, přičemž my víme, že konstruuje instanci některého z jejích potomků. Na první pohled by nás mohlo napadnout, že bychom vlastně v rodičovské třídě měli definovat dva konstruktory: první bude pomocný a bude provádět pouze nejzákladnější operace spojené s konstrukcí objektu, tj. zřídí danou instanci a přiřadí jí počáteční hodnotu. Druhý konstruktor bude plnohodnotný, tj. provede všechny námi požadované operace včetně tisku.

Abychom mohli obdobným způsobem konstruovat instance i u „vnoučat“ a „pravnoučat“, musíme ovšem obdobnou dvojicí konstruktorů vybavit každou generaci potomků.

Jak jsme si řekli, pomocný konstruktor definujeme pouze proto, abychom jej mohli používat v konstruktorech potomků. Z toho by zároveň měla vyplývat i jeho přístupová práva. Teoreticky by k němu neměl mít přístup nikdo jiný, než právě konstruktory potomků. Toho však dosáhnout nelze. O tom, jaká přístupová práva tedy zvolit, si povíme v následujících pasážích věnovaných specifičnostem probíraných jazyků.

Asi vás již napadlo, že v C++ bude nejlepší deklarovat pomocný konstruktor jako chráněný. To znamená, že k němu budou mít přístup pouze metody dané třídy a metody jejich potomků (a samozřejmě také jejich přátelé). Výsledná deklarace by tedy mohla vypadat následovně:

```
/*   Příklad C2 – 5   */
class /*****/ cObjekt_1 /*****/
{ public:
 cObjekt_1( const char *Jm="???" )     //Úplný konstruktor
 :Jmeno( Jm ),
  RodCis( ++Vzniklo )
 {Tiskni( Jm ); }
 void Tiskni( const char *Text         //Pomocné tisky - parametr
           = "===" );                 //definuje nadpis pomocného tisku
 static char *JmenoTridy()
 {return "cObjekt_1"; }
protected:
 cObjekt_1( int, const char *Jm="???" )
 :Jmeno( Jm ),
  RodCis( ++Vzniklo )
 {}
 const char *Jmeno;                    //Jméno instance
 const word RodCis;                    //Pořadí vzniku objektu
 static word Vzniklo;                  //Počet dosud zkonstr. objektů
};
word cObjekt_1::Vzniklo = 0;
```

```
/********** class cObjekt_1 **********/

class /*****/ cZivy_1 /*****/
: public cObjekt_1
{ public:
 cZivy_1( const char *Jm="???", int Chrom = 0)
 :cObjekt_1( 0, Jm ),
 Chromozomu( Chrom )
   {Tiskni( Jm ); }
 void Tiskni( const char *Text = "===" );
 static char *JmenoTridy()
   {return "cŽivý_1"; }
protected:
 cZivy_1( int, const char *Jm="???", int Chrom = 0)
 :cObjekt_1( 0, Jm ),
 Chromozomu( Chrom )
   {}
 int Chromozomu;
};
/********** class cZivy_1 **********/

void /*****/ Test_1 /*****/
()
{ cObjekt_1 o1( "První objekt" );
 cZivy_1 z1( "První živý", 1 );
 cObjekt_1 o2( "Druhý objekt" );
 cZivy_1 z2( "Druhý živý", 1 );
};
/********** cTest_1 **********/
```

Možná, že vás napadlo, zda by nebylo možno využít v definici úplného konstruktoru jednodušší definice pomocného konstruktoru. Jistě, jde to. V tělech konstruktorů, podobně jako v tělech ostatních metod, je k dispozici ukazatel **this** na právě konstruovanou instanci. Stačí tedy v úplném konstruktoru přiřadit proměnné ***this** hodnotu vytvořenou pomocným konstruktorem. To znamená, že definice konstruktorů třídy *cZivy* by mohly vypadat např. takto:

```
/*   Příklad C2 – 6   */
// Pomocný konstruktor
cZivy_1( int, char *Jm="???", int Chrom = 0)
: cObjekt_1( 0, Jm ),
  Chromozomu( Chrom )
 {}
// Úplný konstruktor
cZivy_1( char *Jm="???", int Chrom = 0)
{                                  // Zavoláme pomocný konstruktor
 *this = cZivy_1(0,Jm, Chrom);
 Tiskni( Jm );
}
```

Všimněte si, že zde konstruktor vystupuje vlastně jako funkce, která vrací vytvořenou instanci[2].

Toto řešení ovšem není nejlepší, a to hned z několika důvodů. Za prvé se zde vcelku zbytečně vytváří pomocná instance. Za druhé se pomocná instance přiřazuje, přenáší do instance, kterou právě vytváříme. To samozřejmě znamená řadu operací navíc; vedle toho musí být pro naši třídu k dispozici přiřazovací operátor, a to není úplně samozřejmé. Už víme, že pokud jej nedeklarujeme explicitně, pokusí se překladač vytvořit si jej sám. Tento implicitní přiřazovací operátor nám ale nemusí vyhovovat. Může se také stát, že si jej překladač nedokáže sám vytvořit (to je právě případ třídy *cZivý*)[3].

Druhou, univerzálnější možností je definovat pomocnou (a samozřejmě soukromou) funkci, která provede operace, jež se v obou konstruktorech opakují, a kterou pak v obou konstruktorech zavoláme.

V Pascalu je to s přístupovými právy trochu složitější, protože, jak víme, tam se přístupová práva neomezují na třídu, ale na modul. Pokud jsou tedy všechny konstruktory potomků, využívající služeb pomocného konstruktoru, definovány ve stejném modulu, v němž je definován i pomocný konstruktor, můžeme jej deklarovat jako soukromý (**private**). V opačném případě jej musíme deklarovat jako veřejný (**public**).

---

[2] Pokud v předchozím příkladu upravíte uvedeným způsobem deklarace konstruktorů, nepovede se příklad přeložit, neboť překladač nedokáže vytvořit pro třídu *cZivy* přiřazovací operátor. Aby tento příklad fungoval, musíme např. v definicích tříd *cObjekt_1* a *cZivy* nahradit konstantní atributy nekonstantními. Úplný fungující zdrojový text najdete na doplňkové disketě.

[3] Překladač nedokáže vytvořit pro nějakou třídu přiřazovací operátor, jestliže

- ✧ tato třída obsahuje konstantní nebo referenční složky,
- ✧ předek dané třídy obsahuje soukromý přiřazovací operátor,
- ✧ atribut (nezděděná složka) obsahuje přiřazovací operátor, který je vzhledem k přístupovým právům nedostupný,
- ✧ překladač nedokáže vytvořit přiřazovací operátor pro předka dané třídy nebo některý z jejích atributů.

Abychom si zbytečně nekomplikovali výklad, budeme od nynějška deklarovat všechny konstruktory v Pascalu jako veřejně přístupné.

```
(*   Příklad P2 – 5   *)
type
 (*****) cObjekt_1 (******) = object
  Jmeno : String;                          {Jméno dané instance }
  RodCis: word;                            {Pořadí vzniku instance }
  constructor Init( const Jm : String );   {Plnohodnotný konstruktor}
  constructor Init0(const Jm : String );   {Pomocný konstruktor }
  procedure Tiskni( const Txt : String );  {Tisk obsahu složek }
  function JmenoTridy: String;
 end;

 const
 Vzniklo : word = 0; {Počet dosud zkonstruovaných instancí}
(********** cObjekt_1 **********)

type
 (*****) cZivy_1 (******) = object( cObjekt_1 )
  Chromozomu: integer;
  constructor Init( const Jm:String; Chrom:integer );
  constructor Init0( const Jm:String; Chrom:integer );
  procedure Tiskni( const Txt:String );
  function JmenoTridy:String;
 end;
 (********** cZivy_1 **********)

constructor (*****) cObjekt_1.Init (******)
( const Jm:String );
begin
  Jmeno := Jm;
  Inc( Vzniklo );
  RodCis := Vzniklo;
  Tiskni( 'KONSTR' );
end;
(********** cObjekt_1.Init **********)

constructor (*****) cObjekt_1.Init0 (******)
( const Jm:String );
begin
  Jmeno := Jm;
  Inc( Vzniklo );
  RodCis := Vzniklo;
end;
(********** cObjekt_1.Init0 **********)

constructor (*****) cZivy_1.Init (******)
( const Jm:String; Chrom:integer );
begin
  cObjekt_1.Init0( Jm );
  Chromozomu := Chrom;
  Tiskni( 'KONSTR' );
end;
(********** cZivy_1.Init **********)
```

```
constructor (*****) cZivy_1.Init0 (******)
( const Jm:String; Chrom:integer );
begin
  cObjekt_1.Init0( Jm );
  Chromozomu := Chrom;
end;
(********** cZivy_1.Init0 **********)

procedure (*****) Test_1 (******)
;
var
  o1, o2 : cObjekt_1;
  z1, z2 : cZivy_1;
begin
  o1.Init( 'První objekt' );
  z1.Init( 'První živý', 1 );
  o2.Init( 'Druhý objekt' );
  z2.Init( 'Druhý živý', 1 );
end;
(********** Test_1 **********)
```

Možná vás napadlo, že by bylo rozumné využít v definici úplného konstruktoru jednodušší definici pomocného konstruktoru. Vřele vám to nedoporučujeme. V předchozím příkladu by se sice nic nestalo, ale pokud bychom měli v dané třídě nějaké virtuální metody (už brzy se s nimi seznámíme), vznikl by nefunkční program.

Pokud je tělo pomocného konstruktoru složitější, můžeme si ušetřit jeho opisování tím, že definujeme pomocnou proceduru, kterou pak v obou konstruktorech zavoláme.

Jak vidíte, řešení je (zejména v Pascalu) trochu upovídané. Naštěstí v praxi ve většině případů vystačíme s konstruktory, které nejsou o mnoho komplikovanější než naše pomocné konstruktory a které hlavně výše popisované dělení na plnohodnotné a pomocné konstruktory nepotřebují.

3.3 Překrývání metod

Podívejme se nyní na definice obou plnohodnotných konstruktorů. Vidíme, že oba volají metodu **Tiskni**. Každý však volá jinou verzi této metody. Je to tím, že jsme definicí metody **Tiskni** ve třídě *cŽivý_1* **překryli** její definici ve třídě *cObjekt_1*.

Filozofie překrývání má mnoho společného s pojmy, jako je obor viditelnosti, globální a lokální identifikátory konstant, proměnných, podprogramů a typů. Jak víme, objekt, který je definován na nějaké úrovni, je dostupný na všech vnořených úrovních do té doby, dokud na některé z vnořených úrovní nedefinujeme jiný objekt se stejným identifikátorem. Totéž platí i pro třídy, i když se možnosti liší podle použitého programovacího jazyka.

V C++ můžete v potomkovi překrýt kterýkoli z identifikátorů složek kteréhokoli z jeho rodičů a při tomto překrývání nejste vázáni žádnými omezeními na charakter složky, je-

jíž identifikátor překrýváte. Složky, jejichž identifikátory jsou překryty, jsou dostupné postupnou kvalifikací jménem třídy následovaným dvojtečkou. Například takto:

```
/*   Příklad C2 – 7   */
struct /*****/ A /*****/
{
  int i;                          //Celočíselná proměnná
  int f( int a )      //Celočíselná funkce s celočíselným parametrem
  {return -a; }
};
struct /*****/ B:A /*****/
{
  int f;                          //Celočíselná proměnná
  int i( int a )      //Celočíselná funkce s celočíselným parametrem
  {return 100*a; }
};

struct /*****/ C:B /*****/
{
  int i;                          //Celočíselná proměnná
  int f( int a )      //Celočíselná funkce s celočíselným parametrem
  {return -100*a; }
};

int /*****/ TestKryti /*****/ ()
{
  A a = {1 };
  B b;
  C c;
  a.i = a. f( 1000 );
  b.A::i= b. i( 200 );
  b.f = b.A::f( 2 );
  c.B::f= c.B::i( 3 );
  c.A::i= c.A::f( 4 );
  c.i = c. f( 5 );
  return( a.i + b.f + c.i );
}
```

V Pascalu jsme omezeni tím, že nám překladač nedovolí použít identifikátorů datových složek rodičů a prarodičů, stejně jako nám nedovolí definovat datovou složku, jejíž identifikátor by se shodoval s identifikátorem některé zděděné metody. Jediné, co nám Pascal povolí, je překrýt zděděné metody, tedy definovat v potomkovi metody se stejným jménem jako v předkovi. Naštěstí však netrvá na tom, aby měla nově definovaná metoda stejný prototyp, jako metoda překrývaná (ono by to ani nešlo) a dokonce nám dovolí překrýt proceduru funkcí a naopak.

Nepříjemné je ovšem to, že Pascal neposkytuje žádný přímý způsob, jak se odvolat na nějakou překrytou metodu. Podívejte se na následující ukázku:

```
(*   Příklad P2 – 6   *)
type
  int = integer;
```

```
(*****) TA (*****) = object
   i:int;
   function f( p:int ):int;
   procedure g( var a:TA );
  end;
 (********** TA **********)
  PTA = ^TA;

(*****) TB (*****) = object( TA )
   j:int;
   procedure f( b:TB; q:int );
  end;
 (********** TB **********)

function (*****) TA.f (*****)
( p:int ):int;
begin
  f := 10 * (i + p);
end;
(********** TA.f **********)

procedure (*****) TA.g (*****)
( var a:TA );
begin
  i := i + a.i;
end;
(********** TA.g **********)

procedure (*****) TB.f (*****)
( b:TB; q:int );
begin
  {j := j + inherited f( q ); {Metoda instance self - jen Pascal 7.0}
  {j := j + TA.f( q ); {Metoda instance self}
  {j := j + b.TA.f( q ); {Očekáván ident. složky}
  {j := j + TA(b).f( q ); {Nekorektní přetypování}
  j := j + PTA(@b)^.f( q );
end;
(********** TB.f **********)

const
  a1:TA=( i:10 );
  b1:TB=( i:110; j:120 );
  b2:TB=( i:210; j:220 );

procedure (*****) TestPrekryvu (*****)
;
begin
  a1.f( 1 );
  b1.f( b2, 2 );
  b2.g( b1 );
end;
(********** TestPrekryvu **********)
```

V této ukázce jsou definovány třídy *TA* a *TB*, přičemž třída *TB* je potomkem třídy *TA*. Třída *TB* zároveň předefinovává metodu *f*. V nové metodě bychom chtěli využít překrytou metodu třídy *TA*.

Pokud se spokojíme s metodou aktuální instance (*self*) resp. jejího bezprostředního předka, můžeme situaci elegantně vyřešit způsobem, který známe z konstruktorů, tj. pomocí klíčového slova **inherited** nebo kvalifikací jménem třídy – tak, jak to vidíme v prvním a druhém komentáři v těle procedury *TB.f*.

První způsob, tj. použití klíčového slova **inherited**, má ovšem dvě nevýhody: za prvé je podporován až od sedmé verze překladače (to by dnes už nemuselo příliš vadit), ale hlavně nám znepřístupní metody od prarodičů. Proto dáváme přednost druhému způsobu (kvalifikaci jménem předka), který nám navíc připadá průzračnější, protože při něm přesně víme, od koho dědíme.

Potřebujeme-li zavolat funkci jiné instance než *self*, dostáváme se do potíží. První postup, který by vás asi napadl, by mohl být podobný postupu z třetího komentáře, tj. mohl by jím být pokus dodatečně kvalifikovat metodu identifikátorem její třídy. Bohužel, kvalifikaci identifikátorem třídy akceptuje překladač pouze tehdy, pokud je to kvalifikace jediná. V opačném případě nám to „nezbaští“ a oznámí chybu *Field indefier expected*.

Další věc, která by nás mohla napadnout, je pokusit se přetypovat proměnnou, jejíž metodu chceme použít, na odpovídající rodičovský typ, a zavolat metodu přetypované proměnné tak, jak jsme to naznačili ve čtvrtém komentáři. I zde se překladač vzbouří a ohlásí chybu *Invalid type cast*, protože zdrojový a cílový objekt mají jinou velikost a Pascal dovoluje přetypovat pouze na typ se stejnou velikostí objektů (z toho ovšem vyplývá, že kdyby třída *TB* nepřidala žádnou datovou složku, ale pouze modifikovala seznam metod, byla by tato cesta schůdná – vyzkoušejte).

Možná, že se nyní zeptáte, jak to, že proměnnou není možno přetypovat na její rodičovský typ, když jsme předtím tvrdili, že předek může vždy zastoupit potomka. **Tato ekvivalence typů však platí pouze pro předávání daných proměnných jako parametrů nebo pro přiřazování (nebo pro podobné operace s ukazateli)**, jak se o tom můžete ostatně přesvědčit v posledním příkazu procedury *Test_3*, v němž proměnná *b2* volá zděděnou metodu *g* a předává jí jako parametr proměnnou *b1*, přestože metoda *g* vyžaduje parametr typu *TA* a proměnná *b1* je typu *TB*. Překladač prostě vezme odpovídající část proměnné *b1* a předá ji jako parametr typu *TA*.

Jediný způsob volání této překryté metody, který nám překladač akceptuje, je uveden za sérií komentářů, na konci procedury *TB.f*. Musíme získat adresu proměnné, tento ukazatel přetypovat na ukazatel na typ *TA* a ten pak vítězně dereferencovat. Je to trochu krkolomné, ale takový je Pascal.

3.4 Destruktory

Při návrhu destruktorů se budeme setkávat s obdobnými problémy jako u konstruktorů, avšak vzhledem k povaze destruktorů budou tyto problémy tak trochu otočené naruby. I u destruktorů bychom měli počítat s tím, že pracují s objekty sestávajícími ze dvou částí: z části zděděné a z části nově přidané.

Z podstaty činnosti, prováděné destruktorem, je však zřejmé, že volání rodičovských destruktorů by mělo probíhat v obráceném pořadí než u konstruktorů: destruktor by se měl nejprve postarat o úklid přidaných složek (složek deklarovaných v potomkovi) a na zděděné složky by si měl pozvat rodičovský destruktor.

V C++ se o volání rodičovského destruktoru starat nemusíme, tam se o ně za nás postará překladač. V Pascalu si však musíme odpovídající rodičovský destruktor zavolat sami. Oproti C++ však máme v Pascalu tu výhodu, že můžeme definovat několik destruktorů a že tyto destruktory mohou mít dokonce parametry.

Jako příklad si vezmeme třídu *cObjekt_2* a jejího potomka *cZivy_2*, které vzniknou zjednodušením tříd *cObjekt_1 a cZivy_1*. Konstruktor, resp. destruktor každé z těchto tříd zavolá metodu *Tiskni*, která vypíše zprávu o třídě a jménu instance, kterou konstruuje, resp. destruuje. V odvozené třídě deklarujeme navíc metodu *Skoc*, která vypíše sdělení „Hop“ (nic lepšího nás zrovna nenapadlo). Dále pak deklarujeme jedinou instanci *o* typu *cZivy_2* a zavoláme pro ni metodu *Skoc*.

V C++ opravdu stačí deklarovat proměnnou a zavolat její metodu:

```
/*   Příklad C2 – 8   */
#include <iomanip.h>
typedef unsigned word;
class /*****/ cObjekt_2 /*****/
{
  public:
       cObjekt_2(const char * jm);
       ~cObjekt_2();
       void Tiskni(const char *uvod);
  private:
       const char *Jm;
};
/********** cObject_2 **********/

/*****/ cObjekt_2::cObjekt_2 /*****/
( const char * jm )
: Jm(jm)
{
  Tiskni("Třída cObjekt_2, konstruktor instance ");
}/********** cObjekt_2::cObjekt_2 **********/
/*****/ cObjekt_2::~cObjekt_2 /*****/
()
{
  Tiskni("Třída cObjekt_2, destruktor instance ");
}
/********** cObjekt_2::~cObjekt_2 **********/
void /*****/ cObjekt_2::Tiskni /*****/
(const char *uvod)
{
  cout << uvod << Jm << endl;
}
/********** cObjekt_2::Tiskni **********/

class /*****/ cZivy_2 /*****/
: public cObjekt_2
```

```
{
  public:
       cZivy_2(const char * jm);
       ~cZivy_2();
       void Skoc();
  private:
       const char *Jm;
};
/********** cZivy_2 **********/

/*****/ cZivy_2::cZivy_2 /*****/
( const char * jm )
: cObjekt_2(jm)
{
  Tiskni("Třída cZivy_2, konstruktor instance ");
}
/********** cZivy_2::cZivy_2 **********/

/*****/ cZivy_2::~cZivy_2 /*****/
()
{
  Tiskni("Třída cZivy_2, destruktor instance ");
}
/********** cZivy_2::~cZivy_2 **********/

void /*****/ cZivy_2::Skoc /*****/
()
{
  cout << "Hop" << endl;
}
/********** cZivy_2::Skoc **********/

void /*****/ Test_1 /*****/ ()
{
  cZivy_2 o = "Hurá";
  o.Skoc();
}
/********** Test_1 **********/
```

V Pascalu musíme navíc volat konstruktor a destruktror instance *o*. Přitom nesmíme zapomenout v konstruktoru třídy *cZivy_2* zavolat konstruktor předka, třídy *cObjekt_2*, a v destruktoru potomka zavolat destruktor předka:

```
 (*   Příklad P2 – 7   *)
type
(*****) cObjekt_2 (******) = object
       Jm: string;
       constructor Init(pjm: string);
       destructor Done;
       procedure Tiskni(uvod: string);
  end;
(******** cObjekt_2 ********)

(*****) cZivy_2 (******) = object(cObjekt_2)
       constructor Init(pjm: string);
       destructor Done;
       procedure Skoc;
  end;
 (******** cZivy_2 ********)

constructor (*****) cObjekt_2.Init (*****)
( pjm: string );
begin
  jm := pjm;
  Tiskni('Třída cObjekt_2, konstruktor instance ');
end;
(********** cObjekt_2.Init **********)

destructor (*****) cObjekt_2.Done (*****)
;
begin
  Tiskni('Třída cObjekt_2, destruktor instance ');
end;
(********** cObjekt_2.Done **********)

procedure (*****) cObjekt_2.Tiskni (*****)
(uvod: string);
begin
  system.writeln(uvod + Jm);
end;
(********** cObjekt_2.Tiskni **********)

constructor (*****) cZivy_2.Init (*****)
( pjm: string );
begin
  cObjekt_2.Init(pjm);
 Tiskni('Třída cZivy_2, konstruktor instance ');
end;
(********** cZivy_2::cZivy_2 **********)

destructor (*****) cZivy_2.Done (*****);
begin
  Tiskni('Třída cZivy_2, destruktor instance ');
  cObjekt_2.Done;
end;
(********** cZivy_2.Done **********)

procedure (*****) cZivy_2.Skoc (*****);
begin
  system.writeln('Hop');
```

```
end;
(********** cZivy_2::Skoc **********)

procedure (*****) Test_1 (*****)
;
var o: cZivy_2;
begin
  o.Init('Hurá');
  o.Skoc;
  o.Done;
end;
(********** Test_1 **********)
```

Pokud si tyto programy spustíte, zjistíte, že vypíší

```
Třída cObjekt_2, konstruktor instance Hurá
Třída cZivy_2, konstruktor instance Hurá
Hop
Třída cZivy_2, destruktor instance Hurá
Třída cObjekt_2, destruktor instance Hurá
```

3.5 Dědičnost v Object Pascalu (Delphi)

V C++ i v Turbo Pascalu platí, že pokud v deklaraci třídy neuvedeme žádného předka, pak jej nově deklarovaná třída opravdu nemá, i když samozřejmě může být kořenem dědické hierarchie. Totéž platí i pro třídy starého objektového modelu v Object Pascalu v Delphi (deklarovaných pomocí klíčového slova **object**).

V případě nového objektového modelu, tedy tříd deklarovaných pomocí klíčového slova **class**, je situace jiná. Všechny tyto třídy tvoří jedinou dědickou hierarchii se společným prapředkem, třídou *TObject*. Pokud tedy neuvedeme v deklaraci žádného předka, znamená to, že nová třída bude odvozena přímo od třídy *TObject*.

Třída *TObject* je v jednotce *Systém* deklarována takto:

```
type
  TObject = class;
  TClass = class of object;
  TObject = class
       constructor Create;
       destructor Destroy; virtual;
       class function ClassInfo: Pointer;
       class function ClassName: string;
       class function ClassParent: TClass;
       function ClassType: TClass;
       procedure DefaultHandler(var Message): virtual;
       procedure Dispatch(var Message);
       function FieldAddres(const Name: string): pointer;
       procedure Free;
       procedure FreeInstance; virtual;
       class function InheritsFrom(AClass: TClass): boolean;
       class function InitInstance(Instance: pointer): TObject;
       class function InstanceSize: word;
```

```
      class function NewInstance: TObject; virtual;
      class function MethodAddress(const Name: string): pointer);
      class function MethodName(Address: pointer) string;
  end;
```

Tento společný předek neobsahuje žádná data, slouží především k tomu, aby poskytl všem potomkům základní společné rozhraní. To umožňuje zacházet se všemi objekty stejným způsobem, alespoň na jisté základní úrovni. Všechny třídy např. mají konstruktor *Create*; není deklarován jako virtuální, a proto jej můžeme v odvozených třídách definovat s jinými parametry. K vytvoření nové instance lze použít také virtuální metodu *NewInstance* (je deklarována jako metoda třídy). Dále je tu virtuální destruktor *Destroy*, metody, které umožňují zjistit jméno třídy (*ClassName*), předka (*ClassParent*) a další. Metoda *DefaultHandler* obstarává implicitní zpracování zpráv od Windows, tj. zpracování těch zpráv, které my v programu ponecháváme neošetřeny.

Spolu s třídou *TObject* je definován i typ *TClass*, který představuje referenci na třídu *TObject*.

Z pravidel dědičnosti plyne, že pokud např. deklarujeme funkci *F* s parametrem typu *TObject*, může být skutečným parametrem jakákoli třída v Object Pascalu. Podobně proměnné typu *TClass* můžeme přiřadit referenci na jakéhokoli potomka, tedy na jakoukoli v programu deklarovanou třídu.

Podrobné povídání o Delphi ovšem přesahuje rámec naší knihy, a proto odkazujeme čtenáře na firemní dokumentaci.

4. Ukazatele do třídy

V této kapitole krátce odbočíme od problémů s dědičností a podíváme se na práci se složkami tříd pomocí ukazatelů.

4.1 Objekty a „obyčejné“ ukazatele

Ukazatele na data

S nestatickými datovými složkami objektů můžeme samozřejmě zacházet pomocí „obyčejných“ ukazatelů, stejně jako s jinými proměnnými. Jestliže např. definujeme třídu *cTrida*, můžeme adresu datové složky přiřadit ukazateli a s pomocí tohoto ukazatele se složkou pracovat:

```
//  Příklad C3 – 1
class cTrida {
public:
  int a;
  double b;
  cTrida(int i, double d):a(i), b(d){}
};

int main(){
// deklarujeme instanci
  cTrida cT(2, 3.15);
  int *ui = &cT.a;
// a pomocí ukazatele změníme hodnotu její složky
  *ui = 332;
  return 0;
}
```

V Pascalu bude mít táž konstrukce tvar

```
(*  Příklad P3 – 1 *)
{ Deklarace třídy }
type
  cTrida = object
       a: integer;
       b: real;
       constructor Init(i: integer; d: real);
  end;

constructor cTrida.Init(i: integer; d: real);
begin
  a := i;
  b := d;
end;
{ Deklarace instance }
```

```
var    cT: cTrida;
       ui: ^integer;

begin
  cT.Init(2, 3.15);
  ui := @cT.a;
{měníme hodnotu složky pomocí ukazatele}
  ui^ := 332;
end.
```

Doufáme, že není třeba zdůrazňovat, že něco podobného bychom měli dělat jen v nejvyšší programátorské nouzi. Tady totiž porušujeme pravidla zapouzdření – a tedy bezpečnosti programování – hned dvoustupňově. Nejenže zpřístupňujeme složku objektu zvenku, takže objekt nemá kontrolu nad svými daty, ale činíme tak dokonce pomocí ukazatelů, takže v místě, kde pak dojde ke změně, ani není zjevné, jaká nepřístojnost se vlastně děje. Nicméně v praktickém životě se můžeme setkat se situacemi, kdy se něco podobného bude hodit.

Ukazatele na metody

Ani Pascal, ani C++ neumožňuje používat „obyčejné“ ukazatele pro práci s nestatickými metodami. (Statické metody jsou v C++ vlastně normální funkce, pouze podle jména přidružené k nějaké třídě, a proto s nimi můžeme zacházet pomocí „obyčejných“ ukazatelů.)

4.2 Ukazatele do tříd

Jazyk C++ nabízí vedle obyčejných ukazatelů ještě tzv. ukazatele do tříd (*member pointers*). Tyto ukazatele mj. také umožňují pracovat s metodami objektových typů. V Turbo Pascalu nemají přímou analogii; ukazatele na metody, se kterými se setkáme v Object Pascalu v Delphi, jsou koncipovány poněkud jinak.

Ukazatele na data

K čemu to je?

„Obyčejný“ ukazatel obsahuje prostě adresu nějaké proměnné. Použijeme-li jej, říkáme, že chceme pracovat s proměnnou, která leží na udaném místě v paměti.

Ukazatele do tříd (občas jim budeme říkat „třídní“ ukazatele, a pokud vám to připomíná třídního nepřítele, tak je nepoužívejte nebo pro ně vymyslete lepší název) obsahují relativní adresu složky vzhledem k začátku instance. Použijeme-li ukazatel do třídy (např. do třídy *cTrida*), říkáme např., že chceme pracovat s první složkou typu **int** v nějaké instanci této třídy. Abychom takovýto údaj mohli využít, musíme k tomu ještě dodat, kterou instanci má systém použít.

Deklarace

Lehce zjednodušený syntaktický popis deklarace ukazatele do třídy má tvar

Deklarace ukazatele do třídy:
*typ třída***::***iden**

Zde *typ* je typ složky, na kterou chceme ukazovat, *třída* je identifikátor třídy, o jejíž složky půjde, a *ident* je identifikátor nově deklarovaného ukazatele.

Použití

Ukážeme si jednoduchý příklad. Nepátrejte po nějakém hlubším smyslu – nemá jej. Pouze ukazuje, jak třídní ukazatele fungují. Rozsáhlejší příklad najdete v příští kapitole, kde se pokusíme pomocí ukazatelů na metody implementovat polymorfismus.

Nejprve deklarujeme třídu *cTrida* (přesněji strukturu, abychom se nemuseli zatím zdržovat s přístupovými právy) a vytvoříme několik instancí:

```
/*   Příklad C3 – 2   */
// Deklarace třídy
struct cTrida {
  int a, b;
  double d, e;
  cTrida(int, double);
  void f(void);
  void g(void);
};

// Konstruktor
cTrida::cTrida(int u, double v)
:a(u), b(u*u),d(v), e(-v)
{}

// Nějaké metody ...použijeme je ve
// výkladu o ukzatelích na metody
void cTrida::f(void)
{ cout << "metoda f" << endl;
}

void cTrida::g(void)
{ cout << "metoda g" << endl;
}

// Instance a ukazatel na ně
cTrida *ucT;
cTrida c1(3,3), c2(2,4.4), c3(0,8);
cTrida *ucT = &c1;
```

Nyní můžeme deklarovat dva ukazatele do této třídy:

```
int cTrida::*uki;
double cTrida::*ukd;
```

Proměnná *uki* je ukazatel na složku typu **int** ve třídě *cTrida*, ukd je ukazatel na složku typu **double** v téže třídě. Nyní potřebujeme přiřadit těmto proměnným hodnotu. K tomu použijeme obvyklý adresový operátor „&“, ovšem poněkud neobvyklým způsobem. Nejprve ale deklarujeme tři instance třídy *cTrida:*

```
uki = &cTrida::b;      // 1
ukd = &cTrida::d;      // 2
```

Všimněte si, že zde jako operand neuvádíme instanci (resp. složku instance), ale jméno složky, kvalifikované jménem třídy. Ukazatel *uki* bude nyní ukazovat na složku *b* třídy *cTrida*, ukazatel *ukd* bude ukazovat na složku *d*.

Operátory „.*“ a „->*“

Budeme-li chtít ukazatele do třídy použít, musíme je – jako všechny ukazatele – dereferencovat. Ukazatel *uki* ukazuje na složku *b* ve třídě *cTrida* (ve kterékoli instanci), my ale musíme doplnit, ve které instanci jej chceme uplatnit.

K dereferencování ukazatelů do tříd slouží operátory „**.***“ a „**->***“. Jestliže známe přímo instanci, s jejíž složkou chceme pomocí třídního ukazatele pracovat, použijeme operátor „**.***“; známe-li ukazatel na instanci, použijeme operátor „**->***“. Syntax jejich použití je

instance **.*** *třídní_ukazatel*

resp.

ukazatel_na_instanci **->*** *třídní_ukazatel*

Podívejme se na příklad:

```
c3.*uki = 11;          // 3
c2.*uki = c3.*uki;     // 4
c1.*ukd = 6.6;         // 5
ucT ->* ukd = 33;      // 6
```

Ukazatel *uki* ukazuje na složku *b* – to jsme zařídili v příkazu, označeném `// 1` v předchozím odstavci. V příkazu `// 3` tedy přiřadíme složce *b* instance *c3* hodnotu 11. V příkazu `// 4` pak přeneseme hodnotu z *c3.b* do *c2.b*. Příkazem `// 5` uložíme do *c1.d* hodnotu 6,6 (ukazatel *ukd* ukazuje na složku *d* ve třídě *cTrida*).

V příkazu `// 6` použijeme ukazatel *ucT*, který obsahuje adresu instance *c1*. Tento příkaz přiřadí 33,0 složce *c1.d.*

Poznámky

Na rozdíl od „obyčejných“ ukazatelů nelze třídní ukazatele konvertovat na celá čísla (ani naopak). Jedinou výjimkou je hodnota 0, kterou lze přiřadit libovolnému ukazateli do třídy a která znamená, že ukazatel „neukazuje nikam“.

Na ukazatele do tříd také nelze používat adresovou aritmetiku jazyka C. Lze je porovnávat pomocí operátorů „==“ a „!=“. Můžeme je porovnávat mezi sebou nebo s 0.

Nelze však na ně použít operátory „<“, „>“, „<=“ a „>=“.

Třídní ukazatele lze – podobně jako ostatní ukazatele – používat v podmínkách v příkazech **if**, **while** apod. (ale pozor, některé starší překladače měly s takovýmito konstrukcemi potíže – buď hlásily podivné chyby, nebo je nepřekládaly správně).

Poslední upozornění se týká priorit. Operátory „.*“ a „->*“ mají prioritu 3, tedy nižší než např. operátor volání funkce nebo indexování nebo než unární operátory jako „*“ a „->“.

Ukazatele na metody

Třídní ukazatele na metody se deklarují podobně jako třídní ukazatele na data; pouze musíme identifikátor ukazatele spolu se jménem třídy uzavřít do závorek, podobně jako při deklaraci „obyčejného“ ukazatele na funkci. Například ukazatel na metodu typu **void** třídy *cTrida* bez parametrů můžeme deklarovat zápisem

```
void (cTrida::*ukf)(void);
```

Hodnotu mu přiřadíme příkazem

```
ukf = &cTrida::g;      // 7
```

kde je operátor „&“nezbytný, nelze jej vynechat. Nyní můžeme metodu, na kterou *ukf* ukazuje, zavolat. I zde použijeme k dereferencování ukazatele operátory „.*“ nebo „->*“, které nám umožňují doplnit instanci, pro níž chceme danou metodu volat.

```
(c1.*ukf)();           // 8
```

Závorky, které v tomto volání uzavírají zápis `(c1.*ukf)`, jsou nezbytné, neboť operátor „`.*`“ má nižší prioritu než operátor volání funkce.

Poznamenejme, že pomocí třídních ukazatelů můžeme pracovat pouze s nestatickými metodami. Se statickými metodami můžeme pracovat pomocí „obyčejných“ ukazatelů.

4.3 Ukazatele na metody v Object Pascalu

Object Pascal v Delphi umožňuje používat ukazatele na metody (ovšem pouze na metody tříd, deklarovaných pomocí klíčového slova **class**). Deklarují se podobně jako proměnné procedurálních typů, k deklaraci však připojíme frázi **of object**. Přesný syntaktický popis deklarace ukazatele na metodu je

Deklarace ukazatele na metodu:

procedure *seznam_parametrů* $_{opt}$ **of object**
funkcion *seznam_parametrů* $_{opt}$ **:** *typ_výsledku* **of object**

Seznam_parametrů a *typ_výsledku* mají stejný význam jako v deklaraci procedury nebo funkce. Všimněte si, že v deklaraci neuvádíme typ, o jehož metody půjde. Ukazatel na metody může obsahovat adresu metody libovolného typu, záleží pouze na tom, zda jde o proceduru nebo funkci, a na počtu a typu parametrů. Jako příklad si ukážeme následující dvě deklarace:

```
type
  TMetoda = procedure of object;
  TUdalost = procedure(OdKoho: TObject) of object;
```

První je ukazatel na proceduru bez parametrů, druhý je ukazatel na proceduru s jedním parametrem typu *TObject* (to znamená, že vlastně může mít parametr libovolného objektového typu).

Deklarujeme nyní proměnnou typu ukazatel na metody:

```
var
  Kliknuti: TUdalost;
```

Je-li *HlavniOkno* instance třídy *TMainForm*, přiřadíme proměnné *Kliknuti* adresu metody *Botton* příkazem

```
Kliknuti := HlavniOkno.Button;
```

Object Pascal umožňuje zjistit pomocí standardní funkce *Assigned*, zda je ukazateli na metodu přiřazena hodnota. Tato funkce vrací *true*, jestliže jsme dané proměnné nějakou hodnotu již přiřadili, a *false* v opačném případě. To znamená, že můžeme napsat

```
if Assigned(Kliknuti) then Kliknuti(Self);
```

Při volání metody postupujeme stejně jako při volání „obyčejné“ funkce pomocí proměnné procedurálního typu.

Proměnná typu ukazatel na metodu je v paměti uložena jako dvojice ukazatelů. První z nich obsahuje adresu kódu metody, druhý obsahuje adresu instance, pro kterou se má metoda volat.

5. Časná a pozdní vazba

V této kapitole si ukážeme, že dědičnost sama o sobě nestačí. K tomu, aby bylo OOP opravdu použitelné, musíme vystoupit ještě na třetí stupeň, který jsme slíbili v úvodu k prvnímu dílu – musíme si povědět o polymorfismu neboli mnohotvárnosti objektů. Uděláme to tak, že si ukážeme, kde nepolymorfní objekty začnou selhávat, a pokusíme se obejít tyto problémy pomocí našich dosavadních znalostí. Pak si povíme o nástrojích, které nám C++, resp. Turbo Pascal k řešení tohoto problému nabízejí.

Nejprve se ale zastavíme u jednoho z obvyklých problémů v návrhu objektových tříd.

5.1 Je nebo má

Pokusme se využít našich dosavadních znalostí objektového programování a napsat jednoduchý grafický editor. Měl by běžet pod DOSem a umožňovat kreslení základních grafických objektů, jako jsou body, úsečky, kružnice apod. Pro zjednodušení budeme předpokládat, že všechny nakreslené objekty jsou bílé a pozadí je černé. Podívejme se, jak při tom budeme postupovat.

Grafické objekty v obrázku, který pomocí našeho editoru nakreslíme, budeme v programu reprezentovat pomocí instancí objektových typů. Nyní si ale musíme ujasnit, kam je budeme ukládat. Protože předem nevíme, z kolika grafických objektů se bude obrázek skládat, nemá smysl uvažovat o poli, bude rozumné použít spíše některou z dynamických datových struktur – např. seznam.

Podívejme se teď na položku takového seznamu. Co – jaká data – bude obsahovat? Především je asi zřejmé, že to nebude přímo instance představující grafický objekt. Bylo by to nepohodlné, neboť instance jednotlivých grafických objektů budou různě velké (pro zobrazení bodu potřebujeme znát pouze jeho souřadnice na obrazovce, tedy dvě celá čísla; pro zobrazení kružnice potřebujeme znát souřadnice středu a poloměr, tedy tři celá čísla, pro zobrazení úsečky souřadnice počátečního a koncového bodu atd.). Bude proto výhodnější, když budou prvky seznamu obsahovat ukazatele na data.

Až potud je doufejme vše jasné: naše úvahy nás přivedly k závěru, že obrázek bude v programu reprezentován seznamem grafických objektů; prvky tohoto seznamy budou obsahovat ukazatele na instance objektových typů představujících jednotlivé grafické objekty. Jenže se ihned vynoří další otázka: Jaký bude doménový typ použitých ukazatelů? (Tedy: na co budou ukazovat?) Bylo by jistě nejlepší, kdyby všechny prvky seznamu obsahovaly ukazatele s týmž doménovým typem.

Tady se nabízí využití dědičnosti. V první kapitole jsme si vysvětlili, že potomek může vždy zastupovat předka, a podobně že ukazatel na potomka můžeme vždy použít na místě ukazatele na předka. Zkusme tedy vzít jeden z grafických objektů za základ a ostatní od něj odvodit. Jako možný kořen dědické hierarchie se nabízí *bod*.

Deklarujeme tedy objektový typ *bod* např. takto:

```
class bod {
    int x, y;
  public:
   bod(int xx, int yy): x(xx), y(yy) {}
// a další metody
};
```

resp. pokud se rozhodneme použít Pascal

```
type
  bod = object
  x, y: integer;
  constructor Init(xx, yy: integer);
 {a další metody }
end;

constructor bod.Init(xx, yy: integer);
begin
  x := xx; y := yy;
end;
```

Nyní potřebujeme třídu *úsečka*. Chceme ji definovat jako potomka třídy *bod*. Ale tady se dostaneme trochu do problémů. Úsečka zdědí od svého předka podobjekt typu *bod*. Bude to počáteční nebo koncový bod? Nebo střed? Nebo jej budeme ignorovat a počáteční a koncový bod úsečky definujeme jako nové atributy?

Kdybychom se např. rozhodli použít zděděný bod jako počáteční a pro koncový bod zavést nové atributy, dostali bychom v C++

```
class usecka: public bod {
  xk, yk;
 public:
  usecka(int x1, int y1, int x2, int y2);
};

usecka::usecka(int x1, int y1, int x2, int y2)
: bod(x1, y1), xk(x2), yk(y2)
{}
```

a v Pascalu

```
type
  usecka = object(bod)
  xk, yk: integer;
  constructor Init(x1, y1, x2, y2: integer);
  end;

constructor usecka.Init(x1, y1, x2, y2: integer);
begin
  bod.Init(x1, y1);
  xk := x2; zk := y2;
end;
```

V obou jazycích je patrná jistá nesymetrie v zacházení s body úsečky. Zatímco počáteční bod jsme zdědili, koncový jsme „přidělali na koleně“. Zatímco počáteční bod inicializuje konstruktor předka, koncový bod inicializujeme až v konstruktoru potomka. V C++

přibude ještě další nesymetrie: zatímco složky (souřadnice) koncového bodu můžeme v metodách typu úsečka volně používat, složky zděděného počátečního bodu nikoli, neboť jsou soukromou záležitostí zděděného podobjektu.

Mohli bychom to částečně vylepšit, kdybychom koncový bod deklarovali jako složku typu *bod*, např.

```
class usecka: public bod {
  bod konec;
 public:
  usecka(int x1, int y1, int x2, int y2);
};
```

resp.

```
type
 usecka = object(bod)
  konec: bod;
  constructor Init(x1, y1, x2, y2: integer);
 end;
```

ale ani potom by oba body nebyly docela rovnoprávné – a to v obyčejné úsečce jsou.

Zdá se tedy, že něco není s naším návrhem v pořádku. Není těžké přijít na to, co. **Úsečka není bod**. Úsečka není zvláštním případem bodu, a proto nemá smysl definovat ji jako potomka bodu. Na druhé straně **úsečka má dva význačné body**, počáteční a koncový. To znamená, že třída *úsečka* může využít služeb třídy *bod*; deklarujeme tedy úsečku jako složenou třídu, obsahující dva body.

Naše deklarace proto budou mít tvar

```
class usecka {
bod: pocatek, konec;
 public:
  usecka(int x1, int y1, int x2, int y2);
}

usecka::usecka(int x1, int y1, int x2, int y2)
: pocatek(x1, y1), konec(x2, y2)
{}
```

resp. v Pascalu

```
type
 usecka = object(bod)
  pocatek, konec: bod;
  constructor Init(x1, y1, x2, y2: integer);
 end;

constructor usecka.Init(x1, y1, x2, y2: integer);
begin
  pocatek.Init(x1, y1);
  konec.Init(x2, y2);
end;
```

Nyní je zacházení s oběma body i přístup k nim symetrický a navíc je i jasný vztah mezi oběma třídami. Ovšem co s dědičností? Řekli jsme si, že bychom ji potřebovali, aby-

chom mohli v seznamu grafických objektů používat pouze ukazatele na předky. Snadno zjistíme, že podobně jako třída *bod* se za společného předka nehodí ani jiná z tříd konkrétních grafických objektů (kružnice, elipsa, trojúhelník…).

Na druhé straně je jasné, že všechny grafické objekty budou mít mnoho společného. Budeme je kreslit na obrazovku nebo zase mazat, budeme je po obrazovce posunovat, budeme je ukládat do souboru atd. Zavedeme tedy pro všechny grafické objekty společného předka, třídu, kterou vtipně nazveme *grafický objekt* (ale protože to je jako název trochu dlouhé a my se musíme vejít do vymezeného rozsahu, a navíc jsme trochu líní, zkrátíme si to na *go*). Nikdo z vás jistě nepochybuje o tom, že bod, stejně jako úsečka, kružnice nebo pravidelný pětiúhelník, jsou grafické objekty. Náš seznam bude tedy obsahovat ukazatele na grafické objekty – instance třídy *go*.

Poznámka

V teoretických pojednáních o OOP se setkáme s označeními **isa** pro dědičnost a **hasa** pro skládání. **Isa** lze rozložit na anglická slova *is a*, tedy *je*, **hasa** vzniklo ze slov *has a*, tedy *má*. Úsečka *je* grafický objekt, ale *není* to bod. Úsečka *má* význačný bod (dokonce dva). Úsečku tedy můžeme deklarovat jako potomka třídy grafický objekt složeného ze dvou bodů.

Vedle toho se můžeme v souvislosti se skládáním tříd setkat s povídáním o vztahu klient – server nebo prodávající – kupující. Složená třída využívá služeb tříd svých složek, např. třída *úsečka* využívá služeb třídy *bod*.

5.2 Když samotná dědičnost přestane fungovat

Abstraktní třídy

Vraťme se k naší výchozí úloze: chceme napsat grafický editor. Už jsme se dohodli, že všechny třídy popisující grafické objekty budou mít společného předka, třídu *go*. V této třídě definujeme všechny vlastnosti, které budou společné všem grafickým objektům.

Z atributů, tedy datových složek, to může být např. barva, dále pak atribut, který vyjadřuje, zda je daný grafický objekt nakreslen.

Dále bude mít každý grafický objekt metodu *zobraz* s parametrem, určujícím barvu, ve které se má daný objekt vykreslit. Tato metoda bude specifická pro každou z tříd: jinak se kreslí bod, jinak kružnice atd.

Vedle toho se nám budou hodit metody *nakresli* a *smaž*. První z nich nakreslí objekt v barvě bílé (takže jej bude na černém pozadí vidět), druhá jej nakreslí v barvě černé (takže jej vidět nebude). Obě tyto metody budou prostě volat metodu *zobraz* s danou barvou.

Metody *nakresli* a *smaž* budou ve všech třídách naprosto stejné; zdá se tedy, že je můžeme naprogramovat pouze jednou, ve třídě *go*, a ostatní třídy nechat, aby ji zdědily.

Vzniká ovšem otázka, jak naprogramovat metodu *zobraz* pro třídu *go*. Chtít nakreslit jeden obecný grafický objekt, to je něco podobného jako chtít koupit jednu potravinu:

nemá to smysl. Můžeme si koupit jednu housku, ale požádáme-li v obchodě o jednu potravinu, dostane se nám nejspíš rychlé lékařské pomoci.

Grafický objekt je – podobně jako potravina – abstraktní pojem a některé operace pro něj nemusí mít smysl. Má smysl *zobrazit* úsečku, ale nemá smysl zobrazit obecný grafický objekt. (*go* je příkladem toho, čemu říkáme *abstraktní třídy*; tento termín ovšem budeme v C++ používat také pro třídy s tzv. čirými virtuálními metodami, ale o tom si přečtete dále.)

To ovšem neznamená, že si můžeme dovolit metodu *zobraz* u třídy *go* vynechat. Se všemi grafickými objekty chceme pracovat pomocí ukazatelů na třídu *go*, takže se v programu budou objevovat příkazy jako

```
go * ugo;
// ...
ugo -> zobraz(bila);
```

resp.

```
var ugo: ^GO;
{ ...}
ugo^.zobraz(bila);
```

a oba překladače – jak Pascalu tak C++ – by měly vážné námitky, kdyby za těchto okolností třída *go* neměla metodu *zobraz*.

Volání metody *zobraz* třídy *go* je ovšem vždy chyba. Pokud se něco takového stane, měli bychom vypsat chybové hlášení a ukončit program.

První pokus o program

Zkusme tedy na základě těchto úvah napsat jednoduchý program, který bude obsahovat třídy *go*, *bod* a *úsečka*, a vyzkoušejme si, zda jsou naše úvahy správné. Pro grafickou práci použijeme borlandské nástroje pro DOS. Pokud používáte jiný překladač, je třeba přepsat odpovídajícím způsobem metody *zobraz*.

Potřebné třídy a jejich metody deklarujeme v C++ takto:

```
/* Příklad C4 - 1 * /
#include <graphics.h>
#include <iostream.h>
#include <process.h>
#include <conio.h>

// Cesta ke grafickému ovladači - nutno změnit dle
// skutečného stavu na vašem počítači
const char* cesta = "C:\\aplikace\\prekl\\bc31\\bgi";

/**************************************/
// Třída go a její metody
class go
{
 public:
   go();
   ~go();
```

```
  void nakresli();
  void smaz();
  void zobraz(unsigned barva);
 protected:
  int sviti;
};

typedef go* pgo;

go::go()
: sviti(0)
{}

// Destruktor objekt pro jistotu smaže
go::~go()
{ smaz();
}

// Nakreslení = zobrazení v barvě bílé
void go::nakresli()
{ sviti = 1;
  zobraz(WHITE);
}

// Smazání = zobrazení v barvě černé
void go::smaz()
{ sviti = 0;
  zobraz(BLACK);
}

// Zobrazení obecného grafického objektu
// nemá smysl, proto vypíšeme chybovou zprávu
// a ukončíme program
void go::zobraz(unsigned)
{ cout << "Jak se zobrazí obecný grafický objekt?"<< endl;
  getch ();
  exit(1);
}

/*****************************************/
// třída bod a její metody

class bod: public go {
 public:
  bod(int xx, int yy);
  ~bod();
  void zobraz(unsigned barva);
  void moveto();
  void lineto();
 private:
  int x,y;
};
bod::bod(int xx, int yy)
: x(xx), y(yy)
{}

bod::~bod()
{}
```

```
// Nakreslí bod v zadané barvě
// Tato a následující dvě metody jsou
// závislé na použitém překladači
void bod::zobraz(unsigned barva)
{ putpixel(x,y,barva);
}

// Posune grafický kurzor
// do zadaného bodu
void bod::moveto()
{ ::moveto(x,y);
}

// Nakreslí úsečku z aktuální
// pozice do zadaného bodu
void bod::lineto()
{ ::lineto(x,y);
}

/***************************************/
// třída úsečka a její metody
class usecka: public go {
 public:
   usecka(int xx1, int yy1, int xx2, int yy2);
   ~usecka();
   void zobraz(unsigned barva);
 private:
   bod poc, kon;
};

usecka::usecka(int xx1, int yy1, int xx2, int yy2)
: poc(xx1, yy1), kon(xx2, yy2)
{}

usecka::~usecka()
{}

// Nakreslí úsečku v zadané barvě
void usecka::zobraz(unsigned barva)
{ setcolor(barva);
 poc.moveto();
 kon.lineto();
}

/***************************************/
// Testovací procedura
void test()
{  int gd = DETECT, gm;
   initgraph(&gd, &gm, cesta);
   usecka cara(10,20,50,80);
   pgo ugo = &cara;
   ugo->nakresli();
   ugo->smaz();
   closegraph;
};

int main(){
   test();
```

```
  return 0;
}
```

Podobný program v Pascalu bude mít tvar

```
(* Příklad P4 - 1 *)
uses graph, crt;

{ Cesta ke grafickému ovladači. }
{Nutno změnit podle skutečného }
{stavu na vašem počítači. }
const
  cesta = 'C:\aplikace\prekl\bp\bgi';

(*****************************************)
{ typ go a jeho metody}
type
  go = object
  sviti: boolean;
  constructor init;
  destructor done;
  procedure nakresli;
  procedure smaz;
  procedure zobraz(barva: word);
 end;

 pgo = ^go;

constructor go.init;
begin
  sviti := false;
end;

{Destruktor objekt pro jistotu smaže }
destructor go.done;
begin
  smaz;
end;

{Nakreslit objekt znamená zobrazit jej bíle }
procedure go.nakresli;
begin
  sviti := true;
  zobraz(WHITE);
end;

{Smazat objekt znamená zobrazit jej černě }
procedure go.smaz;
begin
  sviti := false;
```

```
       zobraz(BLACK);
end;

{ Zobrazit obecný grafický objekt nemá smysl,}
{ proto volání této procedury způsobí vypsání }
{chybové zprávy a ukončení programu }
procedure go.zobraz;
begin
  outtext('jak se zobrazí grafický objekt?');
  readkey;
  halt(1);
end;

(*****************************************)
{ typ bod a jeho metody}
type
 bod = object (go)
  x,y: integer;
  constructor init(xx, yy: integer);
  destructor done;
  procedure zobraz(barva: word);
  procedure moveto;
  procedure lineto;
 end;

constructor bod.init(xx, yy: integer);
begin
  go.init;
  x := xx;
  y := yy;
end;

destructor bod.done;
begin
  go.done;
end;

{Nakreslí bod se zadanými souřadnicemi }
procedure bod.zobraz(barva: word);
begin
  putpixel(x,y,barva);
end;

{ Přesune grafický kurzor do zadaného bodu }
procedure bod.moveto;
begin
  graph.moveto(x,y);
end;

{Nakreslí úsečku z aktuální pozice grafického kurzoru
 do zadaného bodu }
procedure bod.lineto;
begin
  graph.lineto(x,y);
end;
```

```
(*****************************************)
{ typ úsečka a jeho metody}
type
 usecka = object (go)
   poc, kon: bod;
   constructor init(xx1, yy1, xx2, yy2: integer);
   destructor done;
   procedure zobraz(barva: word);
 end;

constructor usecka.init(xx1, yy1, xx2, yy2: integer);
begin
   go.init;
   poc.init(xx1, yy1);
   kon.init(xx2, yy2);
end;

destructor usecka.done;
begin
   go.done;
end;

{ Nakreslí úsečku v zadané barvě }
procedure usecka.zobraz(barva: word);
begin
   setcolor(barva);
   poc.moveto;
   kon.lineto;
end;

{ instance typu úsečka }
var cara: usecka;
(*****************************************)
{ Testovací procedura }
procedure test;
var
   cara: usecka;
   gd, gm: integer;
   ugo: pgo;

begin
   gd := detect;
   initgraph(gd, gm, cesta);
   cara.init(10,20,50,80);
   ugo := @cara;
   ugo ^.nakresli;
   ugo^.done;
{cara.nakresli;
 cara.smaz;}
   closegraph;
end;

begin
   test;
end.
```

Než se pustíme do rozboru chování tohoto programu, zastavíme se u několika detailů. Jak jsme si řekli předem, v odvozených třídách jsme deklarovali pouze metodu *zobraz*. Metody *nakresli* a *smaž*, které ji využívají a které mají formálně naprosto stejný tvar, jsme deklarovali pouze ve společném předkovi, ve třídě *go*.

Do třídy *bod* jsme přidali metody *moveto* a *lineto*, které přesunou grafický kurzor do zadaného bodu, resp. nakreslí úsečku z aktuální pozice do zadaného bodu. (Dali jsme jim poněkud nelogicky anglické názvy. Všimněte si, že jsme se přesto dovolali globálních funkcí se stejnými jmény; v C++ nám k tomu posloužil unární operátor „::“, v Pascalu kvalifikace jménem jednotky *System*.)

Program nefunguje. Proč?

Jestliže náš první program s grafickými objekty přeložíme a spustíme, dočkáme se zklamání. Vypíše totiž pouze dotaz „Jak se nakreslí obecný grafický objekt?“ a skončí.

Budeme-li jej krokovat v prostředí nebo v Turbo Debuggeru, zjistíme, že příkazy

```
ugo->nakresli();
```

resp.

```
ugo^.nakresli;
```

zavolají metodu *nakresli,* zděděnou po typu *go*, jak jsme očekávali. Metoda *nakresli* ovšem zavolá metodu *zobraz* třídy *go*, nikoli metodu třídy *úsečka*. Podobně dopadneme i v případě, že napíšeme

```
ugo -> zobraz(WHITE); // *
```

resp.

```
ugo^.zobraz(WHITE); {*}
```

I tentokrát se bude volat metoda třídy *go*, nikoli metoda třídy *úsečka*, a to přesto, že ukazatel *ugo* ve skutečnosti obsahuje adresu objektu typu *úsečka*.

Problém je v tom, že překladač nemůže tušit, na jaký objekt bude ukazatel *ugo* ve chvíli volání (tedy za běhu programu) ukazovat. Nemůže také tušit, zda chceme volat v těle metody *nakresli* metodu typu *go* nebo některého z potomků; překladač dokonce ani případné potomky typu *go* nemusí znát, ty můžeme deklarovat v jiném souboru.

Vyjde tedy z informací, které má k dispozici. V metodě *nakresli* typu *go* voláme metodu *zobraz*. To pochopí jako volání

```
this -> zobraz(WHITE);
```

resp.

```
self.zobraz(WHITE);
```

tedy jako volání metody typu *go*. Při volání metody *zobraz* pomocí ukazatele *ugo* překladač vyjde ze skutečnosti, že jsme *ugo* deklarovali jako ukazatel na typ *go*, a přeloží příkazy, označené hvězdičkou v komentáři, prostě jako volání metod třídy *go*.

Poznámka:

Všimněte si, že díky pravidlům dědičnosti v OOP jsme se dostali do situace, že pracujeme (pomocí ukazatelů) s instancí, jejíž skutečný typ vlastně neznáme (a nezná jej ani překladač).

Co s tím?

S dědičností jsme se tedy zatím moc daleko nedostali. Dědění metod nás zklamalo, a stejně nás zklamala i představa, že by potomek mohl vždy zastoupit předka. V čem je chyba? Jsou snad špatné překladače? Nebo ještě hůře – jsou snad jazyky Pascal a C++ nedostatečně objektové?

I to by se možná dalo tvrdit; některé „čistě objektové“ programovací jazyky (např. Smalltalk) by tuto situaci zvládly bez problémů, ovšem za cenu efektivity výsledného programu. V našich dvou jazycích si ale budeme muset poradit jinak.

Než si ukážeme, jak tento problém elegantně zvládnout pomocí prostředků našich dvou programovacích jazyků, zkusíme jej vyřešit pomocí vědomostí, které zatím máme. Snáze tak pochopíme, o co jde.

Jedno z možných řešení je následující: Postaráme se, aby každá instance obsahovala informaci o svém typu. Uděláme to tak, že ve třídě *go* definujeme nový atribut, který nazveme výstižně *typ*. Tento atribut pak zdědí všechny odvozené třídy. Konstruktor do něj uloží hodnotu výčtového typu, který označíme řekněme *druhy* a který bude indikovat druh (tj. skutečný typ) instance.

Při volání metody *zobraz* (nebo jiné podobně problematické metody) se vždy zavolá metoda typu *go*. Dáme jí tedy za úkol, aby si zjistila skutečný typ instance, pro kterou se daná metoda volá, a postarala se o zavolání správné metody.

Podívejme se na řešení v obou programovacích jazycích. (Zde si ukážeme pouze ty části programu, které se liší od předchozí verze. Úplné znění tohoto příkladu najdete v souboru *C4-02.CPP* resp. *P4-02.PAS* na doplňkové disketě.) Deklarace třídy *go* bude mít v C++ tvar

```
/* Příklad C4 - 2 */
// Výčtový typ pro druhy
// grafických objektů
enum druhy {tGo, tBod, tUsecka};

class go
{
 public:
   go();
   ~go();
   void nakresli();
   void smaz();
   void zobraz(unsigned barva);
 protected:
   int sviti;
//Nový atribut, popisující DRUH grafického objektu
   druhy typ;
};

typedef go* pgo;
```

V Pascalu to bude

```
{ Výčtový typ pro identifikaci instancí}
type druhy = (tGo, tBod, tUsecka);

{ V deklaraci typu go přibude nový atribut typ }
go = object
   sviti: boolean;
   typ: druhy;
   constructor init;
   destructor done;
   procedure nakresli;
   procedure smaz;
   procedure zobraz(barva: word);
 end;

 pgo = ^go;
```

Deklarace dalších dvou tříd – *bod* a *úsečka* – zůstanou stejné jako v předchozím (nefungujícím) příkladu. Také deklarace většiny metod zůstanou stejné. Změny se dotknou pouze konstruktorů (všech tříd) a metody *zobraz* třídy *go*. Konstruktory budou mít v C++ tvar

```
// Konstruktor uloží do atributu
// typ informaci o skutečném typu
go::go()
: sviti(0), typ(tGo)
{}

bod::bod(int xx, int yy)
: x(xx), y(yy)
{ typ = tBod;
}

usecka::usecka(int xx1, int yy1, int xx2, int yy2)
: poc(xx1, yy1), kon(xx2, yy2)
{ typ = tUsecka;
}
```

a v Pascalu

```
{ Konstruktor má za úkol vložit do instance
 příznak typu }
constructor go.init;
begin
   sviti := false;
   typ := tGo;
end;

constructor bod.init(xx, yy: integer);
begin
   go.init;
   typ := tBod;
   x := xx;
   y := yy;
end;
```

```
constructor usecka.init(xx1, yy1, xx2, yy2: integer);
begin
  go.init;
  typ := tUsecka;
  poc.init(xx1, yy1);
  kon.init(xx2, yy2);
end;
```

Zbývá metoda *zobraz* třídy *go*, která má za úkol postarat se o volání metody *zobraz*, odpovídající skutečnému typu instance. V C++ můžeme napsat

```
// Metoda go::zobraz se postará
// o zavolání správné metody
// podle skutečného typu instance
void go::zobraz(unsigned barva)
{
  switch(typ)
  {
       case tGo:
           cout << "Jak se zobrazí obecný grafický objekt?"<< endl;
           getch ();
           exit(1);
           break;
        case tBod:
           ((bod*)this) -> zobraz(barva);
           break;
        case tUsecka:
           ((usecka *)this) -> zobraz(barva);
            break;
  }
}
```

Zde jsme prostě přetypovali ukazatel **this** na ukazatel na potomka a zavolali metodu *zobraz*. Pouze v případě, že skutečný typ instance je *go*, ukončíme program s chybovým hlášením.

V Pascalu bude situace trochu složitější; *self* totiž není ukazatel, ale instance předávaná odkazem, a tu nám Pascal nedovolí přetypovat na potomka (instance potomků mají jinou velikost než instance třídy *go*). Musíme proto nejprve získat adresu instance, tu přetypovat (to nám Pascal dovolí) a přetypovaný ukazatel použít k volání správné metody.

```
type
  pbod = ^bod;
  pusecka = ^usecka;

{ Metoda go.Nakresli má za úkol zavolat
 správnou metodu potomka }
procedure go.zobraz;
begin
  case typ of
       tGo: begin
         writeln('Jak se zobrazí grafický objekt?');
         readkey;
         halt(1);
       end;
```

```
       tBod: pbod(@self)^.zobraz(barva);
        tUsecka: pusecka(@self)^.zobraz(barva);
  end;
end;
```

Tento program již funguje, tedy nakreslí úsečku, jak jsme si přáli.

Je ale jasné, že toto řešení má řadu nevýhod. Nejnápadnější z nich asi je, že pokud se rozhodneme kdykoli později definovat dalšího potomka třídy *go*, musíme zasáhnout i do deklarace typu *druhy* a do definice metody *zobraz* třídy *go*. Kromě toho budeme muset podobným způsobem zacházet se všemi metodami, které bychom chtěli v potomkovi předefinovat.

Řešení pomocí třídních ukazatelů v C++

V C++ můžeme problém s voláním metod, jejichž implementace v potomkovi se liší od implementace v předkovi, řešit také pomocí tabulek třídních ukazatelů. Program bude na první pohled vypadat méně přehledně než předchozí příklad, ale ve skutečnosti napodobuje způsob, jakým překladač zachází s tzv. virtuálními metodami, k nimž v našem výkladu pozvolna spějeme.

Základní myšlenka je jednoduchá: V každé z tříd v dané dědické hierarchii definujeme jako statický atribut tabulku ukazatelů na metody, které se mohou v předkovi a v potomkovi lišit. Tento atribut budeme dále označovat jako „tabulku metod“.

Každá instance bude obsahovat ukazatel na tabulku metod své třídy. Tento ukazatel nazveme *uTM* (ukazatel na tabulku metod) a deklarujeme jej jako atribut ve třídě *go*. Všechny odvozené třídy jej zdědí, bude tedy k dispozici v každé z instancí každé z odvozených tříd. Konstruktor bude mít za úkol uložit do tohoto atributu adresu tabulky metod.

Dále definujeme ve třídě *go* metodu *zavolej*, která bude mít jako parametry index v tabulce metod a skutečné parametry volané metody. Tato metoda bude mít na starosti zavolat odpovídající metodu z tabulky.

Nyní se ale musíme postarat o to, aby naše tabulky obsahovaly adresy odpovídajících metod. Na první pohled by se mohlo zdát, že jde o úlohu pro konstruktor. To je ale ve skutečnosti pozdě; v době, kdy budeme v programu konstruovat první instanci, by už měly být tabulky metod inicializovány. Lepším řešením by bylo použít „startovací" funkci, která by se postarala o inicializaci ještě před spuštěním funkce *main*. (Připomeňme si, že „startovací“ posloupnost funkcí se v Borland C++ deklaruje pomocí direktiv **#pragma startup**.)

V C++ můžeme ovšem deklarovat také globální objekty. Jejich konstruktory se volají v rámci startovací posloupnosti (s prioritou 32). To znamená, že naše inicializační funkce by musela mít vyšší prioritu (tedy v rozmezí 0 – 31).

Nevýhodou tohoto řešení je, že je použitelné jen v Borland C++. Ostatní překladače nemusí nic podobného nabízet. Proto si ukážeme jiné řešení, které je použitelné v jakémkoli překladači jazyka C++. Definujeme pomocnou třídu (nazveme ji *go_init*), jejíž konstruktor se postará o inicializaci tabulek metod všech našich tříd. Abychom si situaci zpřehlednili, definujeme v každé z tříd soukromou statickou metodu *init( )*, která bude

mít na starosti inicializaci tabulky metod své třídy. (Metoda *init()* musí být statická, neboť nebude pracovat se žádnou konkrétní instancí své třídy. Budeme ji volat ještě před tím, než první instanci dané třídy vůbec vytvoříme.)

Podívejme se tedy na deklaraci třídy *go* (úplný program najdete na doplňkové disketě v souboru *C4-03.CPP*).

```
/* Příklad C4 - 4 */
// Výčtový typ pro indexy metod, předefinovaných
// v odvozených třídách
enum met {ZOBRAZ, H_POSUN, _met };

// Obecný grafický objekt
class go {
 public:
   go();
   ~go();
   void nakresli();
   void smaz();
   void zobraz(int barva);
   void h_posun(int okolik);
// Typ: ukazatel na metodu
   typedef void (go::*prvtab)(int);
// Nová metoda
   void zavolej(int met, int par);
 protected:
   int sviti;
// ukazatel na první prvek tabulky
   prvtab *uTM;
 private:
// Pole ukazatelů na předefinované metody
   static prvtab TM[_met];
   static void init();
   friend class go_init;
};

// Definiční deklarace tabulky metod
go::prvtab go::TM[_met] = {0,};

typedef go* pgo;
```

Zde jsme nejprve deklarovali výčtový typ *met* pro indexy metod, které chceme v odvozených třídách předefinovat.

Dále jsme pomocí deklarace **typedef** zavedli označení *prvtab* pro typ „ukazatel na metodu třídy *go* s jedním parametrem typu **int**“. Do třídy *go* jsme přidali metodu *h_posun()*, která posune ve vodorovném směru nakreslený objekt o zadanou vzdálenost. (Především proto, abychom měli více metod, které se v odvozených typech liší.)

Ukazatel *uTM*, který bude obsahovat adresu tabulky metod, jsme ve třídě *go* deklarovali jako chráněný (**protected**) atribut. To proto, aby jej mohly bez problémů používat i odvozené třídy.

Tabulku metod *TM* jsme deklarovali jako pole ukazatelů na metody (na typ *prvtab*). Konstruktor třídy *go* uloží do každé z instancí adresu této tabulky:

```
go::go()
: sviti(0), uTM(&go::TM[0])
{}
```

Metoda *go::zavolej()* se bude starat o volání metod z tabulky. Bude mít tvar

```
void go::zavolej(int metoda, int param)
{
   (this->* uTM[metoda])(param);
}
```

Její první parametr je index v tabulce metod, popsaný výčtovým typem *met*, druhý parametr je skutečný parametr metody, kterou chceme volat. Tuto metodu mohou odvozené třídy bez obav zdědit; vzhledem k tomu, že se skutečná metoda volá pomocí ukazatelů, nemůže zde překladač nic zkazit.

Metoda *go::zobraz()* se bude starat – na rozdíl od předchozího příkladu – již zase jen o zobrazení instance své třídy, což znamená, že vynadá tomu, kdo ji zavolal:

```
void go::zobraz(int)
{
   cout << "Jak se zobrazí obecný grafický objekt?"<< endl;
   getch ();
   exit(1);
}
```

Podobně se bude chovat i metoda *go::h_posun()*.

Deklarace třídy *bod* bude mít tvar

```
class bod: public go {
 public:
   bod(int xx, int yy);
   ~bod();
   void zobraz(int barva);
   void moveto();
   void lineto();
// Změna horizontální souřadnice a
// horizontální posun bodu
   void h_zmena(int okolik);
   void h_posun(int okolik);
// Typ: ukazatel na metodu
   typedef void (bod::*prvtab)(int);
 private:
   int x,y;
// Pole ukazatelů na metody
   static prvtab TM[_met];
   friend class go_init;
};

bod::prvtab bod::TM[_met] = {0,};
```

Všimněte si, že jsme znovu deklarovali typ *prvtab*, tentokrát jako ukazatel na metodu třídy *bod*. Ukazatel na tabulku metod není třeba v odvozené třídě deklarovat, neboť jej zdědí po předkovi, třídě *go*.

Konstruktor třídy *bod* nemůže – na rozdíl od konstruktoru třídy *go* – inicializovat ukazatel *uTM* na tabulku metod ve své inicializační části, neboť *uTM* jsme v této třídě nedeklarovali, je to zděděná složka. To znamená, že konstruktor bude mít tvar

```
bod::bod(int xx, int yy)
: x(xx), y(yy)
{
   (prvtab *)uTM = TM;
}
```

Podívejme se na tento konstruktor podrobněji. Z toho, co jsme si řekli ve druhé kapitole, již víme, že nejprve se zavolá konstruktor předka, třídy *go*. Ten uloží do atributu *uTM* adresu tabulky metod třídy *go*. Teprve pak proběhne tělo konstruktoru odvozené třídy *bod*, ve kterém se do *uTM* uloží adresa tabulky metod třídy *bod*. To ale znamená, že v různých fázích inicializace bude atribut *uTM* obsahovat různé hodnoty. My v konstruktorech žádné z „problematických" metod nevoláme, takže nám to zatím nevadí; je ale rozumné si to uvědomit pro případ, že bychom se někdy později rozhodli program nějak podstatně změnit.

Podívejme se nyní na třídu *go_init*. Ve třídě *go* a v jejích potomcích jsme ji deklarovali jako spřátelenou, neboť používá soukromý statický atribut *TM* (tabulku metod). Tato třída obsahuje pouze konstruktor, který se stará o inicializaci tabulek metod:

```
class go_init {
 public:
  go_init();
};

go_init::go_init(){
   static int hotovo = 0;
   if(!hotovo) {
        hotovo = 1;
 // inicializace tabulky třídy go
        go::TM[ZOBRAZ] = &go::zobraz;
        go::TM[H_POSUN] = &go::h_posun;
 // inicializace tabulky třídy bod
        bod::TM[ZOBRAZ] = &bod::zobraz;
         bod::TM[H_POSUN] = &bod::h_posun;
 // inicializace tabulky třídy usecka
        usecka::TM[ZOBRAZ] = &usecka::zobraz;
        usecka::TM[H_POSUN] = &usecka::h_posun;
   }
}
```

Lokální statická proměnná *hotovo* zabezpečuje, že se tělo konstruktoru provede jen jednou, a to i v případě, že omylem deklarujeme několik instancí třídy *go_init*.

V programu musíme deklarovat alespoň jednu globální instanci třídy *go_init*. Tato deklarace by měla předcházet před jakoukoli deklarací instance třídy *go* nebo některého z potomků nebo před vytvořením jakékoli dynamické instance některé z těchto tříd.

Metoda *go::nakresli()* nebude metodu *zobraz()* volat přímo, ale prostřednictvím metody *zavolej()*:

```
void go::nakresli()
{
   sviti = 1;
   zavolej(ZOBRAZ, WHITE);
}
```

Podobně i metoda *smaz()* použije služeb metody *zavolej()*. Takto deklarovanou metodu *nakresli()* můžeme zdědit do potomků a bude bez problémů fungovat. O tom se přesvědčíme např. pomocí testovací procedury, kterou jako obvykle nazveme velice výstižně *test()*. Bude mít tvar

```
void test()
{
   int gd = DETECT, gm;
// Inicializace grafiky
   initgraph(&gd, &gm, cesta);
// Instance třídy usecka
   usecka cara(10,20,50,80);
   pgo ugo = &cara;
// voláme zděděnou metodu
   ugo->nakresli();
   getch();
// voláme metodu posun
   ugo->zavolej(H_POSUN,100);
   getch();
   ugo->smaz();
   getch();
   closegraph;
};
```

Funkce *test()* čeká vždy na stisknutí klávesy, pak provede další akci (přemístění úsečky, smazání úsečky, ukončení programu).

Je jasné, že ani tato cesta není příliš pohodlná a bezpečná. Problémy a nebezpečná místa můžeme shrnout do následujících bodů:

- ✧ Přidáme-li do naší hierarchie novou metodu, která bude v každé ze tříd jiná, musíme změnit deklaraci výčtového typu *met*, který používáme pro indexování metod.
- ✧ Uvedený postup, založený na nepřímém volání, nepůjde použít pro destruktory, neboť ty se volají automaticky při zániku instance nebo při použití operátoru **delete**. Nicméně pokud mají destruktory nějaké netriviální úkoly, může se snadno stát, že se budou v různých odvozených třídách lišit.
- ✧ Přidáme-li do již hotové dědické hierarchie novou třídu (nového potomka), musíme mj. změnit (a překompilovat) i konstruktor třídy *go_init*, který se stará o inicializace tabulky metod.
- ✧ Jestliže se metody, které chceme v potomcích předefinovat, budou lišit v počtu parametrů, budeme muset definovat několik metod *zavolej* (nebo použít pro předávání parametrů výpustku nebo si pomoci jiným nepohodlným trikem).

Zapomeneme-li na některý z těchto bodů, může program provádět neuvěřitelné věci.

5.3 Virtuální metody

Jak jsme viděli v předchozí podkapitole, překladače Pascalu i C++ za normálních okolností používají tzv. časnou vazbu. To znamená, že při volání metody vyhodnotí typ instance, pro kterou danou metodu voláme, již v době překladu. Pokud ovšem pracujeme s potomkem pomocí ukazatele na předka (nebo pokud předáváme potomka jako parametr odkazem a formální parametr je deklarován jako předek), můžeme narazit na problémy: Překladač zavolá metodu předka, i když bychom potřebovali, aby volal metodu potomka.

Ukázali jsme si sice, jak tuto potíž obejít, ale žádné řešení nebylo dobré, neboť vyžadovalo pro každou z předefinovávaných metod mnoho únavně stejných operací v programu, operací, které by koneckonců mohl a měl zařídit překladač sám.

Proto oba naše jazyky nabízejí tzv. **virtuální metody**. To jsou metody, které se v programu volají pomocí tzv. **pozdní vazby**: v případě virtuálních metod se typ instance, pro kterou se daná metoda volá, určuje až za běhu programu („dynamicky“).

Deklarace virtuální metody

V C++ deklarujeme virtuální metodu pomocí klíčového slova **virtual**, které se chová podobně jako specifikátor paměťové třídy (zapisuje se před prototyp metody v deklaraci třídy). Deklarace třídy *go* s virtuálními metodami by tedy mohla mít tvar

```
class go
{ public:
 go();
 ~go();
 void nakresli();
 void smaz();
 // deklarace virtuálních metod
 virtual void zobraz(int barva);
 virtual void h_posun(int okolik);
 protected:
 int sviti;
};
```

Jak vidíte, vrátili jsme se v podstatě k prvnímu návrhu třídy *go*, pouze jsme metody, které chceme v potomcích předefinovat, deklarovali jako virtuální.

V definiční deklaraci metody již klíčové slovo **virtual** neopakujeme. To znamená, že např. metodu *zobraz()* bychom mohli deklarovat takto:

```
void go::zobraz(int barva)
{
   cout << "Jak se zobrazí obecný grafický objekt?"<< endl;
   getch ();
   exit(1);
}
```

Virtuální metoda v potomkovi (jazyk C++)

Pokud chceme v potomkovi virtuální metodu předefinovat (a to obvykle chceme, neboť jinak bychom ji nedeklarovali jako virtuální), musí mít kromě stejného jména také stej-

ný počet a typ parametrů a musí vracet hodnotu stejného typu. Pokud by se deklarace metody *f( )* v předkovi a v potomkovi lišily ve specifikaci parametrů, pochopil by to překladač jako definici homonyma (které mimochodem zastíní metodu, zděděnou z předka). Pokud se budou lišit jen v typu vracené hodnoty, ohlásí překladač chybu.

V pozdějších verzích jazyka C++, např. v Borland C++ 4.x a novějších nebo ve Watcom C++ 10.5, je pravidlo o vracené hodnotě trochu volnější: Jestliže virtuální metoda *f()* vrací v předkovi ukazatel, resp. referenci na objektový typ *A*, musí v potomkovi vracet předefinovaná metoda *f()* ukazatel, resp. referenci na typ *A* nebo na typ, který je veřejným potomkem typu *A*; to znamená, že v C++ může potomek zastoupit předka i v hodnotě vracené virtuální funkcí (pokud se výsledek vrací jako ukazatel nebo reference).

Poznámka:
V deklaraci potomka nemusíme opakovat klíčové slovo **virtual**. *Jestliže jsme např. v typu* go *deklarovali metodu*

```
void go::zobraz(int barva)
```

jako virtuální, bude automaticky virtuální ve všech potomcích. Přesto vřele doporučujeme klíčové slovo ***virtual*** *pro přehlednost opakovat.*

Poznámka:
Pravidla jazyka C++ nedovolují deklarovat konstruktory a statické metody jako virtuální. Statické metody totiž nejsou volány pro žádnou konkrétní instanci své třídy. Mohou být dokonce volány i v situaci, kdy žádná instance dané třídy neexistuje. Proto u nich nemá smysl ani určovat typ instance za běhu programu.
U konstruktorů je situace na první pohled trochu složitější, ale jen na první pohled. Konstruktory jsou v C++ totiž vlastně statické metody. V době, kdy konstruktor voláme, ještě instance neexistuje; konstruktor ji teprve musí vytvořit.

V Pascalu deklarujeme virtuální metody pomocí direktivy *virtual*, kterou zapíšeme v deklaraci třídy za hlavičku metody. To znamená, že deklarace třídy *go* by mohla mít tvar

```
type
 go = object
   sviti: boolean;
   constructor init;
   destructor done;
   procedure nakresli;
   procedure smaz;
   procedure zobraz(barva: word); virtual;
   procedure h_posun(okolik: integer); virtual;
 end;
```

V definici metody tuto direktivu již neopakujeme. Definice metody *zobraz* by tedy mohla mít tvar

```
procedure go.zobraz;
begin
 outtext('jak se zobrazí grafický objekt?');
```

```
 readkey;
 halt(1);
end;
```

Turbo Pascal nedovoluje definovat virtuální konstruktory. V době, kdy konstruktor voláme, totiž vlastně ještě instance neexistuje; konstruktor ji teprve musí z „prázdné“ paměti vytvořit. Nemá tedy smysl určovat typ instance v době volání[4].

Poznámka:
Před prvním voláním virtuální metody ***musíme*** *v Turbo Pascalu zavolat pro danou instanci konstruktor, jinak se program zhroutí! (To platí i při předávání parametrů objektových typů hodnotou. Nejprve je třeba zavolat konstruktor, pak teprve smíme volat virtuální metody.) Později si vysvětlíme, proč tomu tak je.*

Virtuální metoda v potomkovi (jazyk Pascal)
Pokud chceme v potomkovi virtuální metodu předefinovat (a to obvykle chceme, neboť jinak bychom ji nedeklarovali jako virtuální), musí mít kromě stejného jména také stejný počet a typ parametrů a musí vracet hodnotu stejného typu. Pokud by se deklarace metody *f* v předkovi a v potomkovi lišily ve specifikaci parametrů nebo v typu vracené hodnoty, ohlásil by překladač chybu.

Poznámka:
V Object Pascalu v Delphi deklarujeme virtuální metodu ve společném předkovi pomocí direktivy **virtual** *nebo* **dynamic**. *(Obojí znamená v podstatě totéž, rozdíl je jen v technických detailech provedení pozdní vazby.) V potomcích pak specifikujeme virtuální metody pomocí direktivy* **override**, *která je společná pro virtuální i dynamické metody.*

5.4 Nevirtuální metody

Metody, které nejsou virtuální, se obvykle v literatuře o OOP (a také v příručkách o Turbo Pascalu) označují jako **statické**. V publikacích o C++ se však označení **statická metoda** používá pro metody tříd; proto abychom se vyhnuli zmatkům, budeme pro metody, které nejsou virtuální, používat označení **nevirtuální metody**.

[4] Trochu jiná je situace v Object Pascalu v Delphi u „nového modelu“ objektových typů deklarovaných pomocí klíčového slova **class**. Při volání konstruktorů tříd – stejně jako při volání metod třídy – kvalifikujeme identifikátor metody referencí na třídu. Ovšem proměnná typu reference na třídu může obsahovat odkaz na „svou“ třídu nebo na kteréhokoli z potomků. To znamená, že i při volání konstruktorů nebo metod třídy se můžeme dostat do situace, kdy nevíme, s jakou třídou pracujeme. Proto mohou být v Delphi i metody tříd a konstruktory virtuální.

5.5 Polymorfismus

Zavedením virtuálních metod jsme vystoupili na slíbený třetí schod OOP. Třídy, které mají virtuální metody, označujeme také jako **polymorfní** (**mnohotvárné**). Pracujeme-li s instancemi pomocí ukazatelů (a to je při objektovém programování velice běžné), může jeden ukazatel ukazovat postupně na instance mnoha různých typů, což lze interpretovat také tak, že jeden objekt mnohokrát změní tvářnost – je polymorfní.

Teprve polymorfismus umožňuje opravdu využívat dědičnosti.

Proč nejsou všechny metody virtuální

Mnohé z vás jistě napadá, proč nejsou všechny metody virtuální, tedy proč se neurčuje typ instance, pro který danou metodu voláme, vždy až za běhu programu. Důvodem je efektivita přeloženého programu. Na konci této kapitoly, v podkapitole *Jak to funguje*, si povíme, jak se polymorfismus v C++ a Pascalu implementuje. Uvidíme, že při volání virtuálních metod je potřeba o něco více operací než při volání nevirtuálních metod; rozdíl sice není velký, ale přece jen je, a proto ponechávají tvůrci překladače na programátorovi, aby určil, kdy je pozdní vazba nezbytná.

Abstraktní a instanční třídy

Než přepíšeme náš prográmek s grafickými třídami pomocí virtuálních metod, musíme se ještě seznámit se dvěma důležitými pojmy – v nadpisu oddílu jste si mohli přečíst, o jaké půjde.

Při návrhu třídy *go* jsme narazili na problém, co s metodou *zobraz*.

Má smysl zobrazovat bod nebo úsečku, nemá ale smysl zobrazovat obecný grafický objekt. Z tohoto hlediska je tedy nesmyslné zařazovat do definice třídy *go* metodu *zobraz*.

Na druhé straně tuto metodu nemůžeme z definice třídy vyhodit, neboť pak bychom nemohli používat ukazatel na třídu *go* k manipulaci s jakýmkoli grafickým objektem.

Tento rozpor je důsledkem skutečnosti, že *go* je **abstraktní třída**, jejíž instance nebudeme nikdy deklarovat. V programu se setkáme s ukazateli na třídu *go* nebo s referencemi na třídu *go* (při předávání parametrů odkazem), ale nikdy se skutečnou instancí třídy *go*. Můžeme dokonce tvrdit, že výskyt skutečné instance třídy *go* v programu bude znamenat vždy chybu (možná syntaktickou, možná dokonce logickou) v návrhu programu.

Toto dilema jsme zatím řešili tím, že jsme sice ve třídě *go* metodu *zobraz* deklarovali, ale implementovali jsme ji tak, že pouze vypsala výkřik o chybě a ukončila program.

Jazyk C++ nabízí pro tuto situaci tzv. čiré (čisté) virtuální metody (*pure virtual methods*). Čirá virtuální metoda nebude mít implementaci a nezamýšlíme ji volat. Deklarujeme ji konstrukcí

prototyp_metody = 0;

Je jasné, že např. metoda *zobraz()* ve třídě *go* je žhavým kandidátem na pozici čiré virtuální metody. Třídu *go* můžeme pak deklarovat takto:

```
class go
{
 public:
   go();
   ~go();
   void nakresli();
   void smaz();
 // deklarace čiré virtuální metody
   virtual void zobraz(int barva) = 0;
 protected:
   int sviti;
};
```

Třída, která obsahuje alespoň jednu čirou metodu, se v terminologii jazyka C++ označuje jako **abstraktní**; překladač nedovoluje definovat instance abstraktních tříd. Ostatní třídy se označují jako **instanční**, neboť od nich můžeme deklarovat instance.

Čiré metody nemají definiční deklaraci a nelze je volat[5]. Pokud v programu dojde k volání čiré metody, skončí program chybou.

Poznámka:
Starší překladače C++ vyžadují, abychom čiré metody v potomkovi buď znovu definovali jako čiré nebo předefinovali. Novější překladače na opakování deklarace čiré metody v potomkovi netrvají, lze je dědit stejně jako ostatní metody.

Poznámka k terminologii (R. P.):
Někteří autoři (např. M. V.) překládají termín „pure methods" jako „čisté metody". Mně se takovýto překlad nelíbí, protože ve mně vyvolává dojem, že by měly existovat také nějaké „špinavé metody". Takovýto termín sice v češtině existuje, ale jak víme, s programováním nemá nic společného.

Poznámka spoluautora (V.M.):
No a? Anglická terminologie je plná narážek, dvojsmyslů a občas i nesmyslů, a vůbec to není na závadu. Proč se bránit něčemu podobnému v češtině? Ostatně čisté virtuální metody nic nedělají, takže mohou zůstat čisté. Kdo něco dělá, obvykle si – alespoň trochu – zamaže ruce.

Turbo Pascal syntaktický prostředek pro práci s abstraktními metodami nenabízí. V knihovně Turbo Vision je sice definována bezparametrická procedura *Abstract*, kterou lze volat z metod, které bychom v C++ deklarovali jako čiré. Tato procedura má jediný úkol – způsobit běhovou chybu 211. Pokud ovšem knihovnu Turbo Vision z jakýchkoli důvodů nepoužíváme, nezbývá nám, než abychom si ji naprogramovali sami. Jistě to bez problémů zvládnete.

[5] Ve skutečnosti mohou čiré metody mít definiční deklaraci. Nelze je ovšem volat pomocí pozdní vazby.

Object Pascal zná analogii čirých virtuálních metod. Označují se jako „abstraktní“, lze je používat v „novém“ objektovém modelu a deklarují se pomocí vyhrazeného slova *abstract*, které zapisujeme za direktivu *virtual*. Např. takto:

```
{ jen v DELPHI }
type
 go = class
   constructor Create;
   destructor Destroy;
   procedure nakresli;
   procedure smaz;
 {deklarace abstraktní metody }
   procedure zobraz(barva: integer); virtual; abstract;
 protected:
   sviti: integer;
 end;
```

Třídy, které obsahují alespoň jednu abstraktní metodu, se označují jako abstraktní; ostatní třídy se označují jako instanční, neboť od nich lze deklarovat instance.

Opět grafický editor

Nyní se můžeme konečně vrátit k našemu původnímu programu a vyzkoušet, jak budou grafické třídy fungovat s virtuálními metodami.

V předchozím povídání jsme si ujasnili, že v naší hierarchii grafických objektů musí být virtuální metoda *zobraz*, jež implementuje zobrazení grafických objektů, a proto se bude v jednotlivých třídách lišit. Ve společném předkovi, třídě *go*, ji deklarujeme jako čirou metodu. Také destruktor deklarujeme jako virtuální – později si povíme proč.

Ostatní metody virtuální být nemusí, protože jejich definice jsou pro všechny třídy stejné (nebudeme je v potomcích předefinovávat).

To znamená, že deklarace tříd *go*, *bod* a *usecka* budou mít v C++ tvar

```
/* Příklad C4 - 4 */
class go
{
 public:
   go();
   virtual ~go();
   oid nakresli();
   oid smaz();
   irtual void zobraz(unsigned barva) = 0;
 protected:
   int sviti;
};

class bod: public go{
 public:
   bod(int xx, int yy);
   virtual ~bod();
   virtual void zobraz(unsigned barva);
   void moveto();
```

```
  void lineto();
 private:
  int x,y;
};

class usecka: public go {
 public:
  usecka(int xx1, int yy1, int xx2, int yy2);
  virtual ~usecka();
  virtual void zobraz(unsigned barva);
 private:
  bod poc, kon;
};
```

Definice jednotlivých metod a testovací procedury se nebudou lišit od příkladu C4 – 01, pouze destruktor třídy *go* bude „prázdný“:

```
// Destruktor tentokrát nedělá nic
go::~go()
{ }
```

V odstavci *Konstruktory, destruktory a virtuální metody* si povíme, proč jsme z destruktoru odstranili volání metody *smaz()*. (Úplnou podobu tohoto příkladu najdete na doplňkové disketě v souboru *C4–04.CPP*.)

V Turbo Pascalu si nejprve deklarujeme pomocnou proceduru *pure*, kterou použijeme v těle čirých metod:

```
procedure pure;
begin
 writeln('Volání abstraktní metody');
 halt(211);
end;
```

Deklarace typu *go* a jeho potomků bude

```
type
 go = object
  sviti: boolean;
  constructor init;
  destructor done; virtual;
  procedure nakresli;
  procedure smaz;
  procedure zobraz(barva: word); virtual;
 end;

 bod = object (go)
  x,y: integer;
  constructor init(xx, yy: integer);
  destructor done; virtual;
  procedure zobraz(barva: word); virtual;
  procedure moveto;
  procedure lineto;
 end;
```

```
 usecka = object (go)
  poc, kon: bod;
  constructor init(xx1, yy1, xx2, yy2: integer);
  destructor done; virtual;
  procedure zobraz(barva: word); virtual;
 end;
```

Procedura *go.zobraz* prostě zavolá pomocnou proceduru *pure* a tím ukončí program;

```
procedure go.zobraz;
begin
 pure
end;
```

Deklarace ostatních metod a testovací procedury se neliší od příkladu P4 – 01. (Úplnou podobu tohoto příkladu, tj. definice všech metod a testovací procedury, najdete na doplňkové disketě v souboru *P4–01.PAS*.)

Virtuální destruktory

Mnoho programátorů zapomíná na možnost, a často nutnost, používat virtuální destruktory – nejspíš proto, že pravidla obou jazyků svorně zakazují virtuální konstruktory. Jistě si všichni snadno představíme situaci, kdy voláme destruktor pro instanci, jejíž skutečný typ v průběhu výpočtu neznáme – je to stejné jako u ostatních metod. (Budeme např. chtít vyprázdnit seznam, který obsahuje ukazatele na grafické objekty. Pro všechny uložené grafické objekty je třeba zavolat destruktor a přitom o žádném z nich v době kompilace nevíme, jakého bude doopravdy typu.)

Pokud destruktor každé z odvozených tříd dělá něco jiného, musí se skutečný typ destruované instance určit až v průběhu výpočtu, a v takovém případě musí být destruktor virtuální.

V našem případě bychom se zatím bez virtuálních destruktorů obešli, neboť žádný z nich vlastně nedělá nic. Přesto je deklarujeme jako virtuální – pro případ, že bychom v některém z odvozených typů od destruktoru něco potřebovali.

Jak to funguje

Podívejme se nyní, jak virtuální metody fungují. Můžeme sice úspěšně programovat, aniž bychom o tom měli nejmenší tušení, ale pochopíme-li, o co jde, vyhneme se snáze některým chybám a bude nám jasná i cena, kterou za polymorfismus platíme.

Pro každou třídu, která obsahuje alespoň jednu virtuální metodu, vytvoří překladač tabulku virtuálních metod. To je tabulka, která obsahuje adresy všech virtuálních metod v dané třídě. Tato tabulka je pro všechny instance společná (mohli bychom ji označit za skrytý atribut třídy). Dále pro ni budeme často používat zkratku VMT (z anglického *Virtual Method Table*).

Dále bude každá instance obsahovat ukazatel na tabulku virtuálních metod dané třídy. Tento ukazatel přidá do instance překladač automaticky a programátorovi je nepřístupný. Překladač se také postará, aby tento ukazatel byl ve všech instancích na stejném místě vzhledem k počátku instance.

Adresu VMT uloží do instance konstruktor. V C++ se to stane automaticky při vytvoření instance, v Pascalu musíme konstruktor zavolat sami.

Podíváme se na příklad. Budeme předpokládat, že *ugo* je ukazatel na typ *go* (může tedy obsahovat adresu instance typu *bod* nebo *usecka*). Podívejme se, co se bude dít při volání virtuální metody *zobraz* příkazem

```
ugo -> zobraz(WHITE);
```

resp.

```
ugo^.zobraz(WHITE);
```

(podle toho, kterému jazyku dáme přednost).

Překladač ví, že adresa metody *zobraz* je v tabulce virtuálních metod třídy *go* i všech jejích potomků uložena jako první. Proto se její volání přeloží tak, že

- ✧ program nejprve vezme instanci, na kterou ukazuje *ugo*, a v ní vyhledá ukazatel na VMT (jak víme, je ve všech instancích třídy *go* i jejích potomků na stejném místě, takže přitom není potřeba znát skutečný typ instance),
- ✧ na základě adresy, zjištěné v instanci, najde v tabulce virtuálních metod adresu první metody, tj. metody *zobraz*,
- ✧ tuto metodu zavolá se zadanými parametry.

To znamená, že pokud bychom se v Turbo Pascalu pokusili zavolat virtuální metodu pro instanci, pro níž jsme dosud nezavolali konstruktor, nenašel by program správnou adresu tabulky virtuálních metod a program by se zhroutil.

5.6 Cena polymorfismu

Dosud jsme hovořili pouze o výhodách virtuálních metod. Podívejme se nyní, co nás stojí.

Z toho, co jsme si dosud pověděli, plyne, že instance polymorfních tříd (tedy tříd s alespoň jednou virtuální metodou) budou větší než instance třídy se stejnými složkami, ale bez virtuálních metod. Navíc je zde ukazatel na VMT. Kromě toho vytvoří překladač pro každou třídu tabulku virtuálních metod; i když je pro každou třídu jen jedna, společná pro všechny instance, zabírá místo v paměti a zvětšuje nároky programu. Kromě toho je volání virtuálních metod pomalejší než volání obyčejných metod nebo funkcí, neboť program musí předem zjistit adresu volané metody.

Odtud také plyne, že pro každou třídu, která má alespoň jednu virtuální metodu, musíme v Turbo Pascalu deklarovat konstruktor.

V C++ máme situaci jednodušší: konstruktory se volají automaticky při deklaraci objektu, To znamená, že nemáme možnost pracovat s objektem, ve kterém by nebyla správně inicializována adresa tabulky virtuálních metod. Navíc pokud pro nějakou třídu konstruktor nedeklarujeme, vytvoří si jej překladač sám (a pokud to z nějakých důvodů nezvládne, upozorní nás na to chybovým hlášením).

Kdy se pozdní vazba uplatní

Jestliže při volání virtuální metody nepoužijeme ukazatele, nemusí se pozdní vazba uplatnit. Je-li např. *b* instance typu *bod* a napíšeme-li

```
b.nakresli(BLACK);
```

nepoužije Borland C++ pozdní vazbu, neboť je to zbytečné: typ instance, pro kterou danou metodu voláme, je znám již v době překladu. Na druhé straně Borland Pascal 7.0 zde pozdní vazbu celkem zbytečně použije.

Musíme ale zdůraznit, že ukazatele se používají (a tedy pozdní vazba se uplatní) také při použití referencí – tedy v případě, že voláme virtuální metodu pro parametr předávaný odkazem.

Také volání jedné metody v těle jiné metody je vlastně volání pomocí ukazatelů. Připomeňme si, že zápis v C++

```
void bod::smaz(){
  nakresli(BLACK);
}
```

se interpretuje jako

```
void bod::smaz(){
  this->nakresli(BLACK);
}
```

a zápis v Pascalu

```
procedure bod.smaz;
begin
 zobraz(BLACK);
end;
```

znamená vlastně

```
procedure bod.smaz;
begin
 self.zobraz(BLACK);
end;
```

kde **this**, resp. *self* jsou (skryté) parametry metod; připomeňme si, že **this** je ukazatel a *self* instance předávaná odkazem.

Konstruktory, destruktory a virtuální metody

Můžeme v konstruktorech a v destruktorech volat virtuální metody? Samozřejmě, můžeme, ale… Podívejme se, jak to je přesně. Situace je totiž v každém z jazyků jiná.

Konstruktory, destruktory a virtuální metody v C++

V okamžiku, kdy se začíná provádět tělo konstruktoru, jsou již inicializace hotové. To znamená, že je mj. inicializován skrytý atribut obsahující adresu VMT. Proto můžeme v těle konstruktoru volat i virtuální metody.

Musíme si ale uvědomit, že při konstrukci potomka se vždy nejprve zavolá konstruktor předka a ten uloží do odkazu na VMT adresu tabulky virtuálních metod předka. Teprve pak přijde ke slovu konstruktor potomka, který uloží do odkazu na VMT adresu tabulky potomka. (Jinými slovy: konstruktor se v C++ nestará o to, zda je volán pro konstrukci zděděného podobjektu nebo pro konstrukci samostatné instance.) To znamená, že se virtuální metody v konstruktoru chovají „nevirtuálně".

Důvod je jednoduchý: v okamžiku, kdy se volá konstruktor předka, ještě není zkonstruována instance potomka. Nemá tedy také smysl pokoušet se volat virtuální metody potomka.

Podobná je situace i v destruktoru. Destruktor volá vždy virtuální metody své třídy, nikoli virtuální metody potomka, a to i při destrukci zděděného podobjektu.

I tady je vysvětlení jednoduché. Jak víme, při destrukci potomka se nejprve provede destruktor potomka a teprve pak se volají destruktory zděděných podobjektů. To znamená, že v okamžiku volání destruktoru pro zděděný podobjekt již instance předka jako celek neexistuje a nemá tedy smysl volat pro ni virtuální metody.

Podívejme se na jednoduchý příklad. Deklarujeme třídu *A*, která bude mít kromě konstruktoru a destruktoru jednu virtuální metodu *f()*; kterou bude konstruktor i destruktor volat.

Tato třída bude mít potomka *B*, ve kterém metodu *f()* předefinujeme. Metoda *f()* (jak verze z třídy *A* tak i verze z *B*) vypíše svou identifikaci, tj. řetězec ″A::f" nebo "B::f", a pak svůj parametr, který určuje místo, odkud byla metoda volána.

```
/* Příklad C4 - 5 */
#include <iostream.h>
class A {
 public:
   virtual void f(char *);
// Konstruktor i destruktor volá
// virtuální metodu
   A() {f("Konstruktor třídy A");}
   ~A(){f("Destruktor třídy A");}
};

// Virtuální metoda předka
void A::f(char * c){
   cout << "A::f: " << c << endl;
}

class B: public A {
 public:
   virtual void f(char *);
// Konstruktor i destruktor opět
// volá virtuální metodu
   B() {f("Konstruktor třídy B");}
   ~B(){f("Destruktor třídy B");}
};

// Virtuální metoda potomka
void B::f(char * c){
   cout << "B::f: " << c << endl;
```

```
}
int main(){
// Zde se volá konstruktor
// a hned také destruktor
  B b;
  return 0;
}
```

Tento program vypíše

```
A::f: Konstruktor třídy A
B::f: Konstruktor třídy B
B::f: Destruktor třídy B
A::f: Destruktor třídy A
```

Při konstrukci zděděného podobjektu typu *A* se zavolala metoda *A::f()*, jako kdyby nebyla virtuální. Podobně i při destrukci zděděného podobjektu se zavolala metoda *A::f()*.

V odstavci *Opět grafický editor* jsme slíbili, že vysvětlíme, proč destruktor třídy *go* nemůže volat metodu *smaz()*. Podívejme se, co by se stalo, kdybychom v destruktoru třídy *go* ponechali volání metody *smaz()*. Podívejme se, jak by vypadal zánik instance *cara* třídy *usecka*:

- ✧ Nejprve se zavolá destruktor *usecka::~usecka()*.
- ✧ Po jeho skončení se zavolá destruktor *go::~go()* zděděného podobjektu typu *go*. **Tento destruktor si změní odkaz na VMT tak, aby ukazoval na tabulku virtuálních metod třídy** *go*, a pak zavolá metodu *smaz()*.
- ✧ Metoda *smaz()* zavolá virtuální metodu *zobraz()*. Protože ale odkaz na VMT ukazuje na tabulku třídy *go*, bude se volat čirá virtuální metoda *go::zobraz()* a program skončí chybou.

Doporučujeme vám, abyste si to vyzkoušeli. Vezměte program v příkladu C4 – 04, do destruktoru doplňte volání metody *smaz()* a program přeložte a spusťte. Dostanete chybové hlášení

```
"Pure virtual function called".
```

Konstruktory, destruktory a virtuální metody v Turbo Pascalu

V Turbo Pascalu se konstruktory a destruktory zděděných podobjektů chovají poněkud jinak než v C++.

Připomeňme si, že konstruktor, resp. destruktor předka voláme buď pomocí klíčového slova **inherited** (jen ve verzi 7.0 a v Delphi) nebo tím, že identifikátor metody kvalifikujeme jménem objektového typu. To u konstruktoru potlačí nastavování odkazu na VMT, takže i v konstruktoru předka bude odkaz na VMT ukazovat na tabulku potomka. Podobně i v těle destruktoru potomka bude odkaz na VMT ukazovat na tabulku potomka.

Podívejme se na jednoduchý příklad (v podstatě stejný jako v C++). Deklarujeme třídu *A*, která bude mít kromě konstruktoru a destruktoru jednu virtuální metodu *f*, kterou bude konstruktor i destruktor volat.

Tato třída bude mít potomka *B*, ve kterém metodu *f* předefinujeme. Metoda *f* (jak verze z předka, třídy *A*, tak i verze z potomka, třídy *B*) vypíše svou identifikaci, tj. řetězec 'A::f' nebo 'B::f', a pak svůj parametr, který určuje místo, odkud byla metoda volána.

```
(* Příklad P4 - 5 *)
{ Volání virtuálních metod v konstruktorech a v destruktorech }
{Třída A má pouze konstruktor, destruktor a virtuální metodu f }
type A = object
 procedure f(s: string); virtual;
 constructor init;
 destructor done;
end;

constructor A.init;
begin
 f('Konstruktor třídy A');
end;

destructor A.done;
begin
 f('Destruktor třídy A');
end;

procedure A.f(s: string);
begin
 writeln('A.f: ' + s);
end;

{ Také třída B má pouze konstruktor, destruktor a virtuální metodu f }
type
 B = object(A)
 procedure f(s: string); virtual;
 constructor init;
 destructor done;
end;

constructor B.init;
begin
 A.init;
 f('Konstruktor třídy B');
end;

destructor B.done;
begin
 f('Destruktor třídy B');
 A.done;
end;

procedure B.f(s: string);
begin
 writeln('B.f: '+s);
end;
```

```
var
 bb: B;
begin
 {Zde voláme konstruktor a hned pak destruktor třídy B }
bb.init;
 bb.done;
end.
```

Tento program vypíše

```
B.f: Konstruktor třídy A
B.f: Konstruktor třídy B
B.f: Destruktor třídy B
B.f: Destruktor třídy A
```

Všimněte si, že na rozdíl od C++ se v konstruktoru a destruktoru předka volala metoda potomka *B.f* (ovšem s parametry, které popisují místo volání).

Mohlo by se stát, že se virtuální metoda, zavolaná v destruktoru předka, pokusí použít data, která již neexistují – která již destruktor potomka „zlikvidoval". Podobně by se mohlo stát, že se virtuální metoda potomka, zavolaná v konstruktoru předka, pokusí použít data potomka, která ještě konstruktor potomka nevytvořil. Vzhledem k tomu, že si ale v Turbo Pascalu můžeme předepsat, kdy budeme konstruktory, resp. destruktory předků volat (a zda je vůbec chceme volat), měli bychom se podobným problémům dokázat vyhnout.

5.7 Delphi: Metody pro ošetření zpráv od Windows

Na závěr připojíme ještě poznámku o metodách pro ošetřování zpráv od Windows v Object Pascalu v Delphi. Tyto metody jsou totiž také implementovány jako virtuální, i když se po formální stránce poněkud liší.

Metodu pro ošetření zpráv od Windows deklarujeme vždy jako proceduru s jedním parametrem předávaným odkazem. V deklaraci třídy musí za hlavičkou metody následovat direktiva *message*, následovaná celočíselnou konstantou v rozsahu 0 .. 32767, určující identifikační číslo zprávy, kterou bude daná metoda ošetřovat. Pro standardní zprávy můžeme použít předdefinovaných konstant.

Podívejme se na příklad:

```
type
  TBox = class(TCustomControl)  TCustomControl je předdefinovaná třída }
       private
        procedure WMChar(var Msg: TWMChar); message WM_CHAR;
{...}
end;
```

WM_Char je konstanta, definovaná v jednotce *Windows,* která označuje zprávu, vzniklou zpracováním stisku klávesnice.

Všimněte si, že ač jsou metody pro zpracování zpráv implementovány jako virtuální, nepoužívá se v jejich deklaraci žádná z direktiv *virtual*, *dynamic* nebo *override*. Dokonce platí, že metoda pro ošetření zpráv může mít v potomkovi jiné jméno než v předkovi. Pokud chceme použít zděděnou metodu, můžeme ji v potomkovi volat pomocí samotného klíčového slova **inherited**. Např. takto:

```
procedure TBox.WMChar(var msg: TWMChar);
begin
  if(Chr(msg.CharCode) = #13 then ZpracujEnter
  else inherited;
end;
```

Pokud instance třídy *TBox* přijme zprávu *WM_Char*, zavolá se automaticky tato metoda. Obsahuje-li parametr *msg* znak #13, tj. kód stisknutí klávesy ENTER, zavolá metoda *WMChar* proceduru *Zpracuj*, jinak zavolá zděděnou metodu; její jméno nepotřebujeme znát, je jednoznačně určena tím, že zpracovává zprávu *WM_Char*. Pokud žádný z předků metodu pro ošetření zprávy *WM_Char* neobsahuje, nevadí – zavolá se virtuální metoda *DefaultHandler*, implementovaná ve společném předkovi, třídě *TObject*.

6. Příklad: jednoduchý grafický editor

V předchozí kapitole jsme sice uvažovali o možnosti napsat jednoduchý grafický editor, ale stále jsme neznali vše, co k tomu bylo třeba. Nyní však již víme o zapouzdření, dědičnosti a polymorfismu vše, co je třeba, a proto se k našemu nápadu vrátíme.

Vzhledem k rozsahu knihy zde samozřejmě nemůžeme uvést zdrojové texty všech částí programu v obou jazycích. Najdete je na doplňkové disketě v adresářích *CPP\C5*, resp. *PAS\C5*.

6.1 Organizace objektového programu

Náš program nebude nijak zvlášť velký, ale přesto bude obsahovat několik různých tříd. Nejvhodnější by tedy bylo ukládat každou třídu do zvláštního souboru. (Někdy je ovšem přehlednější ponechat v jednom souboru třídy, které spolu bezprostředně souvisejí – záleží na okolnostech, zejména na velikosti souborů.)

Připomeňme si, že v C++ zapisujeme deklaraci třídy do hlavičkového souboru (*.H*) a definice metod a případně definiční deklarace statických atributů do souboru *.CPP*. Hlavičkový soubor pak vložíme pomocí direktivy **#include** do všech souborů, ve kterých budeme danou třídu používat, mezi jiným také do souboru s definicemi metod.

V Pascalu zapisujeme obvykle každou ze tříd do samostatné jednotky (*unit*).

V C++ nám nic nebrání deklarovat třídu *XXX* v souboru *XXX.H* a její metody popsat v *XXX.CPP*; je to přehledné a v našem příkladu toho také využijeme. V Pascalu je situace poněkud složitější, neboť jednotka se musí jmenovat stejně jako soubor, ve kterém je uložena (jinak ji překladač nenajde), ale objektový typ nemůže mít stejné jméno jako jednotka nebo program, uvnitř něhož jej deklarujeme. Proto se budou názvy souborů v Pascalu poněkud lišit od názvů tříd.

6.2 Zadání

Pokusíme se tedy napsat jednoduchý grafický editor. Položíme si následující minimální požadavky:

- ✧ Program poběží v reálném režimu pod DOSem.
- ✧ Kreslit (umísťovat objekty) budeme pomocí myši. Některé další operace budeme zadávat prostřednictvím menu nebo z klávesnice.
- ✧ Budeme kreslit bílé objekty na černém pozadí, jiné barvy nebudeme používat.
- ✧ Základní nabídka typů grafických objektů musí obsahovat alespoň bod, úsečku a kružnici.
- ✧ Musíme mít možnost nakreslený objekt dále upravovat, tzn. přemísťovat, zvětšovat nebo zmenšovat, otáčet, smazat (odstranit).

V tomto seznamu chybí řada důležitých specifikací: nehovoříme zde o ukládání obrázků do souboru, o možnostech exportu obrázků do některých známých formátů, o importu, nespecifikujeme podrobnosti ovládání myší ani vzhled uživatelského rozhraní atd.

Při vytváření objektů se spokojíme s velice primitivním postupem: po kliknutí myší se vytvoří jakýsi „standardní“ objekt (např. kružnice o poloměru 50 pixelů se středem v daném bodě) a tento objekt si teprve dále upravíme – zmenšíme, posuneme, otočíme apod.

K tomu máme dva dobré důvody: za prvé, příliš složitý příklad by byl nepřehledný a tím pádem by pro nás – a především pro vás – ztratil význam, a za druhé se s tímto příkladem musíme vejít do vymezeného rozsahu knihy. Můžete se ovšem pokusit sami napsat podobný program, který nabídne uživateli více možností, lepší ovládání atd. – fantazii se meze nekladou.

6.3 Základní schéma programu

Začneme návrhem architektury programu. Ten se skládá z řady kroků, ve kterých postupně zpřesňujeme popis programu a ujasňujeme si postupy, algoritmy a datové struktury, které použijeme.

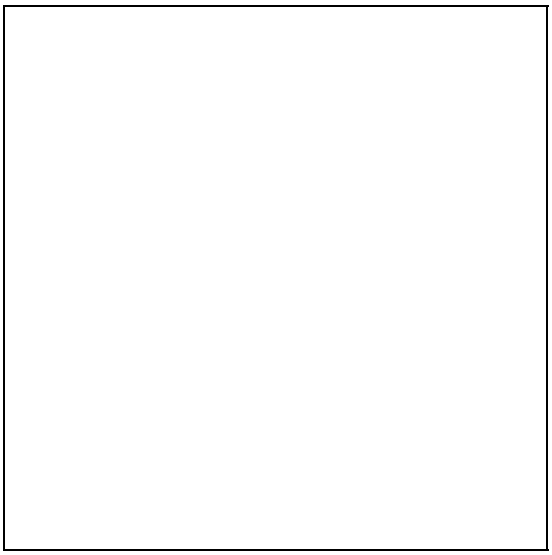
Náš grafický editor bude mít – ostatně jako téměř každý program – dvě základní části: uživatelské rozhraní a výkonnou část (obr. 5.1). Ty by měly být pokud možno oddělené, nezávislé, a komunikovat spolu pouze prostřednictvím komunikačního rozhraní (kanálu).

Protože vycházíme z objektového návrhu, bude editor instancí nějaké třídy (třídu nazveme *edit*, instanci *e*), která bude mít základní složky *rozhraní*, *komunikační kanál* a *výkonnou část*. Vedle toho musíme počítat i s *administrativní částí*, která bude obsahovat různé pomocné složky a funkce (jako např. inicializaci grafického režimu). Schématická deklarace takové třídy tedy bude v C++

```
class edit {
  // rozhraní
  // komunikační kanál
  // výkonná část
  // administrativní část
};
```

a v Pascalu

```
type
  edit = object
       {rozhraní}
       {komunikační kanál }
       {výkonná část }
       {administrativní část }
  end;
```



Obr. 5.1 Základní schéma našeho grafického editoru

Program poběží takto: Rozhraní bude přijímat pokyny od uživatele a transformovat je na zprávy pro výkonnou část. Zprávy bude rozhraní ukládat do komunikačního kanálu. Výkonná část si zprávy z kanálu vyzvedne a zpracuje (vytvoří grafický objekt, přesune jej, smaže atd.)

To znamená, že základní výkonná metoda (zapneme fantazii na plné obrátky a vymyslíme pro ni příznačné jméno *běh*, neboť má na starosti běh našeho programu) bude mít tvar:

```
void edit::beh(){
   // Úvodní operace
   while(!konec){
        // Příjem zpráv od klávesnice a myši
        // Transformace na zprávy pro výkonnou část
        // Zpracování výkonnou částí
   }
}
```

resp. v Pascalu

```
procedure edit.beh;
begin
   {Úvodní operace }
   while not konec do begin
        {Příjem zpráv od klávesnice a myši }
        {Transformace na zprávy pro výkonnou část }
        {Zpracování výkonnou částí }
   end
end;
```

Hlavní program v C++ bude v vypadat velice jednoduše:

```
edit e;

int main(){
  e.beh();
  return 0;
}
```

Jeho pascalská varianta bude jen o málo složitější, neboť musí obsahovat explicitní volání konstruktorů a destruktorů:

```
var e: edit;

begin
  e.init;
  e.beh;
  e.done;
end.
```

Uživatelské rozhraní

Při dalším zpřesňování návrhu začneme uživatelským rozhraním. Řekli jsme si, že chceme program ovládat jak pomocí myši, tak i pomocí klávesnice. Co to znamená?

Zprávy od uživatele můžeme rozdělit na tři základní skupiny: určení typu grafického objektu, manipulace s grafickým objektem nebo prostředím editoru, vytvoření grafického objektu. Požadavek na vytvoření grafického objektu zadá uživatel kliknutím myší na pracovní plochu editoru. Ostatní pokyny bude předávat kliknutím myší na odpovídající položku menu nebo stisknutím některé klávesy.

Z tohoto popisu vyvodíme, že uživatelské rozhraní bude obsahovat menu, prostředky pro ovládání myši a prostředky pro zpracování přicházejících zpráv.

Definujeme tedy třídu *menu*, která nám poskytne nástroje pro tvorbu nabídek a pro zpracování příkazů, které uživatel jejich prostřednictvím zadá. Podobně definujeme třídu *mys*, do které ukryjeme procedury pro práci s hlodavcem.

(Všimněte si, že jsme se zatím vyhnuli např. bližší specifikaci operací s menu. Řekli jsme, že třída *edit* bude mít nějaké složky typu *menu*, a způsob zacházení s menu popíšeme později. Třída *menu* – resp. její instance – převezmou zodpovědnost za některé z funkcí rozhraní.)

Bude asi rozumné oddělit od sebe menu pro specifikaci druhu grafického objektu a menu, určující akci s objektem (např. otočení) nebo s celým editorem (např. skončení programu). Náš program bude tedy mít dvě menu, která pojmenujeme výstižně *Akce* a *Typ*. Tím jsme jinými slovy řekli, že rozhraní bude obsahovat instanci *hlodavec* třídy *mys* a instance *Typ* a *Akce* třídy *menu*. (Pokud jde o myš, budeme muset svůj názor – alespoň v Pascalu – časem poopravit. Ale to se při programování stává.)

Znakové řetězce

V této fázi návrhu se také zamyslíme se nad tím, jak budeme zacházet se znakovými řetězci[6] – v našem programu půjde o texty nabídek a chybových hlášení. Máme několik možností, např.:

- ✧ Příliš se nevzrušovat a zapsat je jako konstanty na místě, kde je potřebujeme. To je nejjednodušší, avšak nejméně výhodná možnost. Řetězce se nejspíš budou několikrát měnit a každá změna bude znamenat náročné vyhledávání, opravu a nový překlad programu.
- ✧ Výhodnější je soustředit řetězce v jednom modulu jako konstanty, např. v C++

```
// Řetězce pro menu AKCE
const char * Nadpis = "AKCE";
const char * nabidka1 = "Smaž";
// ...atd.
```

resp. v Pascalu

```
{ Řetězce pro menu AKCE }
const Nadpis: string = 'AKCE';
const nabidka1: string = 'Smaž';
{ ...atd. }
```

Potřebujeme-li některý řetězec změnit, snadno jej najdeme, přeložíme jediný modul a *jede se dále močálem černým kolem bílých skal* (Jan Werich).

- ✧ Řetězce můžeme uložit do samostatného souboru (nebo do několika souborů) a přečíst si je teprve za běhu programu. To umožní upravovat znakové řetězce, aniž bychom museli znovu překládat program. Může si je dokonce měnit i uživatel, aniž by k tomu potřeboval vědět cokoli bližšího o programu samotném. Na druhé straně je to samozřejmě pomalejší.
- ✧ V programech pro Windows můžeme řetězce uložit do tabulek řetězců (*stringtable*), které jsou součástí prostředků programu (*resources*). Prostředky jsou připojeny ke spustitelnému souboru, ve kterém je lze editovat pomocí speciálních editorů prostředků (jako je borlandský *Resource Workshop*).

Pro nás samozřejmě připadají v úvahu pouze první tři možnosti. Použijeme třetí, tj. budeme texty číst ze souborů.

---

[6] I když se to na první pohled možná nezdá, představuje zacházení se znakovými řetězci dosti důležitou součást návrhu programu. Řetězce jsou poměrně nápadnou součástí výstupu programu a požadavky na jejich změny jsou dosti časté – mohou záviset nejen na náladě programátora, ale i na potřebách nebo vkusu zákazníka. Také při lokalizaci programu, tedy při převodu do jiného jazyka, se mění právě znakové řetězce.

Ošetření chyb

Také ošetřování chyb za běhu programu patří mezi strategická rozhodnutí, která nelze příliš dlouho odkládat. My zvolíme nejjednodušší možný postup: v případě chyby (nelze otevřít soubor, nelze inicializovat grafiku apod.) vypíšeme zprávu o chybě a skončíme. Můžeme si to dovolit, neboť náš program – alespoň jak jej zde vytvoříme – stejně nebude umět ukládat data, takže nehrozí, že bychom o nějaká přišli.

V C++ tím pověříme statickou metodu *chyba( )* třídy *edit* (musí být statická, jinak bychom ji totiž nemohli volat již v konstruktorech atributů třídy *edit*). V Pascalu statické metody nemáme k dispozici, proto použijeme globální proceduru *chyba_proc()*.

Komunikační kanál

Komunikační kanál, to je datová struktura, do které bude rozhraní ukládat příkazy pro výkonnou část. Bude obsahovat souřadnice bodu, ve kterém uživatel stiskl myš, a výsledky zpracování vstupu – příznak druhu grafického objektu a příznak požadované akce. Pro takto fungující kanál není třeba (alespoň zatím) definovat žádné metody – bude to jen skupina atributů ve třídě *edit*.

Výkonná část

V zadání se hovoří o bodu, kružnici a úsečce. To budou třídy, tedy objektové typy. Editor bude pracovat s jejich instancemi; ty si musí ukládat tak, aby se k nim mohl později vracet.

Základem výkonné části bude dvousměrný seznam grafických objektů. Grafické objekty v něm budou uloženy v pořadí, v jakém je uživatel vytvořil.

Každý nově vytvořený grafický objekt se po vytvoření vloží na konec seznamu. S tímto objektem budeme po vytvoření chtít ještě manipulovat – přemísťovat jej, zvětšovat nebo zmenšovat, možná ho budeme chtít i smazat a nakreslit místo něj jiný objekt. Proto bude vhodné přidat do výkonné části ještě ukazatel na aktuální objekt.

Zatím se nám tedy rýsuje jediná metoda výkonné části: zpracování informací, uložených v rozhraní. Ta bude využívat služeb seznamu grafických objektů a jednotlivých grafických objektů.

6.4 Další zpřesňování návrhu

Nyní bychom již měli navrhnout rozhraní jednotlivých tříd, které jsme se rozhodli v programu použít. Může se samozřejmě stát, že později vyvstane potřeba zavést další třídy.

Obvykle se začíná návrhem rozhraní. To je rozumné i v našem případě; začneme od nabídek.

Třída menu

Nyní začínáme v určitém smyslu znovu, od začátku. Stojíme před úkolem navrhnout třídu *menu*, jejíž instance budou sloužit jako menu neboli nabídky v našem grafickém editoru.

Abychom si co nejvíce zjednodušili situaci (a vešli se do vymezeného rozsahu knihy), nebudeme dělat rozbalovací menu; nabídky budou stále na obrazovce.

Menu bude obsahovat nadpis (např. „TYP“) a výčet nabídek. V menu, obsahujícím typy objektů, by měla být nějak vyznačena aktuální hodnota, tedy typ objektu, který se bude kreslit. Na druhé straně v nabídce akcí takovéto označení postrádá smysl.

Každá instance třídy *menu* bude obsahovat jednak pole řetězců, obsahujících nabídky, jednak pole znaků, obsahujících horké klávesy. Dále tu bude atribut vyznačující, zda se má v menu barevně odlišovat naposledy zvolená položka.

Další atributy budou spíše administrativní: souřadnice levého dolního a pravého horního rohu menu na obrazovce, odsazení od okraje a mezi řádky, skutečný počet položek menu a délka nejdelší z nich.

Vedle toho je nezbytné, aby instance třídy *menu* mohly pracovat s myší[7] přímo (před nakreslením nebo překreslením menu je třeba odstranit z obrazovky grafický kurzor a po ukončení operace ho musíme zase zobrazit.)

V C++ to vyřešíme tím, že ve třídě *menu* deklarujeme ukazatel na instanci třídy *mys* (nazveme ho třeba *krysa*). Stačí samozřejmě jeden, společný pro všechny instance třídy *menu* (*krysa* bude tedy statický atribut třídy *menu*).

V Pascalu ovšem nemůžeme používat statické atributy, takže použijeme globální proměnnou. Instanci třídy *tmys* pro práci s myší a ukazatel *krysa* definujeme v jednotce *mys* a inicializujeme je v inicializační části jednotky:

```
unit mys;

interface

type
  pmys = ^tmys;
  tmys = object
       {...}
       constructor init;
  end;

var
  krysa: pmys;
  hlodavec: tmys;

implementation
  {...}
begin
  {inicializace myši }
  hlodavec.init;
  krysa := @hlodavec;
```

[7] Podrobnější informace o práci s myší viz *Dodatek*.

```
end.
```

Z popisu fungování editoru plyne, že „odchycení“ kliknutí myší nebo stisknutí klávesy bude mít na starosti třída *edit*. Ta pošle přijatou zprávu menu – vlastně se menu zeptá, zda pro něj tato zpráva něco znamená.

To znamená, že menu musí mít metodu – nazvěme ji třeba *hodnota* – která bude mít jako vstupní parametr souřadnice kliknutí myší a která vrátí buď pořadové číslo zvolené položky nebo oznámí, že se ho tato zpráva netýká.

Podobně potřebujeme metodu pro vyhodnocení horkých kláves, tedy klávesových zkratek pro jednotlivé položky menu. I tato metoda bude vracet pořadové číslo zvolené položky nebo oznámení, že se ho tato zpráva netýká. Pojmenujeme ji tedy také *hodnota*.

Samozřejmě potřebujeme také metodu pro zobrazení menu (pojmenujeme ji výstižně *nakresli*), konstruktor a destruktor.

Spojení s menu myší zařídí v C++ statická metoda *nastavMys*. Hlavičkový soubor *menu.h* tedy může vypadat takto:

```
// Tyto direktivy zabrání, aby se hlavičkový
// soubor vložil víckrát do jednoho souboru
#ifndef _MENU_H_
  #define _MENU_H_

// počet položek menu
#define PPM 10

// max. počet znaků nabídky
#define MAX_ZN 20

#include "mys.h"

class menu {
  static mys *krysa;             // Odkaz na myš
  char text[PPM][MAX_ZN];        // Nabídky
  char hor_kl[PPM];              // Horké klávesy
  int pocet_pol, maxdel;    // Skutečný počet položek, maximální délka
  int Lhx, Lhy, Pdx, Pdy, od;    // Souřadnice levého horního a pravého
                                 // dolního rohu, odsazení od kraje
  int znaceni;  // Určuje, zda má menu označovat zvolenou položku

public:
  menu(char *soub, int ozn);
  int nakresli(int, int, int);
  void nakresli();
  int hodnota(int ch);
  int hodnota(int x, int y);
  static void nastavMys(mys* m){ krysa = m; }
};
#endif
```

Všimněte si, že jsme deklarovali dvě metody pro kreslení menu. První z nich má jako vstupní parametr souřadnice pravého horního rohu a požadované „odsazení“, tedy odstup menu od okraje kreslicí plochy. Tato metoda vypočte souřadnice *Lhx*, *Lhy*, *Pdx* a *Pdy*. Zavoláme ji pouze napoprvé. Pro případné překreslování budeme používat metodu bez parametrů, která použije uložené souřadnice.

Deklarace třídy *menu* (soubor *NABIDKY.PAS*) v Pascalu bude velmi podobná:

```
const VP = 8;                {velikost písma }
PPM = 10;                    { počet položek menu }
MAX_ZN = 20;                 {max pocet znaků nabídky }

type menu = object
   text: array [0..PPM] of string[MAX_ZN];   {texty nabídek }
   hor_kl: array [1..PPM] of char;     {znaky pro horké klávesy }
   pocet_pol, maxdel: integer;    {počet položek, max. délka textu
                                    v nabídce (skutečná)}
   Lhx, Lhy, Pdx, Pdy, od: integer;    {souřadnice a odsazení }
   znaceni: integer;   {označovat barevně vybranou položku? Kterou ?}
   constructor init(soub: string; ozn: integer);
   function nakresli1(phx: integer; phy:integer; ods:integer): integer;
   procedure nakresli2;
   function hodnota1(ch: char): integer;
   function hodnota2(x: integer; y: integer): integer;
end;
```

I zde potřebujeme dvě metody pro kreslení menu – jednu, která je nakreslí napoprvé a přitom vypočte jeho souřadnice na obrazovce a uloží je do instance, a druhou, která je pouze překreslí, pokud je poškodí např. objekt, nakreslený přes menu. Pascal ovšem nedovoluje přetěžování funkcí, a proto jsme je pojmenovali *nakresli1* a *nakresli2*. Podobně jsme deklarovali i metody *hodnota1* a *hodnota2*.

Konstruktor

Konstruktor třídy *menu* bude mít dva parametry: jméno souboru s texty nabídek a příznak určující, zda se má zvolená nabídka označovat.

Soubor s texty musí obsahovat název menu a jednotlivé položky, každou na zvláštním řádku, řekněme takto:

```
 DRUH
------------
Bod      (B)
Kruznice (K)
Usecka   (U)
```

Za položkou je v závorkách znak, označující horkou klávesu, která dané položce menu odpovídá. Tyto znaky uložíme do atributu *hor_kl* (pole znaků).

Ke čtení použijeme v C++ standardní funkci *fgets*, která přečte ze souboru zadaný počet znaků nebo celý řetězec až po konec řádku včetně[8]. Pokud tato funkce narazí na konec souboru, vrátí *NULL*.

Zde je deklarace konstruktoru třídy *menu* v C++:

```
menu::menu(char *soub, int ozn)
: pocet_pol(0), maxdel(0), znaceni(ozn)
{
   // Otevřeme soubor s texty
   FILE *F;
   F = fopen(soub, "r");
   if(!F) edit::chyba(soubor);
   int i = 0;
// Čteme položky menu ze souboru
   while(fgets(text[i], MAX_ZN-1, F)){
        pocet_pol++;
        // Odstraníme znak konce řádku
        for(int j=0; text[i][j]; j++)
            if(text[i][j] == '\n')text[i][j] = 0;
        // Zjistíme maximální délku
        int s = strlen(text[i]);
        if(s > maxdel) maxdel = s;
        i++;
   }
   // Určíme horké klávesy
   for(i = 0; i < PPM; i++) hor_kl[i] = 0;
   for(i = 0; i < pocet_pol; i++){
        // Najdi závorku a přečti znak za ní
        for(int j = 0; (text[i][j] != '(') && text[i][j]; j++)
        ;
        if(text[i][j+1]) hor_kl[i] = tolower(text[i][j+1]);
        else hor_kl[i] = 1;
   }
   close(F);
}
```

V Turbo Pascalu bude mít tento konstruktor tvar

```
constructor menu.init(soub: string; ozn: integer);
var
   F:       system.text;
   i,j,s:   integer;
   pom:     string;
begin
```

[8] Mohli bychom samozřejmě také použít objektového proudu *fstream* a jeho metody *istream get(char* buf, int len, char c = '\n')*, která přečte do pole *buf* nejvýše *len* znaků (pokud narazí na konec souboru nebo na znak *c*, skončí dříve. Tato metoda ovšem nepřečte znak konce řádku.)
O proudech budeme podrobně hovořit v kapitole 8. Zde použijeme tradiční prostředky jazyka C.

```
  pocet_pol:=0; maxdel:=0; znaceni:=ozn;
  {čtení textů menu ze souboru }
  assign(F, soub);
  {$I-}
  reset(F);
  {$I+}
  if(ioresult <> 0) then chyba_proc(soubor);
  i := 0;
  {čtení položek menu ze souboru }
  while not eof(F) do begin
       readln(f, pom);
       text[i] := pom;
       inc(pocet_pol);
       s := length(text[i]);
       if s > maxdel then maxdel := s;
       inc(i);
  end;
  {horké klávesy jsou v závorce za nabídkou }
  for i := 1 to PPM do hor_kl[i] := #0;
  for i := 1 to pocet_pol-1 do begin
       {najdi '('}
       s := 0;
       for j := 1 to length(text[i]) do
           if text[i][j] = '(' then s:= j;
       {pokud je, převeď následující znak na
        malé písmeno a ulož do pole hor_kl}
       if (text[i][s+1] <> #0) and (s <> 0) then
           hor_kl[i] := tolower(text[i][s+1])
       else
           hor_kl[i] := #1;
  end
end;
```

Třída mys (čti myš)

Třída *mys* bude obsahovat funkce pro práci s myší. Podrobnější informace o používání myši v dosovském programu najdete v *Dodatku*; zde si popíšeme pouze rozhraní této třídy.

Konstruktor třídy *mys* zjistí, zda je myš instalována a „resetuje“ ji (k tomu použije soukromé metody *Zjisti*). Kromě toho nastaví příznak, že je myš k dispozici.

Dále bude třída *mys* obsahovat metody *ZobrazKurzor()* a *SkryjKurzor()*, jejichž názvy není třeba vysvětlovat.

Vedle toho potřebujeme metody, které budou zjišťovat stav myši, přesněji kolikrát bylo stisknuto, resp. uvolněno levé tlačítko, a souřadnice kurzoru při této události.

Deklarace třídy *mys* může v C++ vypadat takto:

```
class mys{
       int jeMys;
       int Zjisti();
       enum {MysInt=0x33,};
  public:
       enum tlacitko {leve=1, prave=2, prostredni=3};
       mys();
```

```
        void ZobrazKurzor();
        void SkryjKurzor();
        int je() {return jeMys;}
        int PusteniLeveho(int &x, int &y);
};
```

Všimněte si deklarací výčtových typů. Zde nahrazují lokální konstanty; mohli bychom je použít např. i k definici mezí polí uvnitř tříd.

V Pascalu je deklarace podobná:

```
type
  pmys = ^tmys;            { ukazatel na myš }
  tmys = object
  jeMys: integer;          { indikace, zda je myš k dispozici }
  function Zjisti: integer;
  constructor init;
  function Stav(var x, y: integer): integer;
  procedure ZobrazKurzor;
  procedure SkryjKurzor;
  function PusteniLeveho(var x, y: integer): integer;
  function je: boolean;    { vrátí informaci, zda je myš k dispozici}
end;
```

Zvláštní klávesy

Stisknutí některých kláves může znamenat příkazy, které nemají ekvivalent v menu. Pokud tedy rozhraní přijme od klávesnice znak, který ani jedno menu nerozezná jako klávesovou zkratku některého ze svých příkazů, předáme jej funkci *ZvlastniKlavesa()*. Pokud jej ani ta nedokáže zpracovat, nemá daný znak v našem programu význam.

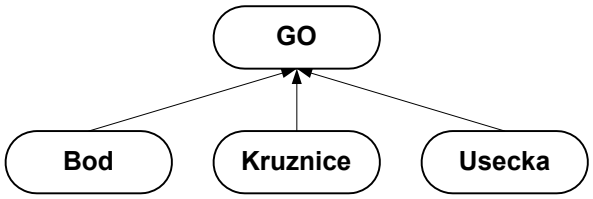
Kanál

Podívejme se nyní na zprávy, které budou proudit kanálem. Předávání souřadnic myši je jasné – k tomu postačí dvojice čísel typu **int**. Typ objektu popíšeme výčtovým typem *typ_objektu*, požadované akce popíšeme výčtovým typem *akce*. Tyto typy spolu s některými dalšími deklarujeme dalšími typy v souboru *enums.h*.

Grafické objekty

Zatím jsme hovořili o *bodu, úsečce a kružnici*. Z minulých kapitol již víme, že pro ně bude výhodné zavést společného předka, abstraktní třídu *GO* (tedy grafický objekt). Dostaneme tedy hierarchii, kterou vidíte na obr. 5.2.

Každý grafický objekt má nějaký význačný bod; jeho souřadnice budou atributy třídy *GO* (použijeme pro ně tradiční identifikátory *x* a *y* a budou typu *int*).



Obr. 5.2 Dědická hierarchie grafických objektů

Spolupráci s myší umožníme v C++ opět pomocí statického atributu *krysa* typu *mys**. V Pascalu jsme tento problém obešli již dříve pomocí globální proměnné.

Atributy třídy *GO* budou v C++ chráněné (**protected**), aby je mohly používat i metody potomků. V Pascalu je ponecháme veřejně přístupné, aby si tento program mohli přeložit i čtenáři, kteří mají k dispozici některý ze starších překladačů.

V jednotlivých třídách budeme potřebovat metody pro nakreslení a smazání objektů. Z požadavků, které jsme si na počátku položili, vyplývá, že nakreslit objekt znamená zobrazit jej v barvě bílé a smazat objekt znamená zobrazit jej v barvě černé (v barvě pozadí). Metody *nakresli* a *smaž* budou stejné ve všech třídách; to znamená, že je můžeme deklarovat již ve třídě *GO* a zdědit do ostatních.

Metoda *zobraz* se ovšem bude u jednotlivých tříd lišit a pro kořenovou třídu *GO* nemá vlastně smysl. Proto ji deklarujeme ve třídě *GO* jako čirou virtuální (v Pascalu jako virtuální, která zavolá chybovou proceduru) a v potomcích ji pak předefinujeme tak, aby zobrazovala daný objekt.

Podobně metody *zvetsi* a *otoc* (význam netřeba komentovat) budou virtuální, neboť každý z objektů bude na tyto příkazy reagovat po svém (a v *GO* musí být čiré virtuální).

Konstruktor má za úkol vytvořit příslušný objekt; destruktor jej před zrušením smaže (odstraní z obrazovky). Poznamenejme, že destruktor musí být také virtuální, neboť s objekty budeme zacházet jedině pomocí ukazatelů na *GO*.

Deklarace třídy *GO* vypadá v C++ takto:

```
// společný abstraktní předek - grafický objekt
class GO {
  protected:
       int x, y;            // střed objektu nebo jiný význačný bod
       static mys* krysa;
  public:
       static void nastavMys(mys *);
       GO(int, int);
       virtual ~GO();
       void nakresli();
       void smaz();
       virtual void zobraz(int barva) = 0;
       virtual void zvetsi(int) = 0;
       virtual void otoc(double) = 0;
       void posun(int, int);
};
```

Uvedeme i deklaraci v Pascalu (je na disketě v souboru *GRAFOBJ.PAS*):

```
{společný abstraktní předek – grafický objekt }
type
  GO = object
       x, y: integer;            {souřadnice středu }
       constructor init(xx, yy: integer);
       destructor done; virtual; {destruktor musí být virtuální !!! }
       procedure nakresli;
       procedure smaz;
       procedure zobraz(barva: integer); virtual;
       procedure zvetsi(n: integer); virtual;
       procedure otoc(s: real); virtual;
       procedure posun(dx, dy: integer);
  end;
```

Třída *bod* musí definovat vlastní konstruktor, destruktor a virtuální metody *zobraz*, *otoc* a *zvetsi*. Její deklarace v C++ je

```
class Bod: public GO {
  public:
       Bod(int xx, int yy);
       ~Bod();
       void zobraz(int bar);
       void zvetsi(int){}
       void otoc(double){};
};
```

a v Pascalu

```
type
  Bod = object(GO)
       constructor init(xx, yy: integer);
       destructor done; virtual;
       procedure zobraz(barva: integer); virtual;
       procedure zvetsi(n: integer); virtual;
       procedure otoc(s: real); virtual;
  end;
```

Třída *Kruznice* bude mít navíc atribut *r* (poloměr kružnice); musíme v ní definovat tytéž metody jako ve třídě *bod*. Její deklaraci najdete na disketě.

Úsečky budeme ve třídě *Usecka* popisovat pomocí počátku (uloženého ve zděděných složkách), směrového vektoru (dvojice *u, v* čísel typu **double** resp. *real*) a délky (atribut *del* typu **double**). Souřadnice počátku úsečky jsou tedy (x, y) a souřadnice konce *jsou* ($x+u*del$, $y+v*del$).

Deklarace třídy *Usecka* vypadá v C++ takto:

```
class Usecka: public GO {
       double u,v, del;
  public:
       Usecka(int xx, int yy, int l);
       ~Usecka();
       void zobraz(int b);
       void zvetsi(int d);
       void otoc(double fi);
```

```
};
```

Deklarace v Pascalu má tvar

```
type
  Usecka = object(GO)
       u,v, del: real;
       constructor init(xx, yy, l: integer);
       destructor done; virtual;
       procedure zobraz(barva: integer); virtual;
       procedure zvetsi(d: integer); virtual;
       procedure otoc(fi: real); virtual;
  end;
```

Naprogramování jednotlivých metod grafických objektů přenecháváme čtenářům (líněj-ší si je mohou najít na disketě).

6.5 Dokončení

Nyní máme hotové jednotlivé podstatné složky; zbývá vyjasnit, jak je bude využívat tří-da *edit*.

Zastavme se ještě u administrativních složek této třídy. Budeme potřebovat rozměry obrazovky (atributy *max_x* a *max_y*), příznak, že je spuštěn grafický režim, konstantu *odsazeni,* která bude určovat vzájemné rozestupy obou menu a položek v něm, a řetězce s chybovými hlášeními. Ty musí být k dispozici co nejdříve po spuštění programu, proto je zapouzdříme do třídy *zpravy* a statický atribut tohoto typu deklarujeme jako jeden z prvních hned v úvodu deklarace třídy *edit.*

Podívejme se, jak vypadá konečná verze deklarace třídy *edit* v C++:

```
class edit{
  friend menu;

  // administrativní složky
  int max_x, max_y; // rozměry obrazovky
  char grafika_bezi;
  const int odsazeni;
  static zpravy ChyboveHlasky;

  // složky rozhraní
  menu Typ;
  menu Akce;
  mys hlodavec;
  int c;                    // vstup z klávesnice

  // komunikační kanál
  int x,y;                  // souřadnice
  akce DruhAkce;            // co se má provést
  typ_objektu TypObjektu;   // jaký objekt vytvářet

  // složky výkonné části
  seznam S;                 // seznam graf. objektů
  GO* aktual;               // ukazatel na graf. objekt
```

```
  //metody rozhraní
  int zpracujZnak(int);
  void zpracujHodnotuAkce(int);
  void zpracujHodnotuTypu(int);
  int zpracujMys(int x, int y);
  int ZvlastniKlavesa(int c);

  // metody výkonné části
  void zpracujAkci(akce &Druh);
  void Vytvor();
  void obnov();

  // administrativní metody
  void inicializuj_gr();
  void prostredi();        // nakreslí prostředí editoru
 public:
  edit();
  ~edit();
  static void chyba(chyby);// tisk hlášení o chybě
  void beh();
};
```

a v Pascalu (je v souboru *GED.PAS*):

```
type edit = object
  {administrativní složky}
  max_x, max_y: integer;
  grafika_bezi: boolean;
  odsazeni: integer;

  { složky rozhraní }
  Typ, Akce: menu;
  c: char; {vstup z klávesnice }

  {komunikační kanál }
  x,y: integer;            {souřadnice }
  DruhAkce: akce;          {co se má provést }
  TypObjektu: typ_objektu; {jaký objekt vytvářet }

  {složky výkonné části }
  S: seznam;
  aktual: pgo;             {ukazatel na aktuální grafický objekt }

  {metody rozhraní }
  function zpracujZnak(ch: char):integer;
  procedure zpracujHodnotuAkce(n: integer);
  procedure zpracujHodnotuTypu(N: integer);
  function zpracujMys(xx, yy: integer): integer;
  function ZvlastniKlavesa(ch: char): integer;

  { metody výkonné části }
  procedure zpracujAkci;
  procedure Vytvor;
  procedure obnov;

  { administrativní metody }
  procedure inicializuj_gr;
  procedure prostredi;
  constructor init;
```

```
  destructor done;
  procedure beh;
end;
```

Konstruktor inicializuje grafický režim, zajistí spojení menu a grafických objektů s myší a nakreslí prostředí editoru. (Vedle toho samozřejmě zavolá konstruktory svých atributů, takže přečte texty chybových hlášení, zkonstruuje obě menu a vytvoří prázdný seznam grafických objektů. V C++ také inicializuje myš.)

Destruktor vyprázdní seznam grafických objektů (zničí je) a ukončí grafický režim. V pokročilejší verzi editoru by se také mohl starat o případné uložení práce do souboru v případě, že uživatel svoji práci od poslední změny neuložil.

Podívejme se na metodu *beh()*, která je vlastně výkonným jádrem programu. V úvodu nakreslí menu, pak v cyklu „odchytává“ zprávy od klávesnice a myši a zpracovává je. V C++ vypadá takto:

```
void edit::beh()
{
  // Nakresli menu
  int posun = Typ.nakresli(max_x-2*odsazeni,2*odsazeni, odsazeni);
  Akce.nakresli(max_x-2*odsazeni,posun + 2*odsazeni, odsazeni);
  // Je-li k dispozici mys, zobraz kurzor
  if(hlodavec.je()) hlodavec.ZobrazKurzor();
  // Ošetření zpráv menu a myši
  while(DruhAkce != konec) {
       // Vstup z klávesnice?
       if(kbhit()) {c = getch();
           zpracujZnak(c);
       }/* if kbhit */

       // od myši
       if(hlodavec.je()){
           if((hlodavec.PusteniLeveho(x,y))){
                int osetreno = zpracujMys(x,y);
                if(!osetreno) {
                     DruhAkce = vytvor_objekt;
                }
           }/* if stav*/
       }/*if hlodavec je*/

       // Zpracování zpráv od rozhraní
       zpracujAkci(DruhAkce);
  }/* while DruhAkce =! konec*/
}
```

Její podoba v Pascalu je

```
{ Řídicí procedura editoru: cyklus odchytává akce
  myši a klávesnice a reaguje na ně }
procedure edit.beh;
var
  posun, osetreno: integer;
  ch: char;
  pom: integer;       {pomocná proměnná, jen z důvodu syntaxe}
begin
```

```
  {Nakresli menu }
  posun := Typ.nakresli1(max_x-2*odsazeni,2*odsazeni, odsazeni);
  pom := Akce.nakresli1(max_x-2*odsazeni,posun + 2*odsazeni, odsazeni);
  { Je-li k dispozici mys, zobraz kurzor }
  if(hlodavec.je) then hlodavec.ZobrazKurzor;
  { Ošetření zpráv menu a myši }
  while DruhAkce <> konec do begin
       {od klávesnice }
       if keypressed then begin
            c := readkey;
            pom := zpracujZnak(c);
       end; {if keypressed }
       {od myši }
       if hlodavec.je then begin
            if hlodavec.PusteniLeveho(x,y) <> 0 then begin
                 osetreno := zpracujMys(x,y);
                 if osetreno = 0 then DruhAkce := vytvor_objekt;
            end; {if stav }
       end; {if hlodavec je }
       {zpracování zpráv od rozhraní }
       zpracujAkci;
  end; {while DruhAkce =! konec }
end;
```

Tato metoda předává případné vstupy z klávesnice funkci *zpracujZnak( )* a pokyny, zadané prostřednictvím myši, předává funkci *zpracujMys( )*. Pokyny od myši jsou dvojího druhu: jde-li o příkaz menu, nastaví se odpovídající příznak, jinak jde o příkaz k vytvoření objektu. To rozlišujeme pomocí proměnné *osetreno*.

Nakonec se volá metoda *zpracujAkci( )*, která zpracuje nastavený příznak akce. Jestliže jsme např. prostřednictvím menu přikázali smazat objekt, smaže aktuální objekt. Jestliže jsme pomocí menu změnili typ grafického objektu, nastaví menu příznak tohoto objektu a druh akce bude *nedelej_nic*. Další možné druhy akcí jsou *konec* (ukončení programu), *vytvor_objekt* (předepisuje vytvoření grafického objektu atd. – viz soubor *enums.h* resp. *ENUMS.PAS*).

Podívejme se nyní podrobněji na jednotlivé funkce, které se volají v metodě *beh*. Funkce *zpracujZnak( )* se dotáže obou menu, zda náhodou nejde o zprávu pro ně (volá metodu *hodnota1( )*). Pokud dostane kladnou odpověď, zavolá metodu *zpracujHodnotuAkce( )*, která nastaví příznak typu akce do atributu *DruhAkce*, nebo *zpracujHodnotuTypu( )*, která nastaví příznak typu (a příznak akce *nedelej_nic*). Pokud znak neošetří ani jedno menu, předá jej ještě metodě *zvlastniKlavesa( )*, která ošetřuje klávesy jako DEL, ESC, kurzorové klávesy, PGUP, PGDN apod.

Zdrojový text metody *zpracujZnak* je v C++

```
int edit::zpracujZnak(int c){
  int ak = Akce.hodnota(c);
  if(ak) {
       zpracujHodnotuAkce(ak);
       return 1;
  }/*if ak*/
  ak = Typ.hodnota(c);
  if(ak) {
```

```
        zpracujHodnotuTypu(ak);
        return 1;
   }/*if ak*/
   ak = ZvlastniKlavesa(c);
   return ak;
}
```

Ani v Pascalu není složitější:

```
{ zpracuje znak z klávesnice }
function edit.zpracujZnak(ch: char): integer;
var ak: integer;
begin
   ak := Akce.hodnota1(ch);       {je to klávesová zkratka menu Akce? }
   if ak <> 0 then begin
        zpracujHodnotuAkce(ak);   {pokud ano, zpracuj ji }
        zpracujZnak := 1;
   end else begin
        ak := Typ.hodnota1(ch);   {je to klávesová zkratka menu Typ? }
        if ak <> 0 then begin
            zpracujHodnotuTypu(ak);     {pokud ano, zpracuj ji }
            zpracujZnak := 1;
        end else begin
            ak := ZvlastniKlavesa(ch);  {je to jiná důležitá klávesa?}
            zpracujZnak := ak;
        end
   end
end;
```

Text metody *zpracujMys( )* je velice podobný, proto jej zde neuvádíme. Metoda *zpracujHodnotuAkce()* je jednoduchá. V C++ má tvar

```
void edit::zpracujHodnotuAkce(int n){
   switch(n) {
        case 1: DruhAkce = otoceni; break;
        case 2: DruhAkce = smazat; break;
        case 3: DruhAkce = konec; break;
   }
}
```

v Pascalu pak

```
{ nastav příznak akce, kterou je třeba provést a která
  byla zvolena v menu }
procedure edit.zpracujHodnotuAkce(n: integer);
begin
   case n of
        1: DruhAkce := otoceni;
        2: DruhAkce := smazat;
        3: DruhAkce := konec;
   end
end;
```

Podobné jsou i metody *zpracujHodnotuTypu()* a *ZvlastniKlavesa()*.

Metoda *zpracujAkci()* volá podle okolností výkonné metody grafických objektů, metodu *Vytvor()* nebo metodu pro smazání posledního prvku v seznamu grafických objektů. Nakonec nastaví příznak akce *nedelej_nic*, čímž říká, že poslední příkaz byl ošetřen.

Její zdrojový text nebudeme uvádět celý ani v C++, ani v Pascalu:

```
// Proveď akci, jejíž příznak byl nastaven
void edit::zpracujAkci(){
  switch (DruhAkce){
       case nedelej_nic:
       case konec:
           return;
       case vytvor_objekt:
           Vytvor();
           break;
       case zvetsit:
           aktual -> zvetsi(DELTA); obnov();
           break;
       // ... a další akce ...
       case otoceni:
           aktual -> otoc(FI); obnov();
           break;
       case smazat:
           aktual = S.smaz();
  }
  DruhAkce = nedelej_nic;             // ošetřeno
}
```

```
{ Proveď akci, jejíž příznak byl nastaven }
procedure edit.zpracujAkci;
begin
  if (aktual = nil) and (DruhAkce <> konec) and
       (DruhAkce <> vytvor_objekt)
  then DruhAkce := nedelej_nic;
  case DruhAkce of
       nedelej_nic,
       konec:
           exit;
       vytvor_objekt:
           Vytvor;
       zvetsit:
           begin
                aktual^.zvetsi(DELTA);
                obnov;
           end;
       {...a další akce ...}
       otoceni:
           begin
                aktual^.otoc(FI);
                obnov;
           end;
       smazat:
           aktual := S.smaz;
  end; {case }
{ Nakonec zakaž další akce až do chvíle, než se znovu nastaví příznak
  pro další akci }
```

```
  DruhAkce := nedelej_nic;
end;
```

Metoda *Vytvor()* má za úkol vytvořit grafický objekt, nakreslit jej a uložit jej do seznamu. Pro práci s tímto objektem slouží ukazatel *aktual* (aktualizuje se i při mazání objektu):

```
void edit::Vytvor(){
  hlodavec.SkryjKurzor();
  switch(TypObjektu){
       case bod:                        // Vytvoř objekt
           aktual = new Bod(x,y);
           break;
       case kruznice:
           aktual = new Kruznice(x,y,50);
           break;
       case usecka:
           aktual = new Usecka(x,y,25);
           break;
  }
  aktual->nakresli();                   // Nakresli ho
  DruhAkce = nedelej_nic;               // Ošetřeno
  hlodavec.ZobrazKurzor();
  S.vloz(aktual);                       // Ulož ho do seznamu
}
```

Její podoba v Pascalu se opět příliš neliší:

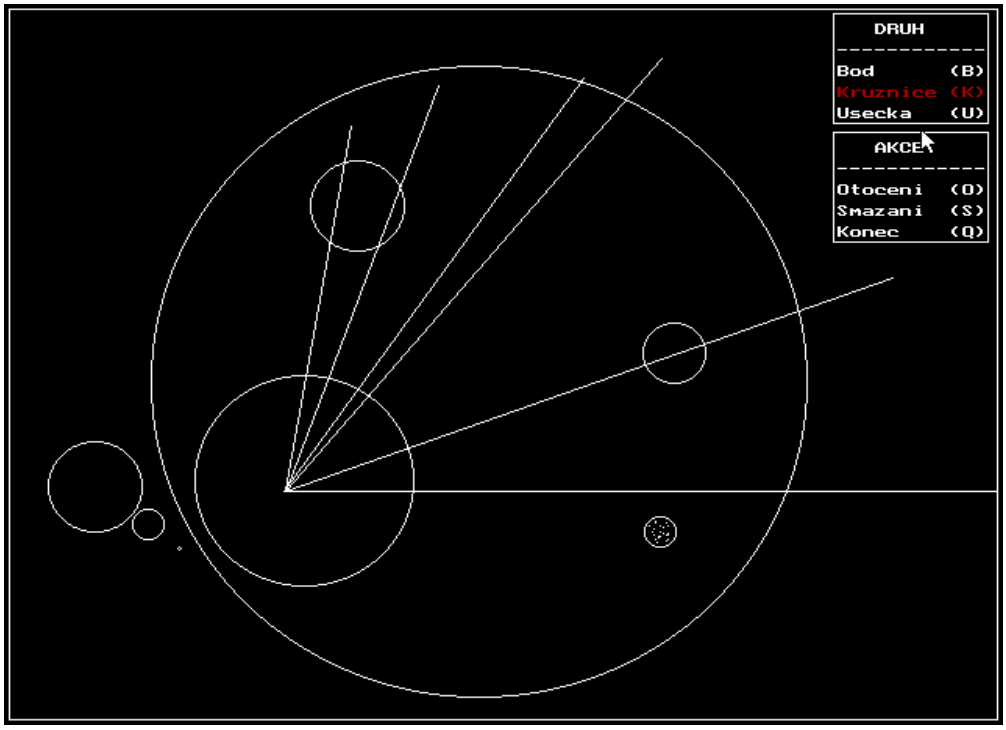
```
{ vytvoř objekt podle nastaveného příznaku }
procedure edit.Vytvor;
begin
  hlodavec.SkryjKurzor;
  case TypObjektu of
       bod_t:
           aktual := new(pbod, init(x,y));
       kruznice_t:
           aktual := new(pkruznice, init(x,y,50));
       usecka_t:
           aktual := new(pusecka, init(x,y,25));
  end;
  aktual^.nakresli;
  DruhAkce := nedelej_nic;
  hlodavec.ZobrazKurzor;
  S.vloz(aktual);
end;
```

6.6 A můžeme si kreslit...

Jestliže si tento program spustíte, měli byste umět bez problémů nakreslit něco podobného obrázku 5.3.

Ještě poznámka k ovládání programu: měl by fungovat s libovolným ovladačem myši (zkoušeli jsem ho s ovladači *Microsoft*, *Genius* a *Mtb*).

Kliknutím myši mimo menu vytvoříte „implicitní“ grafický objekt. Pomocí menu nebo stisknutím *O* (písmeno „o“) ho lze otáčet, PGUP resp. PDDN ho zvětšuje nebo zmenšuje, kurzorovými šipkami se posunuje. DEL nebo ESC smaže poslední nakreslený objekt.



Obr. 5.3 A můžeme si kreslit ...

Fantazii se meze nekladou

Zkuste si zavést nějaký další druh grafických objektů – např. čtverec. Budete muset:

- ✧ odvodit od třídy *GO* nového potomka, třídu s názvem řekněme *ctverec*, a definovat pro něj konstruktor, destruktor a metody *zobraz*, *posun* a *otoc*,
- ✧ připsat do souboru *Menu.typ* možnost *ctverec* (a v závorce uvést horkou klávesu),
- ✧ rozšířit definici výčtového typu *typ_objektu* o položku *ctverec*,
- ✧ c metodě *zpracujHodnotuTypu()* připsat řádek, který ošetří typ *ctverec*.

A to je vše.

Můžete se pokusit i o další rozšíření: zkuste přidat další akce do menu i do možností editoru. Zkuste přepsat ovládání pomocí myši tak, aby se např. úsečka kreslila kliknutím na počáteční a na koncový bod, nebo ještě lépe, aby se po stisknutí pravého tlačítka my-

ši nakreslil počáteční bod a po puštění tohoto tlačítka koncový bod. Fantazii se meze nekladou.

7. Vícenásobná dědičnost

Dalším tématem, kterému se nelze při výkladu OOP vyhnout, je vícenásobná dědičnost. Objekty v C++ mohou mít více než jednoho předka. Je samozřejmě otázka, zda je něco podobného potřeba; nikdo totiž dosud nevymyslel příklad, ve kterém by byla vícenásobná dědičnost nezbytná. Představuje ale v některých případech snadnou cestu k řešení problémů, které bychom museli jinak složitě obcházet – a v programování, stejně jako v mnoha dalších oborech, má rychle nalezené řešení často větší cenu než řešení dokonalé, ale opožděné. Jako příklad na vícenásobnou dědičnost použijeme mj. objektové datové proudy jazyka C++.

Celá tato kapitola se bude zabývat jazykem C++, neboť Turbo Pascal (ani Object Pascal implementovaný v Delphi) vícenásobnou dědičnost nenabízí.

7.1 Jak je to s vícenásobnou dědičností

V C++ může mít potomek větší počet přímých předků (na rozdíl od lidí více než dva). To s sebou nese řadu problémů, o kterých si musíme povědět.

Deklarace

Deklarace potomka s více předky se příliš neliší od deklarace potomka s jedním předkem. Specifikace jednotlivých potomků oddělujeme čárkami. Přístupová práva musíme specifikovat pro každého z předků zvlášť. Podívejme se na jednoduchý příklad:

```
class Potomek
: public Předek1, Předek2, protected Předek3
{ // ...
};
```

Takto definovaná třída *Potomek* má tři předky. Veřejně přístupného předka *Předek1*, soukromého předka *Předek2* (u něj jsme neuvedli přístupová práva, proto použil překladač implicitní specifikaci – v případě objektového typu **class** je to **private**), a chráněného předka *Předek3*.

Žádný z přímých předků se v deklaraci nesmí opakovat.

Význam

Jednoduchá dědičnost představuje, jak víme, vlastně specializaci. Potomek, odvozená třída, představuje podtřídu předka. Jak je to ale u vícenásobné dědičnosti?

Představme si, že máme dvě třídy: *vůz* a *vlek*. Od nich odvodíme společného potomka, třídu *souprava*:

```
class souprava: public vůz, public vlek {
// ...
};
```

Třída *souprava* zdědí jak vlastnosti *vozu* tak i *vleku*. Nelze ovšem s dobrým svědomím tvrdit, že by *souprava* byla zvláštním případem *vozu* nebo *vleku*. Dědění zde tedy vpodstatě nahrazuje skládání objektů.

Přesto jazyk C++ umožňuje i v případě vícenásobné dědičnosti, aby potomek zastupoval předka. To např. znamená, že proměnné typu *vůz* můžeme přiřadit hodnotu typu *souprava*. Při takovémto přiřazení se do proměnné typu *vůz* přenesou pouze datové složky ze zděděného podobjektu typu *vůz*.

Instance, konstruktory a destruktory

Podobně jako při jednoduché dědičnosti obsahuje instance potomka podobjekty všech předků, uvedených v deklaraci. V paměti jsou uloženy ve stejném pořadí jako v deklaraci. To znamená, že instance třídy *souprava* z předchozího odstavce se bude skládat z podobjektu typu *vůz*, za kterým bude v paměti uložen podobjekt typu *vlek*. Teprve za nimi budou v paměti uloženy datové složky, deklarované přímo ve třídě *souprava*.

Při konstrukci instance potomka se budou volat konstruktory předků v pořadí, v jakém jsou předkové uvedeni v deklaraci. Destruktory se budou při zániku instance volat v obráceném pořadí. Pokud měl předek nějaké virtuální metody, zdědí je potomek i při vícenásobné dědičnosti.

Přetypování ukazatelů

Už jsme si řekli, že i v případě vícenásobné dědičnosti může potomek vždy zastoupit předka. To platí i v případě ukazatelů: ukazateli na předka můžeme přiřadit adresu potomka. Přetypování ukazatele na potomka na ukazatel na veřejně přístupného předka je v C++ automatické. Přetypování zpět automatické není, musíme si je explicitně předepsat.

Při přetypování ukazatele na potomka na ukazatel na předka se změní adresa v přetypovávaném ukazateli tak, aby ukazovala na zděděný podobjekt.

Příklad

Následující prográmek nám předvede, jak je to s uložením zděděných podobjektů v paměti a s přetypováním ukazatelů při vícenásobné dědičnosti. Deklarujeme si třídy *A* a *B* (abychom nemuseli vypisovat specifikaci **public**, použijeme struktury):

```
/*   Příklad C6 – 1  */
struct A {
  int a;
  A() {cout << "konstruktor třídy A" << endl; }
  ~A(){ cout << "destruktor třídy A" << endl; }
  virtual void fa(){}
};

struct B {
  int b;
  B() {cout << "konstruktor třídy B" << endl; }
  ~B() {cout << "destruktor třídy B" << endl; }
  virtual void fb(){}
```

```
};
```

Obě tyto třídy obsahují vedle jedné datové složky typu **int** ještě konstruktor, destruktor a jednu virtuální metodu. Od nich pak odvodíme společného potomka, třídu *C*:

```
struct C: A, B {
  int c;
  C() {cout << "konstruktor třídy C" << endl; }
  ~C() {cout << "destruktor třídy C" << endl; }
  virtual void fc(){}
};
```

Dále deklarujeme ukazatele na tyto třídy:

```
C *uc;
A *ua;
B *ub;
```

V programu si nejprve vypíšeme velikost instancí tříd *A*, *B* a *C*:

```
cout << "velikost A: " << sizeof(A) << endl;
cout << "velikost B: " << sizeof(B) << endl;
cout << "velikost C: " << sizeof(C) << endl;
```

Dostaneme (v Borland C++ 3.1, v malém modelu pro DOS)

```
velikost A: 4
velikost B: 4
velikost C: 10
```

Velikost typu **int** je 2 B. Instance tříd *A* a *B* jsou dvojnásobné; to proto, že kromě jednoho atributu typu **int** obsahují také adresu tabulky virtuálních metod.

Instance třídy *C* zabírají 10 B (4 B zděděný podobjekt třídy *A*, 4 B zděděný podobjekt třídy *B*, a 2 B atribut *c* typu **int**). Všimněte si, že instance třídy *C* obsahuje pouze zděděné odkazy na tabulku virtuálních metod v podobjektech typu *A* a *B*. Pro potomka, třídu *C*, se nový odkaz nevytvořil, i když jsme v této třídě definovali novou virtuální metodu.

Dále vytvoříme instanci typu *C*:

```
C* uc = new C;
```

Protože konstruktory těchto tříd hlásí, co dělají, dostaneme

```
konstruktor třídy A
konstruktor třídy B
konstruktor třídy C
```

Všimněte si, že se nejprve provedly konstruktory obou předků, a to v pořadí, předepsaném deklarací, a teprve pak se provedlo tělo konstruktoru potomka.

Dále si deklarujeme ukazatele na typy *A* a *B*, přiřadíme jim adresu naší instance typu *C* a vypíšeme si je. Vypíšeme si také adresu složky *uc->c* naší instance:

```
cout << "adresa c: " << uc << endl;
A* ua = uc;
cout << "adresa zděděného podobjektu A: " << ua << endl;
B* ub = uc;
cout << "adresa zděděného podobjektu B: " << ub << endl;
cout << "adresa složky C::c " << &(uc->c) << endl;
```

Tak dostaneme výstup

```
adresa c: 0x0ff6
adresa zděděného podobjektu A: 0x0ff6
adresa zděděného podobjektu B: 0x0ffa
adresa složky C::c 0x0ffe
```

Nejprve jsme si vypsali adresu celé instance. Pak jsme ji přiřadili ukazateli na *A* a vypsali ji. Výstup je stejný jako v předešlém případě, neboť zděděný podobjekt typu *A* je uložen v instanci typu *C* jako první (začínají tedy na stejné adrese). Pokud si na svém počítači spustíte program C6–01.CPP (celý je na doplňkové disketě), dostanete možná jiné adresy. Na smyslu příkladu se tím ovšem nic nezmění.

Dále jsme přiřadili adresu instance typu *C* ukazateli na typ *B*. Při přetypování se daná adresa změnila tak, že nyní ukazuje na zděděný podobjekt typu *B* (tedy nikoli už na počátek instance typu *C*, ale kamsi dovnitř).

Poslední výpis ukazuje, že atribut *uc->c* leží v paměti až za zděděnými podobjekty.

Nakonec instanci třídy *C* zničíme:

```
delete uc;
```

Nato program vypíše

```
destruktor třídy C
destruktor třídy B
destruktor třídy A
```

neboť nejprve zavolá destruktor třídy *C*, a ten pak destruktory předků v pořadí opačném než v deklaraci.

Konflikty jmen

Při vícenásobné dědičnosti se může občas stát, že dva předkové obsahují složku (atribut nebo metodu) téhož jména. Obvykle se můžeme konstrukcím, podobným jako v následujícím příkladu, vyhnout. Ovšem ne vždy.

```
/*   Příklad C6 – 2   */
struct A {int a; };
struct B {int a; };
struct C: A, B {
   int c;
   void f();
};

void C::f(){
// Nejednoznačné
   cout << a;
```

```
}
```

Nyní obsahuje třída *C* dvě zděděné složky, které se obě jmenují velice vtipně *a*. To překladač jazyka C++ nepovažuje za chybu, dokud se nedostane do situace, kdy se neumí rozhodnout, kterou ze složek se stejným jménem použít.

Jednou takovou situací je použití *a* v metodě *C::f()*. Překladač neví, zda chceme vypsat složku *a*, zděděnou po *A*, nebo složku, zděděnou po *B*, a ohlásí chybu. Musíme mu pomoci kvalifikací jménem předka. Jestliže napíšeme

```
void C::f(){
// Tohle je v pořádku
   cout << A::a;
}
```

je vše jasné (nám i překladači).

Může se ale stát, že složku jménem *a* deklarujeme z jakýchkoli důvodů i v odvozené třídě *C*:

```
struct A {int a; }
struct B {int a; };
struct C: A, B {
int a;
   void f();
};
void C::f(){
// OK
   cout << a;
}
```

Tentokrát bude překladači pro změnu vše jasné a program v metodě *C::f()* vypíše složku *C::a*. Deklarace *C::a* totiž zastínila deklarace stejnojmenných složek ve všech předcích. To samozřejmě neznamená, že by zděděné složky byly nepřístupné; musíme je kvalifikovat jménem předka. (To ale známe už z výkladu o jednoduché dědičnosti.)

Trochu složitější situace nastane, jestliže budou mít třídy *A* a *B* společného předka, řekněme třídu *AA*, a společného potomka, třídu *C*:

```
/*   Příklad C6 – 3   */
struct AA{ int x; };
struct A: AA {int a; };
struct B: AA {int a; };
struct C: A, B {
int c;
   void f();
};
```

```
void C::f(){
// I toto je nejednoznačné
   cout << x;
}
```

Instance třídy *C* obsahuje dvě složky *x* zděděné po prapředkovi *AA*. Zde tedy kvalifikace jménem třídy *AA* nepomůže; musíme použít kvalifikaci jménem třídy, jejímž prostřednictvím jsme složku zdědili. Např. takto:

```
cout << A::x;
```

7.2 Problémy s vícenásobnou dědičností: datové proudy

V příští kapitole budeme hovořit o datových proudech jazyka C++. Jak již určitě víte, používá jazyk C++ pro vstup a výstup služeb knihovny objektových typů, jejichž deklarace najdeme v hlavičkových souborech *iostream.h*, *fstream.h*, *iomanip.h* a dalších.

Základem této knihovny je třída *ios* (zkráceno ze slov *input / output stream*, tedy vstupní / výstupní proud). Tato třída definuje vlastnosti, společné všem datovým proudům. Mezi jejími atributy najdeme např. ukazatel na sdruženou vyrovnávací paměť, slovo se stavovými příznaky (popisují aktuální stav proudu), příznaky pro formátování vstupů a výstupů atd.

Od třídy *ios* je odvozena řada dalších – specializovanějších – tříd. Mezi nimi najdeme i třídy *istream* a *ostream*, které definují vlastnosti vstupních, resp. výstupních datových proudů. Společným potomkem těchto dvou tříd je třída *iostream* pro proudy, které umožňují zároveň vstup a výstup dat. Třída *iostream* představuje pouhé spojení vlastností vstupních a výstupních proudů – prostě zdědí jejich atributy i metody (nic nepředefinuje, nic nepřidá).

Z pravidel pro dědění mezi objekty, která jsme dosud probrali, by ovšem plynulo, že instance třídy *iostream* bude obsahovat dva zděděné podobjekty třídy *ios*. To zatím nezní nijak strašně: prostě pouze trochu plýtváme pamětí. Jenže z toho také plyne, že instance třídy *iostream* bude obsahovat dvakrát formátovací příznaky a dvakrát stavové příznaky. Takže by se mohlo stát, že jeden formátovací příznak nastavíme, ale metoda, která by se jím měla řídit, se podívá na druhý a udělá něco jiného, než si přejeme. Taková představa je dost nepříjemná.

Zde nabízí jazyk C++ řešení v podobě tzv. virtuálního dědění. Tento mechanismus zajistí, že se vícekrát zděděné podobjekty sloučí. Deklarujeme-li třídu *ios* jako virtuálního předka tříd *istream* a *ostream*, bude jejich společný potomek, třída *iostream*, obsahovat pouze jeden podobjekt typu *ios*, tedy pouze jedny formátovací příznaky, pouze jedny stavové příznaky atd.

7.3 Virtuální dědění

Jestliže potřebujeme, aby se podobjekt třídy *A* v odvozených třídách vyskytoval vždy jen jednou, musíme třídu *A* deklarovat v jejích bezprostředních potomcích jako virtuálního předka. K tomu poslouží klíčové slovo **virtual**, které uvedeme ve specifikaci předka (podobně jako uvádíme specifikaci přístupových práv pro předka).

Podobně jako specifikace přístupových práv pro předka, i specifikace **virtual** se vztahuje pouze na předka, u kterého ji uvedeme. Pokud zároveň uvádíme obě specifikace, nezáleží na pořadí. V následujícím příkladu vezmeme podobnou hierarchii jako v příkladu C1 – 03, použijeme ale virtuální dědičnost:

```
/*   Příklad C6 – 4   */
// Základ hierarchie
class AA {
  double x;
 public:
  AA(): x(0){};
};

// Ve třídách A a B deklarujeme AA jako virtuálního, veřejně
// přístupného předka:
class A: public virtual AA {
  double a;
 public:
  A(){};
};

class B: public virtual AA {
  double b;
 public:
  B(){};
};

// Ve třídě D se podobjekty třídy AA
// sloučí, protože jsou virtuální
class D: public A, public B
 {double d;};

D d;
```

Instance *d* třídy *D* bude nyní obsahovat pouze jediný podobjekt třídy *A*. To mj. znamená, že zápis *d.a* je nyní (na rozdíl od příkladu C1 – 3) naprosto jednoznačný a překladač nevyžaduje dodatečnou kvalifikaci (nemusíme psát *d.B::a* nebo *d.C::a*).

Jak vypadá potomek, který má virtuální předky

V předchozích odstavcích této kapitoly jsme si ukázali, že při „obyčejném“, nevirtuálním dědění obsahuje instance potomka podobjekty, které jsou instancemi předků. V případě virtuálního dědění je struktura instance potomka složitější; jde v podstatě o jinou koncepci stavby instance.

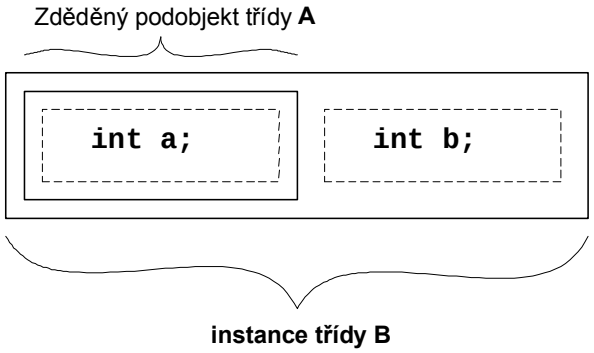
Ukážeme si, jak to je v Borland C++ 3.1. Použijeme-li jiný překladač (téže nebo jiné firmy), mohou se lišit detaily, základní princip by však měl zůstat stejný.

Virtuálně zděděný podobjekt musí být v instanci potomka obsažen pouze jednou. Proto uloží překladač na místo, kde by tento podobjekt byl při nevirtuálním dědění, pouze odkaz na místo (adresu) , kde je zděděný podobjekt doopravdy uložen.

Podívejme se nejprve na jednoduchý příklad:

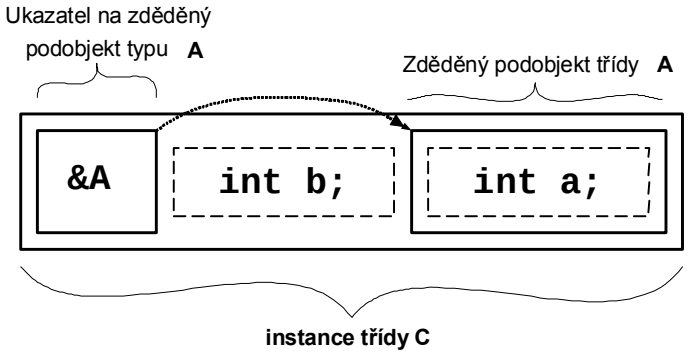
```
struct A {int a;};
struct B: A {int b;};
struct C: virtual A {int c;};
```

Třída *A* je obyčejným předkem třídy *B*, takže instance třídy *B* obsahuje zděděný podobjekt třídy *A* a za ním následují složky, deklarované v *B* (viz obr. 6.1).



Obr. 6.1 Struktura instance třídy B („obyčejná“ dědičnost)

Třída *C* je virtuálním potomkem *A*, takže obsahuje nejprve adresu místa, kde je skutečně uložen zděděný podobjekt *A*, pak atribut *c*, deklarovaný ve třídě *C*, a nakonec zděděný podobjekt *A* (viz obr. 6.2).

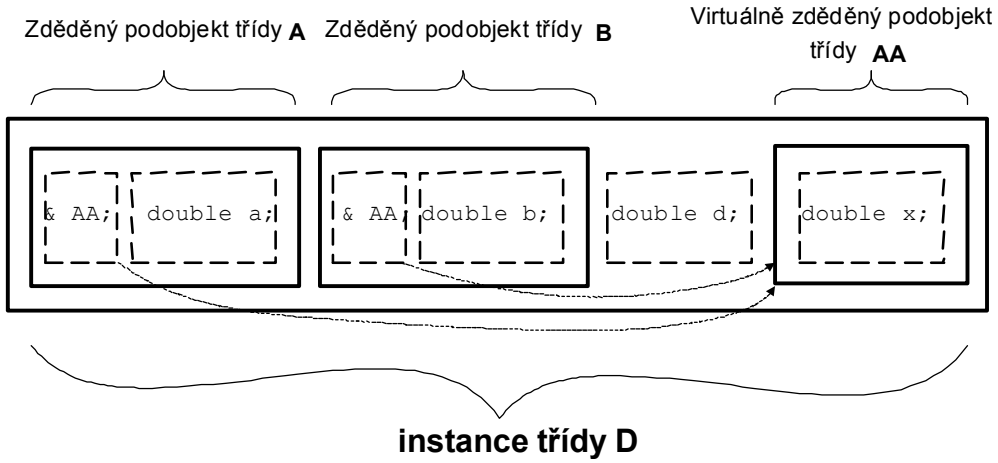


Obr. 6.2 Struktura instance třídy C (virtuální dědičnost)

Nyní se podíváme na instanci třídy *D* z příkladu C6 – 4 (obr. 6.3). Instance se skládá z podobjektu třídy *A* a z podobjektu třídy *B*, ve kterých je ale podobjekt třídy *AA* nahra-

zen odkazem na místo, kde se skutečně nachází. Dále pak obsahuje atributy deklarované ve třídě *D*, a konečně podobjekt třídy *AA*.

Změnám ve struktuře instance musí odpovídat i změny v chování konstruktorů a destruktorů – ale o tom si povíme o několik odstavců dále.



Obr. 6.3 Struktura instance třídy D z příkladu C6 – 4

Virtuální a nevirtuální předkové

Může se stát, že vzdáleného předka – například třídu *AA* – zdědíme několika způsoby, a to jak virtuálně tak i nevirtuálně. Jak bude pak vypadat instance potomka?

Všechny virtuálně zděděné podobjekty třídy *AA* se sloučí. Nevirtuálně děděné podobjekty se pochopitelně slučovat nebudou, takže výsledná třída bude obsahovat tolik podobjektů typu *AA*, kolikrát je tato třída nevirtuálním předkem, a jeden společný podobjekt *AA* za všechna virtuální dědictví dohromady.

V následujícím příkladu použijeme opět struktury, abychom se nemuseli starat o přístupová práva:

```
/*   Příklad C6 – 5   */
struct A {int a; A(){} };
struct B: virtual A {int b; B(){}};
struct C: virtual A {int c; C(){}};
struct D: A {int d; D(){}};
struct E: B, C, D {E(){} };

E ee;
```

Struktura *A* je virtuálním předkem *B* a *C* a nevirtuálním předkem *D*. Struktura *E* má nevirtuální předky *B*, *C* a *D*.

Celkem tedy *E* dědí *A* třikrát. Z toho dvakrát virtuálně (prostřednictvím *B* a *C*) a jednou „obyčejně“, nevirtuálně, prostřednictvím třídy *D*. To znamená, že instance třídy *E*

obsahuje dva podobjekty třídy *A*: jeden vznikne sloučením podobjektů zděděných virtuálně a druhý zděděný nevirtuálně prostřednictvím *D*.

Konstruktory a destruktory při virtuálním dědění

Pro třídy, které mají virtuální i nevirtuální předky, platí následující pravidlo: Nejprve se volají konstruktory virtuálních předků v pořadí, ve kterém jsou předkové zapsáni v deklaraci. Pak teprve se volají konstruktory předků nevirtuálních, opět v pořadí určeném deklarací.

To znamená, že např. pro třídu *H* deklarovanou zápisem

```
class H: A, virtual B, C, virtual D
 {/* ...*/};
```

se nejprve zavolá konstruktor předka *B*, pak *D*, (nejprve konstruktory virtuálních předků) a teprve pak se bude volat konstruktor *A* a nakonec *C*. (Jsou-li předkové také odvozené třídy, uplatní se toto pravidlo rekurzivně na předky předků atd.)

Jako složitější příklad vezmeme třídu *E* z příkladu C6 – 5. Při konstrukci instance této třídy se nejprve zavolá konstruktor virtuálního předka *A* a pak se zkonstruují podobjekty *B* a *C*. Pak se zavolá konstruktor nevirtuálního předka, třídy *D*, a ten si zavolá opět konstruktor svého nevirtuálního předka, třídy *A*.

Destruktory se vždy volají v obráceném pořadí než konstruktory.

Inicializace nepřímého virtuálního předka

Podívejme se na následující zdánlivě jasnou konstrukci:

```
/*   Příklad C6 – 6   */
struct A {
int a;
  A(int i):a(i){}
  A():a(0){}
};

struct B: virtual A {
  int b;
  B(int i):A(i), b(i){}
};

struct C: B{
  int c;
  C(int i):B(i), c(i){}
};

void main(){
  B bb(7);
  C cc(9);
  cout << cc.a;
}
```

Inicializace instance *bb* proběhne bez problémů a jak zděděný atribut *bb.a* tak i *bb.b* bude obsahovat hodnotu 7. Jestliže si ale necháme vypsat obsah instance *cc*, zjistíme, že zděděný atribut *cc.a* obsahuje nulu!

Krokováním zjistíme, že při inicializaci *cc* se volá implicitní konstruktor, a to i přes to, že v konstruktoru třídy *B* přikazujeme použít konstruktor *A(int)*.

Důvod je jednoduchý: Třída C může podobjekt *A* dědit virtuálně prostřednictvím několika předků a v každém z nich bychom mohli předepisovat jiný konstruktor nebo alespoň jiné parametry pro konstruktor třídy *A*.

Překladač proto použije při inicializaci virtuálního prapředka implicitní konstruktor (pokud bychom jej nedefinovali, ohlásí chybu).

Pokud nám takovýto postup nevyhovuje, máme možnost předepsat, který konstruktor třídy *A* použít, přímo v inicializační části konstruktoru třídy *C*. Bude-li mít deklarace třídy *C* tvar

```
struct C: B{
  int c;
  C(int i): A(i), B(i), c(i){}
};
```

zajistíme inicializaci virtuálně zděděného podobjektu tak, jak to potřebujeme. (V inicializační části nesmíme uvést konstruktor nepřímého *nevirtuálního* předka.)

Obecně platí pravidlo, že za inicializaci svých virtuálních předků zodpovídá „nejvíce odvozená“ třída (tedy třída, jejíž objekt deklarujeme), nikoli třídy podobjektů. V příkladu C6 – 6 se tedy o inicializaci virtuálních předků musí postarat třída *C*). Pokud v nejvíce odvozené třídě nespecifikujeme konstruktor pro virtuálně zděděné předky, použije se implicitní bez ohledu na to, co předepisují „méně odvozené“ třídy.

Nad virtuální dědičností

Virtuální dědičnost je nezbytným doplňkem vícenásobné dědičnosti. Vede ovšem k dosti komplikované struktuře instancí a ke složitějším pravidlům pro volání konstruktorů a destruktorů. Vedle toho může vést i k problémům při přetypování, při práci s ukazateli na složky apod.

Terminologická poznámka

Slovo *virtuální* se v češtině často používá ve významu *nekonečně malý* nebo *zdánlivý*. V souvislosti s virtuálními metodami má však spíše smysl význam *takový, který má schopnost něco konat* (slovník spisovné češtiny – viz [2] – uvádí tento význam dokonce na prvním místě.) Podobný význam má i anglický termín *virtual*.

Název *virtuální metoda* nebo *virtuální funkce* tedy vyjadřuje – zhruba řečeno – skutečnost, že virtuální funkce jsou ty, které mají schopnost správně reagovat.

Použití slova *virtuální* v souvislosti s vícenásobnou dědičností je poněkud problematičtější. B. Stroustrup, tvůrce C++, ve své knize o vzniku a vývoji tohoto jazyka (viz [3]) říká, že je zde použil na základě analogie: virtuální dědičnost je podobně jako virtuální metody skrytě řízena ukazateli. Nezasvěceným prý říká, že „virtuální znamená magický“.

8. Šablony

V této kapitole si budeme povídat o šablonách. Jde o konstrukci, která je součástí C++ podle specifikace Cfront 2.1 a pozdějších. Najdeme ji např. v Borland C++ počínaje verzí 3.0, ve Visual C++ a ve Watcom C++ 10.5. V průběhu standardizace jazyka zde (bohužel) došlo k několika změnám, takže některé konstrukce, které fungovaly v prvních překladačích, dnešní překladače odmítnou. Samozřejmě na ně upozorníme.

To znamená, že tato kapitola se bude týkat pouze jazyka C++, neboť v Pascalu nic podobného nenajdeme. Přesto si dovolíme doporučit tuto kapitolu alespoň k letmému přečtení i těm, kteří dávají přednost jazyku Pascal. Vždy se vyplatí vědět o možnostech, které nabízí konkurence.

Výklad v tomto dílu kursu se opírá o referenční popis jazyka C++ a o zkušenosti s implementací šablon v borlandských překladačích.

8.1 K čemu to?

Představme si, že potřebujeme v programu dvousměrný seznam, do kterého budeme jako „užitečná data“ ukládat celá čísla. Není nic jednoduššího: definujeme třídu *seznam* – s příklady jsme se setkali v prvním dílu.

Jenže o chvíli později budeme v témže programu potřebovat opět dvousměrný seznam, do kterého budeme pro změnu ukládat znakové řetězce. Někdy příště budeme potřebovat dvousměrný seznam, do kterého budeme ukládat objekty nějakého podivného typu *JakSeHonemJmenuje* … atd.

Je jasné, že všechny tyto seznamy se liší pouze typem ukládané hodnoty. Zápis třídy *seznam* i metod, které s instancemi *seznamu* pracují, bude patrně stejný bez ohledu na skutečný typ ukládané hodnoty.

Na podobný problém narazíme, budeme-li psát funkci – řekněme *Trideni()* – pro třídění pole, tedy funkci, která seřadí prvky pole podle velikosti. Napišme *funkci Trideni(int[], int n)*, která utřídí pole s *n* prvky typu **int** metodou přímého výběru[9]:

```
/*  Příklad C7 – 1  */
// Třídění úseku pole přímým výběrem
// a je tříděné pole, u je počáteční index, v je index prvku
// ZA posledním tříděným prvkem

void Trideni(int *a, int u, int v){
// dokud jsou nějaké neseřazené prvky
  for(int w = u; w < v; ++w) {
```

[9] Připomeňme si ideu tohoto algoritmu: Na počátku projdeme celé pole a najdeme v něm nejmenší prvek. Tento prvek prohodíme s prvkem na prvním místě. Pak najdeme nejmenší prvek ve zbývající části pole a ten prohodíme s prvkem na druhém místě. Nyní už máme na prvních dvou místech pole dva nejmenší prvky ve správném pořadí; zbývá tedy utřídit prvky s indexy 2, ..., *n*-1. Postup bude stejný: najdeme mezi nimi nejmenší prvek atd.

```
        int k = w;           // index nejmenšího prvku
        // Najdi nejmenší zbývající prvek
        for(int l= w+1; l < v ; l++) if(a[l] < a[k]) k = l;
        // Pokud je menší než prvek na řadě, vyměň je
        if(k != w) {
            int s = a[w];
            a[w] = a[k];
            a[k] = s;
        }
   }
}
```

Takovouto proceduru bychom si nejspíše chtěli uložit do knihovny. Přece jen třídění pole je činnost natolik obvyklá, že se to vyplatí. Problém ale je, že ne vždy budeme třídit pole typu **int**.

Uschovejme si tedy proceduru z příkladu C7 – 1. Pokud budeme potřebovat třídit pole jiného typu, stačí nahradit všechna klíčová slova **int** označením nového typu. To ovšem není nejlepší nápad: co když jedno **int** zapomeneme? Překladač nemusí takovou chybu odhalit a my se pak budeme jen tiše divit, co se to vlastně děje – představte si např., že bychom k záměně dvou hodnot typu **double** použili pomocnou proměnnou typu **int**. Nebo naopak, co když nahradíme **int**, které jsme nahradit neměli?

Zde se nabízejí dvě „klasická“ řešení:

✧ Všechny výskyty klíčového slova **int**, které označují typ tříděného pole, nahradíme nějakým obecným identifikátorem, např. *TYP*. Před překladem funkce *Trideni()* dáme tomuto identifikátoru význam pomocí deklarace **typedef** (komentáře k vlastnímu třídění již vynecháme):

```
/*    Příklad C7 – 2   */
// Definujeme význam identifikátoru TYP
typedef int TYP;

void Trideni(TYP * a, int u, int v){
  for(int w = u; w < v; ++w){
        int k = w;
        for(int l= w+1; l < v; ++l) if(a[l] < a[k]) k = l;
        if(k != w) {
            TYP s = a[w];
            a[w] = a[k];
            a[k] = s;
        }
   }
}
```

✧ Deklarujeme celou funkci jako makro, jehož parametrem bude typ tříděného pole:

```
/*    Příklad C7 — 3    */
#define TRIDENI(TYP) void Trideni(TYP * a, int u, int v){   \
     for(int w = u; w < v; ++w){                            \
          int k = w;                                        \
          for(int l= w+1; l < v; ++l) if(a[l] < a[k]) k = l;        \
          if(k != w) {                                      \
               TYP s = a[w];                                \
               a[w] = a[k];                                 \
               a[k] = s;                                    \
          }                                                 \
     }                                                      \
}

// Rozvineme makro v definici potřebné funkce
TRIDENI(double)
```

Připomeňme si, že obrácené lomítko na konci řádky znamená, že definice makra pokračuje i na následujícím řádku.

Budeme-li ve svém programu potřebovat několik homonym naší třídicí funkce, bude první možnost poněkud nešikovná.

Použijeme-li druhou možnost, tedy makro, můžeme sice snadno definovat řadu homonym pro různé typy tříděných polí, budeme ale mít problémy při jejich krokování. Makra totiž symbolické ladicí programy zpravidla „nevidí“, neboť je nezpracovává překladač, ale preprocesor.

V případě seznamů (a jiných kontejnerů) se nabízí možnost využít vlastností objektů – zejména skutečnosti, že potomek může vždy zastupovat předka. Mohli bychom definovat seznam, jehož prvky budou obsahovat ukazatele na společného předka všech typů, které do něj chceme ukládat. Taková třída by se mohla v duchu nejlepších tradic OOP jmenovat *Objekt* (nebo třeba *cObject, tObject,* … jak je ctěná libost).

To ale znamená, že pokud se rozhodneme do tohoto seznamu ukládat čísla typu **int**, musíme definovat třídu (nazvěme ji třeba **cInt**), která by byla potomkem třídy *Objekt* a do které naše celá čísla zapouzdříme. Budeme pro ni muset nejspíš přetížit řadu operátorů, předefinovat několik virtuálních funkcí atd.

Takovým způsobem jsou opravdu organizovány knihovny kontejnerů v Borland C++ 3.0 a starších.

Ovšem i toto řešení má řadu nevýhod. Například:

✧ Nebude nejefektivnější, neboť nás nutí používat přetížené operátory i tam, kde by jinak stačily standardní operátory pro standardní typy (např. porovnávání při třídění seznamu).

✧ Pro práci s prvky seznamu budeme muset používat virtuální funkce.

✧ Nebude příliš přehledné, neboť nejprve se musíme seznámit se standardními vlastnostmi typu *Objekt* a teprve pak je můžeme využívat.

Jazyk C++ ale nabízí ještě jednu možnost, a to použití šablon. Naprogramujeme funkci *Trideni( )* nebo typ *seznam* jen jednou, jako šablonu. Typ dat, tříděných v poli nebo ukládaných do seznamu, bude parametrem této šablony. Překladač pak v případě potřeby vytvoří podle této šablony funkci nebo objektový typ s požadovanými vlastnostmi (tedy založený na typu, který je parametrem dané šablony).

Šablony se v mnohém podobají makrům; zpracovává je ale překladač, nikoli preprocesor, a proto je mohou „vidět“ symbolické ladicí programy (jako Turbo Debugger, CodeView aj.).

8.2 Trocha teorie

Nejprve si řekneme několik slov k názvosloví. Anglický termín *template* se obvykle překládá jako **šablona** nebo **vzor**. V českých publikacích se lze setkat s oběma termíny. Teoretici hovoří o šablonách také jako o **generických** nebo o **parametrizovaných konstrukcích**.

Deklarace šablony v programu představuje abstraktní vzor, podle kterého je překladač schopen definovat celé skupiny funkcí nebo objektových typů. Funkce nebo typy vytvořené podle šablony označujeme jako **instance šablony**. V C++ můžeme deklarovat šablony řadových funkcí, šablony objektových typů a jejich metod.

Z hlediska objektově orientovaného programování se šablony tříd podobají metatřídám[10]. Je tu ale jeden podstatný rozdíl: zatímco instance metatříd v „čistě objektových“ jazycích mohou vznikat za běhu programu, instance šablon vytváří již překladač.

Deklarace šablony

Deklaraci šablony můžeme v programu v C++ zapsat pouze na úrovni souboru. (Je to jeden z mála druhů deklarací, která je v C++ takto diskriminována.) Má obecný tvar

template<*seznam_par*> *deklarace_šablony*

Za klíčovým slovem **template** následuje v lomených závorkách (tvořených znaky „menší než“ a „větší než“) seznam formálních parametrů šablony *seznam_par*. To jsou samozřejmě jednotlivé formální parametry oddělené čárkami. Šablony mohou mít jednak tzv. *typové parametry*, jednak *hodnotové parametry*.

Hodnotové parametry jsou parametry, s jakými se setkáváme u obyčejných funkcí. Mohou být skalárních typů (tj. **int**, **unsigned**, ukazatele, třídní ukazatele …). Nelze použít objektové typy nebo pole. Ve specifikaci formálních parametrů šablony nelze také uvést výpustku (…). U hodnotových parametrů můžeme předepsat implicitní hodnoty[11].

---

[10] Metatřída je třída, jejíž instance jsou opět třídy; s metatřídami se setkáme v „čistě objektových“ programovacích jazycích, jako je např. Smalltalk nebo Actor.

[11] V ANSI C++ (tj. např. v Borland C++ 5.0) můžeme předepsat implicitní hodnoty i u typových parametrů.

Typové parametry jsou uvedeny klíčovým slovem **class** (v ANSI C++, tj. v nejnovějších překladačích, také klíčovým slovem **typename**) a specifikují datové typy. Skutečným parametrem šablony, který odpovídá typovému formálnímu parametru, může být označení libovolného datového typu. Ve starších verzích C++ nelze pro typové parametry předepisovat implicitní hodnoty.

Deklarace_šablony znamená deklaraci řadové funkce, objektového typu (struktury, třídy nebo unie) nebo metody podle obvyklých pravidel. V této deklaraci mohou být některé konstanty nahrazeny hodnotovými parametry a některá označení typů typovými parametry.

8.3 Šablony řadových funkcí

Výklad o šablonách začneme u šablon řadových funkcí. (Připomeňme si, že jako „řadové“ označujeme funkce, které nejsou metodami objektových typů.)

Deklarace

Deklarace šablony řadové funkce se podobá „obyčejné“ deklaraci; má tvar

template<*seznam_par***>** *deklarace*

Šablona řadové funkce může mít – alespoň v současné verzi jazyka C++ – pouze typové parametry.

Deklarace znamená obvykle definiční deklaraci funkce, ve které mohou být některá označení typů nahrazena formálními parametry šablony. Přitom ve starších překladačích platí, že **všechny formální parametry šablony funkce musíme použít jako typy formálních parametrů šablonové funkce**. V překladačích, které respektují současný stav normy ANSI C++, je toto omezení odstraněno; k tomu se ještě vrátíme v odstavci *ANSI C++: explicitní kvalifikace*.

Jako příklad si deklarujeme šablonu funkce *Trideni():*

```
/*    Příklad C7 – 4   */
template<class TYP>
void Trideni(TYP * a, int u, int v){
  for(int w = u; w < v; ++w){
       int k = w;
       for(int l= w +1; l < v; ++l) if(a[l] < a[k]) k = l;    // *
       if(k != w) {
           TYP s = a[w];                                    // **
           a[w] = a[k];
           a[k] = s;
       }
  }
}
```

Šablona *Trideni* má jeden formální parametr *TYP*, který představuje typ. Zdůrazněme, že klíčové slovo **class** zde nemá nic společného s objektovými typy; skutečným parame-

trem této šablony může být jakýkoli datový typ, pro který je definován operátor „<“. (Tento operátor potřebujeme v řádku, označeném jednou hvězdičkou, v příkazu **if**.) Může to samozřejmě být i objektový typ, pro který jsme tento operátor přetížili.

V ANSI C++ můžeme klíčové slovo **class**, specifikující typový parametr šablony, nahradit novým klíčovým slovem **typename**.

Formální typ *TYP* můžeme v *deklaraci* používat podobně jako jiná označení typů. Např. v řádku, označeném dvěma hvězdičkami, jsme deklarovali pomocnou proměnnou typu *TYP*.

Poznámka:
Deklarace šablony řadové funkce může také obsahovat jen prototyp. Taková deklarace pak má tvar

```
template<class TYP> void Trideni(TYP *a, TYP *b);
```

Takováto šablona slouží ke generování odkazu na instanci. V podstatě tedy oznamuje překladači, že se v programu někde vyskytuje šablona Trideni *s danými parametry. Překladač ovšem na základě takovéto deklarace nemůže vytvořit instanci.*

Instance šablony řadové funkce

Instance šablony řadové funkce budou funkce se stejným identifikátorem, které se budou lišit typem parametrů. Můžeme je generovat dvěma způsoby – buď explicitně nebo implicitně.

Implicitní generování instancí

Začneme u implicitního generování instancí. Předpokládáme, že jsme v programu deklarovali výše uvedenou šablonu *Trideni*. Je-li *a* pole typu **int**, pak příkaz

```
Trideni(a, 0, 10);
```

způsobí, že překladač vytvoří automaticky instanci *void trideni(int *, int, int)*. Podobně deklarujeme-li v programu šablonu

```
template<class T> T max(T a, T b)
{ return a < b ? b : a;
}
```

resp. v ANSI C++

```
template<typename T> T max(T a, T b)
{ return a < b ? b : a;
}
```

a proměnné *x* a *y* typu **int**, způsobí zápis

```
int k = max(x,y);
```

že se proměnné *k* přiřadí hodnota většího z čísel, uložených v *x* a *y*. Bude-li ale *x* typu **int** a *z* typu *char*, způsobí zápis

```
r = max(x, z);        // Nelze
```

chybu, neboť takovou funkci podle naší šablony vytvořit nelze. Šablona *max* totiž předpokládá, že typy obou parametrů jsou shodné. **Šablona nenahrazuje prototyp funkce, takže nelze spoléhat na konverzi parametrů**. Při generování instance šablony se neprovádějí ani triviální konverze parametrů. To znamená, že pokud bychom deklarovali v programu konstantu

```
const int N = 1000;
```

a pokusili se vytvořit instanci šablony *Trideni* zápisem

```
Trideni(a, 0, N);
```

vynadal by nám překladač, že neumí najít funkci *Trideni(int*, int, const int)*, a my bychom si museli pomoci přetypováním

```
Trideni(a, 0, (int)N);
```

nebo explicitním vytvořením instance, jak si povíme dále.

Explicitní generování instancí

Zde se poněkud liší starší verze překladačů C++ od nejnovějších, založených na návrhu normy ANSI C++. Podívejme se nejprve na způsob, který se používal ve starších verzích (např. Borland C++ 3.x a 4.x).

Starší překladače

Pokud chceme v některém z těchto překladačů generovat instanci šablony řadové funkce explicitně, uvedeme **na úrovni souboru v oboru viditelnosti deklarace šablony** její prototyp. Je-li např. *vektor* objektový typ, pro který jsme definovali operátor „<“, způsobí prototyp

```
vektor max(vektor, vektor);
```

vytvoření instance šablony *max* s parametry typu *vektor,* která také vrací hodnotu typu *vektor*. Zde již překladač zná prototyp, takže může provádět i konverze skutečných parametrů.

Ovšem pozor: implicitní generování funkce podle šablony má přednost před konverzí parametrů podle prototypu. Podívejme se na trochu rozsáhlejší příklad:

```
/*    Příklad C7 – 5   */
// Šablona funkce max
template<class T>
T max(T a, T b)
{ /* ...*/ }

// Prototyp na úrovni souboru
double max(double, double);

int main(){
 int x = 11, y = 23, z;
 double d, dd = 11.1, ee = 22.1;
 d = max(dd, ee);                // **1**
 d = max(x, dd);                 // **2**
```

```
 z = max(x,y);                    // **3**
 return 0;
}
```

Zápis prototypu *double max(double, double)* způsobí, že se generuje odpovídající instance, a to i v případě, že ji v daném oboru viditelnosti nepoužijeme.

Ve funkci *main()* voláme funkci *max()* třikrát. V příkazu označeném **1** se použije instance *double max(double, double)*, generovaná podle šablony na základě prototypu.

Táž instance se použije i v příkazu **2**. Zde je jeden parametr typu **double** a druhý typu **int**, takže šablonu nelze použít. Proto překladač použije známý prototyp, konvertuje parametr *x* na typ **double** a zavolá již existující instanci *double max(double, double)*.

Příkaz označený **3** způsobí, že překladač generuje instanci *int max(int, int)*, neboť vytvoření instance, která přesně odpovídá šabloně, má přednost před konverzí parametrů na základě známého prototypu.

Poznamenejme, že kdybychom v tomto příkladu odstranili prototyp, způsobil by příkaz **2** chybu.

ANSI C++
Norma jazyka C++ zavádí pro explicitní generování instancí šablon syntax

template *prototyp*;

To znamená, že instanci šablony *Trideni* vytvoříme zápisem

```
template void Trideni(int*, int*);
```

a instanci šablony *max* zápisem

```
template double max(double, double);
```

tato deklarace nahrazuje prototyp, takže překladač může provádět konverze parametrů.

Instance, které nerespektují šablonu

Někdy se stane, že nám pro určitý typ parametrů instance vytvořená podle šablony nevyhovuje. Zůstaňme u šablony *max*, kterou jsme deklarovali o několik odstavců výše. Nic nám samozřejmě nebrání generovat instanci *char* max(char*, char*)*, která bude porovnávat dva znakové řetězce podle adresy. Je ale otázka, k čemu nám taková funkce bude; na druhé straně bychom nejspíš ocenili funkci, která by dokázala porovnat dva znakové řetězce podle abecedy. Přitom i pro funkci, která porovná dva znakové řetězce (určené ukazateli) a vrátí ten, který je lexikograficky větší, je docela logické jméno *max*.

V C++ nám naštěstí deklarace šablony nebrání, abychom definovali funkci se stejným jménem a s potřebným typem parametrů:

```
char *min(char *a, char *b)
{ // zde řetězce
  // porovnáme lexikograficky
}
```

Tato deklarace zastíní šablonu, takže se nemusíme obávat, že by překladač vytvořil podle šablony nesmyslnou funkci. (Přesněji řečeno: překladač bude takto deklarovanou funkci *char* max(char*, char*)* pokládat za instanci šablony *max*.)

Omezení v Borland C++

V deklaraci šablony řadové funkce nesmíme specifikovat paměťovou třídu **static**. Můžeme však použít modifikátory **extern** nebo **inline**. Borland C++ také zakazuje používat v těle šablonové funkce příkaz **asm**.

Bezpečnost práce

Nepozornost je, jak známo, zlým nepřítelem programátorů (a nejen jich) a jen málokdo může sebevědomě prohlásit, že je před ní naprosto bezpečný. My sami bychom se o sobě rozhodně neodvážili něco takového tvrdit.

Implicitní generování instancí šablony řadových funkcí nabízí krásný zdroj chyb z nepozornosti. Je-li šablona viditelná v celém programu, snadno se stane, že si vytvoříme a zavoláme instanci, o kterou nestojíme a která bude dělat něco trochu jiného než potřebujeme.

Jedna z cest, jak se podobným problémům vyhnout, spočívá v tom, že šablonu funkce umístíme do samostatného souboru. V témže souboru vytvoříme i potřebné instance.

V ostatních souborech programu tyto instance deklarujeme tím, že uvedeme prototypy, stejně jako u jakýchkoli jiných externích funkcí. (V takovém případě ale nemůžeme použít modifikátor **inline**.)

ANSI C++: explicitní kvalifikace

Starší překladače jazyka C++ požadují, abychom všechny parametry šablony řadové funkce použili jako typy jejích formálních parametrů. To umožňuje překladači stanovit skutečné parametry šablony podle typů skutečných parametrů generované instance.

Současná norma jazyka C++ od tohoto požadavku upustila. Pokud nemůže překladač při volání instance určit všechny parametry šablony na základě parametrů volané funkce, můžeme mu napovědět tzv. explicitní kvalifikací: za identifikátor šablonové funkce zapíšeme v lomených závorkách potřebné parametry, samozřejmě v pořadí, které odpovídá deklaraci.

Podívejme se na příklad. Deklarujeme šablonu funkce *prevod*:

```
/*   Příklad C7 — 6  */
template<class T, class V> T prevod(V v)
{
 return v;
}
```

Něco takového nám starší verze překladačů netolerovaly: parametr *T* této šablony nepředstavuje typ parametrů funkce *prevod*. Nicméně např. Borland C++ 5.0 tuto konstrukci přijme bez námitek.

Jestliže se ale pokusíme tuto šablonu použít a napíšeme

```
int i = 11;
char x = prevod (i);
```

bude zle. Překladač nemá z čeho určit typ vracené hodnoty (že ji přiřazujeme proměnné typu **char** ho nezajímá) a ohlásí chybu. Musíme mu pomoci explicitní kvalifikací:

```
char x = prevod<char, int> (i);
```

Poslední parametr můžeme vynechat, neboť překladač ho může určit podle typu parametrů volané funkce. Můžeme tedy také napsat

```
char x = prevod<char> (i);
```

Určení volané instance podle explicitní kvalifikace má přednost před určováním podle typu skutečných parametrů; jestliže napíšeme

```
int i = 11;
char x = prevod<long, double> (i);
```

opravdu se použije instance *long prevod<long, double> (double)*, i když je skutečný parametr typu **int**.

8.4 Šablony objektových typů a jejich metod

Také výklad o šablonách objektových typů začneme u deklarace.

Deklarace

Deklarace šablony objektového typu nebo metody má opět tvar

template<*sez_par***>** *deklarace*;

středník na konci je podle normy nezbytný.

Šablony objektových typů a jejich metod mohou ovšem mít – na rozdíl od šablon řadových funkcí – nejen typové, ale i hodnotové parametry i ve starších verzích překladačů.

Skutečné hodnotové parametry při použití šablony (při generování instance nebo při odkazu na existující instanci) musí být konstantní výrazy, tj. musí je umět vyhodnotit již překladač.

Deklarace představuje v tomto popisu deklaraci objektového typu nebo deklaraci metody. V této deklaraci můžeme k označení typu použít typové formální parametry a jako konstanty použít hodnotové formální parametry.

V *deklaraci* objektového typu můžeme zapsat i vložené (**inline**) metody a vložené spřátelené funkce. Ostatní metody musí mít své vlastní šablony.

V následujícím příkladu deklarujeme šablonu objektového typu pro práci s dynamicky alokovaným jednorozměrným polem proměnné délky. Jako parametry šablony zadáme počet prvků pole a jejich typ.

```
/*   Příklad C7 – 7   */
```

```
template<class T, int N=2>
class Pole {
  int delka;
  T* p;                    // Ukazatel na pole typu T délky N
 public:
  Pole(T r=0);             // Implicitní konstruktor
  Pole(Pole&);             // Kopírovací konstruktor
  ~Pole() {delete p; }     // Destruktor
  Pole& operator=(Pole&);
  T& operator[] (int i) {return p[i]; }
  Pole operator*(double);  // násobení zprava
 // Operátor násobení zleva jako řadová fce
  friend Pole<T,N> operator*(double d, Pole<T,N>& Q)
       {return Q*d; }
};
```

Tato šablona má typový parametr *T* a hodnotový parametr *N* (u něj jsme deklarovali implicitní hodnotu 2). Identifikátor *Pole* je jméno šablony. V deklaraci šablony objektového typu je lze používat bez parametrů, při ostatních použitích však musí být spojeno se skutečnými parametry.

Pro objektový typ, který obsahuje ukazatele na dynamicky alokované pole, jsme museli deklarovat kopírovací konstruktor, destruktor a přiřazovací operátor. Snadný přístup ke složkám tohoto pole nám umožní operátor indexování. Dále jsme zde deklarovali operátor „*“, který vynásobí všechny prvky pole daným číslem. Operátor násobení číslem zprava deklarujeme jako metodu (první operand je *Pole*), operátor násobení číslem zleva – v pořadí *číslo* * *pole* – deklarujeme jako spřátelenou funkci.

Operátor indexování a destruktor jsme zde deklarovali jako vložené metody, neboť jsme zapsali jejich definiční deklarace v těle šablonové třídy.

Pro ostatní metody musíme deklarovat zvláštní šablony. Podívejme se na šablonu implicitního a kopírovacího konstruktoru a operátoru násobení číslem (zprava):

```
/*   Příklad C7 – 7 */
// Implicitní konstruktor
// v případě neúspěšné alokace voláme funkci void Chyba()
template<class T, int N>
Pole<T,N>::Pole(T r)
:delka(N) {
  p = new T[delka]; // Inicializuje pole hodnotou r
  if(!p) Chyba();
  for(int i = 0; i < N; i++) p[i] = r;
}
// Kopírovací konstruktor
template<class T, int N>
Pole<T,N>::Pole(Pole<T,N>& P)
:delka(P.delka) {
  p = new T[delka];
  if(!p) Chyba();
  for(int i = 0; i < N; i++) p[i] = P.p[i];
}
// Násobení číslem zprava
```

```
template<class T, int N>
Pole<T,N> Pole<T,N>::operator*(double d)
{
  Pole<T,N> Q = *this;
  for(int i = 0; i < N; i++) Q.p[i] = Q.p[i]*d;
  return Q;
}
```

Zopakujme si, že mimo deklaraci šablony objektového typu musíme spolu se jménem šablony uvádět vždy parametry. Jedinou výjimkou je šablona konstruktoru, kde se parametry uvádějí pouze jednou, ve jménu typu vlevo od čtyřtečky.

Je asi jasné, že šablony metod, které jsou deklarovány samostatně, nejsou součástí deklarace šablony objektového typu. Méně zřejmé ale může být, že součástí deklarace šablony objektového typu není ani deklarace vložené spřátelené funkce – v našem případě druhého operátoru „*“. Nicméně je to tak, a proto jsme v deklaraci druhého operátoru „*“ použili zápis *Pole<T, N>*.

Instance šablony objektového typu

Vytvoříme-li instanci šablony objektového typ, vznikne objektový typ. Pod tím se skrývá nejen jméno a struktura třídy, ale také kódy metod a případně instance statických atributů.

Nejjednodušší způsob jak vytvořit instanci objektového typu je přímé použití v deklaraci. Např. zápis

```
Pole<int, 15> Ppp(99);
```

způsobí vytvoření typu jménem *Pole<int, 15>*. Současně samozřejmě vznikne instance *Ppp* nově vytvořeného typu. Obvykle je ale výhodnější vytvořit nejprve nový typ, tedy instanci šablony, pomocí deklarace **typedef**[12]:

```
typedef Pole<int, 15> intpole15;
```

Příkazem

```
typedef Pole<long> lpole2;
```

vytvoříme typ *Pole<long, 2>*, neboť překladač použije implicitní hodnotu parametru *N*.

ANSI C++

Norma ANSI C++ umožňuje – podobně jako u šablon řadových funkcí – předepsat explicitní generování instance zápisem

```
template class Pole<int, 2>;
```

Klíčové slovo **class** před identifikátorem šablony je nezbytné.

[12] To ale – aspoň v BC++ – funguje pouze při jistém nastavení přepínačů překladače.

Statické atributy

V šabloně objektového typu můžeme samozřejmě specifikovat také statické atributy. Každá instance takovéto šablony, tedy každý objektový typ, který podle této šablony vytvoříme, bude obsahovat své vlastní instance statických atributů.

V takovém případě ovšem potřebujeme pro každou instanci šablony také definiční deklaraci těchto statických atributů; ta může obsahovat i inicializaci. I zde si ovšem můžeme vypomoci šablonou, tentokrát šablonou statického atributu.

Podíváme se opět na jednoduchý příklad.

```
// Šablona třídy, která obsahuje statický atribut
template<class T>
class vektor {
  static T pocet;
  T p[10];
 public:
  vektor(){for(int i=0; i < 10; i++) p[i] = 0; pocet++;}
  // ...a další metody
};

// šablona statického atributu
template<class R> R vektor<R>::pocet = 0;

// Generujeme instance
vektor<int> t;
vektor <double> r;
```

Instance *vektor<int>* bude mít statický atribut *int vektor<int>::pocet*; instance *vektor<double>* bude mít statický atribut double *vektor<double>::pocet*. Oba budou inicializovány hodnotou 0. Instance statických atributů se zde vytvoří automaticky při vytváření instancí šablony vektor.

Šablonu statického atributu můžeme pochopitelně nahradit „obyčejnou“ definiční deklarací. To znamená, že bychom předchozí příklad mohli zapsat také takto:

```
template<class T>
class vektor {
  static T pocet;
  // atd....
};

vektor<int> t;
int vektor<int>::pocet = 0;

vektor <double> r;
double vektor<double>::pocet = 0;
```

ANSI C++

Norma ANSI C++ chápe inicializační hodnotu, uvedenou v šabloně statického atributu, jako implicitní hodnotu, kterou překladač použije, pokud mu někde dále nepřikážeme něco jiného. To znamená, že v ANSI C++ můžeme napsat

```
// Šablona statického atributu
template<class R> R vektor<R>::pocet = 0;
```

pak generovat instanci a ještě upravit počáteční hodnotu statického atributu:

```
// Generujeme instanci
vektor<int> t;
int vektor<int>::pocet = 22;
```

Takováto explicitní definice překryje implicitní hodnotu uvedenou v šabloně statického atributu.

Instance, které nerespektují šablonu

Pokud nám pro určité hodnoty parametrů nevyhovuje instance vytvořená podle šablony, můžeme ji deklarovat podle svých představ. Má-li překladač chápat nově deklarovaný typ jako instanci existující šablony, musí se jméno typu shodovat se jménem šablony a musí obsahovat skutečné parametry. Příklad:

```
class Pole<float, 100> {
  int p[100];
 public:
  int& operator[] (int i);
  Pole<float, 100>::Pole(int m);
};

Pole<float, 100>::Pole(int m = -1)
{
  for(int i = 0; i < 100; i++) p[i] = m;
}
```

Pro parametry **float** a 100 použije překladač instanci, kterou jsme mu zde předložili; pro ostatní hodnoty vytvoří instance podle šablony.

Podobně se můžeme rozhodnout, že pro parametry **unsigned** a 3 nám vyhovuje definice třídy i všech metod podle šablony *Pole* – až na jedinou výjimku, a to konstruktor. C++ nám dovoluje v takovém případě deklarovat samostatně konstruktor pro tyto hodnoty parametrů:

```
// Samostatná definice konstruktoru
Pole<unsigned, 3>::Pole(unsigned r)
{
  delka = 4;
  p = new unsigned[4];
  for(int i = 0; i < 4; i++) p[i] = r/2;
  p[3] = r;
}
```

Pro typ *Pole<unsigned, 3>* se všechny metody, kromě uvedeného konstruktoru, vytvoří podle šablony. Šablonový konstruktor bude nahrazen explicitně deklarovanou instancí. I tento konstruktor se ale pokládá za instanci šablony, takže pro něj platí mj. implicitní hodnota parametru $r = 0$, předepsaná v deklaraci šablony *Pole*. Proto jej můžeme použít i jako implicitní konstruktor a deklarovat např. instance

```
Pole<unsigned, 3> r,s,t;
```

Vložené spřátelené funkce

Některé operátory nemůžeme deklarovat jinak než jako spřátelené funkce. Typickým příkladem je funkce *operator*(double, Pole)*, který jsme deklarovali v šabloně *Pole* (jeho první operand není objektového ani výčtového typu).

Pokud se v deklaraci spřátelené funkce v šablonovém typu potřebujeme na tento šablonový typ odvolat, musíme jej použít i s parametry:

```
template<class T, int N=2>
class Pole {
// ...
  friend Pole<T,N> operator*(double d, Pole<T,N>& Q) {
       return Q*d;
  }
};
```

Typ *T*, konstantu *N* a také šablonový typ *Pole<T,N>* můžeme použít i v těle spřátelené funkce. Deklaraci operátoru „*“ v šabloně *Pole* bychom mohli upravit např. takto:

```
friend Pole<T,N>
operator*(double d, Pole<T,N>& Q) {
  Pole<T,N> t;
  t = Q*d;
  return t;
}
```

8.5 Šablony v rozsáhlých programech

Smysluplné programy obvykle nebývají malé; zpravidla jsou uloženy v mnoha souborech. Je jasné, že šablony můžeme použít v několika různých souborech, které se navíc ani nemusí překládat zároveň. Pak se může snadno stát, že vytvoříme několikrát tutéž instanci. Něco takového ale sestavovací programy obvykle pokládají za chybu. Co s tím?

Jazyk C++ nabízí následující řešení: oddělíme od sebe deklarace a definice šablon. (Jako deklaraci budeme označovat šablonu řadové funkce, která obsahuje pouze prototyp, nebo deklaraci šablony objektového typu bez šablon metod a statických atributů.

V celém programu zpřístupníme pomocí hlavičkových souborů pouze deklarace; definice šablon uvedeme v jednom souboru, ve kterém generujeme zároveň i potřebné instance.

Šablony v borlandských překladačích

Borlandské překladače nabízejí další možnosti, založené na přepínačích překladače **-Jg**, **-Jgx** a **-Jgd**. Ty lze zadat v příkazovém řádku samostatného překladače nebo kdekoli v programu pomocí direktivy **#pragma option**. Tyto přepínače ovlivňují funkci překladače i sestavovacího programu.

V programovém prostředí BC++ 3.1 lze tyto přepínače nahradit volbami v dialogovém okně *Options* | *Compiler* | *C++ Options* | *Template Generation*, v BC++ 4.0 a pozdějších *Options* | *Project* | *C++ Options* | *Templates.*

Implicitní přístup představuje přepínač **-Jg**, kterému odpovídá v prostředí volba *Smart*. Tento přepínač způsobí, že se v jednotlivých modulech budou generovat instance podle okamžité potřeby a linker pak sloučí ty, které se opakují.

Tento přístup je nepochybně nejpohodlnější, vzdáváme se při něm však kontroly nad vytvářením instancí.

Pokud chceme řídit vytváření instancí, použijeme přepínačů **-Jgx** (kterému odpovídá v IDE volba *External*) a **-Jgd** (kterému odpovídá *Global*).

Přepínač **-Jgx** přikazuje překladači, aby vytvářel pouze odkazy na instance, které budou vytvořeny někde jinde. Překladač tedy nebude vytvářet instance šablony.

Přepínač **-Jgd** přikazuje překladači, aby vytvářel podle potřeby veřejně přístupné instance (tj. instance, které budou dostupné i v jiných modulech). Pokud se nám ovšem přitom stane, že vznikne několik shodných instancí, bude to linker pokládat za chybu.

Jak tedy postupovat, jestliže si chceme v BC++ řídit vznik instancí šablon?

- ✧ Šablonu objektového typu – bez šablon (nevložených) metod a bez šablon statických atributů – umístíme do hlavičkového souboru.
- ✧ Šablony metod a statických atributů dáme do samostatného souboru .CPP, který bude součástí projektu a který přeložíme s volbou *Global* nebo s přepínačem **-Jgd**. Do tohoto souboru vložíme direktivou **#include** hlavičkový soubor se šablonou objektového typu. V tomto souboru také vytvoříme potřebné instance.
- ✧ Do ostatních souborů vložíme hlavičkový soubor se šablonou objektového typu a přeložíme je s přepínačem **-Jgx** nebo s volbou *External*.

Všechno lze zkazit

Podívejme se na několik drobných zrad, se kterými se můžeme v souvislosti se šablonami setkat.

Skutečným parametrem šablony může být také typ, vytvořený podle šablony. Ovšem uzavírací závorku „>“ nesmíme zapsat dvakrát vedle sebe. Zápis

```
// Nelze
typedef Pole<Pole<long, 22>> lPole22;
```

způsobí chybu, neboť překladač pochopí „>>“ jako operátor bitového posunu (pamatujme na céčkovskou lexikální konvenci). Mezi znaky „>“ musíme vložit čárku nebo alespoň jednu mezeru, např. takto:

```
// To lze
typedef Pole<Pole<long, 22>,> lPole22;
typedef Pole<Pole<short, 31> > sPole31;
```

Skutečným parametrem šablony může být i výraz, pokud jej dokáže překladač vyhodnotit již v době kompilace. Jestliže tento výraz obsahuje operátor >, musíme jej uzavřít do závorek:

```
Pole< double, (M>N+32) > Ppp;
```

Pokud na to zapomeneme, bude překladač tento operátor pokládat za závorku, která ukončuje parametry šablony.

Také vložené spřátelené funkce mohou způsobit problémy. Nejsnáze si to ukážeme na příkladu:

```
template<class R> class NoACo {
R r;
// ...a další složky
  friend int Fun() {return 45;}
};
```

Na první pohled se zdá být vše v pořádku; bohužel jen na první pohled a při generování první instance šablony *NoACo*. Jakmile se totiž pokusíme generovat druhou instanci, ohlásí překladač chybu – tělo funkce *Fun* již bylo definováno.

Problém je v tom, že typ žádného z parametrů funkce *Fun* nezávisí na šabloně *NoACo* ani na jejích parametrech. Takže když napíšeme např.

```
NoACo<long> noacol;
NoACo<float> noacof;
```

bude se překladač snažit vytvořit dvě funkce se stejným jménem a stejnými typy parametrů. A to pochopitelně nejde.

Na podobný problém bychom narazili i v případě, že by se spřátelené funkce v různých instancích lišily pouze typem vracené hodnoty, např.

```
template<class S> class NoACo2 {
  S s;
  // ...a další složky
  friend S Fun() {/* ...*/}
};
```

Řešení je obvykle jednoduché: v těle šablony ponecháme pouze prototyp spřátelené funkce a definiční deklaraci zapíšeme mimo.

8.6 Šablony v knihovnách

Už z úvodních úvah je jasné, že šablony představují mimořádně vhodný nástroj pro vytváření knihoven. Nelze se proto divit, že např. funkce *max()* nebo *min()*, které počítají maximum nebo minimum ze dvou čísel, jsou v knihovnách deklarovány jako šablony. Také kontejnerové třídy (třídy, které představují zásobníky, fronty, seznamy, stromy, rozšiřitelná pole a jiné struktury pro ukládání dat) se vyplatí implementovat jako šablony.

Řadu příkladů šablon najdete v borlandské knihovně kontejnerů[13]. Hlavičkové soubory *arrays.h, stacks.h*, *queues.h* atd. s jejich definicemi najdete v podadresáři *\CLASSLIB\INCLUDE* v domovském adresáři *BC++ 3.1* a pozdějších.

S jiným příkladem použití šablon se setkáme v kapitole o datových proudech, až si budeme povídat o implementaci parametrických manipulátorů na datových proudech v Borland C++ 4.x.

8.7 Příklad

Na závěr této kapitoly se pustíme do většího příkladu, na kterém si předvedeme, že šablony umožňují psát velmi univerzální programy. Přepíšeme algoritmus třídění přímým výběrem tak, aby jej bylo možno použít i pro třídění seznamů a jiných kontejnerů, pokud mají vhodně definované iterátory. Jednosměrný seznam, jeho iterátor a proceduru pro třídění definujeme jako šablony. Zdrojový text všech příkladů z této podkapitoly najdete na doplňkové disketě v souboru *C7–8.CPP*.

Třídění

Nejprve se tedy podíváme na třídění přímým výběrem. V příkladu C7 – 4 jsme předávali víceméně zbytečně pole, index počátečního prvku tříděného úseku a index prvního prvku **za** tříděným úsekem. Ve skutečnosti by stačilo předávat pouze ukazatel na první prvek tříděného úseku a ukazatel na první prvek za tímto úsekem. Tyto ukazatele již v sobě nesou informaci o poli, se kterým pracujeme.

Přepíšeme proto šablonu *Trideni* následujícím způsobem:

```
/*    Příklad C7 – 8   */
template<class TYP>
void Trideni(TYP u, TYP v){
  for(TYP w = u; w != v; ++w){
       TYP k = w;
       TYP l = w;
       for(++l; l != v; ++l) if(*l < *k) k = l;
       if(k != w) Prohod(*k, *w);                          // **
```

[13] Kontejnerové třídy jsou počínaje BC++ 4.0 implementovány pouze pomocí šablon. Také standardní knihovna jazyka C++, předepsaná normou ANSI, obsahuje šablonové implementace kontejnerů, jejich iterátorů a řady základních algoritmů.

```
   }
};
```

K prohození obsahu dvou prvků jsme v řádku označeném dvěma hvězdičkami použili funkci *Prohod()*. Zatím to zdůvodníme třeba tak, že tím zpřehledníme zápis; brzy však uvidíme, že nám to pomůže vyřešit jeden problém.

Prohlédněme si nyní tento algoritmus pozorněji. Snadno zjistíme, že na parametry *u* a *v* se můžeme dívat jako na dvojici iterátorů, které ukazují do nějakého kontejneru; *u* určuje první prvek tříděného úseku, *v* první prvek **za** tříděným úsekem.

Přitom vlastně vůbec nezáleží na tom, o jaký kontejner jde; jedinou podmínkou je, aby umožňoval seřazení prvků za sebou (může to být pole, seznam apod.) a aby byly pro náš iterátor, který s tímto kontejnerem pracuje, definovány alespoň tyto operace:

= – přiřazení,

!= – zjištění, zda dva iterátory ukazují na různé prvky kontejneru,

++ – přesun iterátoru na následující prvek kontejneru,

* – zpřístupnění dat v prvku kontejneru.

Také na typu uložených dat vlastně nezáleží, pokud jsou pro ně definovány operace „=“ a „<“.

Pokud ale použijeme jakýkoli kontejner, dostaneme se při prohazování prvků v třídicí proceduře do potíží. Formální parametr *TYP* šablony *Trideni* totiž nemá s typem uložených dat téměř nic společného, takže v příkazu, označeném dvěma hvězdičkami, vlastně nemůžeme definovat pomocnou proměnnou, kterou k prohození potřebujeme (viz např. pomocnou proměnnou *s* v příkladu C7 – 4).

Zde nám mohou opět pomoci šablony. My sice v tomto místě programu neznáme typ prohazovaných hodnot, ale víme, že zápis

```
w*
```

pro libovolný iterátor *w* představuje hodnotu právě toho typu, o který nám jde. Jestliže tedy deklarujeme šablonu

```
template<class TYP>
void Prohod(TYP& a, TYP& b){
   TYP c = a;
   a = b; b = c;
}
```

způsobí příkaz

```
Prohod(*k, *w);
```

vytvoření instance, která naše proměnné prohodí. Překladač bude v okamžiku překladu tohoto řádku již znát typ, který představuje zápis ***k* resp. ***w*.

Prvek seznamu

Podívejme se nyní, jak by mohla vypadat šablona jednosměrného seznamu, na kterém bychom náš třídicí algoritmus vyzkoušeli. Nejprve deklarujeme šablonu prvku seznamu.

```
// Předběžné deklarace seznamu a iterátoru – použijeme je
// v deklaraci přátel
template<class Typ> class seznam;
template<class Typ> class SezIter;

// Šablona prvku
template<class Typ>
class prvek {
   Typ d;
   prvek *nasl;
 public:
   prvek(Typ data):d(data), nasl(0) {}
   prvek():d(0), nasl(0) {}
   ~prvek(){}
   friend seznam<Typ>;
   friend SezIter<Typ>;
};
```

Prvek seznamu obsahuje pouze užitečná data (atribut *d*) a ukazatel na následující prvek. Konstruktory tyto složky inicializují. Další metody, jak uvidíme, nepotřebujeme.

Seznam

Šablona seznamu bude také jednoduchá. Použijeme seznam se zarážkou, abychom mohli snadno používat iterátory, které ukazují za poslední platný prvek seznamu. Třída *seznam<Typ>* bude obsahovat ukazatel na hlavu seznamu a na zarážku.

Dále deklarujeme konstruktor, který vytvoří prázdný seznam (bude obsahovat jen zarážku), destruktor, který zruší všechny prvky seznamu včetně zarážky, a metodu *vlož*, která vloží nový prvek za počátek seznamu. Další metody si můžete zkusit přidat sami.

Třída *seznam<Typ>* deklaruje jako přítele *SezIter<Typ>*, tedy třídu iterátorů, kterou definujeme dále.

Šablona seznamu bude mít tvar

```
template<class Typ>
class seznam {
   prvek<Typ>* hlava, *zarazka;
 public:
   seznam();
   ~seznam();
   void vloz(Typ);
   friend SezIter<Typ>;
};
```

Pokud jde o metody, ukážeme si alespoň šablonu konstruktoru:

```
// Konstruktor vytvoří prázdný seznam se zarážkou
template<class Typ>
seznam<Typ>::seznam()
```

```
: hlava(new prvek<Typ>)
{
  zarazka = hlava;
 // pokud není paměť ...
  if(!hlava) Chyba(1);
}
```

Ostatní metody najdete na doplňkové disketě. Najdete na ní také implementaci funkce *Chyba()*, která se volá v případě, že se nepodaří alokovat paměť pro prvek seznamu.

Iterátor

Podívejme se nyní na šablony třídy *SezIter<Typ>*, která definuje iterátory na seznamech, vytvořených podle šablony *seznam*.

Iterátor bude obsahovat referenci na instanci seznamu, se kterou pracuje, a ukazatel na aktuální prvek seznamu. Dále musí obsahovat několik konstruktorů, a to:

- ✧ konstruktor, který nastaví vytvořený iterátor na daný prvek seznamu,
- ✧ konstruktor, který nastaví vytvořený iterátor na i-tý prvek seznamu od počátku nebo na zarážku,
- ✧ kopírovací konstruktor (ten je nezbytný, neboť překladač neumí vytvořit implicitní kopírovací konstruktor pro třídy, které obsahují reference).

V úvodu této podkapitoly jsme si také řekli, že pro iterátor musí být k disposici

- ✧ operátor „++“, který jej posune na následující prvek v seznamu,
- ✧ operátor „*“, který zpřístupní hodnotu uloženou v prvku, na nějž iterátor ukazuje,
- ✧ přiřazovací operátor „=“ (ten je opět nezbytný, neboť třída *SezIter<Typ>* obsahuje referenční složku, a v takovém případě překladač odmítne vytvořit implicitní přiřazovací operátor),
- ✧ operátor „!=“, který potřebujeme při třídění.

Šablona iterátoru může mít tvar

```
template<class Typ>
class SezIter {
seznam<Typ> &TenSeznam;
  prvek<Typ>* Ten;
 public:
  SezIter(prvek<Typ>*, seznam<Typ>&);
  SezIter(int, seznam<Typ>&);
  SezIter(SezIter<Typ>&);
  Typ& operator*();
  SezIter<Typ>& operator++();
  int operator!=(SezIter& sit)
  {
       return Ten != sit.Ten;
  }
  SezIter<Typ> operator+(int i);
  SezIter<Typ>& operator=(SezIter<Typ>&);
};
```

Operátor „=!“ jsme definovali jako vloženou funkci. Tento operátor porovná adresy prvků seznamu, na které ukazují oba iterátory (levý a pravý operand).

Konstruktor, který nastaví iterátor na i-tý prvek seznamu (pokud v něm takový prvek existuje), má tvar

```
template<class Typ>
SezIter<Typ>::SezIter(int i, seznam<Typ> &s)
: TenSeznam(s)
{
  if (i == -1) Ten = TenSeznam.zarazka;
  else {
       Ten = TenSeznam.hlava;
       for(int j = 0; j < i; j++) ++(*this);
  }
}
```

Má-li první parametr hodnotu -1, bude iterátor ukazovat na zarážku, tedy **za** poslední platný prvek seznamu. Jinak nastavíme iterátor na první prvek (hlavu) a *i*-krát „popojdeme o krok“, tedy přesuneme iterátor na následující prvek.

Parametr *s*, který určuje seznam, s nímž definovaný iterátor pracuje, musíme předávat odkazem. Jinak by se totiž vytvořila uvnitř konstruktoru lokální kopie seznamu a iterátor by se nastavil na ni. Ovšem tato kopie po ukončení těla konstruktoru zanikne, takže vytvořený iterátor by nepracoval se žádným seznamem, takže by vlastně nebyl k ničemu. Navíc jsme ve třídě *seznam<typ>* nedefinovali kopírovací konstruktor, takže by si překladač vytvořil svůj vlastní, který by mohl napáchat další škody.

O posun iterátoru na následující prvek seznamu se stará operátor „++“. Jeho definice je jednoduchá: pokud to jde, najde následující prvek a jeho adresu uloží do atributu *Ten*.

```
template<class Typ>
SezIter<Typ>& SezIter<Typ>::operator++()
{
  if(Ten->nasl) Ten = Ten->nasl;
  return *this;
};
```

Nesmíme také zapomenout na přiřazovací operátor. Budeme ho často potřebovat a přitom překladač si ho nedokáže sám vytvořit, neboť třída *SezIter<Typ>* obsahuje referenci.

Přiřazování má ovšem smysl, pouze pokud oba iterátory pracují s týmž seznamem. Tuto podmínku otestujeme porovnáním adres seznamů (reference na ně je uložena v atributu *TenSeznam*). Pokud je přiřazení možné, přenese se adresa prvku seznamu uložená v atributu *Ten*. Jinak se zavolá funkce *Chyba()*. Šablona tohoto operátoru má tvar

```
template<class Typ>
SezIter<Typ>& SezIter<Typ>::operator=(SezIter<Typ>& si){
  if(&TenSeznam != &si.TenSeznam) Chyba(2);
  Ten = si.Ten;
 return *this;
}
```

Poslední z operátorů, o kterém si povíme, je operátor „*“. Ten má za úkol – podobně jako u ukazatelů – zpřístupnit data, na která iterátor ukazuje. V našem případě tedy bude ukazovat na atribut *d* v prvku seznamu. Abychom jej mohli používat ke všem běžným operacím s daty, tedy abychom jej mohli zapisovat i na levé straně přiřazovacího příkazu, musí vracet referenci na *Typ* (typ, který je parametrem šablony a jehož hodnoty ukládáme do seznamu). Jeho šablona je jednoduchá:

```
template<class Typ>
Typ& SezIter<Typ>::operator*()
{
  return Ten->d;
}
```

Šablony dalších metod najdete na doplňkové disketě.

Nyní můžeme vyzkoušet, jak naše šablony fungují. Napíšeme jednoduchý testovací program, ve kterém vytvoříme seznam celých čísel, uložíme do něj čísla 1000, 999, ... , 1, a tento seznam setřídíme pomocí funkce *Trideni(SezIter<int>, SezIter<int>)* generované na základě šablony *Trideni*. Pak vypíšeme (pomocí iterátoru) první 4 prvky seznamu.

Abychom si ukázali, že šablonu *Trideni* můžeme použít i ke třídění polí, deklarujeme pole typu **double** o 1000 prvcích, uložíme do něj čísla 1000.5, 999.5, ..., 1.5, a toto pole setřídíme pomocí funkce *Trideni(double*, double*)*, generované opět pomocí šablony *Trideni*. Pak vypíšeme (zase pomocí iterátoru, což je tentokrát ukazatel) první 4 prvky pole.

```
int main(){
  seznam<int> S;                          // Vytvoříme seznam S
  for(int i = 0; i < N; i++) S.vloz(i+1);  // Uložíme do něj čísla.
// Iterátory ukazující na počátek (hlavu) S a ZA konec (na zarážku) S
  SezIter<int> sZac(0,S), sKon(-1,S);
  Trideni(sZac, sKon); // Nyní seznam setřídíme
// A nyní pomocí iterátoru vypíšeme první 4 prvky seznamu
  for(SezIter<int> q1(0, S), q2(4,S); q1 != q2; ++q1)
       cout << *q1 << endl;
  double a[N];            // Pole čísel typu double naplníme čísly
  for(i = 0; i < N; i++) a[i] = N-i+0.5;
  Trideni(&a[0], &a[N]);                  // a setřídíme
// Vypíšeme první 4 prvky pole
  for(double * r1(&a[0]), *r2(&a[4]); r1 < r2; ++r1)
  cout << *r1 << endl;
  return 0;
}
```

Poznamenejme, že podobným způsobem je implementována řada základních algoritmů ve standardní šablonové knihovně jazyka C++, která je součástí ANSI C++.

9. Datové proudy v jazyce C++

Tradičním příkladem na použití vícenásobné dědičnosti jsou objektové datové proudy jazyka C++. Jejich implementace je ovšem založena i na rafinovaném použití přetěžování operátorů.

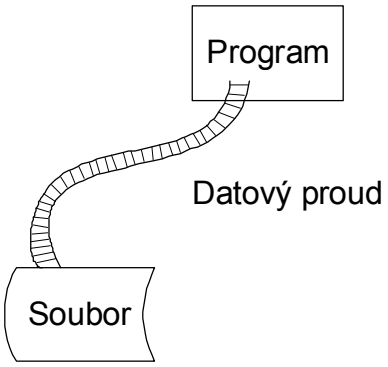
Náš výklad se bude opírat o implementaci datových proudů v Borland C++ 3.1 a 4.5. V jiných překladačích můžeme narazit na drobné odlišnosti, princip by ale měl být stejný, a v době, kdy budete toto povídání číst, už možná zakotvený v normě ANSI.

Vedle samotných datových proudů si povíme také o tzv. manipulátorech a aplikátorech. Část výkladu v této kapitole se bude skládat z poněkud nezáživných výčtů tříd, atributů, metod atd. Zkuste to vydržet, s tím se bohužel nedá nic dělat.

9.1 Soubory a proudy

Pokud jste začínali programovat v některém jiném jazyku, možná vás zarazilo, že se v Céčku a v C++ nehovoří o práci se soubory, ale o datových proudech.

Jazyk C, a po něm i C++, totiž vychází z představy, že data proudí ze *zdroje* do *spotřebiče*. Zdrojem dat může být program, spotřebičem soubor, tiskárna apod., nebo naopak, zdrojem dat může být soubor a spotřebičem program. Mezi zdrojem a spotřebičem je tedy *proud* dat.



*Obr. 8.**1** Datový proud, spojující soubor a program*

O soubory, které jsou na jednom konci této „komunikační linky“, se stará operační systém. Také datové struktury v programu, které jsou na druhé straně, jsou věc poměrně jasná. Program se ale musí postarat o zabezpečení datového proudu mezi daty a souborem (obr. 8.1). Tento proud můžeme zpravidla připojit k různým souborům a k různým datům a můžeme jej také využívat pro přenos dat oběma směry – jak z programu do souboru tak i naopak.

Jazyk C používá k práci s datovými proudy především standardní datovou strukturu *FILE* a funkce, které s ní pracují (*printf*, *scanf* apod. – najdeme je ve standardní knihovně *stdio.h*).

Jazyk C++ všechny tyto prostředky zdědil a my je můžeme v plné míře využívat. Vedle toho ale přináší C++ prostředky nové, založené na objektových prostředcích jazyka a na přetěžování operátorů a funkcí.

My jsme objektové datové proudy využívali již v předchozích dílech. Standardní proudy *cin*, *cout* atd. a vstupní a výstupní operátory << a >> pro nás nejsou žádnou novinkou. Setkali jsme se i s některými manipulátory; např. manipulátor *endl* způsobuje ve výstupních proudech přechod na nový řádek. Další manipulátory mohou měnit formátování (my jsme dosud používali formátování implicitní) nebo měnit stav proudů.

Objektová koncepce knihovny datových proudů a využití přetížených operátorů přináší řadu výhod. Uveďme namátkou:

- ✧ Můžeme si definovat vlastní verzi vstupního, resp. výstupního operátoru >>, resp. << pro naše vlastní objektové typy.
- ✧ Můžeme si definovat vlastní datové proudy (např. proud na tiskárnu).
- ✧ Můžeme definovat vlastní manipulátory.

9.2 Základní informace

Začneme trochou teorie. Datové proudy jazyka C++ jsou založeny na dvou hierarchiích objektových typů (obr. 8.2). Jednodušší z nich je odvozena od třídy *streambuf* a obsahuje objekty, které tvoří vyrovnávací paměti (*buffer*) pro datové proudy. Vytváří vlastně rozhraní mezi pamětí a fyzickými zařízeními a obsahuje základní metody pro práci s datovými proudy.

Potomci třídy *streambuf* obsahují metody specifické pro proudy orientované na soubory, do řetězců a na konzolu.

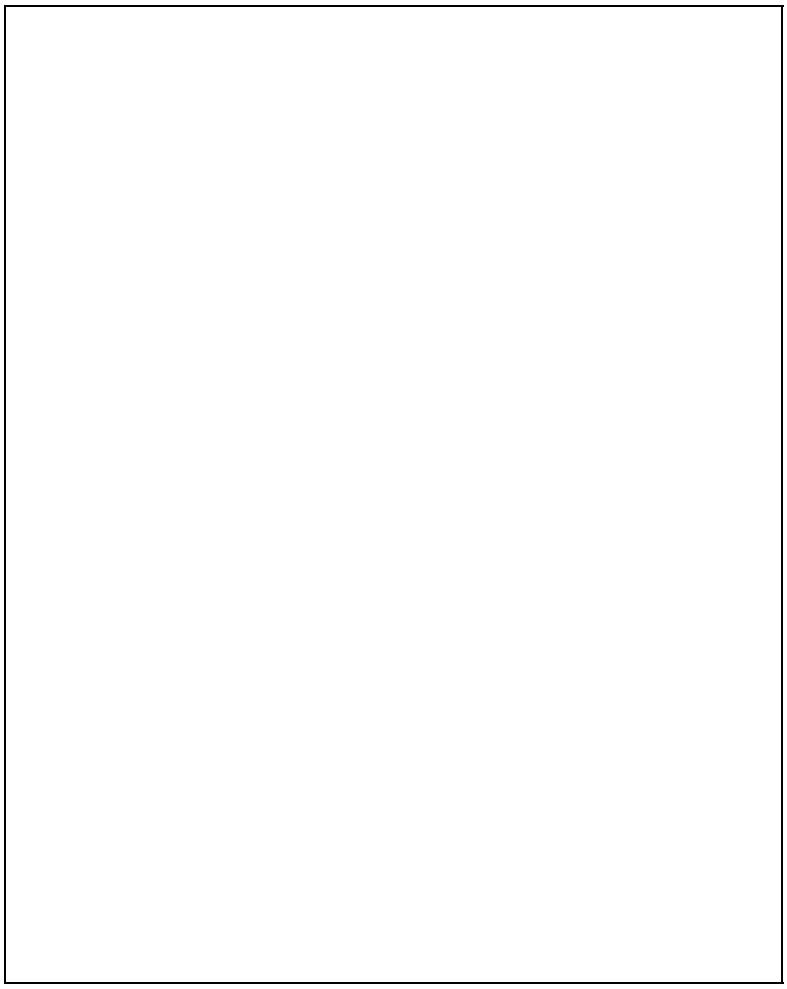
Programátor obvykle o těchto třídách nepotřebuje vědět nic více, než že existují. Pracují s nimi metody třídy *ios* a jejích potomků, které programátor opravdu využívá. Hierarchie, odvozená od třídy *ios*, je podstatně rozvětvenější.

Hlavičkové soubory

Základní prostředky pro datové proudy jazyka C++ najdeme v souboru iostream.h. Tyto prostředky umožňují vstup ze souboru *stdin* a výstup do *stdout*.

Pokud chceme používat manipulátory s parametry (co to je, to si povíme dále), musíme použít hlavičkový soubor *iomanip.h*.

Pro práci s proudy orientovanými na ostatní soubory potřebujeme hlavičkový soubor *fstream.h*.



Obr. 8.2 Hierarchie tříd tvořících datové proudy v Borland C++ 4.5

Paměťové proudy, tedy datové proudy, které umožňují vstup ze znakových řetězců a výstup do řetězců, jsou popsány v souboru *strstrea.h.* (Tyto proudy jsou obdobou „klasických" funkcí *sprintf* a *sscanf*.)

Proud pro výstup na konzolu (tedy na obrazovku PC) a manipulátory na tomto proudu jsou v Borland C++ definovány v souboru *constrea.h*. (Tento proud obsahuje obdobu funkcí z borlandské knihovny *conio.h*.)

Všechny zmíněné soubory se odvolávají na *iostream.h* (vkládají jej direktivou **#include**). To znamená, že pokud do svého programu vložíme kterýkoli z dalších souborů, máme k dispozici i prostředky popsané v *iostream.h*.

Třída ios

Objekty třídy *ios* se v programech nepoužívají; přesto jí věnujeme zdaleka nejvíce místa, neboť definuje vlastnosti, které jsou společné všem datovým proudům. Je virtuálním předkem dalších tříd a je definována v *iostream.h*. Povězme si o některých jejích atributech a metodách.

Všechny následující atributy jsou chráněné (protected); jsou tedy přístupné v odvozených třídách, ale nikde jinde.

Třída *ios* obsahuje ukazatel *bp* na sdružený objekt typu *streambuf*, tedy na sdružený buffer. Tento atribut využívají především metody potomků.

Dále zde najdeme atribut *state* typu **int**, který obsahuje příznaky možných chybových stavů proudu (tedy vlastně příznaky toho, zda se poslední vstupní nebo výstupní operace s tímto proudem podařila nebo k jaké chybě došlo). Tyto příznaky popisuje veřejně přístupný výčtový typ *ios:: io_state*.

```
enum io_iostate {
goodbit = 0,     // vše je OK
 eofbit = 1,     // nalezen konec souboru
 failbit = 2,    // poslední operace se nezdařila
 badbit = 4,     // pokus o neplatnou operaci
 hardbit = 8     // hardwarová chyba
};
```

Atribut *x_flags* je typu **long**. Obsahuje formátovací příznaky. Ty jsou popsány pomocí nepojmenovaného veřejně přístupného výčtového typu deklarovaného taktéž ve třídě *ios*:

```
enum {
skipws = 1,           // při vstupu budou přeskočeny bílé znaky
 left = 2,            // výstup bude zarovnán vlevo
 right = 4,           // výstup bude zarovnán vpravo
 internal = 8,        // první znak výstupního pole je znaménko
                      // nebo označení číselné soustavy (např. 0x)
 dec = 16,            // desítková soustava
 oct = 32,            // osmičková soustava
 hex = 64,            // šestnáctková soustava
 showbase = 128,      // při výstupu se indikuje číselná soustava
                      // podle pravidel C/C++ (např. 0x11)
 showpoint = 256,     // zobrazuje se desetinná tečka
 uppercase = 512,     // Při výstupu v šestnáctkové soustavě
                      // se použijí velká písmena
 showpos = 1024,      // kladná čísla vystupují se znaménkem "+"
 scientific = 2048,   // výstup v semilogaritnickém tvaru
 fixed = 4096,        // racionální čísla ve tvaru s pevnou
                      // řádovou čárkou
 unitbuf = 8192,      // spláchnutí proudů po výstupu
 stdio =16384         // spláchnutí proudů sdružených se
                      // stdout a stderr po výstupu
};
```

Další tři atributy, *x_precision, x_width* a *x_fill*, jsou typu **int** a obsahují přesnost (počet zobrazovaných desetinných míst), šířku výstupního pole a vyplňovací znak. Implicitní hodnota prvních dvou je 0, u posledního je to mezera.

S uvedenými atributy lze pracovat buď pomocí metod nebo pomocí manipulátorů. Podívejme se nyní na některé z metod.

Většina metod třídy *ios* nastavuje nebo vrací hodnoty atributů (a tak zjišťuje stav proudu nebo určuje formátování vstupu a výstupu).

Hodnoty jednotlivých stavových bitů v atributu *state* lze zjišťovat pomocí funkcí *bad(), eof(), fail()* a *good()*, které vracejí hodnoty typu **int**. Např. příkaz

```
if(cout.bad()) Konec();
```

způsobí volání funkce *Konec( )*, jestliže v proudu *cout* došlo k závažné chybě (tj. je-li v *io_state* nastaven příznak *badbit*).

Voláním metody *ios::clear( )* lze nastavit nebo vynulovat příznaky chyb (kromě příznaku *hardfail*).

Metoda *int ios::rdstate( )* vrací slovo obsahující všechny chybové příznaky daného proudu (tedy atribut *io_state*).

Funkce *long ios::flags( )* vrátí hodnotu formátovacích příznaků, uložených v atributu *x_flags*. Metoda *long ios:: flags(long Priznaky)* vrátí původní hodnotu příznaků a nastaví nové, dané jednotlivými bity parametru *Priznaky*.

Metody *int ios::precision( )* a *int ios::precision(int P)* vracejí přesnost (počet desetinných míst při výstupu reálných čísel). Druhá z nich také nastavuje novou hodnotu přesnosti.

Metody *int ios::width( )* a *int ios::width(int s)* vracejí nastavenou šířku vstupního nebo výstupního pole. Druhá z nich také šířku výstupního pole nastavuje.

Metoda *long ios::setf(long priznaky)* vrátí předchozí nastavení formátovacích příznaků a nastaví novou hodnotu, danou parametrem *priznaky*.

Metoda *char ios::fill(char Vypln)* vrátí předchozí vyplňovací znak a nastaví nový, daný parametrem *Vypln*. Přetížená metoda *char ios::fill( )* pouze vrátí původní vyplňovací znak.

Pro testování stavu datových proudů se často využívají přetížené operátory „!“ a (**void***). Operátor „!“ vrací 1, jestliže se poslední vstupní nebo výstupní operace s proudem nepodařila (pokud je v atributu *state* nastaven některý z příznaků *eofbit, failbit* nebo *badbit*). Operátor přetypování na **void*** vrací nulu, pokud se poslední operace nepodařila, a ukazatel na proud, jestliže proběhla v pořádku. Vrácený ukazatel nelze dereferencovat.

Další proudové třídy

Od třídy *ios* jsou odvozeny třídy *istream* a *ostream*, které představují základ vstupních a výstupních proudů, třída *fstreambase*, která je základem proudů, orientovaných na soubory, a třída *strstreambase*, která je základem paměťových proudů. Také tyto třídy se v programech přímo nepoužívají.

Od nich jsou pak odvozeny třídy *fstream, strstream, istream_withassign, ostream_withassign* a některé další, které již opravdu slouží ke vstupním a výstupním operacím.

Pokud výslovně neuvedeme něco jiného, jsou následující třídy deklarovány v hlavičkovém souboru *iostram.h*.

Třída istream

Tato třída je základem vstupních proudů. V této třídě je pro účely formátovaného vstupu přetížen operátor „>>“. (Občas je označován jako *extraktor*, neboť vyjímá data ze vstupního proudu.) Deklarace tohoto operátoru pro typ **int** má tvar

```
istream & istream::operator>> (int&);
```

Tento operátor vrací odkaz na datový proud, pro který jej zavoláme. To znamená, že např. výraz

```
cin >> i;
```

představuje odkaz na proud *cin*. Díky tomu můžeme přetížené operátory zřetězovat. Jestliže napíšeme

```
cin >> i >> j;
```

vyhodnotí to překladač jako

```
(cin >> i) >> j;
```

neboť operátor >> se vyhodnocuje v pořadí zleva doprava. To znamená, že se přečte hodnota do proměnné *i* a jako výsledek se vrátí odkaz na proud *cin*. Takto vrácený odkaz na proud pak slouží jako levý operand při následujícím čtení do proměnné *j*. Jestliže se při čtení do proměnné *i* nějakým způsobem změnil stav proudu *cin*, bude následující operace probíhat již se změněným proudem.

Ve třídě *istream* pro vstupní proudy jsou mimo jiné definovány metody *istream::tellg( )* a *istream::seekg( )*. První z nich umožňuje zjistit aktuální pozici v souboru, druhá umožňuje tuto pozici změnit.

Třída ostream

Tato třída je pro změnu základem výstupních proudů. Jak víme, je v ní přetížen operátor "<<", který slouží k formátovanému výstupu. (Bývá také označován jako *insertor*, neboť vkládá data do proudu.) Definice tohoto operátoru pro typ **int** má tvar

```
inline ostream&
ostream::operator<< (int _i)
{
  return *this << (long) _i;
}
```

Tento operátor konvertuje levý operand na hodnotu typu **long** a použije operátor „<<“ pro tento typ; pak vrátí odkaz (referenci) na proud, pro který jsme jej zavolali. To opět umožňuje zřetězení několika výstupních operátorů v jednom výrazu.

V této třídě jsou také definovány metody *ostream::tellp()* a *ostream::seekp()*. První z nich zjistí aktuální pozici v proudu, druhá z nich umožňuje aktuální pozici změnit.

Třída iostream

Třída *iostream* je společným potomkem tříd *istream* a *ostream*. Spojuje jejich vlastnosti, obsahuje tedy prostředky pro vstup i pro výstup. Ke zděděným vlastnostem nepřidává nic nového.

Třída ostream_withassign

Je potomkem třídy *ostream*. Navíc je v ní definován přiřazovací operátor, který umožňuje sdružit objekt této třídy s objektem typu *streambuf*. Tím, že se změní vyrovnávací paměť, se proud přesměruje.

V hlavičkovém souboru *iostream.h* jsou definovány standardní instance

```
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;
```

Proud *cout* slouží, jak víme, k formátovanému výstupu do souboru *stdout* (na PC je to tedy přesměrovatelný výstup na obrazovku). Proudy *cerr* a *clog* představují standardní chybový výstup. Proud *cerr* není vybaven vyrovnávací pamětí, proud *clog* je.

Třída istream_withassign

Tato třída je potomkem třídy *istream*. Podobně jako u třídy *ostream_withassign* je v ní definován přiřazovací operátor, který umožňuje sdružit instanci této třídy s objektem typu *streambuf* a tak jej přesměrovat.

V hlavičkovém souboru *iostream.h* je definována standardní instance

```
extern istream_withassign cin;
```

která slouží k formátovanému vstupu ze souboru *stdin* (na PC je to přesměrovatelný vstup z klávesnice).

Poznámka:
V některých překladačích (např. Watcom C++ 10.5) jsou objekty cout, cerr a clog instance třídy ostream a objekt cin je instancí třídy istream.

Třídy fstream, ifstream a ofstream

Třída *fstream* je definována v hlavičkovém souboru *fstream.h*. Je potomkem tříd *iostream* a *fstreambase*. Obsahuje prostředky pro formátovaný vstup a výstup do externích souborů. Jsou v ní k dispozici mj. oba operátory „>>“ a „<<“, zděděné po třídě *iostream*.

V této třídě je definováno několik konstruktorů. Konstruktor

```
fstream::fstream(const char* jmeno, int rezim, int pristup =
                 filebuf::openprot)
```

vytvoří proud, sdruží jej se souborem *jmeno* a tento soubor otevře z režimu *rezim* s atributem *pristup*. Např. příkazem

```
fstream F("data.dta", ios::in);
```

vytvoříme souborový proud *F*, sdružíme jej se souborem *data.dta* a tento soubor otevřeme pro vstup. Jako *atribut* přitom necháváme implicitní hodnotu *filebuf::openprot*, která specifikuje režim "čtení i zápis".

Režim otevření souboru je popsán výčtovým typem *ios::openmode* definovaným ve třídě *ios*. Hodnoty tohoto výčtového typu lze skládat pomocí operátoru „|“ jako bitové příznaky.

```
enum open_mode {
   in = 1,        // otevře soubor pro čtení
   out = 2,       // otevře soubor pro zápis
   ate = 4,       // po otevření najde konec souboru;
                  // to však lze změnit pomocí metody
                  // ostream::seekp
   app = 8,       // Po otevření najde konec souboru;
                  // veškerý výstup půjde na konec souboru
                  // a nelze to změnit
   trunc = 16,    // pokud soubor existuje, smaže se
   nocreate = 32,      // pokud soubor neexistuje, nastane chyba
   noreplace= 64,      // pokud soubor existuje, nastane chyba
   binary = 128        // Soubor se otevře jako binární (implicitně
                       // se otevírá v textovém režimu)
};
```

Jako atribut souboru lze použít hodnoty *S_IREAD* (pouze čtení) a *S_IWRITE* (pouze zápis) definované v souboru *sys\stat.h*. Implicitní hodnota *filebuf::openprot* odpovídá *S_IREAD | S_IWRITE*.

Můžeme také použít konstruktor bez parametrů. Ten pouze vytvoří proud, nic více. Takový „bezprizorní“ proud můžeme později sdružit se souborem pomocí metody *open*, která má stejné parametry jako výše uvedený konstruktor.

Po použití můžeme soubor, sdružený s proudem, uzavřít (a odpojit od proudu) pomocí metody *close*.

```
fstream F;
F.open("data.dta", ios::in|ios::out);

// nějaké operace

F.close();
F.open("data1.dta", ios::ate);
```

Pokud potřebujeme otevírat soubor pouze pro vstup, resp. pouze pro výstup, můžeme použít tříd *ifstream*, resp. *ofstream*. Jejich konstruktory lze volat s jediným parametrem, jménem souboru.

Třídy strstream, istrstream a ostrstream

Třída *strstream* slouží pro práci s paměťovými proudy. Je potomkem tříd *iostream* a *strstreambase* a najdeme ji v hlavičkovém souboru *strstrea.h*. Konstruktor této třídy má tvar

```
strstream::strstream(char* pole, int delka, int rezim).
```

Parametr *pole* je ukazatel na znakový řetězec (pole znaků), ke kterému bude proud připojen. Toto pole musí být dostatečně dlouhé, aby se při výstupních operacích nepřekročily jeho meze.

Je-li parametr *delka* kladný, určuje délku daného pole; 0 znamená řetězec, zakončený '\0', a záporné číslo znamená pole nekonečné délky.

Parametr *rezim* musí být jedna z hodnot výčtového typu *ios:: openmode* uvedených v tabulce 8.1.

Příznak	Význam
ios::in	Vstupní proud, čte se od počátku pole.
ios::out	Výstupní proud, zapisuje se od počátku pole.
ios::ate	Výstupní proud. Pole je řetězec ukončený '\0', zapisuje se od tohoto znaku.
ios::app	Znamená totéž co ios::ate.

Tab. 8.1 Režimy otevření paměťového proudu

Potřebujeme-li paměťový proud pouze pro vstup, resp. pouze pro výstup, můžeme použít proudů *istrstream*, resp. *ostrstream*. V jejich konstruktorech neuvádíme parametr *rezim*.

Třída constream

Tato třída je borlandským rozšířením knihovny datových proudů. Obsahuje prostředky pro formátovaný výstup na konzolu (tedy na obrazovku v textovém režimu). Umožňuje pracovat s okny, měnit barvy vystupujícího textu apod.

Tato třída poskytuje podobné prostředky jako borlandská knihovna *conio.h*. Je definována v hlavičkovém souboru *constrea.h*, který obsahuje též definice řady speciálních manipulátorů.

Konstruktor této třídy je bez parametrů. Proud lze používat ihned po vytvoření.

Z metod této třídy uvedeme *constream::window*, která je analogií funkce *window* z knihovny *conio.h* a která umožňuje omezit výstup proudu do okna na obrazovce. Lze vytvořit i několik proudů, z nichž každý bude pracovat s jinou částí obrazovky.

Dále jsou v této třídě definovány metody *clrscr* a *textmode*, které jsou přesnou analogií stejnojmenných funkcí z knihovny *conio.h*.

Formátování

Podíváme se nyní na prostředky pro formátování vstupů a výstupů. K tomu se používají především manipulátory (viz tab. 8.2), což jsou objekty, které lze vkládat do proudů a tím nějak ovlivnit stav proudu – např. změnit formátovací příznaky.

Manipulátor	Proud	Význam

Manipulátor	Proud	Význam
dec, hex, oct	i,o	následující vstupy nebo výstupy v tomto proudu budou probíhat v desítkové, resp. šestnáctkové, resp. osmičkové soustavě
setbase(n)	i,o	předepisuje číselnou soustavu (při n = 8, 10 nebo 16) nebo implicitní stav (při n = 0)
endl	o	vloží do proudu znak pro přechod na nový řádek a vypíše řádek do souboru (spláchne vyrovnávací paměť)
ends	o	vloží na konec řetězce znak '\0'
flush	o	spláchne proud (vypíše obsah vyrovnávací paměti do souboru a tím ji vyprázdní)
resetiosflags(n)	i,o	vynuluje formátovací příznaky (uložené v atributu x_flags), určené parametrem n; např. je-li nultý bit n roven 1, vynuluje nultý bit x_flags
setiosflags(n)	i,o	nastaví formátovací příznaky (uložené v atributu x_flags), určené parametrem n; např. je-li nultý bit n roven 1, nastaví nultý bit x_flags na 1
setfill(n)	o	definuje vyplňovací znak
setprecision(n)	o	nastaví přesnost (počet desetinných míst) pro výstup reálných čísel
setw	o	nastaví šířku výstupního pole (minimální počet znaků, které vystoupí)
ws	i	přikazuje okamžitě přeskočit na vstupu bílé znaky

Tab. 8.2 Přehled manipulátorů; i resp. o znamení vstupní, resp. výstupní proud

Číselná soustava

Celá čísla implicitně vystupují v desítkové soustavě; při vstupu se implicitně používá konvence jazyka C, tj. čísla začínající *0x* jsou chápána jako čísla v šestnáctkové soustavě, čísla začínající 0 jsou brána jako osmičková a ostatní jako desítková.

Můžeme ovšem explicitně předepsat číselnou soustavu; k tomu poslouží manipulátor *dec*, který specifikuje desítkovou soustavu, *hex* (šestnáctková soustava) nebo *oct* (osmičková soustava. Například příkazy

```
int i = 12;
cout   <<  dec << i << ' '<< oct << i << ' ' << hex << i;
```

způsobí výstup

```
12 14 c
```

Číselnou soustavu lze také předepsat pomocí manipulátoru *setbase(n),* kde *n* je jedno z čísel 0, 8, 10, 16. Hodnota 0 znamená obnovení implicitního stavu.

Šířka výstupního pole

Šířka výstupního pole znamená minimální počet vytištěných znaků. Pokud má vystupující hodnota více znaků, výstup „přeteče“, zabere tolik, kolik potřebuje.

Upozornění:
Na rozdíl od ostatních formátovacích příznaků se po každé výstupní operaci nastavuje šířka výstupního pole na implicitní hodnotu 0 (to znamená použít tolik znaků, kolik je třeba). Šířku tedy musíme nastavovat pro každou vystupující hodnotu znovu.

Šířku výstupního pole nastavíme na *n* znaků pomocí manipulátoru *setw(int n)*. Zopakujme si, že šířku pole můžeme také nastavit pomocí metody *width(int)*, zděděné po třídě *ios*. Metoda *width()* bez parametrů umožňuje zjistit nastavenou hodnotu šířky pole.

Manipulátor *setw(n)* lze sice použít i pro vstupní proudy, nemá ale žádný účinek. Při čtení končí vstupní pole buď bílým znakem nebo znakem, který nepatří do reprezentace čtené hodnoty.

Přesnost

Pod „přesností“ rozumíme počet míst za desetinnou tečkou při výstupu reálných čísel. K tomu nám poslouží manipulátor *setprecision(int n)*, kde *n* je počet desetinných míst.

Vyplňovací znak

Pokud výstup potřebuje méně místa, než kolik předepisuje šířka, doplní se vyplňovacím znakem; implicitně je to mezera. Jinou hodnotu lze nastavit pomocí manipulátoru *setfill(int)*. Například příkazy

```
int i = 22;
cout << setw(11) << setfill('*') << i;
```

způsobí výstup

```
*********22
```

(Podobné triky se uplatní při tisku peněžních částek.)

Aktuální hodnotu vyplňovacího znaku zjistíme pomocí metody *fill()*, zděděné po třídě *ios*.

Další formátovací možnosti

K podrobnějšímu formátování lze použít manipulátorů *setiosflags(long n)* a *resetiosflags(long n)*. První z nich nastavuje formátovací příznaky, specifikované parametrem *n*, druhý tyto příznaky nuluje. Hodnotu parametru poskládáme jako bitový součet příznaků uvedených v tabulce 8.2.

Aktuální hodnotu atributu, obsahujícího všechny formátovací příznaky, zjistíme pomocí metody *flags()* deklarované ve třídě *ios*.

Chceme např., aby čísla typu **double**, uložená v poli *a*, vystupovala v proudu *F* v semilogaritmickém tvaru, aby se přitom vždy zobrazovala desetinná tečka a aby se u kladných čísel tisklo znaménko „+“.

```
long form = ios::scientific | ios::showpoint | ios::showpos;
```

```
F << setiosflags(form) << setprecision(5);
for(int i = 0; i < N; i++)
  F << setw(10) << a[i] << endl;
```

Pokud bychom se po vypsání pole *a* chtěli vrátit k předchozímu formátu výstupu, vynulujeme nastavené příznaky příkazem

```
F << resetiosflags(form);
```

Pozor: manipulátor *setiosflags(n)* nastaví na 1 bity v atributu *x_flags*, které jsou v *n* rovny 1, a ostatní ponechá. To znamená, že

```
// neudělá nic !!
F << setiosflags(0);
```

formátovací příznaky proudu vůbec nezmění. Pokud bychom chtěli vynulovat *F.x_flags*, tedy nastavit všechny formátovací příznaky proudu *F* na 0, museli bychom napsat

```
F << resetiosflags(0xFFFFFFFF);
```

Spláchnutí proudu

Manipulátor *flush* „spláchne“ proud, to znamená, že přenese všechna data z vyrovnávací paměti do souboru.

Příklady

Jako první příklad napíšeme kratičký program, který vypíše na obrazovku tabulku funkce sinus s krokem 5°.

```
/*  Příklad C8 – 1   */
#include <iomanip.h>
#include <fstream.h>
#include <math.h>

#define PI 3.14159265358

int main(){
  // Příznaky
  cout << setprecision(5) << setiosflags(ios::showpoint);
  // Hlavička tabulky
  cout << endl << " alfa sin(alfa)" << endl
       << "-----------------" << endl;
  // Výpis hodnot
  for(int i = 5; i <= 30; i += 5){
      cout << setw(4) << i << setw(11) << sin(i/180.0*PI) << endl;
  }
  return 0;
}
```

Nejprve jsme nastavili přesnost 5 desetinných míst. Aby se vypisovaly i koncové nuly, museli jsme nastavit i příznak *ios::showpoint*. Pak jsme vypsali hlavičku tabulky.

Nakonec v cyklu vypisujeme *i*, které nabývá hodnot 5, 10, ... , 30, a sinus tohoto úhlu. Funkce *sin* ovšem očekává úhel v obloukové míře (v radiánech). Proto jej musíme nejprve přepočítat; platí, že $1° = \pi/180$ radiánu.

Pro úhel jsme předepsali šířku pole 4 znaky; vystupující hodnota se implicitně zarovná doprava. Pro hodnoty funkce *sin* jsme předepsali šířku pole 11 znaků. Tím dosáhneme patřičného odsazení od hodnoty úhlu.

Manipulátor *endl* zabezpečuje přechody na nový řádek. Výstup tohoto programu je

```
alfa sin(alfa)
-----------------
  5    0.08716
 10    0.17365
 15    0.25882
 20    0.34202
 25    0.42262
 30    0.50000
```

Druhý příklad bude jednodušší: nejprve vypíšeme do souboru *C:\WORK\DATA.DTA* čísla od 10 do 19, každé na jednu řádku. Pak tento soubor uzavřeme, znovu jej otevřeme pro vstup, jeho obsah přečteme a vypíšeme na obrazovku.

```
/*  Příklad C8 – 2   */
#include <fstream.h>
void Chyba(){ /* ...*/ }

fstream F;

int main(){
  F.open("c:\\work\\data.dta",ios::out);
  if(!F) Chyba();
  for(int i = 10; i < 20; i++)
       F << i << endl;
  F.close();

  F.open("c:\\work\\data.dta",ios::in);
  if(!F) Chyba();
  int j;
  while(F >> j){
       cout << j << endl;
  }
  return 0;
}
```

Datový proud *F* jsme deklarovali jako globální. V prvním příkazu jsme jej sdružili s daným souborem a ten jsme otevřeli pro výstup. (Pokud už takový soubor existoval, jeho obsah se smazal, takže nyní je soubor *DATA.DTA* prázdný.)

Vzápětí jsme testovali, zda se otevření souboru podařilo; k tomu jsme využili přetížený operátor „!“.

Následujícím příkazem **for** jsme do souboru zapsali potřebná data. Pak jsme soubor voláním metody *close* uzavřeli a dalším příkazem jsme jej otevřeli pro vstup. Výsledek jsme opět prověřili pomocí operátoru „!“.

Všimněte si cyklu, kterým jsme ze souboru četli:

```
while(F >> i){
  // Zpracuj přečtenou hodnotu
 }
```

Přetížený operátor „>>“, jak víme, vrací odkaz na svůj levý operand. To znamená, že výraz *F >> i* má hodnotu *F*. Protože v podmínce příkazu **while** smí být číslo nebo ukazatel (nebo hodnota typu **bool**, pokud použijeme ANSI C++), pokusí se překladač převést *F* jeden z těchto typů. Jedinou možností, kterou překladač najde, představuje operátor přetypování na **void*** zděděný po třídě *ios*.

Tento operátor ovšem vrací 0 v případě, že se poslední operace nepodařila. Jakmile se tedy při čtení narazí na konec souboru, cyklus **while** skončí.

Na konci jsme soubor neuzavřeli. O to (a o spláchnutí vyrovnávacích pamětí, tedy výstup všech dat, která jsme do proudu vložili) se postará destruktor, který se volá automaticky při zániku instancí *F* a *cout*.

Paměťové proudy

Paměťové proudy se chovají podobně jako proudy orientované na soubory. Nelze je ovšem uzavírat nebo otevírat pomocí metod *close* nebo *open*, neboť ty zde nejsou k dispozici. Můžeme ale zjistit aktuální pozici v řetězci pomocí funkcí *tellg* (při čtení) resp. *tellp* (při zápisu). Aktuální pozici v řetězci lze také změnit pomocí metod *seekg* (pro čtení) resp. *seekp* (pro zápis). Ukážeme si jednoduchý příklad.

```
/*  Příklad C8 – 3   */
// VÝSTUP DO ŘETĚZCE
#include <iomanip.h>
#include <strstrea.h>

int main(){
  int i;
  char s[100];                              // Buffer
  double d = 3.1415926;

  // Definujeme proud a otevřeme ho pro vstup i výstup
  strstream Str(s, 100, ios::out|ios::out);

  // Vypíšeme dvakrát d, pokaždé s jinou přesností
  Str << setprecision(3) << d << ' ' << setprecision(5) << d << ends;
  double dd;

  // Přečteme hodnotu zpět
  Str >> dd;

  // Zjistíme pozici v proudu a změníme ji
  i = Str.tellg();
  Str.seekg(2);
  // a přečteme znovu obsah bufferu
  Str >> d;
  return 0;
}
```

Zde jsme deklarovali proud *Str* napojený na pole *s* o délce 100 znaků. Proud *Str* jsme otevřeli pro vstup i výstup. Do proudu *Str* (tedy do pole *s*) jsme vložili Ludolfovo číslo s přesností na 3 desetinná místa, mezeru a Ludolfovo číslo s přesností na 5 míst. Výstup jsme ukončili zápisem znaku '\0' pomocí manipulátoru *ends*.

Pole *s* tedy nyní obsahuje řetězec "3.142 3.14159".

Dalším příkazem jsme z proudu *Str* (tedy z pole *s*) přečetli jednu hodnotu typu **double** do proměnné *dd*. Protože jsme z tohoto proudu dosud nečetli, je aktuální pozice pro čtení na počátku u znaku 0. Proměnná *dd* tedy bude obsahovat hodnotu 3.142 a aktuální pozice pro čtení bude u 5 znaku – o tom se přesvědčíme, prohlédneme-li si hodnotu vrácenou funkcí *tellg* a uloženou do proměnné *i*.

Následujícím příkazem, ve kterém voláme metodu *seekg*, změníme aktuální vstupní pozici na 2 (tj. na číslici „1“ za desetinnou tečkou). Poslední čtení proto uloží do proměnné *d* hodnotu 142.0.

Konzolové proudy

Proudy, orientované na výstup na konzolu, jsou borlandským rozšířením C++. Vedle metod, o kterých jsme se zmínili v úvodním přehledu tříd, se pro práci s tímto proudem používá řada manipulátorů. V tabulce 8.3 najdete jejich přehled spolu s funkcemi z *conio.h*, kterým odpovídají.

Manipulátor	Odpovídá funkci	Význam
clreol	*clreol()*	vymaže znaky do konce řádku
delline	*delline()*	odstraní řádek
highvideo	*highvideo()*	kontrastní zobrazení
insline	*insline()*	vloží řádek
lowvideo	*lowvideo( )*	nekontrastní zobrazení
normvideo	*normvideo( )*	normální zobrazení
setattr(int)	*textattr( )*	nastaví atributy vystupujícího textu
setbk(int)	*textbackground( )*	nastaví barvu pozadí
setclr(int)	*tectcolor( )*	nastaví barvu textu
setcrstype	*setcursortype( )*	nastaví typ kurzoru
setxy(int, int)	*gotoxy( )*	umístí kurzor na obrazovce

Tab. 8.3 Manipulátory na proudu constream

Jako parametry manipulátorů *setclr* a *setbk* můžeme použít symbolické konstanty *WHITE, BLACK* atd., podobně jako u funkcí *textcolor()* a *textbackground()*.

Již v teoretické části této kapitoly jsme si řekli, že můžeme definovat několik proudů na obrazovku. Jejich okna se mohou překrývat, i když to obvykle není nejlepší nápad.

Podívejme se na jednoduchý příklad. Definujeme dva proudy na obrazovku, *C* a *D*, a každému přidělíme jedno okno. Tato okna se budou částečně překrývat. V každém z nich definujeme jinou barvu pozadí a textu. Nejprve necháme vystoupit řetězec do jednoho proudu, pak do druhého a pak zase do prvního.

```
/*  Příklad C8 – 4   */
#include <constrea.h>
constream C, D;

int main(){
   C.window(10, 10, 30, 20);
   C << setbk(WHITE) << setclr(BLUE);
   C.clrscr();
// Odstraníme kurzor v proudu C
   C << setcrsrtype(_NOCURSOR);
   C << "Toto je první výstup do proudu C.";
   D.window(15, 8, 35, 18);
   D << setbk(BLUE) << setclr(WHITE);
   D.clrscr();
   D << "Toto je první výstup do proudu D.";
   C << "A toto je druhý výstup do proudu C.";
   C << setxy(1,1) << delline;;
   return 0;
}
```

V proudu *C* jsme odstranili kurzor, v proudu *D* jsme jej ponechali. Jednotlivé fáze výstupu tohoto programu vidíte na obr. 8.3, 8.4, 8.5 a 8.6.

Obr. 8.3

Obr. 8.4

Obr. 8.5

Obr. 8.6

Neformátované vstupy a výstupy

Prostředky pro neformátované vstupní a výstupní operace jsou definovány ve třídách *istream* resp. *ostream* a jejich potomci je pochopitelně zdědili. Při těchto operacích se ignorují formátovací příznaky nastavené v atributech proudu.

Pro neformátovaný výstup lze použít funkci *ostream& ostream::put(char c)*. Tato funkce vloží do výstupního proudu znak *c*.

Pro výstup většího množství bytů lze použít funkci *ostream& ostream::write(const char *p, int n)*. Tato funkce vloží do výstupního proudu *n* bytů, počínaje tím, na který ukazuje *p*. Poznamenejme, že při výstupu znakového řetězce výstup neskončí, jestliže narazí na znak '\0' označující konec řetězce.

Pro neformátovaný vstup lze použít funkci *get*, která má několik tvarů:

Funkce *int istream::get( )* přečte následující znak ze vstupního proudu a vrátí ho; funkce *istream& istream::get(char& c)* přečte následující znak ze vstupního proudu a uloží ho do proměnné *c*.

Funkce *istream& istream::get(char *p, int i, char c = '\n')* přečte ze vstupního proudu nejvýše *i* znaků a uloží je do pole *p*. Čtení může skončit i dříve, pokud se ve vstupním proudu narazí na znak *c* (implicitně konec řádku). **Pozor: ukončovací znak se již nepřečte a zůstane ve vstupním proudu**. To znamená, že pokud bychom napsali

```
char cc[100];
cin.get(cc, 10);
char t = cin.get();
```

na vstupu z klávesnice zadali

```
qwer
```

a ukončili tento vstup stisknutím klávesy ENTER, bude v poli *cc* řetězec "qwer" a v *t* znak '\n'.

Další funkcí, kterou můžeme použít pro neformátovaný vstup, *je istream& istream::read(char *p, int i)*. Tyto funkce přečtou ze vstupního proudu *i* bytů (pokud nenarazí na *EOF*).

Funkce *int istream::peek( )* vrátí hodnotu následujícího znaku, ale nevyjme jej z proudu (a také nenastavuje žádné chybové příznaky); představuje tedy pouze jakési nezávazné „nakouknutí za roh".

Funkce *istream& istream::ignore(int n=1, int zarazka=EOF)* přeskočí (tj. přečte a zapomene) *i* znaků ve vstupním proudu. Přeskakování může skončit i dříve, pokud tato funkce narazí na znak *zarazka*.

Funkce *istream& istream::putback(char z)* vrátí znak *z* zpět do vstupního proudu. V Borland C++ 3.1 lze vrátit maximálně 4 znaky; pokusíme-li se vrátit pátý, nastane chyba (nastaví se příznak *failbit*) a další operace s proudem se nepodaří, pokud příznak chyby nevynulujeme voláním metody *clear*.

9.3 Vstup a výstup uživatelských typů

Vzhledem k tomu, že knihovna datových proudů je založena na objektech jazyka C++, můžeme relativně snadno rozšířit její možnosti. Nejčastěji se setkáme s rozšířením operátorů „<<" a „>>" na typy, definované uživatelem.

Ukážeme si nejprve příklad. Vezmeme třídu *bod*, která slouží k práci s body ve dvojrozměrném prostoru. Vstupní a výstupní operátor pro tuto třídu deklarujeme jako spřátelenou operátorovou funkci (musí mít přístup k soukromým atributům), která bude mít jako levý operand proud a jako pravý operand instanci třídy *bod*. Vracet bude proud.

Datový proud musíme předávat odkazem a operátory „<<" a „>>" jej musí odkazem vrátit, jinak by nebylo možné operátory zřetězovat.

Celý program bude vypadat takto:

```
/*  Příklad C8 – 5   */
#include <iomanip.h>

class bod{
  double x, y;
  public:
  bod(double, double);
  // ...a další metody
  friend ostream& operator<<
  (ostream& o, bod& b);
  friend istream& operator>>
  (istream& i, bod& b);
};

bod::bod(double a, double b)
: x(a), y(b){}
```

```
istream& operator>>(istream& i, bod& b){
   i >> b.x >> b.y;
   return i;
}

ostream& operator<<(ostream& o, bod& b){
   int w = o.width();
   o << b.x << ' ' << setw(w) << b.y;
   return o;
}

bod b(8.9987, 9.12345);

int main(){
   cout << setprecision(3) << setw(5) << b;
   // ...
   cin >> b;
   // ...
   return 0;
}
```

Vstupní operátor je jednoduchý – prostě z proudu *i* přečte hodnoty složek.

Výstupní operátor je nepatrně složitější. Protože šířka pole se po každé výstupní operaci nastavuje na 0, musíme si ji nejprve zjistit pomocí funkce *width()* a uložit do proměnné *w*. Před výstupem druhé složky pak šířku znovu nastavíme pomocí manipulátoru *setw*. Tím zabezpečíme, že pro obě složky bude vyhrazena stejná šířka pole.

9.4 Manipulátory

V tomto oddílu si ukážeme, jak manipulátory vlastně fungují a jak si můžeme definovat vlastní. Výklad vychází ze zkušeností s Borland C++ 3.1, 4.0 a 4.5.

Manipulátory bez parametrů

Manipulátory bez parametrů jsou zpravidla definovány na proudech *istream* resp. *ostream*. (Manipulátory z tab. 3 jsou ovšem definovány pouze na proudu *constream*.)

Manipulátor bez parametrů na proudu *ostream* je identifikátor funkce, která vrací *ostream&* a má jeden parametr typu *ostream&*. Podobně manipulátor bez parametrů na proudu *istream* je identifikátor funkce, která vrací hodnotu typ*u istream&* a má jeden parametr typu *istream&*.

Tradičním příkladem manipulátoru bez parametru, opisovaným z učebnice do učebnice, se jmenuje *beep* (nebo česky *pipni*) a způsobí, že počítač pípne. Deklarujeme jej jako funkci

```
// Pípací manipulátor
ostream& peepni( ostream & proud )
{
   return proud << '\a';
}
```

Podobně jako u operátorů „<<“ a „>>“ i u manipulátorů bez parametrů je nezbytné, aby se odkazem předával jak parametr, tak i vracená hodnota.

Nyní můžeme napsat

```
cout << peepni << "POZOR, NĚCO SE DĚJE!!!";
```

a počítač na nás opravdu před vypsáním upozornění pípne. (Zdrojový text tohoto manipulátoru najdete na doplňkové disketě v souboru *C8–06.CPP*, spolu s příkladem na manipulátor s parametrem.)

Jak to vlastně funguje? Už jsme si řekli, že ve třídě *ostream* je definována řada variant výstupního operátoru „**<<**“. Jedna z nich má tvar

```
inline ostream& ostream::operator<<
 (ostream & (* _f) (ostream&))
{
  return (* _f)(*this);
}
```

Parametrem (pravým operandem) *_f* tohoto operátoru je ukazatel na funkci typu *ostream&* s jedním parametrem typu *ostream&*. Operátor „**<<**“ tuto funkci zavolá a jako parametr jí předá odkaz na proud, pro který ji voláme. To znamená, že příkaz

```
cout << peepni;
```

způsobí volání funkce *peepni()* s parametrem *cout*. Funkce *peepni()* odešle do proudu, který dostala jako parametr, řídicí znak, který způsobí pípnutí. Potom tato funkce vrátí odkaz na svůj parametr. Hodnotu, kterou vrátila funkce *peepni()*, pak vrátí také operátor „**<<**“.

Podobně pracují i manipulátory bez parametrů na vstupních proudech.

Přetížené operátory „**<<**“ a „>>“, které zabezpečují funkci manipulátorů, se označují jako *aplikátory*.

Manipulátory s jedním parametrem

Manipulátor s parametrem je identifikátor, za kterým následuje parametr v závorkách. Protože to musí být správný zápis v C++, musí to být buď volání funkce, nebo instance objektového typu, na kterou aplikujeme operátor volání funkce.

V Borland C++ můžeme definovat parametrické manipulátory, založené na obou možnostech, i když standardní manipulátory používají pouze první z nich.

První možnost: zápis funkce

V Borland C++ představují standardní manipulátory s jedním parametrem volání funkce, která vytvoří a vrátí instanci některého ze speciálních objektových typů.

Tento objekt obsahuje dva atributy: ukazatel na „výkonnou“ funkci, která provede to, co od manipulátoru požadujeme, a parametr manipulátoru (ten poslouží jako parametr „výkonné“ funkce).

Dále jsou pro tyto speciální datové typy přetíženy operátory „<<“ a „>>“**.** Tyto operátory prostě zavolají „výkonnou“ funkci s daným parametrem a vrátí referenci na datový proud.

Podívejme se na jednu z těchto tříd, *smanip_int*, která se používá pro manipulátory na proudu *ios* s parametrem typu **int**. Aplikátor, který zajišťuje funkci manipulátorů s jedním parametrem na této třídě, je deklarován jako spřátelená funkce.

```
class smanip_int {
   // Ukazatel na výkonnou funkci
   ios &(cdecl * fun)(ios&, int);
   // Parametr manipulátoru
   int p;
 public:
   // Konstruktor
   cdecl smanip_int(ios& (__cdecl * _f)(ios&, int), int _p)
   : fun(_f), p(_p) {}
   // Aplikátory
   friend ostream& operator<<(ostream&, const smanip_int &);
   friend istream& operator<<(istream&, const smanip_int &);
};

ostream& operator<<(ostream& o, const smanip_int & s){
   return (* s.fun)(o, s.p);
}

// definice operátoru >> je podobná
```

Jako příklad manipulátoru s jedním parametrem použijeme standardní manipulátor *setw*, který určuje šířku výstupního pole. Je definován pro třídu *ios*, takže jej mohou používat všechny proudy. V Borland C++ 3.1 je *setw* funkce, která vrací hodnotu typu *smanip_int*:

```
smanip_int setw(int n)
{
   return smanip_int(swidth, n);
}
```

Tato funkce prostě zavolá konstruktor třídy *smanip_int* a vytvořený objekt vrátí. Konstruktor uloží do vytvořené instance ukazatel na výkonnou funkci *swidth()* a hodnotou parametru *n*.

Výkonná funkce *swidth()* zavolá metodu *width()* třídy *ios* a nastaví jeden z atributů datového proudu:

```
// Výkonná funkce setw
static ios & swidth(ios & io, int n)
{
   // Nastaví šířku pole
   io.width(n);
   // a vrátí odkaz na proud
   return io;
}
```

To znamená, že když napíšeme

```
cout << setw(10);
```

zavolá se nejprve funkce *setw()*, která vytvoří objekt typu *smanip_int*, a do něj uloží adresu funkce *swidth()* a hodnotu 10.

Pak přijde na řadu volání operátoru „<<“. Jeho prvním parametrem je proud *cout*, druhým nově vytvořená instance typu *smanip_int*.

Operátor „<<“ si ze druhého parametru vyzvedne ukazatel na výkonnou funkci *swidth()* a parametr, se kterým ji má zavolat. Pak funkci *swidth()* opravdu s parametry *cout* a 10 zavolá.

Funkce *swidth()* změní stav datového proudu a vrátí odkaz na něj. Vrácený odkaz na tento proud pak vrátí i operátor „<<“.

Třídy pro další manipulátory

Podobným způsobem jsou definovány i manipulátory na ostatních datových proudech. V Borland C++ 3.1 k tomu slouží třídy

- ✧ *iomanip*_typ pro manipulátory na třídě **iostream**,
- ✧ *imanip*_typ pro manipulátory na třídě **istream**,
- ✧ *omanip*_typ pro manipulátory na třídě **ostream**,
- ✧ *smanip*_typ pro manipulátory na třídě **ios**.

Přípona „*_typ*“ určuje typ parametru manipulátoru. V souboru *iomanip.h* jsou tyto třídy definovány pro typy **int** a **long** (tedy *iomanip_int, iomanip_long* atd.).

Vlastní manipulátor s jedním parametrem typu int nebo long (BC++ 3.1)

Chceme-li napsat vlastní manipulátor s jedním parametrem typu **int** nebo **long**, musíme napsat funkci, která vytvoří objekt některé z předdefinovaných tříd, a k tomu „výkonnou“ funkci, která provede to, co vlastně potřebujeme.

Jako příklad napíšeme manipulátor *linky(n),* který vloží do výstupního proudu *n* přechodů na novou řádku a přitom proud spláchne.

Můžeme očekávat, že vkládaný počet přechodů na novou řádku bude menší než 32767, takže vystačíme s parametrem typu **int**. Náš manipulátor chceme používat na proudu *ostream* nebo na některém z jeho potomků, takže použijeme pomocnou třídu *omanip_int*.

Výkonnou funkci pojmenujeme třeba *_linky()*. Funkci, která vytvoří pomocný objekt, musíme pochopitelně nazvat *linky()*:

```
/*  Příklad C8 – 6   */
#include <iomanip.h>
// Výkonná funkce
ostream& _linky( ostream& os, int m )
{
  while(m>0){
       os << endl;
       m--;
  }
  return os;
```

```
}

// Manipulátor
omanip_int linky( int m )
{
  return omanip_int(_linky, m);
}
```

To je vše, o ostatní se postará překladač (použije k tomu samozřejmě deklarace z hlavičkových souborů). Jestliže nyní napíšeme

```
cout << "Jdi do háje" << linky(8) << "a zůstaň tam.";
```

bude mezi oběma doporučeními 7 volných řádků (8 přechodů na nový řádek).

Manipulátory s parametry jiných typů

Třídy *..manip_*typ jsou v hlavičkovém souboru *iomanip.h* v Borland C++ 3.1 vytvořeny makrem *IOMANIPdeclare(typ)*. To zároveň vytvoří přetížené operátory „<<“ a „>>“, které slouží jako aplikátory.

Pokud bychom se tedy z nějakých důvodů odhodlali deklarovat např. na proudu *ios* manipulátor *blabla,* který by měl jeden parametr **int***, museli bychom si nejprve vytvořit pomocnou třídu pomocí makra *IOMANIPdeclare*. Označení typu ale musí být jednoslovné a nesmí obsahovat modifikátory „*“ apod., jinak by toto makro generovalo syntaktické nesmysly. Proto pro typ **int*** zavedeme deklarací **typedef** nové jméno:

```
typedef int* uint;
IOMANIPdeclare(uint);
```

Toto makro generuje všechny potřebné pomocné třídy.

Další postup je pak již stejný jako v případě manipulátorů s parametry typu **int** nebo **long** – deklarujeme výkonnou funkci a vlastní manipulátor, tedy funkci, která vrátí pomocný objekt:

```
// Výkonná funkce
ios& _blabla(ios& p, uint i){
  // Zde se udělá, co je třeba
  return p;
}

// Vlastní manipulátor
smanip_uint blabla(uint ii){
  return smanip_uint(_blabla, ii);
}
```

A to je opět vše.

V Borland C++ 4.0 a pozdějších je situace poněkud jiná – makra jsou nahrazena šablonami. Hlavičkový soubor *iomanip.h* obsahuje deklarace šablon *smanip<typ>*, *omanip<typ>*, *imanip<typ>* a *iomanip<typ>*, které umožňují generovat pomocné třídy pro manipulátory s jedním parametrem typu *typ*.

Budeme-li chtít v Borland C++ 4.0 definovat na proudu *ostream* manipulátor *linky*, změní se deklarace funkce *linky()* takto:

```
omanip<char> linky ( char m ){
  return omanip<char>(_linky, m);
}
```

Deklarace manipulátoru *blabla* s parametrem typu **int*** na proudu *ios* se bude skládat pouze z deklarace funkce *_blabla*, stejné jako předtím, a z deklarace funkce, která vytvoří pomocný objekt

```
smanip<int*> blabla(int *ii){
  return smanip<void*>(_blabla, ii);
}
```

Při použití šablon, tedy v Borland C++ 4.0 a pozdějších, můžeme pro označení ukazatele na **int** použít zápis **int***, nemusíme je přejmenovávat pomocí deklarace **typedef**. Podobně můžeme použít i složitější označení typů.

Druhá možnost: operátor volání funkce

V úvodu povídání o manipulátorech s parametry jsme si řekli, že to také mohou být objekty, na které se použije operátor volání funkce. Borland C++ obsahuje prostředky, které zpřístupňují programátorovi i tuto cestu vytváření manipulátorů; pro programátora je o něco pohodlnější, ale na druhé straně je o něco méně efektivní.

I zde se liší starší verze jazyka, které používají makra, od novějších verzí, ve kterých se pomocné třídy vytvářejí na základě šablon. Vrátíme se k deklaraci manipulátoru *linky*, který vloží do výstupního proudu *n* přechodů na nový řádek.

Začneme opět u starší verze. Také v tomto případě musíme deklarovat výkonnou funkci *ostream& _linky(ostream& o, int m)* stejnou jako prve. Dále použijeme makro

```
OAPP(int) linky(_linky);
```

Makro *OAPP(int)* se rozvine v identifikátor třídy *oapply_int* definované (spolu s makrem *OAPP* a dalšími podobnými třídami) v hlavičkovém souboru *iomanip.h*.

Zápis tohoto makra tedy představuje deklaraci instance *linky* pomocné třídy *oapply_int*. Příkaz

```
cout << linky(n);
```

znamená, že se na instanci *linky* třídy *oapply_int* použije operátor volání funkce. Tento operátor je ve třídě *oapply_int* deklarován tak, že vytvoří instanci třídy *omanip_int*, uloží do ní parametr *n* a ukazatel na funkci *_linky*. Vytvořenou instanci třídy *omanip_int* pak zpracuje operátor „**<<**“ stejně jako v předchozím případě.

V souboru *iomanip.h* najdeme makra, která umožňují deklarovat tímto způsobem jednoparametrické manipulátory i na dalších proudech:

- ✧ *SAPP(typ)* pro proud *ios*,
- ✧ *IAPP(typ)* pro proud *istream*,
- ✧ *OAPP(typ)* pro proud *ostream*,
- ✧ *IOAPP(typ)* pro proud *iostream*.

Počínaje Borland C++ verze 4.0 jsou tato makra opět nahrazena šablonami:

- *sapp*<*typ*> pro proud *ios*,
- *iapply*<*typ*> pro proud *istream*,
- *oapp*<*typ*> pro proud *ostream*,
- *ioapp*<*typ*> pro proud *iostream*.

Na závěr se podívejme opět na deklaraci manipulátoru *blabla* na proudu *ios* s parametrem typu **int***. V Borland C++ 4.0 ji zapíšeme ve tvaru

```
ios& _blabla(ios& proud, int* u){
// Zde se udělá, co je třeba
  return proud;
}

sapp<void*> blabla(_blabla);
```

Deklarace výkonné funkce je stejná jako předtím; vedle toho deklarujeme instanci *blabla* třídy *sapp*<*int**>.

10. Výjimky

V této kapitole si budeme povídat o vyvolávání a ošetřování výjimek. Jde – zhruba řečeno – o nástroj, který umožňuje mimořádný přenos řízení uvnitř programu „na velké vzdálenosti“, tedy mezi funkcemi.

Prostředky pro práci s výjimkami jsou součástí návrhu C++ jazyka od roku 1989; první implementace ovšem pochází až z roku 1992 (Hewlet-Packard). V borlandských překladačích máme výjimky k dispozici počínaje verzí 4.0. Najdeme je také například v microsoftském překladači Visual C++ 2.0, ve Watcom C++ 10.5 aj.

Vedle toho se v posledních verzích microsoftských a borlandských překladačů jazyka C setkáme s tzv. strukturovanými výjimkami. Jde o microsoftské rozšíření jazyka C, které bylo zavedeno převážně kvůli multithreadovému prostředí Windows NT a Windows 95.

V Turbo Pascalu se s aparátem výjimek nesetkáme; najdeme je až v Object Pascalu v Delphi. Přesto si zde o nich alespoň krátce povíme.

10.1 O co vlastně jde

Proč výjimky vůbec zavádíme? Není to zbytečné? Nestačily by „obyčejné“ mechanismy přenosu řízení, jako volání funkce, návrat z funkce, nanejvýš ještě skok? V této podkapitole se pokusíme ukázat, k čemu jsou výjimky dobré.

Tolerance vůči chybám

Čím více se počítače stávají součástí každodenního života a přebírají zodpovědnost za řadu činností, tím častěji se setkáváme s požadavkem, aby programy, které je řídí, byly tolerantní vůči chybám.

Přitom máme na mysli nejen chyby uživatele nebo prostředí, ve kterém se software používá, ale i chyby samotného programu. **Program by měl počítat nejen s tím, že chybu udělá uživatel, ale i s tím, že on sám obsahuje chyby**.

Pod chybami prostředí nebo uživatele můžeme chápat takové události, jako je vyčerpání operační paměti, poškození souboru, se kterým program pracuje, přerušení síťového spojení s jiným počítačem, výpadek napájení atd. Tyto chyby jsou obvykle předvídatelné a většina programů s nimi nějakým způsobem počítá.

Podstatně horší jsou ovšem vlastní chyby programu. To jsou vlastně důsledky chyb v analýze problému, v návrhu aplikace, nebo třeba i přehlédnutí při psaní (kódování) a ladění. Program s obzvláště záludnými chybami může dlouho běžet bez problémů; nakonec ovšem dojde k dělení nulou, odmocňování záporného čísla, dereferencování ukazatele, který neukazuje nikam (nebo ještě hůře – který ukazuje bůhvíkam) apod.

Takovéto chyby jsou v každém větším programu, to ví každý, kdo si pozorně prostudoval Murphyho zákony [5]. Potvrzují to i odborné prameny, které uvádějí, že v softwa-

ru špičkové kvality připadá jedna chyba přibližně na 10 000 – 20 000 řádek kódu (nepočítaje komentáře) – viz [6].

Software, jehož selhání by mohlo způsobit škody (ať už pád letadla nebo chybu v bankovních operacích), musí předpokládat výskyt chyb a musí vůči nim být tolerantní – to znamená, že v případě, že k chybě dojde, nesmí způsobit katastrofu.

Ošetření chyby může mít různou podobu: program se může pokusit výpočet zopakovat, může provést nějakou implicitní akci, která sice neposkytne optimální výsledek, ale zaručeně nezpůsobí škodu, může podrobně popsat problém a skončit – opět závisí na okolnostech. Pochopitelně také požadujeme jiné chování programu při ladění a jiné při „ostrém“ používání.

Výjimka: co to je?

Za *výjimku* označujeme situaci, která nastane v průběhu normálního chodu programu a která způsobí, že program nemůže obvyklým způsobem pokračovat. Přeloženo do češtiny: výjimka je běhová chyba. U programů tolerantních vůči chybám ovšem běhová chyba nemusí (a často ani nesmí) způsobit předčasné ukončení programu (představte si, že by se pilotovi při přistávání objevila na obrazovce zpráva *Run time error 511* a palubní počítač odmítl vysunout podvozek).

Pokud vznikne výjimka, je třeba ji co nejdříve programově ošetřit. Problém ovšem je, kde. Běhová chyba – např. dělení nulou – může být důsledkem logické chyby v úplně jiné části programu.

Představme si například, že se ve funkci *f()* pokusíme dělit nulou. Chyba ve funkci *f()* je důsledkem špatných hodnot parametrů, které jsme jí předali – a tyto parametry jsme vypočítali ve funkci *g()*, která *f()* volala. Chybný výpočet v *g()* ovšem může být důsledkem problémů, které vznikly ještě dříve. Chceme-li chybu ošetřit, musíme se obvykle dostat k místu jejího vzniku. To ale znamená, že při ošetřování výjimky bychom se měli umět snadno, rychle a také beztrestně přenést na úplně jiné místo v programu.

Chyby v knihovnách

Představte si, že píšeme programovou knihovnu, např. knihovnu kontejnerů. Co dělat, jestliže některá z funkcí nebo metod v knihovně zjistí chybu, kterou nedokáže sama napravit?

Zůstaňme u příkladu s kontejnery: co když se uživatel knihovny pokusí vyjmout data z prázdného zásobníku? Odpověď bude záviset na okolnostech, za kterých byla knihovní funkce volána. Možná, že to znamená fatální chybu a je třeba celý program spustit znovu s jinými parametry; možná, že se vlastně nic důležitého neděje, a program může pokračovat.

O tom ale nemůže rozhodnout autor knihovny, to bude muset vyřešit její uživatel.

Tady ovšem narazíme na další problém. Jak ho o vzniklé situaci informovat? V knihovnách různých programovacích jazyků se obvykle setkáme s některou z následujících možností:

1. Funkce, která problém zjistila, ukončí program (v C++ např. voláním některé z funkcí *exit()*, *abort()*, *terminate(),* apod.). V lepším případě vypíše před ukončením podrobné informace o místu a původu chyby.
2. Funkce vrátí hodnotu, která indikuje chybu.
3. Funkce vrátí použitelnou (nebo alespoň v nějakém smyslu neškodnou) hodnotu a nechá program pokračovat.
4. Funkce, která chybu zjistila, zavolá pomocnou funkci, která se pokusí vzniklou chybu ošetřit.
5. Funkce, která chybu zjistila, vyvolá výjimku.

První možnost lze prakticky vždy zavrhnout. Takto napsanou knihovnu lze používat při ladění, nikoli však v „ostré“ verzi programu. Tolerance vůči chybám znamená, že po chybě se program nějakým způsobem zotaví a poběží dál.

S druhou možností se občas setkáme ve standardních knihovnách C/C++. Připomeňme si např. operátor **new**, jenž v případě neúspěšné alokace vrátí hodnotu 0, která nepředstavuje žádnou přípustnou adresu[14]. Toto řešení lze ale použít jen někdy. Co by měl vrátit přetížený operátor indexování, jestliže hodnota indexu leží mimo meze pole? Jakou hodnotu by měla vrátit funkce *vyjmi()*, která vyjímá prvek na vrcholu zásobníku, je-li zásobník prázdný? Co v případě konstruktoru nebo destruktoru, které prostě nemohou vracet nic? A nakonec, co uživatel naší knihovny: zkontroluje si vrácenou hodnotu?

Se třetí možností se setkáme ve standardní knihovně jazyka C poměrně často. Jistě víte, že mnohé knihovní funkce ukládají v případě neúspěchu kód chyby do globální proměnné *errno*.

Jenže tady se skrývá z hlediska bezpečnosti programu problém: jak donutit programátora, který naši knihovnu používá, aby si zjistil, zda nedošlo k chybě a program náhodou není v havarijním stavu? Co když si programátor ani nepřečte manuál a nebude o uvedené globální proměnné vůbec vědět? (Známý problém uživatele jakéhokoli programového produktu: *RTFM*.[15]) Kromě toho, i když o uvedené globální proměnné bude vědět, bude ji kontrolovat?

Se čtvrtou možností se setkáváme např. u operátoru **new**: pomocí knihovní funkce *set_new_handler()* můžeme určit funkci (handler), která se pokusí neúspěšnou alokaci napravit. Jenže takovéto řešení lze použít spíše výjimečně: Máte představu, jak by měl vypadat handler, který by se pokusil napravit prázdný zásobník? Kromě toho, pokud se náprava nepodaří, musí handler stejně zvolit jednu ze zbývajících možností.

Pátá možnost, použití výjimek, představuje velice rozumnou alternativu zejména k první, třetí a čtvrté možnosti.

---

[14] Podle současného návrhu normy může operátor **new** v případě neúspěšné alokace buď vrátit hodnotu, která nepředstavuje žádnou platnou adresu (tedy 0), nebo vyvolat výjimku.

[15] *RTFM* je zkratka anglických slov *Read The Fucking Manual*. Překlad do češtiny by mohl jemnocitnější čtenáře urazit, a proto jej neuvádíme.

Z předchozího povídání opět vyplynulo, že se při vzniku výjimky potřebujeme rychle a beztrestně přenést z místa, kde jsme problémy zjistili, do místa, kde je můžeme ošetřit. Tedy např. z knihovní funkce do funkce, která ji (přímo nebo nepřímo) zavolala, nebo třeba až do hlavního programu.

Na první pohled by se mohlo stát, že při takovémto mimořádném skoku mezi funkcemi vystačíme s funkcemi *setjmp()* a *longlmp()*, které byly již součástí ANSI C a o kterých si můžete přečíst v *Dodatku*.

Situace je ale trochu složitější. Jde o to, co udělat s prostředky přidělenými ve funkcích mezi místem chyby a místem ošetření. Než program dospěl do místa, kde došlo k chybě, mohli jsme alokovat dynamickou paměť, otevřít soubor, vytvořit časovač v programu pro Windows atd. Při rychlém návratu je tedy potřeba uvolnit paměť, uzavřít soubor, zrušit časovač atd.

To bohužel při dlouhém skoku nejde automaticky zařídit. Záznam stavu pomocí funkce *setjmp()* obsahuje vlastně jen stav registrů procesoru, nic více. To zajistí korektní ošetření zásobníku (uvolnění lokálních proměnných a parametrů), nikoli však ošetření ostatních prostředků.

Na druhé straně mechanismus práce s výjimkami, jak je implementován v C++, zajistí kromě ošetření zásobníku i volání destruktorů všech lokálních objektů v opouštěných blocích. Kromě toho umožňují výjimky v C++ přenést z místa chyby do místa ošetření řadu informací, které můžeme později využít při ošetřování vzniklé situace.

10.2 Výjimky v C++

Jazyk C++ umožňuje pracovat pouze s tzv. synchronními výjimkami, tj. s výjimkami, které vzniknou uvnitř programu. Pomocí výjimek v C++ tedy nemůžeme zpracovávat události, které vzniknou mimo program – jako např. stisknutí klávesové kombinace CTRL+BREAK. Také běžné chyby, jako je dělení nulou nebo aritmetické přetečení, nezpůsobí v C++ vznik výjimky.

První přiblížení

Všechny operace, které by se nemusely podařit, budeme provádět v tzv. **hlídaném bloku** (anglicky se nazývá *guarded block*). *Hlídaný blok* se skládá z *pokusného bloku* (anglicky nazývaného *try block*) a z jednoho či několika *handlerů*[16] (*exception handler*).

V pokusném bloku se pokusíme provést ony „nebezpečné“ operace, tedy operace, které by mohly vyvolat výjimku. Pokud výjimka nenastane, proběhnou v pořádku všechny příkazy pokusného bloku, po jeho ukončení bude program pokračovat za hlídaným blokem a handlery se prostě přeskočí.

[16] Pokud víme, korektní český termín pro handler v této souvislosti zní asi tak „úsek programu, určený pro ošetření výjimky“. Vzhledem k tomu, že je to poněkud zdlouhavé, zůstaneme u slangového termínu „handler“. Máte-li lepší návrh, dejte vědět.

Pokud se ale některá z operací v pokusném bloku nepodaří, skončí provádění tohoto bloku předčasně a řízení převezme některý z handlerů tohoto hlídaného bloku. Neukončí-li handler běh programu, může program po provedení tohoto handleru pokračovat za hlídaným blokem.

Výjimka může vzniknout přímo v pokusném bloku nebo v některém z vnořených bloků. Může vzniknout i ve funkci, kterou voláme v pokusném bloku.

Když vznikne výjimka, začne systém hledat vhodný handler, který by ji ošetřil. Přitom postupuje zpět podle posloupnosti vnořených bloků a volání funkcí. Pokud tedy nenajde potřebný handler v bloku, kde výjimka vznikla, přejde do bloku nadřízeného. Pokud nenajde handler v dané funkci, vrátí se do funkce, která ji zavolala a bude jej hledat tam. O tomto procesu hovoříme jako o *šíření výjimky* do nadřízeného bloku, resp. do volající funkce.

Výjimka se může postupně rozšířit až do hlavního programu, tedy do funkce *main* (pokud programujeme pro Windows, tak do funkce *WinMain*).

Při vyvolání výjimky posíláme handleru hodnotu, která nese informace o povaze výjimky a okolnostech jejího vzniku. K tomu se často využívají objektové typy. Typ hodnoty, kterou posíláme handleru při vyvolání výjimky, označujeme jako *typ výjimky*.

Syntax výjimek

V C++ slouží pro práci s výjimkami trojice klíčových slov **try**, **catch** a **throw**.

První z nich, klíčové slovo **try**, používáme jako prefix pokusného bloku. Klíčové slovo **catch** uvádí handler, blok pro ošetření výjimky. Handlery následují bezprostředně za pokusným blokem. Klíčové slovo **throw** představuje operátor, který vyvolá výjimku.

Poznamenejme, že *try* znamená v angličtině *zkusit*, *pokusit se*; zde tedy přikazujeme „zkus následující příkazy“. *Catch* znamená mj. *(za)chytit*; handler tedy zachytí vzniklou výjimku a pokusí se s ní něco provést. Překlad slova *throw* je asi nejproblematičtější. V angličtině znamená *hodit*, *vrhnout* – třeba také vrhnout mláďata. Ani jedna možnost se nám příliš nezamlouvá, takže říkáme, že výjimka *vznikne*, *nastane* nebo jsme výjimku *vyvolali*.

Uvedené konstrukce popisují následující syntaktické definice:

hlídaný_blok:
    pokusný_blok seznam_handerů

pokusný_blok:
    try *složený_příkaz*

Seznam_handlerů je jeden nebo několik handlerů za sebou.

seznam_handlerů:
    handler
    seznam_handlerů handler

Jednotlivé handlery popíšeme takto:

handler:

catch(*specifikace_výjimky***)** *složený_příkaz*

Specifikace_výjimky se podobá specifikaci jednoho formálního parametru v deklaraci funkce – obsahuje buď určení typu a identifikátor parametru nebo jen určení typu. Může obsahovat i výpustku (…).

Výjimku vyvoláme pomocí výrazu, který se pro účely popisu syntaxe označuje jako *výraz throw*:

výraz_throw:
throw *přiřazovací_výraz*$_{opt}$

Index $_{opt}$ říká, že *přiřazovací_výraz* můžeme vynechat. Příkaz **throw** vyvolá výjimku. Hodnota výrazu, uvedeného v příkazu **throw**, ponese informace o výjimce. Typ výjimky je určen typem hodnoty tohoto výrazu. Hodnotu tohoto výrazu můžeme v handleru použít a podle ní se rozhodnout, jak výjimku ošetříme.

Vše, co jsme si dosud řekli, ukazuje následující schematický příklad:

```
try {  // Pokusný blok:
       // zde jsou operace, které se nemusí podařit
       if(problem_1) throw Vyj(1);
       // ...
}catch(Vyj v) {
   // Handler:
   // Zde se pokusíme chybu napravit
}
```

Funkce, které šíří výjimky

Klíčové slovo **throw** se také používá jako přípona v deklaraci funkce. Zde specifikuje typ výjimek, které mohou v dané funkci vzniknout a rozšířit se z ní (tzn. nejsou v ní ošetřeny).

Deklarace funkce se specifikací výjimek, které se z ní mohou šířit, může mít jeden z následujících tvarů:

dekl_fce_s_výjimkami:
hlavička_funkce
hlavička_funkce **throw()**
hlavička_funkce **throw(***seznam_výjimek***)**

Seznam_výjimek:
typ
seznam_výjimek, typ

Seznam_výjimek je vlastně seznam označení typů výjimek, které mohou v dané funkci vzniknout a nejsou v ní ošetřeny. Pokud za hlavičkou funkce klíčové slovo **throw** neuvedeme vůbec, znamená to, že ve funkci může vzniknout jakákoli výjimka.

Ukážeme si jednoduché příklady:

```
void F();
int G() throw();
char H() throw(int, double, Info);
```

Při volání funkce *F* může nastat výjimka jakéhokoli typu a může se z této funkce rozšířit.

Při volání funkce *G* nemůže nastat žádná výjimka. (Přesněji řečeno: z funkce *G* se nemůže rozšířit žádná výjimka; v těle funkce *G* může nastat jakákoli výjimka, musí v něm ale být zachycena a ošetřena.) Pokud by se z této funkce přece jen nějaká výjimka rozšířila, bude ji systém pokládat za *neočekávanou výjimku* (*unexpected exception*). K neočekávaným výjimkám se ještě vrátíme.

Z funkce *H* se může rozšířit výjimka typu **int**, **double** nebo *Info*. Pokud by se z této funkce rozšířila jakákoli jiná výjimka, bude s ní systém opět zacházet jako s neočekávanou výjimkou.

Poznamenejme, že specifikace typu výjimek není součástí jména funkce – tedy neodrazí se ve vnitřním jménu, které používá např. linker (a které umožňuje mj. rozlišovat přetížené funkce podle počtu a typu parametrů). Proto můžeme deklarovat jiný typ výjimek pro virtuální metodu v předkovi a jiný typ výjimek pro virtuální metodu stejného jména v potomkovi. Není to ale dobrý nápad, neboť tím především zmateme sami sebe a vykoledujeme si nejspíš nějakou záhadnou chybu.

Příklad

Jako první příklad si vezmeme implementaci jednosměrného seznamu, podobného jako v příkladu C7 – 8. Pro jednoduchost do něj budeme ukládat pouze celá čísla. Navíc do něj přidáme metodu *vyjmi_prvni()*, která bude mít za úkol odstranit ze seznamu první prvek (hlavu) a vrátit data, která v něm byla uložena. Pokud je seznam prázdný, vyvolá tato metoda výjimku typu *Chyba*. Úplný zdrojový text tohoto příkladu najdete v souboru *C9–01.CPP* na doplňkové disketě.

Nejprve deklarujeme třídu, jejíž instance nám budou sloužit pro přenos informací o výjimkách. Pojmenujeme ji vtipně *Vyjimka*.

```
/*  Příklad C9 – 1   */
class Vyjimka{
  char* zprava;            // Text chybové zprávy
 public:
  Vyjimka(char* c): zprava(c){}
  char* text(){ return zprava; }
};
```

Tato třída je velice jednoduchá: obsahuje pouze ukazatel na textový řetězec s chybovou zprávou. Konstruktor do tohoto ukazatele uloží adresu řetězce se zprávou, metoda *text()* tento ukazatel vrátí. Pro naše účely to zatím postačí.

Dále definujeme třídu *prvek* (téměř stejnou jako v příkladu C7 – 8, až na to, že nepoužijeme šablony) a třídu seznam:

```
class seznam {
  prvek* hlava, *zarazka;
 public:
  seznam();
```

```
  ~seznam();
  void vloz(int);
  int vyjmi_prvni() throw (Vyjimka);
};
```

Také konstruktor, destruktor a metoda *vloz()* jsou prakticky stejné jako v příkladu
C7 – 8, až na to, že pracujeme s hodnotami typu **int**, a tudíž nepoužíváme šablony.

Metoda *vyjmi_prvni()* nejprve prověří, že seznam není prázdný, a pak z něj odstraní první prvek. Hodnotu, která je v něm uložena, si uschová v pomocné proměnné a pak ji vrátí. Je-li seznam prázdný, vyvolá výjimku.

```
int seznam::vyjmi_prvni() throw (Vyjimka)
{
  if(hlava == zarazka) throw Vyjimka("prázdný seznam");
  prvek* pom = hlava;
  hlava = hlava -> nasl;
  int vysledek = pom->d;
  delete pom;
  return vysledek;
}
```

Dále použijeme v programu instanci *S*:

```
seznam S;
```

Nyní chceme ze seznamu vyjmout všechny prvky a vypsat jejich hodnoty. Při odebírání prvků ze seznamu může ovšem vzniknout výjimka, proto uzavřeme tuto operaci do pokusného bloku:

```
try{                              // Pokusný blok
  for(int i = 0; i < N+1; i++)    // ***
       cout << S.vyjmi_prvni() << endl;
}
catch(Vyjimka v){                 // Handler
  cout << v.text();
  exit(1);
}
Dalsi();
```

V souboru *C9–02.CPP* na doplňkové disketě najdete podobný příklad řešený pomocí šablon.

Když dojde k výjimce

Výjimku, jak víme, způsobíme v C++ příkazem **throw**. Provedení tohoto příkazu způsobí, že se ihned přeruší zpracování pokusného bloku a řízení se přenese do nejbližšího následujícího handleru pro výjimku daného typu. Pokud nastane výjimka ve funkci, ve které není k dispozici odpovídající handler, ukončí se celé tělo funkce a handler se bude hledat v nadřízeném bloku.

Pokud nalezený handler program neukončí voláním některé z funkcí *terminate()*, *abort()*, *exit()* ap., vrátí se řízení za poslední handler hlídaného bloku. Zbytek kódu v pokusném bloku se již v žádném případě neprovede.

Vraťme se k příkladu C9 – 1 z minulého odstavce, k vybírání ze seznamu. V příkazu **for**, označeném třemi hvězdičkami, se pokoušíme vybrat ze seznamu *N*+1 hodnot. Pokud seznam obsahuje kromě zarážky opravdu alespoň *N*+1 prvků, bude vše v pořádku, žádná výjimka nevznikne a všechny vyjmuté hodnoty se vypíší na obrazovku. Po skončení cyklu **for** skončí i pokusný blok, následující handler se přeskočí a program bude pokračovat voláním funkce *Dalsi()*.

Předpokládejme ale, že seznam obsahuje pouze *N*-1 záznamů. Při *N*-tém volání bude seznam již prázdný a vznikne výjimka typu *Výjimka*; v příkazu **throw** voláme konstruktor této třídy.

To znamená, že se přeruší provádění těla metody *vyjmi_prvni()* a systém začne hledat handler, který by vzniklou výjimku ošetřil. V těle této metody ji nenajde, proto ji ukončí a přejde do nadřízeného bloku.

Také provádění nadřízeného bloku příchodem výjimky skončí, zbylé příkazy se již neprovedou. To znamená, že např. nedojde k výstupu do proudu *cout*. (Nemělo by to ani smysl, neboť metoda *vyjmi_prvni()* neskončila řádně a nevrátila žádnou hodnotu.)

Nadřízený blok byl v našem případě již pokusný blok, za kterým následuje handler typu *Vyjimka*. Řízení tedy přejde do tohoto handleru. Zde vypíšeme text, uložený v instanci, a ukončíme program.

Pošli to dál…

Někdy se stane, že v handleru potřebujeme výjimku částečně ošetřit (např. pozavírat soubory), nemůžeme ale udělat vše, co je třeba. Jinými slovy: potřebovali bychom jednu výjimku zpracovávat nadvakrát, na dvou různých úrovních.

To jde. V handleru můžeme v případě potřeby znovu vyvolat tutéž výjimku. K tomu poslouží příkaz

```
throw;
```

ve kterém neuvedeme *výraz*. Parametrem takto vzniklé výjimky bude parametr právě ošetřované výjimky. (V tomto případě musí být handler, ve kterém vznikne výjimka, vnořen v dalším pokusném bloku.)

Handler

Handler, blok pro ošetření výjimky, musíme zapsat bezprostředně za pokusný blok nebo za jiný handler.

Jak víme, začíná handler klíčovým slovem **catch**, za kterým je v závorkách specifikován typ výjimky. Specifikace typu výjimky se podobá specifikaci (jednoho) formálního parametru funkce; jméno tohoto parametru můžeme vynechat.

Jestliže jméno parametru uvedeme, můžeme se na ně uvnitř handleru odvolávat. Jestliže toto jméno vynecháme, znamená to, že je pro nás důležitý jen typ výjimky.

Parametr handleru můžeme specifikovat jako konstantní nebo referenční (tj. předávaný odkazem). Předávání parametrů odkazem oceníme především v případě parametrů objektových typů. O tom si povíme dále.

Vraťme se k funkci *vyjmi_prvni()*. Pokud by tato funkce mohla vyvolat nejen výjimky typu *Vyjimka*, ale i typu **int**, mohli bychom předchozí příklad upravit např. takto:

```
try{                                // Pokusný blok
   for(int i = 0; i < N+1; i++)     // ***
   cout << S.vyjmi_prvni() << endl;
}
catch(int) {                        // Zde nás hodnota parametru nezajímá
   cout << "Nějaké problémy v seznamu" << endl;
}
catch(Vyjimka v){                   // Handler
   cout << ch.text();
   exit(1);
}
Dalsi();
```

V tomto příkladu je hlídaný blok tvořen pokusným blokem a dvěma handlery. Pokud vznikne při vybírání ze seznamu výjimka typu **int**, přejde řízení do odpovídajícího handleru. Po jeho skončení přejde řízení za poslední handler – na funkci *Dalsi()*.

Univerzální handler

Specifikaci typu výjimky v deklaraci handleru můžeme nahradit výpustkou (…). Takový handler označujeme jako **univerzální**, neboť zachytává všechny výjimky bez výjimky. Proto jej musíme uvádět vždy jako poslední, za handlery se specifikovaným typem.

Konverze typu parametrů v handleru

Nastane-li výjimka, hledá systém vhodný handler. Přitom jednotlivé handlery rozlišuje podle typu parametrů. Tento mechanismus může na první pohled připomínat rozlišování přetížených funkcí s jedním parametrem – ale jen na první pohled. Při hledání odpovídajícího handleru se téměř neuplatňují konverze parametrů. Povíme si přesná pravidla.

Handler

```
catch(XXX){
   // ...
}
```

ve kterém *XXX* je specifikace typu *T*, *T&*, **const** *T* nebo **const** *T&*, přijme jako skutečný parametr objekt *b*, který je

- ✧ typu shodného s typem *T*,
- ✧ objektového typu, který je veřejně přístupným předkem typu *T*,
- ✧ typu ukazatel na objekt, který lze konvertovat na typ *T* (také ukazatel) pomocí standardních konverzí ukazatelů v místě vyvolání výjimky.

Žádné jiné konverze se neprovádějí, ani takové zdánlivě samozřejmé, jako je konverze **short** na **int** nebo **char** na **signed char** nebo **unsigned char**.

Všimněte si, že vlastně jediné konverze jsou dány tradičním objektovým pravidlem o tom, že potomek může zastoupit předka. Díky tomu může handler zachytit všechny výjimky, jejichž typy jsou členy stejné dědické posloupnosti, a pokud předáváme parametr odkazem, můžeme při ošetřování využít jejich virtuálních metod. Ukážeme si na příklad:

```
// Třídy, které přenášejí informace o výjimce
class Vyjimka1 {
   // ...
   virtual char* Info();
};

class Vyjimka2: public Vyjimka1 {
   // ...
   virtual char* Info();
};

class Vyjimka3: public Vyjimka2 {
   // ...
   virtual char* Info();
};

// ...
try {
   ZkusTo();
   // ...
}
catch(Vyjimka1 &v) {
   // zachytí výjimky typů Vyjimka1, Vyjimka2 i Vyjimka3
   int i = v.Info();
}
```

Podívejme se, co se stane, jestliže ve funkci *ZkusTo()* vznikne výjimka některého z typů *Vyjimka1*, *Vyjimka2* nebo *Vyjimka3*. Protože typy *Vyjimka2* a *Vyjimka3* jsou veřejně odvozenými potomky třídy *Vyjimka1,* zachytí vzniklou výjimku náš handler. Jeho parametr jsme předali odkazem, takže se zavolá správná virtuální metoda *Info* podle typu zachycené výjimky.

Z pravidel pro konverze plyne, že pokud používáme k přenosu informace objektové typy, které tvoří dědickou hierarchii, musíme handler, jehož parametrem je předek, uvést *před* handlerem, jehož parametr je typu potomek. Pokud bychom tedy chtěli v předchozím příkladu mít zvláštní handler pro každý z typů výjimky, museli bychom je napsat takto:

```
// Deklarace tříd Vyjimka1, Vyjimka2 a Vyjimka3 stejné jako prve
// ...
try {
   ZkusTo();
   // ...
}
catch(Vyjimka3 &v) {
   // ...
}
catch(Vyjimka2 &v) {
   // ...
}
catch(Vyjimka1 &v) {
   // ...
}
```

Pokud bychom jako první uvedli např. handler s parametrem typu *Vyjimka1*, zachycoval by i výjimky typu *Vyjimka2* a *Vyjimka3*.

Uvnitř handleru můžeme měnit hodnotu parametru. V příkazu **throw** můžeme pochopitelně zapsat i globální objekt, takže by se mohlo zdát, že pokud tento parametr předáváme odkazem, mohl by handler tento globální objekt změnit. To však není pravda; příkazem **throw** se vždy přenáší kopie hodnoty „vrhaného“ výrazu.

Výjimky a bloková struktura programu

Pokusný blok a těla handlerů se chovají jako nezávislé bloky. To znamená, že každý z nich může mít své vlastní lokální proměnné.

Jak pokusný blok tak i handlery jsou samozřejmě součástí svého nadřízeného bloku, takže i v těle pokusného bloku nebo handleru se můžeme odvolávat na proměnné z nadřízeného bloku a na globální proměnné.

Z pokusného bloku i z handleru můžeme vyskočit příkazem **return**, **break**, **continue** nebo **goto**. Skákat dovnitř pokusného bloku nebo dovnitř handleru je zakázáno.

Vyvolání výjimky znamená vždy ukončení pokusného bloku. **Než se ale předá řízení handleru, který výjimku ošetří, zavolají se destruktory instancí lokálních automatických objektů, které jsme v pokusném bloku deklarovali**. To znamená, že se např. uzavřou otevřené datové proudy.

Výjimky uvnitř funkcí

Již víme, že vyvolání výjimky v těle funkce, která ji sama neošetří, znamená ukončení této funkce, ovšem předtím se zavolají destruktory lokálních objektů. Předvedeme si to na následujícím příkladu. Nehledejte v něm žádný hlubší smysl, není v něm – pouze ukazuje, jak to funguje.

```
/*  Příklad C9 – 2   */
// Konstruktor a destruktor této třídy o sobě "dají vědět"
class Blabla {
public:
  Blabla(){ cout << "Konstruktor třídy Blabla" << endl; }
  ~Blabla(){ cout << "Destruktor třídy Blabla" << endl; }
};

// Vyvolá výjimku typu int
void F(int j) throw(int)
{
  Blabla a;
  cout << "Funkce F, před místem výjimky" << endl;
  if (j == chyba) throw 1;
  cout << "Funkce F, za místem výjimky" << endl;
};

// Volá F, ale výjimku neošetří
void G(int j) throw(int) {
  Blabla b;
  try{
```

```
        cout << "Funkce G, před voláním F" << endl;
         F(j*j);
        cout << "Funkce G, za voláním F"<< endl;
   }
   catch(double){
        cout << "Funkce G, zachycena výjimka typu double" << endl;
   }
   catch(int){
        cout << "Funkce G, zachycena výjimka typu int, posílám ji dál"
             << endl;
        throw;
   }
}

int main()
{
   try {
        Blabla c;
        cout << "Funkce main, před voláním G" << endl;
        G(0);
        cout << "Funkce main, za voláním G" << endl;
   }
   catch(...) {
        cout << "Funkce main: zachycena nějaká výjimka" << endl;
   }
   cout << "Funkce main, za handlerem" << endl;
   return 0;
}
```

Třída *Blabla* má v tomto příkladu jediný smysl: její konstruktor a destruktor o sobě dá vědět tím, že vypíše zprávu na obrazovku.

V pokusném bloku ve funkci *main* deklarujeme instanci *c* třídy *Blabla* a pak zavoláme funkci *G()*. Funkce *G()* deklaruje instanci *b* třídy *Blabla* a zavolá funkci *F()*. Také ve funkci *F* deklarujeme instanci třídy *Blabla*.

Podívejme se nyní, co se bude dít, jestliže ve funkci *F()* vznikne výjimka typu **int**:

Provádění těla funkce *F()* skončí provedením příkazu **throw**. Zbylé příkazy v těle této funkce se přeskočí, ale zavolá se destruktor lokální instance *a* třídy *Blabla*. Pak se uvolní zásobník, tj. odstraní se z něj lokální proměnné a skutečné parametry a obsah registrů *SP* a *BP* se nastaví na hodnotu před voláním *F()*. (Připomeňme si, že jde vlastně o standardní ošetření zásobníku při návratu z funkce).

Protože výjimka nebyla ve funkci *F()* ošetřena, rozšíří se do *G()*, která funkci *F()* zavolala. Volání *F()* je v *G()* vloženo do pokusného bloku, ke kterému je připojen handler typu **int**.Tento handler výjimku zachytí, vypíše upozornění a pošle ji dál (neošetří ji).

Výjimka se tedy bude šířit dále. To znamená, že také provádění těla funkce *G()* skončí. Příkazy, které následují za voláním funkce *F()*, se již neprovedou. Samozřejmě také funkce *G()* bude ukončena korektně – zavolá se destruktor lokální instance *b* třídy *Blabla* a upraví se zásobník.

Tím se řízení vrátí do pokusného bloku ve funkci *main()*. Teprve zde je k dispozici handler, který tuto výjimku ošetří. Řízení tedy opustí i tento pokusný blok (zpráva

Funkce main, za voláním G() se nevypíše) a přejde do příslušného handleru. Ten vypíše upozornění a řízení bude pokračovat za handlerem.

Výstup tohoto programu, pokud vznikne ve funkci *F()* výjimka, je

```
Konstruktor třídy Blabla
Funkce main, před voláním G
Konstruktor třídy Blabla
Funkce G, před voláním F
Konstruktor třídy Blabla
Funkce F, před místem výjimky
Destruktor třídy Blabla
Funkce G, zachycena výjimka typu int, posílám ji dál
Destruktor třídy Blabla
Destruktor třídy Blabla
Funkce main: zachycena nějaká výjimka
Funkce main, za handlerem
```

Výjimky a konstruktory

Co když výjimka vznikne v konstruktoru instance objektového typu? Taková instance ještě není hotová, proto se pro ni nebude volat destruktor. (Případné „zbytky“ musíme uklidit sami v příslušném handleru.)

Může se také stát, že výjimka nastane v konstruktoru instance, která má několik předků. V tom případě se zavolají destruktory těch předků, kteří již byli zcela zkonstruováni.

Podívejme se na schematický příklad, ve kterém pro jednoduchost vynecháme těla tříd (úplné znění najdete opět na doplňkové disketě):

```
/*  Příklad C9 — 3   */
class A{/*...*/};
class B{/*...*/};
class C: public A{/*...*/};
class D: public B, public C
 {/*...*/};

void main()
{
  try{
       D d;
  }
  catch(int){
       // ...
  }
}
```

Předpokládejme, že při konstrukci instance *d* třídy *D* nastane výjimka, a to v těle konstruktoru předka *C*. To znamená, že je již zkonstruován předek *B* a nepřímý předek *A*. Pro tyto dva předky se tedy zavolají destruktory. Předek *C* a samozřejmě instance *d* ještě nejsou hotovy, proto se jejich destruktory volat nebudou.

Výjimky a destruktory

Z destruktoru se v C++ nesmí rozšířit výjimka. Pokud se to stane, program skončí voláním standardní funkce *terminate()*.

Důvod tohoto na pohled drastického opatření je jednoduchý. Představte si, že při volání destruktoru instance *h* třídy *H* vznikne výjimka a rozšíří se. V té době ještě není instance zdestruována, a proto se zavolají destruktory všech lokálních instancí, mimo jiné i instance *h*. V jejím destruktoru může ovšem vzniknout znovu výjimka a celý proces by se mohl do nekonečna opakovat. (V těle destruktoru nemáme možnost rozlišit, zda je volán za „normálních“ okolností nebo při výjimce. Musíme tedy vždy počítat s jeho voláním při ošetřování výjimky.)

Výjimky v handlerech

Výjimka může vzniknout kdekoli, třeba také v bloku handleru. Připomeňme si, že v handleru lze také právě ošetřovanou výjimku „poslat dál“ příkazem **throw;**.

Jenže pozor: handler může zachytit pouze výjimku, která vznikla v předchozím pokusném bloku. To znamená, pokud může v handleru vzniknout výjimka, musíme celý hlídaný blok vložit do dalšího pokusného bloku.

Podívejme se na příklad:

```
try {
   // ...
}
catch(long j) {
   // ...
   if(neumim) throw;
}
catch(...) {
   // ...
}
```

Předpokládejme, že v hlídaném bloku vznikne výjimka typu **long**. Je-li v handleru typu **long** splněna podmínka *neumim*, vyvolá se opět výjimka typu **long**. Tuto výjimku ale nezachytí ani handler, ve kterém vznikla, ani následující univerzální handler.

Celý tento hlídaný blok může ale být vnořen do jiného pokusného bloku, který tuto výjimku zachytí.

Neošetřené a neočekávané výjimky

Při troše nepozornosti se může stát, že výjimka vznikne mimo pokusný blok nebo že vznikne sice v pokusném bloku, ale nezachytí ji žádný handler. V takovém případě hovoříme o **neošetřené výjimce** (*unhandled exception*).

Vyskytne-li se neošetřená výjimka, program zavolá standardní funkci *terminate()*. Funkce *terminate()* ukončí program tím, že zavolá funkci *abort()*. Program pak skončí hlášením *Abnormal program termination*.

Chování funkce *terminate()* lze ovšem změnit. Pomocí standardní funkce *set_terminate()* můžeme předepsat, která funkce se má volat místo funkce *abort()*. V každém případě by to ovšem měla být funkce, která ukončí běh programu.

Dalším druhem programátorských nedopatření jsou **neočekávané výjimky** (*unexpected exception*). Neočekávaná výjimka nastane, jestliže se z nějaké funkce rozšíří výjimka jiného typu, než které jsou uvedeny ve specifikaci možných výjimek ve frázi **throw** za hlavičkou. To je zjevně chyba v návrhu programu, a to natolik závažná, že na to program musí radikálně reagovat.

V případě neočekávané výjimky zavolá program standardní Funkci *unexpected()*. Funkce *unexpected()* normálně volá funkci *terminate()*. Pomocí funkce *set_unexpected()* můžeme ale předepsat volání jiné funkce.

Prototypy těchto funkcí a další nástroje, používané při práci s výjimkami, najdeme v hlavičkovém souboru *except.h*. (Pozor: v Borland C++ nebo v Microsoft Visual C++ 2.0 najdeme také hlavičkový soubor *excpt.h*. Ten slouží pro práci se strukturovanými výjimkami jazyka C, o kterých budeme hovořit dále.)

Standardní výjimky

S výjimkami se v C++ budeme setkávat - s tím se musíme smířit, ať se nám to líbí nebo ne. Je jasné, že jich budou využívat především tvůrci knihoven. Také některé z operátorů mohou vyvolávat výjimky. Podívejme se, jaká je situace v dnešním C++.

Standardní operátory

Podle současného návrhu normy může operátor **new** v případě neúspěšné alokace buď vracet 0 nebo vyvolat výjimku typu *bad_alloc* (závisí to na implementaci). Zde se borlandské překladače odchylují od normy; počínaje verzí 4.0 (kdy byly výjimky v BC++ implementovány poprvé) vyvolává operátor **new** při neúspěšné alokaci výjimku typu *xalloc*. Voláním funkce *set_new_handler(0*) si ale můžeme předepsat, že má při neúspěšné alokaci vracet hodnotu 0.

Další dva operátory, které mohou vyvolat výjimky, jsou **typeid**, který může vyvolat výjimku typu *bad_typeid*, a **dynamic_cast**, který může vyvolat výjimku typu *bad_cast*[17]. (O obou operátorech budeme ještě hovořit.)

Standardní knihovna

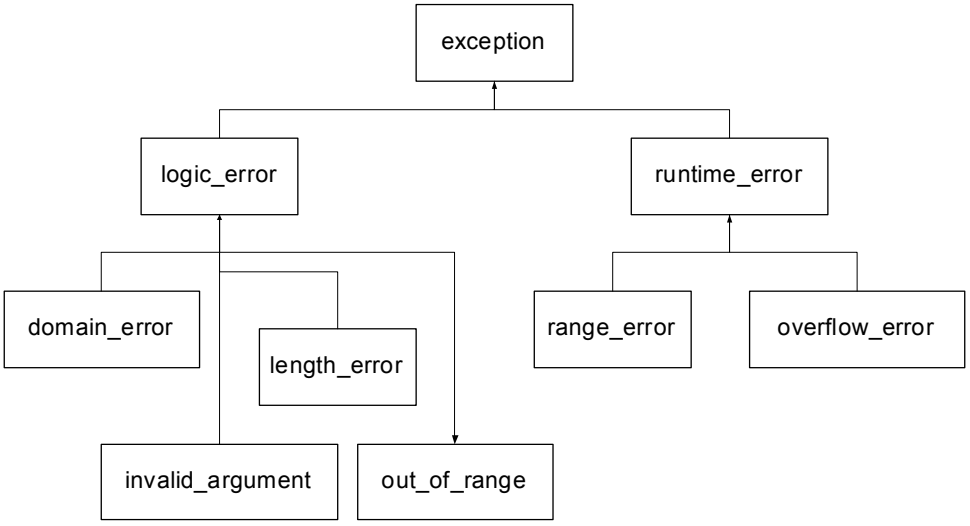
Jedním z výsledků práce standardizační komise, která vyvíjí normu jazyka C++, je i popis standardní knihovny tohoto jazyka. Funkce a metody, které v této knihovně najdeme, mohou vyvolávat výjimky typů, které patří do standardní hierarchie výjimkových typů.

Standardní hierarchie výjimek

Ve standardní knihovně jazyka C++ je definována hierarchie objektových typů pro práci s výjimkami (obr. 9.1). Tato hierarchie obstarává základní třídění chyb na logické, které jsou důsledkem logické chyby v návrhu programu (a kterým se lze v ideálním případě vyhnout) a běhové, kterým se vyhnout nelze, neboť jsou důsledkem např. poruch peri-

[17] Pozor, v Borland C++ jsou jména těchto tříd psána s velkým počátečním písmenem, tj. *Bad_typeid* a *Bad_cast*.

ferních zařízení, chyb uživatele programu apod. V obou skupinách jsou definovány ještě další podtřídy (tedy jemnější členění typů chyb). To dává programátorovi možnost rozhodnout se, jak bude s chybami zacházet.



Obr. 9.1 Hierarchie standardních tříd pro ošetřování výjimek v C++

Deklarace tříd pro práci se standardními výjimkami najdeme ve standardním hlavičkovém souboru *stdexcep.h*. Společným předkem všech těchto typů je třída *exception*, která má dva přímé potomky – třídy *logic_error* (logická chyba) a *runtime_error* (běhová chyba).

Třída *logic_error* má ještě čtyři potomky, kteří definují přesnější třídění logických chyb. Jmenují se *domain_error* (chyba definičního oboru), *invalid_argument* (špatný parametr), *length_error* (chybná délka) a *out_of_range* (mimo rozsah). Třída *runtime_error* má pouze dva potomky, *range_error* a *overflow_error*.

Norma ANSI jazyka C++ zaručuje, že pokud některá z funkcí nebo metod ze standardní (touto normou definované) knihovny C++ vyvolá výjimku, bude to výjimka z této hierarchie. Nic nám samozřejmě nebrání definovat a používat vlastní třídy odvozené od některých členů této hierarchie.

S výjimkou třídy *exception* mají všechny třídy v této hierarchii konstruktor s parametrem typu **char***. Ten umožňuje vložit do instance znakový řetězec s informací o výjimce. Úplně všechny třídy pak mají metodu *what()*, která vrátí objekt typu *string*, obsahující uloženou zprávu. (Má ji i třída *exception*; v tomto případě se vypíše jakási implicitní zpráva.)

V následujícím příkladu vytvoříme instanci standardní třídy *string* a pokusíme se nahradit stou pozici znakem ‘c’. Pokud je řetězec kratší, vyvolá metoda *replace()* jednu ze standardních výjimek:

```
#include <iostream.h>
#include <stdexcept>
#include <string>
int main(){
   string s;
   try{
        s.replace(100,1,1,'c');
   }
   catch(const exception& e){
        cout << "Výjimka: " << e.what() << endl;
   }
   return 0;
}
```

Cena výjimek

Výjimky poskytují jazyku C++ elegantní aparát, s jehož pomocí lze zvládat chybové situace v programech. Patrně nejpůvabnější vlastností je, že se můžeme rozhodnout, kde – na jaké úrovni – budeme výjimky ošetřovat.

Některé výjimky můžeme chytit a ošetřit hned „u zdroje“, v místě, kde vzniknou, ošetření jiných můžeme soustředit třeba až do funkce *main()*. Použijeme-li k přenosu informací o výjimkách vhodně navržené hierarchie objektových typů, výrazně se usnadní třídění chyb a tím i rozhodování, kdy a jak je ošetřit.

Kritické operace lze sdružit do skupin a testy, zda se podařily, lze vynechat: buď dopadly dobře a my se o nic nemusíme starat, nebo se nepodařily, nastala výjimka, a systém přenese řízení do odpovídajícího handleru. Výjimky umožňují řídit se v programování tak trochu filozofií „o problémy se starám, až když nastanou“.

Není to ovšem zadarmo. Přeložené programy s výjimkami jsou rozsáhlejší, i když mohou být rychlejší.

Použijeme-li výjimky, vloží se do kódového segmentu několik tabulek, do kterých si program ukládá potřebné informace. Navíc je třeba připojit nové standardní funkce, které práci s výjimkami umožňují. Také ošetření zásobníku je u funkcí se specifikovanými výjimkami poněkud složitější.

Proto řada překladačů C++ umožňuje používání výjimek zakázat.

10.3 Strukturované výjimky v jazyku C

Strukturované výjimky (*structured exception handling*) nejsou standardní součástí jazyka C; původně byly navrženy pro programy pod Win32 (tedy pod Windows NT nebo Windows 95). Lze je ale s jistými omezeními používat i v programech pro 16bitová prostředí (DOS, Windows 3.1). Setkáme se s nimi např. v Borland C++ počínaje verzí 4.0 a v Microsoft Visual C++ počínaje verzí 2.0.

Ve 32bitovém prostředí můžeme pracovat nejen se softwarovými, ale i s hardwarovými (asynchronními) výjimkami, jako je např. výskyt nesprávné instrukce, dělení nulou apod. V programech pro 16bitová prostředí můžeme pracovat pouze se synchronními výjimkami, tedy s výjimkami vyvolanými v programu voláním funkce

RaiseException(). Pro ošetřování hardwarových výjimek je totiž potřeba spolupráce operačního systému, a tu ani DOS ani obyčejná Wokna nenabízejí.

První přiblížení

Celková koncepce strukturovaných výjimek připomíná výjimky v C++. Protože jsou však určeny pro neobjektové prostředí jazyka C, musí mnohé věci řešit poněkud jinak.

Strukturované výjimky poskytují jednak možnost zachytit a ošetřit **výjimku**, ať už vznikla jakkoli, a za druhé předepsat *koncovku bloku* – skupinu operací, která se provede vždy na závěr bloku, bez ohledu na to, jakým způsobem jej opouštíme. (Přesněji řečeno, koncovka se provede *téměř* vždy; k tomu se ještě vrátíme).

Práce se strukturovanými výjimkami je opět založena na **hlídaném bloku**, který se skládá z **pokusného bloku** a z **handleru** nebo **koncovky** (*termination handler*). K pokusnému bloku smíme tentokrát připojit jen jediný handler nebo jedinou koncovku.

Podobně jako v C++, i tentokrát obsahuje pokusný blok operace, které nemusí dopadnout dobře, tj. při kterých může nastat výjimka. K pokusnému bloku se připojí handler, blok, který případnou výjimku ošetří.

Vznikne-li výjimka, přeruší se operace v pokusném bloku a systém bude hledat vhodný handler. Nenajde-li jej v bloku, ve kterém výjimka vznikla, přejde do bloku nadřízeného – výjimka se **rozšíří** do nadřízeného bloku. Podobně pokud se vhodný handler nenajde ve funkci, ve které výjimka vznikla, rozšíří se výjimka i do funkce, která ji volala atd. – ale to už přece známe: stejným způsobem se šířily i výjimky v C++. Je tu však jeden důležitý rozdíl. Operace v pokusném bloku se pouze přerušily a za jistých okolností se k nim můžeme vrátit. Tuto možnost jsme v C++ neměli.

Jak hledá systém vhodný handler? Zkouší postupně všechny, na které při šíření výjimky narazí. Každý handler obsahuje *vstupní filtr*. To je výraz, který se při vstupu do handleru vyhodnotí a podle něj se systém rozhodne pro některou z následujících možností:

- *Nalezený handler výjimku ošetří*. Pokud tento handler program neukončí, vrátí se po provedení handleru řízení za blok, který výjimku ošetřil. (Pokud není handler v téže funkci jako místo, kde výjimka vznikla, ošetří se také řádně zásobník.)
- *Nalezený handler výjimku odmítne*; výjimka se bude šířit dále, systém bude hledat vhodný handler v nadřazených blocích.
- *Výjimka bude ignorována*. Řízení se po vyhodnocení filtru vrátí na místo, kde výjimka vznikla. Také tohle je ve srovnání s C++ novinka.

Přenos informací o výjimce

Při vzniku výjimky je třeba poslat handleru informace o okolnostech, za kterých výjimka vznikla, o druhu problému apod. Na rozdíl od C++ jsou zde tyto informace předávány v předdefinovaných datových strukturách, takže přístup k nim je vždy stejný.

Stav hardwaru v okamžiku, kdy výjimka vznikla (tj. např. obsahy registrů) popisuje struktura typu *CONTEXT*. Protože Windows NT jsou určena pro počítače s několika

různými typy procesorů (vedle Intelu je to také Alpha a Mips), bude tvar struktury *CONTEXT* záviset na použitém počítači.

Informace o typu výjimky, o tom, zda ji lze ignorovat, a další parametry najdeme ve struktuře typu *EXCEPTION_RECORD*. Tato struktura je v Borland C++ deklarována následujícím způsobem:

```
typedef struct _EXCEPTION_RECORD {
DWORD ExceptionCode;
   DWORD ExceptionFlags;
   struct _EXCEPTION_RECORD __ss *ExceptionRecord;
   LPVOID ExceptionAddress;
   UINT NumberParameters;
   DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;

typedef EXCEPTION_RECORD __ss *PEXCEPTION_RECORD;
```

DWORD je synonymum pro **unsigned long**, zavedené v hlavičkových souborech *windows.h* a *excpt.h* pomocí deklarace **typedef**. Podobně *UINT* je synonymum pro **unsigned int**.

Všimněte si, že struktura *EXCEPTION_RECORD* obsahuje také ukazatel na další strukturu téhož typu. Výjimky se mohou zřetězit, tj. při ošetřování jedné výjimky může nastat další výjimka. Pomocí tohoto ukazatele lze při ošetřování novější výjimky získat přístup k informacím o starší výjimce.

Ukazatele na obě struktury s informacemi o výjimce, *CONTEXT* a *EXCEPTION_RECORD*, jsou uloženy ve struktuře typu *EXCEPTION_POINTERS*. Ta je v Borland C++ deklarována jako

```
typedef struct _EXCEPTION_POINTERS {
   PEXCEPTION_RECORD ExceptionRecord;
   PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Syntax strukturovaných výjimek

Deklarace datových typů a prototypy funkcí, se kterými se při používání strukturovaných výjimek setkáme, jsou v hlavičkových souborech. Programujeme-li pro Win32, vystačíme se souborem *windows.h*, chceme-li používat strukturované výjimky v programech pro DOS, musíme použít soubor *excpt.h*. (Pozor, nespleťte si ho se souborem *exc**e**pt.h*, o kterém jsme hovořili v souvislosti s výjimkami v C++.)

Pokusný blok v Céčku uvádí klíčové slovo **__try**, handler začíná klíčovým slovem **__except**. Koncovku bloku označuje klíčové slovo **__finally**. Všechna tato klíčová slova začínají v microsoftských a borlandských implementacích Céčka pro PC dvěma podtržítky.

V některých implementacích, např. v Microsoft C, můžeme potkat ještě klíčové slovo **__leave**, které přikazuje ihned opustit pokusný blok a přejít do koncovky. Borland C++ toto klíčové slovo neobsahuje.

Syntax hlídaného bloku je

hlídaný blok:
    pokusný_blok handler

pokusný blok:
    __try *blok*

handler:
    __except(*filtr***)** *blok*

filtr:
    výraz

Filtr je výraz, který se vyhodnotí při vstupu do handleru. Musí nabývat jedné ze tří možných hodnot, #definovaných v hlavičkových souborech:

- ✧ Hodnota *EXCEPTION_EXECUTE_HANDLER* (je #definována jako 1) způsobí, že daný handler výjimku ošetří.
- ✧ Hodnota *EXCEPTION_CONTINUE_EXECUTION* (je #definována jako -1) přikazuje tuto výjimku ignorovat. To znamená, že se řízení vrátí na místo, kde výjimka vznikla – a to i v případě, že handler leží v jiné funkci, než ve které výjimka vznikla.
- ✧ Hodnota *EXCEPTION_CONTINUE_SEARCH* (je #definována jako 0) způsobí, že handler výjimku odmítne ošetřit. Systém bude hledat další handler, který by se výjimky ujal.

Všimněte si, že při šíření výjimky se postupně vyhodnotí filtry všech handlerů, na které systém narazí, než najde handler, který se výjimky ujme a ošetří ji.

Schematický příklad práce se strukturovanými výjimkami:

```
__try {
   f(x,y);              /* V těchto funkcích */
   G();                 /* může nastat výjimka */
}
__except(EXCEPTION_EXECUTE_HANDLER) {
       printf("Chyba!!!");
 /* a další akce pro nápravu situace */
}
```

Podobně jako v C++ i zde platí, že dovnitř hlídaného bloku můžeme vstoupit pouze přes jeho otevírací závorku. Dovnitř handleru lze vstoupit pouze přes filtr (tj. když vznikne výjimka, systém najde tento handler a filtr tohoto handleru bude mít hodnotu *EXCEPTION_EXECUTE_HANDLER*). Podobně dovnitř koncovky lze vstoupit pouze po ukončení bloku, ke kterému je připojena. Skoky dovnitř těchto bloků syntaktická pravidla zakazují.

Ven z hlídaného bloku smíme vyskočit pomocí kteréhokoli z příkazů **goto**, **break**, **continue** nebo **return** nebo voláním funkce *longjmp()*. Smíme z něj také pochopitelně odejít „normálně“, tím, že řízení přejde přes ukončovací závorku „}“ bloku.

Příklad

Následující příklad[18] ukazuje, proč bylo v multithreadovém prostředí Win32 potřebné zavést výjimky. Chceme napsat funkci, která by bezpečně kopírovala znakový řetězec z jednoho pole do druhého. Řetězce jsou samozřejmě určeny ukazateli a k vlastnímu kopírování můžeme použít knihovní funkci *strcpy()*. Problém ale je, jak ověřit, zda oba ukazatele ukazují na použitelné oblasti paměti. Pokud bychom použili např. neinicializované ukazatele, skončil by program chybou.

V programech pro 16bitová Windows můžeme použitelnost ukazatelů prověřit pomocí funkcí *IsBadReadPtr()*, resp. *IsBadWritePtr()*. Funkce *SafeCopy()* by tedy mohla vypadat např. takto:

```
char* SafeCopy(char* Cil, char* Zdroj)
{                                   /* Jsou obě pole použitelná? */
  if(Zdroj != NULL && !IsBadReadPtr(Zdroj, strlen(Zdroj)))
       if(Cil != NULL &&! IsBadWritePtr((void*)Cil, strlen(Cil)))
           return strcpy(Cil, Zdroj);
       else
           return NULL;
  else
       return NULL;
}
```

Tato funkce nejprve zjistí, zda ukazatele na zdrojový a cílový řetězec vůbec někam ukazují (porovná je s NULL) a pak otestuje pomocí funkcí *IsBadReadPtr()*, resp. *IsBadWritePtr()* jejich použitelnost pro čtení, resp. zápis. Pokud je něco v nepořádku, vrátí *NULL*, jinak překopíruje zdrojový řetězec do cílového pole a vrátí jeho adresu.

Pod obyčejnými Windows je takovéto řešení naprosto v pořádku. Jenže v prostředí Windows NT, kde může běžet zároveň několik paralelních „vláken“ (threadů) programu, je všechno jinak.

Problém je v tom, že funkce *SafeCopy()* **nejprve** prověří použitelnost obou ukazatelů a **teprve pak** je použije. Budou-li *text1* a *text2* dva globální řetězce, může se stát, že v jednom vláknu (threadu) si funkce *SafeCopy()* pracně ověří, že jsou oba ukazatele v pořádku, a mezitím druhé vlákno jejich stav změní. Při kopírování pak stejně dojde k chybě[19].

Strukturované výjimky umožňují napsat funkci *SafeCopyEx()*, která se postará o bezpečné kopírování, velice jednoduše a přehledně:

```
char * SafeCopyEx(char* Cil, char* Zdroj)
{                                   /* Zkusíme to ...*/
  __ try {
```

[18] Pochází z článku [7].

[19] To je jedna z typických chyb, jakých se lze dopustit, jestliže máme v programu několik paralelně běžících součástí. Pokud mohou jednotlivé součásti zároveň přistupovat ke globálním datům, je téměř jisté, že si budou provádět naschvály.

```
        return strcpy(Cil, Zdroj);
    }
                                    /* A když se to nepovede: */
    __except(EXCEPTION_EXECUTE_HANDLER) {
        return NULL;
    }
}
```

Funkce *SafeCopyEx()* se prostě pokusí řetězce zkopírovat. Pokud se to podaří, vrátí adresu výsledku. Nastanou-li nějaké problémy, vznikne výjimka a handler se postará, aby tato funkce vrátila *NULL*.

Jedinou nevýhodou tohoto řešení je, že je můžeme použít pouze pod Win32, neboť vznik strukturovaných výjimek při událostech, jako je porušení ochrany paměti, vyžaduje spolupráci operačního systému.

Všimněte si, že jsme zde změnili vlastně i filozofii přístupu. Zatímco u funkce *SafeCopy()* jsme se snažili myslet předem na všechny možné obtíže, zde jsme se rozhodli starat se o problémy až ve chvíli, kdy nějaké nastanou.

Navíc i zdrojový text je podstatně přehlednější a výsledný program bude rychlejší, neboť odpadnou předběžné testy.

Jak vznikají strukturované výjimky

Již víme, že pod Win32 můžeme pracovat jak se softwarovými tak i s hardwarovými výjimkami. V ostatních prostředích (16bitová Wokna, DOS) můžeme pracovat pouze se softwarovými výjimkami.

Softwarové výjimky

Softwarovou výjimku způsobíme voláním funkce *RaiseException()*, která má (v Borland C++) prototyp

```
void __cdecl __far
RaiseException ( DWORD kod, DWORD priznak, DWORD pocParam,
                 DWORD poleParam);
```

První parametr, *kod*, udává kód výjimky. Kódy výjimek si definujeme sami. Při vyhodnocování filtru pak můžeme kód, předaný funkci *RaiseException(),* zjistit pomocí funkce *GetExceptionCode()*.

Druhý parametr, *priznak*, určuje, zda jde o pokračovatelnou výjimku (kterou lze ignorovat) nebo o nepokračovatelnou výjimku (která musí být ošetřena). Tento parametr může mít buď hodnotu *EXCEPTION_CONTINUABLE* (pokračovatelná výjimka; tato konstanta je v *excpt.h* #definována jako 0) nebo *EXCEPTION_NONCONTINUABLE* (nepokračovatelná výjimka; je #definována jako 1). Pokračovatelnou výjimku, tj. výjimku s nastaveným příznakem *EXCEPTION_CONTINUABLE*, lze ignorovat.

Má-li tedy filtr hodnotu *EXCEPTION_CONTINUE_EXECUTION*, vrátí se řízení na místo výjimky a program bude pokračovat, jako by se nechumelilo. (To je nadsázka, neboť mezitím se vyhodnotil filtr a tím mohly vzniknout různé vedlejší efekty. Navíc v době, kdy tento díl píšeme, pouze vytrvale prší.)

Jestliže se pokusíme ignorovat nepokračovatelnou výjimku, vznikne výjimka nová.

Poslední dva parametry umožňují přenášet další informace. Ty uložíme do pole *poleParam*. Parametr *pocParam* udává počet parametrů uložených v tomto poli.

Hardwarové výjimky

Připomeňme si, že hardwarové výjimky jsou k dispozici pouze v programech pro Win32, neboť vyžadují spolupráci operačního systému.

Hardwarová výjimka vznikne, jestliže při výpočtu nastane některá ze situací, které znamenají (nebo mohou znamenat) chybu. U některých situací si můžeme předepsat, zda se mají pokládat za chybu (a vyvolávat výjimku) nebo zda se mají přejít bez povšimnutí.

V souboru *excpt.h* je #definováno 16 kódů hardwarových výjimek. Zastavíme se krátce u některých z nich.

Výjimka *EXCEPTION_ACCESS_VIOLATION* vznikne, jestliže se program pokusí číst data z oblasti, do které nemá přístup, nebo data do takové oblasti uložit. Tuto výjimku může způsobit např. dereferencování ukazatele, který obsahuje *NULL*.

Výjimka *EXCEPTION_INT_DIVIDE_BY_ZERO* vznikne jako důsledek celočíselného dělení nulou.

Výjimka *EXCEPTION_PRIV_INSTRUCTION* vznikne, jestliže se program pokusí provést privilegovanou instrukci, tedy instrukci, která je povolena pouze v režimu jádra Windows NT (*kernel mode*). Tuto výjimku může způsobit např. pokus o přímý zápis dat na port:

```
asm{
  out DX, AX
}
```

Něco takového si můžeme dovolit v programu pro DOS, v programu pro 32bitová prostředí však nikoli. Tam je tato instrukce považována za privilegovanou, lze ji použít pouze v režimu jádra (*kernel mode*). Aplikace ovšem běží v uživatelském režimu (*user mode*) a tam pokus o přímý zápis na port skončí výjimkou.

Výjimka *EXCEPTION_ILLEGAL_INSTRUCTION* vznikne, jestliže se program pokusí provést neplatnou instrukci, tedy instrukci, jejíž operační kód nemá pro daný procesor význam. Tato výjimka obvykle znamená ošklivou chybu v programu – např. pokaženou návratovou adresu na zásobníku.

Výjimky také mohou vznikat při operacích s reálnými čísly při přetečení (*EXCEPTION_FLT_OVERFLOW*), podtečení (*EXCEPTION_FLT_UNDERFLOW*), dělení nulou (*EXCEPTION_FLT_DIVIDE_BY_ZERO*) apod. Tyto výjimky jsou však pod Win32 normálně „vypnuty“, systém je nevyvolává. Pokud bychom si přáli, aby koprocesor 80x87 při těchto událostech výjimky generoval, musíme mu to předepsat pomocí funkce *control87*, jejíž prototyp je v hlavičkovém souboru *float.h*.

Filtrování výjimek

Jak název napovídá, filtr filtruje výjimky, tj. určuje, které z nich je daný handler schopen ošetřit. Již víme, že to je výraz, který zapisujeme v závorkách za klíčové slovo *__except* a jehož vyhodnocením musíme dostat jednu z hodnot -1, 0 nebo 1.

Jako filtr můžeme použít jednoduchý výraz, např. konstantu, která přikazuje výjimku ošetřit:

```
__try {
   f(x);          /* příkazy, které mohou */
   g(x);          /* vyvolat výjimku */
}
__except(EXCEPTION_EXECUTE_HANDLER) {
   /* Ošetření výjimky */
   printf("Zkus to znovu.");
   znovu = 1;
}
```

Tento handler zachytí bez výjimky veškeré výjimky. Ovšem takovéto „černé díry“ na výjimky se používají spíše zřídka, neboť obvykle se rozhodujeme podle kódu a případně dalších parametrů výjimky. Proto se jako filtry zpravidla používají složitější výrazy.

Stačí-li nám k rozhodnutí ve filtru znát kód výjimky, použijeme funkci *GetExceptionCode()*. Pokud potřebujeme i další informace o výjimce, použijeme funkci *GetExceptionInformation(),* jež vrací ukazatel na strukturu *EXCEPTION_POINTERS* (v ní jsou uloženy ukazatele na struktury *CONTEXT* a *EXCEPTION_INFORMATION*).

Obě tyto funkce, *GetExceptionCode()* a *GetExceptionInformation()*, jsou bez parametrů a smíme je volat pouze ve filtru, nikde jinde. (Ve skutečnosti to jsou makra, ale to je obvykle nepodstatné.)

Typickým příkladem jednoduchého filtrového výrazu může být následující konstrukce:

```
__try {
   G(z);
}
__except(GetExceptionCode() > KOD_CHYBY_1 ?
        EXCEPTION_CONTINUE_EXECUTION :
        EXCEPTION_EXECUTE_HANDLER) {
   /* Ošetření výjimky */
}
```

Je-li kód výjimky větší než předdefinovaná konstanta *KOD_CHYBY_1*, bude se výjimka ignorovat, jinak se tento handler provede.

Jako filtr smíme také použít zápis funkce, která vrátí některou z dovolených hodnot filtru – *filtrovací funkce*. Této funkci můžeme kód výjimky nebo ukazatele na strukturu *EXCEPTION_POINTERS* předat jako parametry.

Jako ukázku si uvedeme krátký program. Nehledejte v něm žádný hlubší smysl, jeho jediným cílem je předvést, jak program se strukturovanými výjimkami funguje.

V tomto programu použijeme pouze softwarové výjimky, takže jej můžeme přeložit jako dosovskou aplikaci např. pomocí Borland C++ 4.x.

```
/*  Příklad C9 — 4.C  */
#include <excpt.h>
#include <stdio.h>

/* Definujeme chybové kódy */
```

```
#define VYJIMKA_0 0
#define VYJIMKA_1 1
#define VYJIMKA_2 2
#define VYJIMKA_3 3

/* Filtrovací funkce */
int Filtr(EXCEPTION_POINTERS *ep) {
  if(ep->ExceptionRecord->ExceptionFlags == EXCEPTION_NONCONTINUABLE)
       return EXCEPTION_EXECUTE_HANDLER;
  else switch (ep->ExceptionRecord->ExceptionCode)
  {                              /* Hodnota podle kódu chyby */
       case VYJIMKA_0:
           return EXCEPTION_CONTINUE_EXECUTION;
       case VYJIMKA_1:
       case VYJIMKA_2:
           return EXCEPTION_EXECUTE_HANDLER;;
       case VYJIMKA_3:
       default:
           return EXCEPTION_CONTINUE_SEARCH;
  }
}

int main()
{
  int i;
  for(i = 0; i < 5; i++)
  __try {
       printf("Je tu VÝJIMKA_%i\n", i);
       RaiseException(i, 0, 0, 0);          /* *** */
       printf("Necháme ji plavat\n");
  }
  __except(Filtr(GetExceptionInformation())) {
       printf("Ošetřujeme výjimku\n");
  }
  return 0;
}
```

Hlavní program v cyklu vyvolává výjimky s kódy 0 – 4. Ve filtru handleru voláme funkci *Filtr()*, které předáme jako parametr ukazatel na struktury s informacemi o výjimkách. Funkce *Filtr()* si v této struktuře najde kód chyby a podle jeho hodnoty pak přikáže výjimku ignorovat (chyba s kódem 0), ošetřit (kódy 1 a 2) nebo hledat další handler (kód 3). Výstup programu *C9–05.C* bude

```
Je tu VÝJIMKA_0
Necháme ji plavat
Je tu VÝJIMKA_1
Ošetřujeme výjimku
Je tu VÝJIMKA_2
Ošetřujeme výjimku
Je tu VÝJIMKA_3
Abnormal program termination
```

Všimněte si, že v případě výjimky s kódem 0 se program chová, jako by se (téměř) nic nestalo – po vyhodnocení filtru se vrátí za místo, kde výjimka vznikla, a pokračuje vesele dál. V případě kódů 1 a 2 se zavolá handler.

Při kódu 3 vznikne *neošetřená výjimka*, neboť náš handler tuto výjimku odmítl ošetřit a systém jiný handler už nenašel. Systém tedy zavolal funkci *terminate()* a ukončil program.

Nepokračovatelné výjimky

Výjimky, které mají v informační struktuře *EXCEPTION_RECORD* nastaven příznak *EXCEPTION_NONCONTINUABLE* (tj. hodnotu 1), jsou *nepokračovatelné*. Takovou výjimku nesmíme ignorovat. Pokud bychom se o to pokusili, pokud by filtr vrátil hodnotu *EXCEPTION_CONTINUE_EXECUTION*, vyvolal by systém speciální výjimku s kódem *STATUS_NONCONTINUABLE_EXCEPTION*.

Koncovka bloku

Výjimka znamená předčasné ukončení pokusného bloku. To může způsobit řadu problémů. Pokud jsme při vstupu do bloku alokovali paměť, je třeba ji před opuštěním bloku uvolnit; pokud jsme otevřeli soubor, je třeba jej uzavřít, jinak přijdeme o data; lze najít ještě mnoho dalších problémů, které může výjimka způsobit.

Podobné problémy si můžeme ale způsobit i v případě, že neuváženě opustíme blok pomocí některého z příkazů **goto**, **break**, **continue**, **return** nebo voláním funkce *longjmp()*.

Bloky s koncovkou představují možnost, jak se s těmito problémy jednoduše vyrovnat. Koncovka bloku se totiž provede **téměř** vždy. Brzy si povíme, co se skrývá za oním „téměř“.

Syntax koncovek

Blok s koncovkou se skládá z pokusného bloku, stejného jako v případě výjimek, a z koncovky. Syntax bloku s koncovkou popíšeme takto:

blok_s_koncovkou:
    pokusný_blok koncovka

pokusný_blok:
    __try *blok*

koncovka:
    __finally *blok*

Již jsme si řekli, že k pokusnému bloku můžeme připojit vždy buď jeden handler nebo jednu koncovku. Ovšem blok s koncovkou můžeme vložit do hlídaného bloku nebo do jiného bloku s koncovkou a podobně hlídaný blok můžeme vnořit do bloku s koncovkou nebo do jiného hlídaného bloku.

Koncovka bloku se provede, jestliže hlídaný blok skončí

- ✧ normálně, tedy tím, že řízení přejde přes jeho uzavírací závorku "{",
- ✧ vyvoláním výjimky,
- ✧ provedením některého ze skokových příkazů (**break**, **continue**, **goto**, **return**, volání funkce *longjmp()*).

Koncovka bloku se neprovede, jestliže v něm zavoláme některou z funkcí, které ukončují běh programu nebo jednoho threadů (paralelních vláken). To znamená, že pokud program pro Win32 nebo jeho thread skončí např. voláním některé z funkcí *ExitThread()*, *ExitProcess()*, *TerminateThread()* nebo *TerminateProcess()* v pokusném bloku, koncovka tohoto bloku se neprovede.

V programech pro DOS se koncovka neprovede, jestliže v pokusném bloku zavoláme např. funkci *exit()* nebo *_terminate()*.

Připomeňme si, že dovnitř koncovky bloku se nesmí skákat.

Ukážeme si schéma typického použití bloku s koncovkou. Pokud budete tento program překládat jako konzolovou aplikaci pro Win32, je třeba nahradit hlavičkový soubor *excpt.h* souborem *windows.h*.

```
/*  Příklad C9 – 5.C   */
#include <excpt.h>
#include <stdio.h>
#include <alloc.h>

#define N 1000

int *pole;
/* ...*/

void G(int i) {
  __try{                                /* zde alokujeme paměť */
       pole = (int*)malloc(sizeof(int)*N);
       /* Operace, díky kterým může blok skončit předčasně */
       if (i == 1) return;
       if (i == 2) RaiseException(1,0,0,0);
  }
  __finally{
       free(pole);
       printf("úklid v koncovce\n");
  /* zde za sebou uklidíme (uvolníme paměť apod.) */
 }
  /* Výpočet pokračuje */
  printf("normální konec funkce G()\n");
}

int main() {
  int i;
  __try{
       for(i = 0; i < N; i++)
       G(i);
  }
  __except(EXCEPTION_EXECUTE_HANDLER){
       printf("Je tam nějaká výjimka...\n");
  }
```

```
   return 0;
}
```

Ve funkci *G()* alokujeme paměť pomocí standardní céčkovské funkce *malloc()*. Abychom měli jistotu, že se vždy uvolní, vložili jsme volání funkce *free()* do koncovky. To znamená, že se uvolní nejen při normálním ukončení funkce *G()*, ale i v případě, že tato funkce skončí výjimkou nebo předčasným použitím příkazu **return**.

Skončil blok normálně?

Dohodneme se, že pokud řízení v pokusném bloku přejde přes jeho uzavírací závorku „}“, budeme říkat, že tento blok skončil *normálně*. Ukončení pokusného bloku po vzniku výjimky nebo vyskočením z bloku pomocí kteréhokoli z příkazů pro přenos řízení budeme označovat jako *abnormální*.

Při vstupu do koncovky někdy potřebujeme zjistit, zda pokusný blok skončil normálně nebo abnormálně. K tomu můžeme použít funkci *AbnormalTermination(),* jejíž prototyp má tvar

```
int AbnormalTermination(void);
```

Tato funkce vrátí 0, jestliže pokusný blok skončil normálně, a v opačném případě vrátí nenulovou hodnotu. Ukážeme si její použití na jednoduchém příkladu:

```
 FILE *F;
__try{
   F = fopen("C:\\DATA\\SOUB.DTA", "a+");
   /* ...*/
}
__finally{
   if(AbnormalTermination()) {
        fclose(F);
        /* ...a další akce potřebné při abnormálním ukončení bloku */
   }
   /* ...*/
}
```

Jestliže pokusný blok skončil normálně, nemáme důvod soubor zatím uzavírat. Proto při vstupu do koncovky zjišťujeme, zda pokusný blok proběhl v pořádku nebo zda skončil nějak problematicky.

Koncovky a výjimky

Vzhledem k tomu, že k pokusnému bloku můžeme připojit pouze jeden handler nebo pouze jednu koncovku, nezbývá, než tyto bloky do sebe vnořovat. Potom ale mohou vzniknout nejasnosti kolem pořadí, v jakém se budou vyhodnocovat filtry, handlery a koncovky.

Dvě koncovky

Začneme nejjednodušším případem. Vnoříme-li do sebe dva bloky s koncovkou a ve vnitřním nastane událost, která způsobí opuštění obou pokusných bloků, provede se nejprve koncovka vnitřního bloku, pak koncovka vnějšího bloku.

V následujícím kousku programu způsobí nenulová hodnota proměnné *prusvih* předčasný návrat do volající funkce:

```
/*  Příklad C9 — 6.C   */
/* Kombinace dvou koncovek */
int pokus(int prusvih){
  __try{
       printf("Vnější blok\n");
       __try {
           printf("Vnitřní blok\n");
           /* Zabal to a zmiz... */
           if (prusvih) return 0;
       }
       __finally {
           printf("Koncovka vnitřního bloku\n");
       }
  }
  __finally{
       printf("Koncovka vnějšího bloku\n");
  }
  return 1;
}

int main(){
  int x;
  /* zde definujeme hodnotu x */
  pokus(x);
  /* ...*/
  return 0;
}
```

Výstup tohoto programu bude

```
Vnější blok
Vnitřní blok
Koncovka vnitřního bloku
Koncovka vnějšího bloku
```

a to bez ohledu na skutečnou hodnotu *x*, neboť koncovky obou pokusných bloků se provedou v každém případě.

Koncovka a výjimka

Zkombinujeme-li pokusný blok s blokem s koncovkou, situace se zkomplikuje. Podíváme se opět na příklad, který nám ukáže, jak se program v takovém případě chová.

```
/*  Příklad C9 — 7.C   */
/* Handler a koncovka */
#include <excpt.h>
```

```
#include <stdio.h>

int main() {
   __try{
        intf("Vnější blok s handlerem\n");
        __try {
            printf("Vnitřní blok s koncovkou\n");
            RaiseException(0,0,0,0);
            printf("Jsme za výjimkou\n");
        }
        __finally {
            printf("Koncovka vnitřního bloku\n");
        }
   }
   __except(printf("Jsme ve filtru\n"), EXCEPTION_EXECUTE_HANDLER)
   {    printf("Jsme v handleru vnějšího bloku\n");
   }
   printf("Šmytec");
   return 0;
}
```

Ve vnitřním hlídaném bloku vznikne výjimka. Tím provádění tohoto bloku skončí a systém bude hledat vhodný handler. Najde jej u vnějšího pokusného bloku.

To znamená, že se nejprve vyhodnotí filtr nalezeného handleru, a když se zjistí, že tento handler výjimku přijme a ošetří, vrátí se řízení do koncovky pokusného bloku. Teprve potom se přejde do těla handleru a ošetří se výjimka.

Výstup předchozího programu proto bude

```
Vnější blok s handlerem
Vnitřní blok s koncovkou
Jsme ve filtru
Koncovka vnitřního bloku
Jsme v handleru vnějšího bloku
Šmytec
```

Poznámka:
Handler není zápis volání funkce, takže ve filtru můžeme klidně používat operátor čárka.

Situace bude trochu jiná, jestliže filtr přikáže výjimku ignorovat. Zaměníme-li v předchozím příkladu hodnotu *EXCEPTION_EXECUTE_HANDLER* za *EXCEPTION_CONTINUE_EXECUTION*, vrátí se po vyhodnocení filtru řízení za místo, kde výjimka vznikla, a program normálně pokračuje. Handler se tedy neprovede. Provedl se ovšem filtr, takže výstup tohoto programu bude mít tvar

```
Vnější blok s handlerem
Vnitřní blok s koncovkou
Jsme ve filtru
Jsme za výjimkou
Koncovka vnitřního bloku
Šmytec
```

Pokud bychom ve filtru použili hodnotu *EXCEPTION_CONTINUE_SEARCH*, provedla by se koncovka vnitřního pokusného bloku a pak by program skončil chybou, neboť výjimka zůstala neošetřená. V takovém případě bychom dostali

```
Vnější blok s handlerem
Vnitřní blok s koncovkou
Jsme ve filtru
Koncovka vnitřního bloku
Abnormal program termination
```

Neošetřené výjimky

Podobně jako u výjimek v C++, i u strukturovaných výjimek se může stát, že některou výjimku nezachytí žádný handler. Taková výjimka se nazývá *neošetřená* (*unhandled exception*).

Neošetřené výjimky je rozumné „chytat“ v hlavním programu, tj. ve funkci *main()* (nebo *WinMain()*, pokud programujeme pro Windows) pomocí speciálního filtru

```
long __far __pascal
UnhandledExceptionFilter(PEXCEPTION_POINTERS ep);
```

Standardní varianta funkce *UnhandledExceptionFilter()* vypíše upozornění, že byla zachycena neošetřená výjimka, a vrátí hodnotu *EXCEPTION_EXECUTE_HANDLER*; při ladění však vrátí *EXCEPTION_CONTINUE_SEARCH*. Tím upozorní programátora, že vznikla neošetřená výjimka, a donutí ho řádně ji ošetřit.

Můžeme ovšem definovat vlastní filtr pro zpracování neošetřených výjimek a použít jej místo funkce *UnhandledExceptionFilter()*. Nová filtrovací funkce musí ovšem být stejného typu jako funkce *UnhandledExceptionFilter()*.

Nový filtr pro neošetřené výjimky předepíšeme pomocí funkce *SetUnhandled-EcxeptionFiltr(),* jejímž parametrem je ukazatel na nový filtr a která vrací ukazatel na starý filtr. Příklad:

```
/*  Příklad C9 – 8.C   */
#include <excpt.h>
#include <stdio.h>

/* schéma použití vlastního filtru pro neošetřené výjimky */
/* Nový filtr */
long far pascal
Filtr(PEXCEPTION_POINTERS Ei)
{
  printf("filtr pro neošetřené výjimky\n");
  return EXCEPTION_EXECUTE_HANDLER;
}

int main()
{
  SetUnhandledExceptionFilter(Filtr);
  /* Vnější blok pro chytání neošetřených výjimek */
  __try {
       /* Vlastní program */
       RaiseException(1,0,0,0);
```

```
  }
  __except(UnhandledExceptionFilter(GetExceptionInformation())){
       printf("Výjimka\n");
       /* Handler pro neošetřené výjimky */
  }
  return 0;
}
```

I když ve filtru předepisujeme použití standardního filtru pro neošetřené výjimky, bude se volat náš vlastní, funkce *Filtr()*.

Poznamenejme, jak koncovky tak i handlery jsou součástí bloku, do kterého jsou vnořeny. To znamená, že mohou používat jeho lokální proměnné. Na druhé straně lokální proměnné z místa, kde výjimka vznikla, nemusí být v handleru či koncovce k dispozici (to se stane např. v případě, že výjimku ošetřujeme v jiné funkci, než ve které vznikla).

10.4 Strukturované výjimky a C++

Strukturované výjimky byly navrženy jako rozšíření jazyka C. Původní dokumentace[20] o jejich vztahu k jazyku C++ vůbec nehovoří. Nicméně vzhledem ke stále rostoucí oblibě tohoto jazyka se musí každý překladač, který implementuje strukturované výjimky, vyrovnat s jejich použitím v C++. My si zde krátce povíme, jak je to v Borland C++.

Borlandské překladače počínaje verzí 4.0 umožňují používat strukturované výjimky i v programech v C++, avšak s následujícími omezeními:

- ✧ Klíčové slovo **__try** je třeba nahradit pluskovým **try**.
- ✧ Koncovky bloků **(__finally**) nelze v C++ používat.
- ✧ V jednom modulu v C++ lze vedle sebe používat strukturované výjimky a výjimky z C++, avšak výjimky, vyvolané voláním funkce *RaiseException(),* může zachytit a ošetřit pouze céčkovský handler **__except** a pluskové výjimky, vyvolané pomocí klíčového slova **throw**, může zachytit a ošetřit pouze pluskový handler **catch**.

Při vzniku strukturované výjimky v programu v C++ zaručuje Borland C++ volání destruktorů lokálních objektů. Tím lze nahradit koncovky bloků, které v C++ nelze používat.

10.5 Výjimky v Delphi

Implementace výjimek v Delphi nezapře inspiraci jazykem C++. Pokud jde ovšem o syntax, podobají se spíše strukturovaným výjimkám z jazyka C. To je dáno mimo jiné tradičním pascalským způsobem zacházení s objekty.

V Object Pascalu se setkáme nejen s přenosem informací o vzniklé výjimce pomocí objektů a s předdefinovanou hierarchií objektů (typu **class**) pro tento účel, ale i s koncovkami bloků, tedy s úseky kódu, u kterých máme zaručeno, že se provedou, ať skončí

[20] [8], kap. 2.

blok normálně nebo výjimkou. Ve srovnání s výjimkami v C++ zde chybí automatické volání destruktorů (i když to není tak úplně pravda).

Použití objektů, a zejména dědičnosti, umožňuje programátorovi snadno výjimky třídit a rozhodnout se, na jaké úrovni se bude chybami zabývat. Některé chyby – tedy některé druhy výjimek – může ošetřovat bezprostředně v místě vzniku, jiné třeba až na úrovni hlavního programu.

Vznik výjimky

Výjimky v Delphi vznikají především v důsledku chyb při běhu programu. Události, které ve starších verzích Turbo Pascalu způsobily ukončení programu a vypsání zprávy typu *Run time error 203 at ...*, nyní vyvolají výjimku: objeví se dialogové okno oznamující, že program způsobil chybu a končí (obr. 9.2). Při ladění v prostředí se předtím objeví ještě okno, které ohlásí, co se stalo, a umožní program krokovat (obr. 9.3). To platí ovšem pouze v případě, že výjimku neošetříme.

Obr. 9.2 Takhle vypadá neošetřená výjimka v Delphi

Obr. 9.3 Při ladění v prostředí dostaneme toto okno

Výjimku způsobí tedy např. dělení nulou, neúspěch při alokaci paměti nebo při práci se soubory, pokus o výstup prostřednictvím procedury *writeln*, pokud zapomeneme použít jednotku (unit) *WinCrt* a jiná nedopatření. Vedle toho můžeme výjimku v programu vyvolat sami příkazem **raise**. Jeho syntax je

raise *instance*;

nebo

raise;

Za klíčovým slovem **raise** může následovat instance objektového typu **class** (resp. volání konstruktoru, které instanci vytvoří). Tato instance ponese informace o vzniklé výjimce. Pokud bychom např. chtěli vyvolat výjimku typu *ERangeError*, mohli bychom napsat

```
raise ERangeError.Create('Hodnota X mimo povolený rozsah');
```

ERangeError je jeden z předdefinovaných typů pro přenos informací o chybě.

Výjimkové třídy

Ve standardní knihovně Object Pascalu najdeme hierarchii předdefinovaných tříd pro přenos informací o výjimkách (celkem 24 tříd). Jejich společným předkem je třída *Exception* (výjimka). Ta má mimo jiné konstruktor *Create(const Msg: string)* s jedním parametrem, do kterého se ukládá chybové hlášení, a samozřejmě virtuální destruktor *Destroy*. Chybové hlášení se uloží do property *message*.

Od této třídy je odvozena dlouhá řada dalších tříd, umožňujících podrobnější klasifikaci chyb. Jejich hierarchii najdete na obr. 9.4. Například přímými potomky třídy *Exception* jsou *EIntError* pro popis chyb při celočíselných operacích, *EMathError* pro popis chyb při operacích s reálnými čísly, *EProcessorException* pro chyby procesoru a další. Od třídy *EMathError* jsou dále odvozeny *EOverflow* (přetečení v aritmetice reálných čísel), *EZeroDivide* (dělení nulou), *ERangeError* (nastane např. při pokusu o výpočet odmocniny ze záporného čísla) atd. Podobně od třídy *EProcessorFault* jsou odvozeny třídy *EPageFault* (chyba stránkování procesoru), *EGPFault* (porušení obecné ochrany paměti, tedy pokus o neoprávněný přístup do paměti), *EStackFault* (přeplnění zásobníku) atd.

Obr. 9.4 Hierarchie standardních výjimkových tříd v Delphi

Takovéto členění umožňuje programátorovi chyby třídit a vybrat si úroveň, na které bude na jednotlivé chyby reagovat. Jak totiž dále uvidíme, výjimku v Delphi „zachytáváme“ podle typu objektu, který o nich nese informace. Pokud někomu toto členění chyb nevystačí, může si od kterékoli z těchto tříd odvodit další potomky.

Podle typu objektu, který při výjimce vznikne a který ponese informace o ní, hovoříme o typu výjimky.

Jak výjimku zachytit

K zachycení výjimky slouží konstrukce **try/except**. Její syntax je

try
    příkazy
except
    výjimkový blok
end

Část mezi klíčovými slovy **try** a **except** označíme jako *hlídanou sekci*. Pokud při provádění příkazů v hlídané sekci vznikne výjimka, provádění příkazů v ní skončí a řízení přejde na *výjimkový blok*, tedy sekci mezi **except** a **end**. V případě, že tento blok výjimku ošetří a neukončí program, bude jeho provádění pokračovat za ním.

Konstrukce **try/except** mohou být do sebe vnořeny. Pokud jeden výjimkový blok vzniklou výjimku nedokáže ošetřit, bude systém hledat nadřízený výjimkový blok, který

by se jí ujal. Pokud žádný nenajde, ujme se výjimky standardní systémový výjimkový blok, který vypíše zprávu o chybě a ukončí program. (Obrázky 9.2 a 9.3 ukazují vlastně reakce standardního systémového výjimkového bloku.)

Výjimkový blok

Výjimkový blok tedy je úsek programu mezi klíčovými slovy **except** a **end**. Nejjednodušší výjimkový blok má tvar

except
    příkazy
end;

Tento blok zachytí bez výjimky všechny výjimky – patří tedy do kategorie programátorských černých děr. Je jasné, že jde o konstrukci poněkud nebezpečnou, zejména když vznikne výjimka, o které jsme to vůbec nepředpokládali.

Výjimkový blok, který dokáže rozlišovat typy výjimek, obsahuje jeden nebo několik handlerů. Handler je v Delphi konstrukce tvaru

on *ident*: *třída* **do příkaz**

nebo

on *třída* **do** *příkaz*

Výjimkový blok může končit konstrukcí

else *příkaz*

Abychom si mohli snáze povídat o tom, jak handlery fungují, podíváme se na příklad výjimkového bloku s handlery:

```
try
 {...}
except
  on E: EIntOverflow do CelociselnyPrusvih(E);
  on E: EOverflow do RealnyPrusvih(E);
  on MathError do ZpracujMatChybu;
else
  UkonciProgram
end;
```

Za **on** následuje identifikátor objektu, který nese informace o výjimce, a za dvojtečkou pak jméno třídy. Za **do** pak uvedeme příkaz (zpravidla složený příkaz nebo volání procedury), který má výjimku odpovídajícího typu ošetřit. Identifikátor objektu lze vynechat. Klíčové slovo **else** uvádí handler, který se ujme výjimek, jež nemůže jiný handler ošetřit. Část **else** můžeme vynechat.

Jestliže v našem příkladu v hlídané sekci žádná výjimka nevznikne, provedou se všechny její příkazy, přeskočí se výjimkový blok a program bude pokračovat za ním. Jestliže v hlídané sekci vznikne nějaká výjimka, vytvoří se objekt, který o ní ponese informace, přeskočí se zbytek této sekce a řízení přejde do výjimkového bloku.

Výjimku může ošetřit handler, ve kterém jsme uvedli odpovídající typ objektu, který přenáší informace. Jestliže tedy vznikne výjimka typu *EIntOverflow*, ujme se jí v našem příkladu hned první handler a zavolá se procedura *ZpracujInt(E)*. Této proceduře se předá jako parametr objekt, který nese informace o dané výjimce. Odvoláváme se na něj identifikátorem *E*, zavedeným v deklaraci handleru.

Jestliže vznikne výjimka typu *EInvalidOp*, ujme se jí třetí handler. Handlery se testují v pořadí, v jakém byly deklarovány, a výjimku přijme ten, pro který jsme deklarovali stejný typ výjimky nebo některého z předků – a protože je třída *EInvalidOp* je potomkem třídy *EMathError*, ujme se i výjimky typu *EInvalidOp*.

Pokud by v chráněné sekci vznikla např. výjimka typu *EInvalidPointer*, ujme se jí handler za **else**, neboť ten zachytí všechny výjimky, které nezachytí handlery deklarované před ním.

Handler za **else** je tedy opět černá díra na výjimky – tedy potenciálně nebezpečná konstrukce. Můžeme jej ale vynechat.

Poznamenejme, že objekt, přenášející informace o výjimce, se po ošetření výjimky automaticky zlikviduje, systém pro něj zavolá destruktor *Destroy*.

Pošli to dál

Občas se stane, že na určité úrovni můžeme výjimku ošetřit jen částečně a potřebujeme ji „poslat dál“. K tomu použijeme v handleru samotné klíčové slovo **raise**:

```
function Fun(i: integer): Beta;
begin
 try
  Result := Beta.Create;
  while i > 0 do begin
       Result.Bubu(i);
  end;
  except
       on EIntOverflow do begin
            Result.Destroy;
            raise;
       end;
  end;
end
```

Tato funkce na počátku alokuje objekt typu *Beta*, který se chystá po nějakých úpravách vrátit. Může se ale stát, že se tyto úpravy z jakéhokoli důvodu nepodaří a v metodě *Bubu* vznikne výjimka, kterou ovšem ve funkci *Fun* nelze ošetřit – to musí udělat až podprogram, který si *Fun* zavolal. My ovšem musíme tento objekt uvolnit, neboť do volajícího podprogramu se řízení nevrátí obvyklým způsobem. Proto výjimku zachytíme, objekt uvolníme a pošleme výjimku dál.

Koncovka bloku

Pokud v nějakém úseku kódu přidělujeme programu prostředky (alokujeme paměť, otevíráme soubory, vytváříme časovače apod.), je třeba zajistit, aby se tyto prostředky po

ukončení zase uvolnily. Prostředkem, který to v Delphi umožňuje, jsou koncovky bloků, tedy konstrukce **try/finally**. Syntax koncovky je

try
    příkazy
finally
    příkazy
end

V následujícím příkladu může v procedurách *Prohledej*, *Oprav* a *Zapis* vzniknout výjimka, kterou zde nemůžeme ošetřit. Potřebujeme ale uzavřít soubory *f* a *g*:

```
reset(f);
rewrite(g);
try
  Prohledej(f, g);
  Oprav(g);
  Zapis(g);
finally
  close(F);
  close(g)
end;
```

Koncovka bloku (sekce mezi **finally** a **end**) obsahuje operace, které se mají provést vždy. To znamená, že se provedou jak v případě, že hlídaná sekce (příkazy mezi **try** a **finally**) skončí normálně, tak i v případě, že v této sekci dojde k výjimce a část z nich se neprovede. Koncovka bloku se provede dokonce i v případě, že hlídanou sekci ukončíme voláním některé z procedur *Exit*, *Break* nebo *Continue*.

Výjimky v konstruktorech

Jestliže při konstrukci objektu, tedy v těle konstruktoru, vznikne výjimka, zavolá se na „nedodělaný“ objekt automaticky destruktor *Destroy*. Z toho plyne, že destruktory musí počítat s tím, že se po nich bude chtít, aby zničily nehotový objekt. K tomu může napomoci skutečnost, že konstruktor po alokaci inicializuje přidělenou paměť nulami.

11. Dynamická identifikace typů

Hovoříme-li o dynamické identifikaci typů, máme na mysli prostředky, které umožňují určit za běhu programu typ instance polymorfního typu. Setkáváme se s ní v C++ a v Object Pascalu v Delphi.

11.1 Dynamická identifikace typů v C++

S nástroji pro dynamickou identifikaci typů[21] se setkáváme až v ANSI C++. Borlandské překladače ji obsahují počínaje verzí 4.0. Dynamická identifikace typů je v C++ založena na

- ✧ operátoru **typeid**, který umožňuje určit typ objektu za běhu programu,
- ✧ operátoru **dynamic_cast**, který slouží k bezpečnému přetypování mezi objektovými typy uvnitř dědické hierarchie a o kterém budeme hovořit v následující kapitole,
- ✧ třídě *type_info*, která obsahuje výsledek určování typu.

Operátor **typeid** patří podle našeho názoru k nejproblematičtějším konstrukcím v C++ vůbec. Situace, kdy jej opravdu potřebujeme, se vyskytují velice zřídka, neboť k dynamickému určování typů skoro vždy postačí virtuální metody. Na druhé straně operátor **typeid** je součástí standardu jazyka, a proto si o něm v naší knize musíme alespoň v krátkosti povědět.

Operátor typeid

Používáme-li polymorfní typy, tj. objektové typy, které mají virtuální funkce, může se stát, že nebudeme znát přesný typ instance, se kterou pracujeme. Většinou to nevadí, neboť to ani nepotřebujeme, stačí použít virtuální metody.

Přesto se mohou vyskytnout situace, kdy přece jen potřebujeme skutečný typ instance (nebo celého výrazu) zjistit za běhu programu a nemáme k dispozici vhodnou virtuální metodu – třeba proto, že používáme objektovou knihovnu, jejíž tvůrci tam metodu, která by nějak určovala typ instance, prostě nezařadili.

ANSI C++ proto nabízí operátor **typeid**. Chceme-li jej používat, musíme do svého programu vložit hlavičkový soubor *typeinfo.h*. Syntax použití tohoto operátoru je

typeid(*výraz*)

nebo
typeid(*jméno_typu***)**

[21] Občas budeme pro dynamickou identifikaci typů používat označení *RTTI*. Zní to sice, jako když si kluci hrají na vojáky, je to ale zkratka slov *Run Time Type Identification*, tedy identifikace typu za běhu (programu).

První způsob použití je jasný: Pomocí operátoru **typeid** můžeme zjistit typ výrazu. Zpravidla to bude dereferencovaný ukazatel nebo reference na nějakou instanci. Druhý případ, ve kterém jako operand vystupuje označení typu, oceníme při porovnávání typů.

Výsledkem použití operátoru **typeid** je konstantní instance třídy *type_info*, která obsahuje mj. znakový řetězec se jménem typu.

Jestliže použijeme operátor **typeid** na jméno typu, na hodnotu neobjektového typu nebo nepolymorfního objektového typu, vyhodnotí se již v době překladu. Použijeme-li jej ale na výraz, který představuje dereferencovaný ukazatel na polymorfní typ nebo na referenci na instanci polymorfního typu, bude se vyhodnocovat dynamicky, tj. až za běhu programu. (Pouze v tomto případě tedy půjde o dynamickou identifikaci typu.)

Zadáme-li operátoru **typeid** dereferencovaný ukazatel s hodnotou 0 (ukazatel nikam), vyvolá výjimku typu *bad_typeid*[22].

typeid vrací type_info

Jak jsme si již naznačili, vytvoří operátor **typeid** konstantní instanci typu *type_info*. Aby byl tento výsledek vůbec k něčemu, jsou v této třídě přetíženy operátory „==“ a „!=“, které umožňují typy porovnávat, např.

```
// Beta je identifikátor třídy
if(typeid(*pb) != typeid(Beta))
  Bubu(pb);
```

Právě v takovýchto konstrukcích oceníme možnost použít operátor **typeid** i na označení typu (to může být identifikátor typu doplněný případně různými modifikátory jako **__far**, *, [ ] apod.).

Dále je ve třídě *type_info* deklarována metoda *name()*, která vrací ukazatel na řetězec obsahující zápis označení typu, a metoda *before()*, která slouží k určování *lexikálního* pořadí typů. Metoda *before()* vrací v ANSI C++ hodnoty typu **bool**; ve starších implementacích, např. BC++ 4.x, vracela tato metoda hodnoty typu **int**, a to 1 resp. 0 nebo 1.

V ANSI C++ je metoda *before()* specifikována jako implementačně závislá a je určena k porovnání názvů např. při zařazování do hešových tabulek. Určené pořadí nemá nic společného s jejich vzájemným postavením tříd v dědické hierarchii. V BC++ porovnává tato metoda řetězce obsahující označení typů lexikálně podle kódu ASCII.

Podívejme se na jednoduchý příklad použití operátoru **typeid** a metod *name()* a *before()*. Program *C10–01* obsahuje deklaraci dvou tříd, *A* a *Z*, přičemž *A* je potomkem *Z*. Pokud ponecháme direktivu **#define** v řádku označeném třemi hvězdičkami, tak jak je, budou obě třídy polymorfní (obsahují alespoň jednu virtuální metodu), takže se výrazy s operátorem **typeid** budou vyhodnocovat až za běhu programu a vyhodnotí se dynamický (tj. skutečný) typ instance:

[22] Připomeňme si, že v Borland C++ 4.x a 5.0 se tato třída jmenuje *Bad_typeid* a nemá normou předepsané metody.

```
/*  Příklad C10 – 1   */
#include <typeinfo.h>
#include <iostream.h>

// Tato direktiva umožní snadno změnit polymorfní
// třídy na nepolymorfní a naopak
#define VIRTUAL virtual                          // ***

// Dvě třídy – předek a potomek
// Obě polymorfní, pokud nezměníme předchozí direktivu
class Z {
  int i;
 public:
  VIRTUAL void f() {
       cout << "Z" <<endl;
  };
};

class A:public Z {
  int i;
 public:
  VIRTUAL void f(){
       cout << "A" << endl;
  };
};

int main(){
  Z z;
  A a;
  Z* pz1= &z;
  Z* pz2 = &a;
  try{
       cout << "pz2 ukazuje na objekt typu "
            << typeid(*pz2).name() << endl;
       cout << (typeid(*pz1).before(typeid(*pz2))? "ANO" : "NE")<< endl;
  }
  catch(Bad_typeid){
       cout << "Dereferencování 0" << endl;
       return 1;
  }
  return 0;
}
```

Tento program vypíše

```
pz2 ukazuje na objekt typu A
NE
```

tj. určí, že ukazatel *pz2*, ač byl deklarován jako ukazatel na *Z*, obsahuje adresu objektu třídy *A*, a že označení třídy *Z* není v abecedě před označením třídy objektu, na který ukazuje *pz1*.

Nyní změníme typy *A* a *Z* na nepolymorfní tím, že direktivu

```
#define VIRTUAL virtual
```

nahradíme direktivou

```
#define VIRTUAL
```

a program znovu přeložíme. Nyní se budou typy určovat staticky, tj. v době kompilace podle deklarovaného typu ukazatele. To znamená, že náš program pak vypíše

```
pz2 ukazuje na objekt typu Z
NE
```

*Z** je opravdu statický (tedy deklarovaný) typ ukazatele *pz2*.

Třída type_info

Třída *type_info* je v ANSI C++ (viz [9]) definována v hlavičkovém souboru *typeinfo* (ve starších implementacích je uváděn jako *typeinfo.h*). V současném návrhu normy je popsána takto:

```
class type_info {
 public:
   virtual ~typeinfo();
   bool operator==(const type_info &ps) const;
   bool operator!=(const type_info &ps) const;
   bool before(const type_info &) const;
   const char * name() const;
 private:
   type_info(const type_info &ps);
   type_info & operator=(const type_info &ps);
};
```

Poznámky:
1. V Borland C++ 4.0 se tato třída jmenovala Type_info *a hlavičkový soubor, ve kterém je deklarována, se jmenoval* Type_info.h*. V Borland C++ 4.5 a 5.0 se jmenuje* typeinfo*, lze však používat i starší označení. Tyto zmatky v názvech jsou důsledkem změn v návrhu normy.*
2. Všimněte si, že třída type_info *má soukromý kopírovací konstruktor a soukromý přiřazovací operátor. To je jednoduchý způsob, jak zabránit uživatelům této třídy v kopírování instancí.*
3. V ANSI C++ je deklarace třídy type_info *vnořena do standardního prostoru jmen* std*. (O prostorech jmen budeme hovořit v kapitole 12.) Vzhledem k pravidlům pro vyhledávání operátorů v prostorech jmen nás to však zatím nemusí zajímat.*
4. Tato třída může mít další (implementačně závislé) složky.

Na co se RTTI nehodí

Už jsme si řekli, že operátor **typeid** použijeme spíše výjimečně. Následující příklad vám nepochybně připomene čtvrtou kapitolu a problémy, se kterými jsme se potýkali, když jsme se snažili zajistit volání správné metody pro grafické objekty v různých potomcích téže abstraktní třídy. Nicméně takto bychom dynamickou identifikaci typů používat neměli:

```
void posun(const go& rgo){
  if(typeid(rgo) == typeid(kruh) {
       Nakresli_kruh(rgo);
  }
  else if(typeid(rgo) == typeid(usecka) {
       Nakresli_usecku(rgo);
  }
}
```

Zde předpokládáme, že *kruh* a *usecka* jsou třídy odvozené od společného předka *go*. Pokud má dynamická identifikace typů vůbec fungovat, musí být všechny tyto třídy polymorfní, a pak je samozřejmě rozumnější založit jakékoli operace s nimi na virtuálních metodách. Výsledný kód je pak nejen přehlednější, ale nejspíš i rychlejší. Kromě toho takto napsanou funkci *posun()* bychom museli měnit vždy, jakmile od třídy *go* odvodíme nějakého dalšího potomka – a tím si samozřejmě připravujeme živnou půdu pro nejrůznější chyby.

Borlandská rozšíření dynamické identifikace typů

Používání dynamické identifikace typů není zadarmo. Znamená připojení dodatečných dat a kódu k přeloženému programu. Proto např. Borland C++ 4.x a 5.0 nabízejí možnost RTTI nepoužívat. Jestliže dynamickou identifikaci typů zakážeme, můžeme i nadále používat operátor **typeid**, bude však i u polymorfních tříd určovat statický typ (tj. bude se vyhodnocovat již při překladu). Podívejme se na následující prográmek:

```
/*  Příklad C10 – 2   */
// Dynamická identifikace typů v BC++
#include <typeinfo.h>
#include <iostream.h>

// polymorfní třída
struct Alfa {
virtual void Fun(){}
};

struct Beta: Alfa{};

Alfa* pa;

int main(){
  Beta b;
  pa = &b;
// *pa má statický typ Alfa, ale dynamický typ Beta
  cout << typeid(*pa).name();
```

```
  return 0;
}
```

Ukazatel *pa* má statický (deklarovaný) typ *Alfa**, ve funkci *main()* mu však přiřadíme adresu instance potomka typu *Beta*. Jeho dynamický typ je proto *Beta **.

Jestliže tento prográmek přeložíme v BC++ s povolenou RTTI (implicitní nastavení), vypíše přesně podle očekávání „Beta“. Jestliže RTTI zakážeme, např. uvedením přepínače -RT- v příkazové řádce, a znovu jej přeložíme, určí statický typ, tj. vypíše „Alfa“, přestože jsou třídy *Alfa* a *Beta* polymorfní.

__rtti

V Borland C++ můžeme v deklaraci třídy použít modifikátor **__rtti**. Umožňuje nám vynutit si dynamické určování typu. Zapisuje se před identifikátor třídy, ale za klíčové slovo **class** nebo **struct**. Jestliže tedy upravíme v programu *C10–02* deklaraci třídy *Alfa* do tvaru

```
struct __rtti Alfa {
  virtual void f(){}
};
```

bude operátor **typeid** schopen určit dynamický typ dereferencované instance i v případě, že RTTI zakážeme. (To platí samozřejmě pouze v případě, že půjde o polymorfní typy. Modifikátor **__rtti** neudělá z nepolymorfní třídy třídu polymorfní.)

Je-li při překladu dynamická identifikace typů povolena, chovají se všechny třídy tak, jako kdybychom je deklarovali s modifikátorem **__rtti**.

U potomka není třeba modifikátor **__rtti** opakovat, odvozená třída (má-li jediného předka) tuto vlastnost zdědí.

Pokud RTTI zakážeme, můžeme narazit na některá omezení: Např. má-li odvozená třída více předků a jeden z polymorfních předků je deklarován s modifikátorem **__rtti**, musí být takto deklarováni všichni polymorfní předci. Jestliže se pokusíme odvodit společného předka od dvojice tříd, z nichž jedna je deklarována jako **__rtti** a druhá nikoli, může překladač hlásit chyby - záleží na pořadí předků a na tom, zda uvedeme modifikátor **__rtti** také u odvozené třídy.

Podrobnější informace o borlandské implementaci dynamické identifikace typů lze najít ve firemních manuálech.

Norma se vyvíjí

Nástroje pro dynamickou identifikaci typů byly do standardu jazyka C++ zařazeny teprve nedávno a diskuse kolem ní stále ještě pokračují. V důsledku toho se mohou dostupné implementace lišit od současného stavu návrhu normy (a tedy také od budoucích implementací). Proto si musíme při používání dynamické identifikace typů dát pozor, zda se náš překladač chová opravdu tak, jak očekáváme. Na jeden z příkladů jsme narazili v souvislosti s názvem třídy, která nese informace o určeném typu.

Další problém, který může programátorům znepříjemnit život, se může týkat toho, zda se při dynamickém určování typů berou v úvahu modifikátory **const** a **volatile**. Sou-

časný návrh normy totiž předepisuje, že tyto modifikátory na nejvyšší úrovni se vždy ignorují. To znamená, že následující program by měl vypsat dvakrát „ANO“:

```
/*  Příklad C10 – 3   */
// Test, zda implementace vyhovuje současnému návrhu normy ANSI C++

#include <iostream.h>
#include <typeinfo.h>

class D {};

D d1;
const D d2;

// Program by měl vypsat dvakrát "ANO"
int main(){
   cout << typeid(d2).name() << endl;
   cout << ((typeid(d1) == typeid(d2))? "ANO" : "NE") << endl;
   cout << ((typeid(D) == typeid(const D))? "ANO" : "NE") << endl;
   return 0;
}
```

V tomto příkladu definujeme dvě instance třídy *D*, jednu nekonstantní a druhou konstantní, a porovnáme jejich typ, zjištěný operátorem **typeid**. Pak porovnáme označení prostě označení typů *D* a **const** *D*. Přeložíme-li tento příklad např. pomocí Borland C++ 5.0, vypíše dvakrát „NE“, neboť tento překladač vychází ze starší specifikace jazyka.

11.2 Dynamická identifikace typů v Object Pascalu

Dynamickou identifikaci typů lze v Object Pascalu aplikovat pouze na reference na instance „nových“ objektových typů (tedy typů, deklarovaných pomocí klíčového slova **class**). Object Pascal na to zavedl nový operátor **is**. Syntax jeho použití je

reference_na_instanci **is** *reference_na_třídu*

Tento operátor vrací hodnotu typu **boolean**, a to:

- *true*, je-li levý operand reference na instanci třídy uvedené na pravé straně nebo jakéhokoli potomka této třídy,
- *false* v opačném případě.

Má-li levý operand hodnotu **nil**, vrátí operátor **is** vždy *false*. Je-li již v době kompilace jasné, že levý operand není instancí třídy na pravé straně nebo jejího předka či potomka, ohlásí překladač chybu.

Tento operátor se zpravidla používá k testu, zda je přetypování bezpečné. Například takto:

```
if Zdroj is Button then Button(Zdroj).Stisk;
```

Vzhledem k tomu, že *Zdroj* je reference (tedy ukazatel, který se automaticky dereferencuje), je uvedené přetypování v Pascalu dovoleno a nezáleží při něm na skutečné velikosti instancí. Musíme si ale zjistit, zda je *Zdroj* referencí na instanci třídy *Button*, aby vůbec mělo smysl volat metodu *Stisk*.

Poznámka:
Operátor **is** *má prioritu 4, tedy nejnižší – stejnou jako ostatní relační operátory („<“, „<=“,* **in** *a další).*

12. Operátory pro bezpečnější přetypování

12.1 Čtveřice nových přetypovacích operátorů v C++

Součástí nového standardu ANSI jazyka C++ je také čtveřice nových přetypovacích operátorů **dynamic_cast**, **static_cast**, **const_cast** a **reinterpret_cast**. Setkáváme se s nimi např. v borlandských překladačích počínaje verzí 4.0.

Proč to?

Přetypování bylo už v jazyku C tak trochu riziková operace, a v C++ je situace ještě problematičtější. Problém je v tom, že za přetypováním se může skrývat několik různých operací.

Jestliže v programu napíšeme

```
(T) V
```

bude výsledkem prakticky vždy hodnota typu *T*, **nějak** odvozená z hodnoty výrazu *V*. Problém je, **jak**. Může jít o

- ✧ jinou interpretaci bitů, které tvoří hodnotu *v* (např. přetypování ukazatelů **double*** na **int***),
- ✧ aritmetické výpočty, jež mohou být někdy značně komplikované, jako např. při převodu hodnoty typu **double** na hodnotu typu **int**,
- ✧ zúžení nebo rozšíření rozsahu (např. přetypování z **int** na **long** nebo naopak), při kterém se hodnota může, ale nemusí změnit,
- ✧ adresovou aritmetiku při přetypování potomka na předka v dědické hierarchii objektových typů, takže výsledkem je vlastně ukazatel na jiné místo v paměti (připomeňme si, že při přetypování ukazatele na potomka na ukazatel předka se změní hodnota ukazatele tak, aby obsahoval adresu zděděného podobjektu, a ta se při vícenásobné dědičnosti nemusí krýt s adresou celé instance),
- ✧ přidání nebo odstranění modifikátoru **const** nebo **volatile** a s tím související změnu v možnostech použití objektu,
- ✧ volání uživatelem definovaného operátoru přetypování nebo konstruktoru,

atd.

Výsledky některých přetypování jsou předepsány standardem jazyka, výsledky jiných jsou závislé na hardwaru a na implementaci (např. při převodu ze **short** na **int** a naopak nebo při přetypování ukazatelů na **int** a naopak). To ukazuje, že význam přetypování, jak je C++ zdědilo po jazyku C a poté ještě doplnilo o přetypování objektů

a ukazatelů na ně, je zjevně příliš široký, a proto se komise pro standardizaci rozhodla rozdělit jeho činnost mezi čtyři operátory.

Zavedení těchto operátorů by mělo

- ✧ vést ke zpřehlednění programů,
- ✧ přimět programátory, aby si rozmysleli, co vlastně přesně chtějí,
- ✧ umožnit vyhledávání přetypovacích operací v rozsáhlých programech pomocí programů, jako je grep (přetypování je nyní označeno klíčovým slovem, navíc všechny čtyři přetypovací operátory obsahují společnou část **…_cast**).

Syntax použití všech čtyř operátorů je stejná. Za klíčovým slovem, označujícím operátor, následuje v lomených závorkách (podobných jako u šablon) cílový typ, a pak v kulatých závorkách konvertovaný výraz.

Operátor dynamic_cast

Operátor **dynamic_cast** se používá především pro polymorfní třídy; slouží k bezpečnému přetypování mezi předky a potomky v dědických hierarchiích objektových typů. Jako jediný z nově přetypovacích operátorů může využívat dynamické identifikace typů. Není určen k přidávání nebo odstraňování modifikátorů **const** a **volatile**.

Popis

Následující odstavce vycházejí z odstavce č. 5.2.7 současného návrhu normy C++ a v podstatě tvrdí, že operátor **dynamic_cast** umožňuje přetypování ukazatelů (resp. referencí) v rámci dědické hierarchie polymorfních typů. Vedle jednoduchého přetypování ukazatele (resp. reference) na předka na ukazatel (resp. referenci) na potomka nebo naopak umožňuje také přetypovat ukazatel (referenci) na jeden zděděný podobjekt na ukazatel (referenci) na jiný zděděný podobjekt v rámci celého objektu. Přitom kontroluje, zda má daná operace smysl.

Přesněji: chceme-li přetypovat hodnotu *V* na typ *T*, použijeme zápis

```
dynamic_cast <T>(V)
```

a výsledek přetypování je typu *T*. Přitom cílový typ *T* musí být ukazatel, resp. reference na plně definovanou třídu[23] nebo **void***.

Je-li cílový typ *T* ukazatel, musí být hodnotou výrazu *V* ukazatel na plně definovaný objektový typ. Výsledkem bude výraz (r-hodnota) typu *T*. Je-li *T* reference na plně definovaný objektový typ, musí být *V* l-hodnota plně definovaného objektového typu a výsledkem bude l-hodnota typu, na který odkazuje *T*. (Čili: ukazatel lze přetypovat na ukazatel, referenci na referenci.)

Je-li typ *V* stejný jako cílový typ *T*, je výsledkem *V*. (Jestliže typ ve skutečnosti neměníme, nic se nestane.)

[23] To znamená, že nestačí předběžná deklarace.

Je-li *V* ukazatel s hodnotou 0 (ukazatel nikam), bude výsledkem ukazatel s hodnotou 0 typu *T*. (Nula zůstane, změní se její typ.)

Je-li *T* typ „ukazatel na *B*“ a přetypovávaný výraz *V* má typ „ukazatel na *D*“, a přitom *B* je předkem *D*, je výsledkem ukazatel na jediný podobjekt typu *B* v objektu typu *D*, na který ukazuje *V*. Totéž platí i v případě, že místo o ukazatelích budeme hovořit o referencích. (Operátor **dynamic_cast** lze použít k obyčejnému přetypování potomka na veřejně přístupného předka.)

Jinak musí být *V* ukazatel nebo reference na polymorfní typ, tedy na objektový typ, který má alespoň jednu virtuální funkci.

Je-li cílový typ *T* typ **void***, musí být *V* ukazatel. Výsledkem pak bude ukazatel na celý objekt, na který ukazuje *V*. (To už ale musí být *V* ukazatel na polymorfní objekt.)

Jinak se použije dynamická identifikace typů a zjistí se, zda je možné požadovanou konverzi provést. Pokud ne, přetypování se nepodaří. Výsledkem neúspěšného přetypování na ukazatel je hodnota 0. Jestliže se nepodaří přetypování na referenci, vyvolá operátor **dynamic_cast** výjimku typu *bad_cast*[24], který je definován v soubory *typeinfo.h*.

Postup přetypování, založeného na dynamické identifikaci typů, je následující:

Jestliže konvertovaná hodnota *V* ukazuje (jde-li o ukazatel) nebo odkazuje (jde-li o referenci) na zděděný podobjekt v objektu typu *T*, bude výsledkem ukazatel (resp. reference) na tento objekt typu *T*. Tedy z ukazatele na předka uděláme ukazatele na potomka, z reference na předka uděláme referenci na potomka. Dynamická kontrola ale zabezpečí, že se toto přetypování podaří pouze v případě, že má smysl, že se nesnažíme přetypovat samostatnou instanci, ale opravdu součást potomka. (Operátor **dynamic_cast** tedy umožňuje přetypování z předka na potomka, a to i v případě virtuálních předků. Něco podobného „klasický“ operátor přetypování neumí, neboť nemá k dispozici dynamickou identifikaci typů.)

Pokud *V* neukazuje (neodkazuje) na zděděný podobjekt v objektu typu *T*, najde se úplný objekt, na který *V* ukazuje. Jestliže má tento objekt jako jediného veřejně přístupného předka typ *T*, bude výsledkem ukazatel (reference) na tento podobjekt. Jinak se přetypování nepodaří.

Poznamenejme, že operátor **dynamic_cast** neumí odebírat modifikátory **const** a (nebo) **volatile**.

Příklad 1: přetypování ukazatelů

Předchozí povídání je značně nestravitelné; ostatně není se co divit, vymyslela je komise, dokonce mezinárodní. Podívejme se proto na několik příkladů; jejich zdrojové texty najdete na doplňkové disketě. Začneme u nejjednodušší situace – u ukazatelů na nepolymorfní typy:

```
/*  Příklad C11 – 1   */
#include <iostream.h>

// Tato direktiva umožní změnit metody na virtuální
```

[24] Připomínáme, že v Borland C++ 4.x a 5.0 se tato třída píše s velkým počátečním písmenem, tj. *Bad_cast*.

```
#define VIRTUAL

struct A {
   int i;
   A(int y=0):i(y){}
   VIRTUAL void F(){cout << "Třída A " << i << endl;}
};

struct B {
   int ii;
   B(int y=0):ii(y){}
   VIRTUAL void G(){cout << "Třída B " << ii << endl;}
};

// Společný potomek dvou polymorfních tříd
struct C: A, B{
   VIRTUAL void F(){cout << "Třída C " << i << endl;}
};

int main(){
   A a(0), *pa;                              // 1
   B b(8), *pb;
   C c, *pc = &c;
   pa = dynamic_cast<A*>(pc);                // 2
   cout << "adresa, uložená v pc: " << pc << endl;
   pb = dynamic_cast<B*>(pc);                // 3
   cout << "adresa, uložená v pc po přetypování na B*: " << pb << endl;
   A& ra = dynamic_cast<A&>(c);              // 4
   cout << "adresa, uložená v pc po přetypování na A*: " << pa << endl;
   c.i = 9;
   c.F();
   ra.F();
// Následující příkazy lze použít, budou-li třídy A,B a C polymorfní
// void *t = dynamic_cast<void*>(pb);        // 5
// cout << "pb: " << pb << " přetypováno na void*: " << t << endl;
// t = dynamic_cast<B*>(pa);                 // 6
   return 0;
}
```

V programu *C11–01.CPP* jsme deklarovali třídu *C* jako potomka tříd *A* a *B*. Žádná z těchto tříd není polymorfní, tj. žádná neobsahuje virtuální metody. V řádku, označeném v komentáři číslem 1, a ve dvou následujících (bez čísel) deklarujeme instance těchto tříd a ukazatele na ně.

Řádky, označené 2 a 3, obsahují přetypování ukazatele na instanci třídy *C* na ukazatele na předky; následující výpis adresy (nebo krokování v integrovaném prostředí) nás přesvědčí, že výsledkem budou opravdu ukazatele na zděděné podobjekty.

V řádku, označeném 4, jsme objekt *c* třídy *C* přetypovali na referenci na objekt typu *A* a získanou hodnotu použili k definici reference *ra* na tento podobjekt. O tom, že to funguje, nás přesvědčí následující tři příkazy, ve kterých do *c.i* uložíme hodnotu 9 a pak ji vypíšeme pomocí metody *F()*, volané pro *c* a pro *ra.* Vzhledem k tomu, že nejde o virtuální metody, bude se volat *C::F()* a *A::F()*.

Příkazy na řádcích, označených 5 a 6, jsou chybné; třídy *A* ani *B* nejsou polymorfní, a proto nejsou uvedená přetypování dovolena.

Nyní uděláme z *A*, *B* a *C* polymorfní třídy. Jestliže nahradíme direktivu

```
#define VIRTUAL
```

direktivou

```
#define VIRTUAL virtual
```

změní se metody *F()* ve všech třech třídách na virtuální. Proto již budou dovoleny i příkazy v řádcích 5 a 6, takže můžeme odstranit znak komentáře před nimi. Tento program vypsal na našem počítači

```
adresa, uložená v pc: 0xffe4
adresa, uložená v pc po přetypování na B*: 0xffe8
adresa, uložená v pc po přetypování na A*: 0xffe4
Třída C 9
Třída C 9
pb: 0xffe8 přetypováno na void*: 0xffe4
```

Vypsané hodnoty (nebo krokování) nás přesvědčí, že

- ✧ v příkazu *ra.F()* (bezprostředně před řádkem, označeným číslem 5) se volá metoda *C::F()*;
- ✧ v řádku 5 se do ukazatele *t* uloží adresa objektu *c*, nikoli adresa zděděného podobjektu typu *A* (výsledek je ovšem ukazatel bez doménového typu, tedy **void***),
- ✧ v řádku 6 se ukazatel na zděděný podobjekt typu *A* převede na ukazatel na zděděný podobjekt typu *B*.

Podívejme se podrobněji na průběh přetypování v řádku 6. Ukazatel *pa* obsahuje adresu podobjektu typu *A*, který je součástí objektu *c* typu *C*. My si poroučíme přetypování na *B**. Protože *B* není veřejně přístupným předkem *A* (a také *A* není veřejně přístupným předkem *B*), najde se nejprve celý objekt. To je instance *c* třídy *C*. Protože třída *C* obsahuje jediný zděděný podobjekt typu *B*, vrátí operátor **dynamic_cast** adresu tohoto podobjektu.

Nyní přidáme do funkce *main()* v příkladu C11 – 1 těsně před příkaz **return** tyto dva řádky:

```
pa = &c; // Explicitní přetypování zde není třeba
pb = dynamic_cast<B*>(pa); // OK
```

Ukazatel *pa* obsahuje adresu zděděného podobjektu, a tak přetypování na potomka proběhne bez problémů – o tom se můžeme přesvědčit např. opět krokováním. Pokud bychom ovšem napsali

```
pa = &a;
pb = dynamic_cast<B*>(pa); // Nelze
```

bude výsledkem 0 (tedy ukazatel nikam). V tomto případě totiž *pa* obsahuje ukazatel na samostatný objekt třídy *A*, a proto přetypování jeho adresy na ukazatel na typ *B* postrádá smysl, což nám operátor **dynamic_cast** dá najevo tím, že vrátí 0. Podobně se nepodaří přetypování

```
pa = &a;
pc = dynamic_cast<C*>(pa);
```

nebo třeba

```
pb = dynamic_cast<B*>(&a);
```

Poznámka:
Jestliže použijeme operátor **dynamic_cast** *k přetypování ukazatelů, měli bychom vždy kontrolovat, zda se přetypování podařilo, tj. zda jeho výsledkem není 0. Např. takto:*

```
if((pc = dynamic_cast<C*>(pa))==0) Chyba();
else Zpracuj(pc);
```

Příklad 2: přetypování referencí

Ve druhém příkladu se podíváme na přetypování referencí na objekty. Program *C11–02.CPP* vznikne jednoduchými úpravami programu *C11–01.CPP*. Deklarace tříd *A*, *B* a *C* jsou stejné jako v předchozím příkladu, proto zde jejich deklarace vynecháme. Na disketě najdete příklad samozřejmě v úplném znění.

```
/*  Příklad C11 – 2   */
#include <iostream.h>
#include <typeinfo.h>

// Tato direktiva umožní změnit metody na virtuální
#define VIRTUAL virtual
struct A {/* ...*/ }; // Deklarace těchto tříd viz příklad C11 – 1
struct B {/* ...*/ };
struct C: A, B {/* ...*/ };

int main(){
  A a(0), *pa;
  B b(8), *pb;
  C c, *pc= &c;
  A& ra = dynamic_cast<A&>(c);              // 1
  c.ii = 9;                                 // 2
  try{                // Pozor, zde může vzniknout výjimka
       b = dynamic_cast<B&>(ra);            // 3
       b.G();
       cout << " OK " << endl;
       b = dynamic_cast<B&>(a);             // 4
  }
  catch(Bad_cast){
       cout << "Přetypování se nepodařilo – vznikla výjimka" << endl;
  }
  return 0;
}
```

Na našem počítači vypsal tento program

```
Třída B 9
OK
Přetypování se nepodařilo – vznikla výjimka
```

Podívejme se na průběh výpočtu. V řádku, označeném v komentáři číslem 1, definujeme referenci *ra* na zděděný podobjekt třídy *A* v instanci *c*. V řádku 2 změníme hodnotu atributu *c.ii*, zděděného po třídě *B*.

Řádek 3 obsahuje přetypování *ra* na referenci na typ *B*. To jde, neboť *ra* je reference na zděděný podobjekt v instanci třídy *C* a třída *C* obsahuje jediný veřejně přístupný zděděný podobjekt třídy *B*. Výsledkem tedy bude reference na tento zděděný podobjekt třídy *B*, který se přiřadí instanci *b*. Voláním metody *b.G()* se přesvědčíme, že *b.ii* obsahuje opravdu hodnotu 9, kterou jsme uložili do *c.ii*.

Řádek, označený číslem 4, obsahuje také přetypování. Zde ovšem nebudeme mít úspěch, neboť *a* je samostatná instance třídy *A*, nikoli zděděný podobjekt v instanci, která by také obsahovala podobjekt třídy *B*. Takové přetypování nemá smysl, a proto operátor **dynamic_cast** vyvolá výjimku *Bad_cast*.

Operátor static_cast

Operátor **static_cast** slouží především pro běžné konverze objektových typů z předka na potomka a naopak, ovšem **bez dynamické kontroly typů**. Vedle toho umožňuje volat standardní konverze jazyka, jako jsou převody mezi celočíselnými a reálnými typy aj. Není určen k přidávání nebo odstraňování modifikátorů **const** a **volatile**.

Popis

Začneme opět tím, že si povíme, jak to s operátorem **static_cast** vlastně je. Výklad bude tentokrát přece jen jednodušší a stravitelnější než u operátoru **dynamic_cast**. Ve výrazu

```
static_cast<T>(V)
```

musí typ *T* představovat ukazatel, referenci, aritmetický nebo výčtový typ. Typ výrazu *V* musí nějakým způsobem odpovídat typu *T*. Oba typy, jak *T* tak i typ výrazu *V*, musí být v době překladu tohoto přetypování plně známy – to znamená, že opět nelze použít typ, který překladač zná pouze z předběžné deklarace.

Objekt (hodnotu) jednoho typu lze převést pomocí operátoru **static_cast** na objekt jiné (i nesouvisející) třídy, pokud existuje vhodná metoda, která konverzi provede – konverzní konstruktor nebo přetypovací operátor. Operátor **static_cast** tuto metodu zavolá (a to i v případě, že jde o explicitní konstruktor). Toto pravidlo lze formulovat také jinak: k přetypování výrazu *V* na typ *T* můžeme použít operátor **static_cast**, jestliže by byla správná deklarace proměnné *t* typu *T*, inicializované výrazem *V*:

```
T t(V);
```

Pomocí tohoto operátoru můžeme převést celočíselný typ na výčtový, např.

```
enum karty {sedm, osm, devet, deset, spodek,
            svrsek, kral, eso};
// ...
karty vynos = static_cast<karty>(2);
```

pokud hodnota převáděného výrazu leží v rozsahu daného výčtového typu; jinak není výsledek definován.

Podobně můžeme tento operátor používat k převodu celých čísel na reálná nebo naopak. Operátor **static_cast** může také provádět konverze opačné k běžným standardním převodům (až na několik výjimek, jako je konverze funkce nebo pole na ukazatel, konverze číselných a ukazatelových typů na typ **bool**, konverze l-hodnoty na výraz atd.).

Ukazatel na objektový typ *X* můžeme pomocí tohoto operátoru konvertovat na ukazatel na objektový typ *Y*, je-li třída *X* jednoznačným **nevirtuálním** předkem třídy *Y* nebo naopak. Přitom se – na rozdíl od operátoru **dynamic_cast** – nezjišťuje, zda má taková konverze smysl. Podobně lze konvertovat l-hodnotu typu *X* na l-hodnotu typu *Y* nebo naopak.

Poznamenejme, že operátor **static_cast** lze použít i ke konverzi ukazatelů do tříd, pokud jsou oba ukazatele do stejné třídy nebo do různých tříd, z nichž jedna je jednoznačným potomkem druhé.

Příklad

V příkladu C11 – 3 se setkáme opět s třídami *A*, *B* a *C*, budou se však poněkud lišit od tříd v příkladech z předchozího oddílu.

```
/* Příklad C11 – 3 */
#include <iostream.h>

struct B {
   int ii;
   B(int y=0):ii(y){}
   void F(){cout << "Třída B " << ii << endl;}
};

struct A {
   int i;
   A(int y=0):i(y){}
   A(B b);
   operator B&();
   void f(){cout << "Třída A " << i << endl;}
};

A::operator B&() {
   cout << "operator pro prevod B na A" << endl;
   return *reinterpret_cast<B*>(this);
}

A::A(B b):i(b.ii){
   cout << "konstruktor A(B)" << endl;
}

struct C: A, B{
   void f(){cout << "Třída C " << i << endl;}
};

int main(){
   A a(0), *pa;
   B b(8), *pb;
   C c, *pc= &c;
   static_cast<B&>(a)=b;               // 1
   cout << a.i << endl;
```

```
   pa = &c; // 2
   pc = static_cast<C*> (pa);          // 3
   pa = &a; // 4
// pc = static_cast<B*> (pa);          // 5 !!
   static_cast<B&>(a).F();             // 6
   b.ii = 66;
   a = static_cast<A>(b);              // 7
   cout << a.i << endl;
   static_cast<A>(b).i = 11;           // 8
   cout << b.ii << endl;
   return 0;
}
```

Ponechme zatím stranou tělo operátoru *A::operator B&()* – o způsobu, jakým tento operátor vytvoří z instance třídy *A* referenci na *B* si povíme v příštím oddílu, který věnujeme operátoru **reinterpret_cast**, a podívejme se na příklady použití operátoru **static_cast**.

V řádku, označeném číslem 1, konvertujeme instanci třídy *A* na instanci třídy *B*. Přitom se zavolá konverzní operátor, který jsme deklarovali ve třídě *A*. Protože výsledkem konverze je reference, může zápis přetypování stát na levé straně přiřazení.

V řádku 2 uložíme do ukazatele *pa* adresu instance *c* odvozené třídy *C*. Protože *pa* je typu *A**, bude obsahovat adresu zděděného podobjektu. V řádku 3 tuto hodnotu konvertujeme na ukazatel na *C*. To je v pořádku, výsledkem bude skutečně ukazatel na objekt *c*.

V řádku 4 uložíme do *pa* adresu instance *a* a tu v řádku 5 konvertujeme na *C**. Tato operace nemá smysl, neboť *a* je samostatná instance, nikoli zděděný podobjekt. Překladač by zde proto ohlásil chybu – ukazatel na *A* nelze pomocí operátoru *static_cast* konvertovat na ukazatel na *B*. Proto jsme řádek 5 „zakomentovali“.

V řádku 6 přetypujeme instanci *a* třídy *A* na instanci třídy *B* a zavoláme pro ni metodu *B::F()*.

V řádku 7 přetypujeme naopak instanci *b* třídy *B* na instanci třídy *A*. Přitom se zavolá konverzní konstruktor *A::A(B)*. Poznamenejme, že takovouto konverzi nelze použít na levé straně přiřazovacího příkazu, neboť nevytvoří l-hodnotu. Překladač Borland C++ 5.0 sice přijme příkaz v řádku, označeném 8, bez námitek, ale vytvoří si při přetypování pomocnou proměnnou, takže instance *b* se v následujícím přiřazení nezmění (o tom nás přesvědčí následující příklad).

Podívejme se ještě na takovouto konstrukci:

```
struct D;                          // 1
D* dd;                             // 2
dd = (D*)pa;                       // 3
// dd = static_cast<D*>(pa);       // 4 !!
```

Strukturu *D* jsme deklarovali pouze předběžně. To nám umožnilo definovat ukazatel na ni (v řádku 2). Je-li *pa* ukazatel na *A*, je možná konverze v řádku 3, která používá „klasické“ přetypování, avšak konverze v řádku 4 není přípustná, neboť typ *D* nebyl dosud definován.

Operátor reinterpret_cast

Operátor **reinterpret_cast** umožňuje konverze, jejichž výsledek může být implementačně závislý (např. převod ukazatele na celé číslo nebo naopak), převod ukazatele na jednu třídu na ukazatel na jinou naprosto nesouvisející třídu, převod ukazatele na data na ukazatel na funkci apod. Jinými slovy: má na starosti „špinavou práci“. Není určen k přidávání nebo odstraňování modifikátorů **const** a **volatile**.

Ve výrazu

```
reinterpret_cast<T>(V)
```

musí být *T* ukazatel, reference, aritmetický typ, ukazatel na funkci nebo ukazatel do třídy.

Pomocí operátoru **reinterpret_cast** můžeme převést ukazatel na celé číslo a naopak, celé číslo na ukazatel. Převedeme-li pomocí tohoto operátoru ukazatel na celé číslo a získané celé číslo zpět na ukazatel s týmž doménovým typem, získáme tutéž hodnotu, pokud celé číslo, získané převodem ukazatele, vejde do rozsahu typu, na který jsme jej převedli. Jestliže bychom např. pomocí tohoto operátoru převedli vzdálený ukazatel, který zabírá 4 B, na číslo typu **short**, které zabírá pouze 2 B, a pak je převedli zpět, dostaneme nesmysl.

Výsledky použití operátoru **reinterpret_cast** mohou záviset nejen na implementaci, ale třeba také na cílové platformě nebo použitém paměťovém modelu. Podívejme se na následující příklad:

```
/* Příklad C11 – 4 */
#include <iostream.h>

int main(){
   int a;
   int* b=&a;
 // přetypujeme ukazatel tam a zpět a výsledky porovnáme
   a = reinterpret_cast<int>(b);
   int *c = reinterpret_cast<int*>(a);
   cout << "Operátor reinterpret_cast:" << endl;
   cout << "Porovnání ukazatele před převodem a po něm: ";
   cout << ((b == c)? "ANO" : "NE") << endl ;
   return 0;
}
```

Jestliže tento prográmek přeložíme např. v Borland C++ 4.5 jako aplikaci pro DOS v malém modelu, vypíše „ANO“, neboť blízký ukazatel má stejný rozsah jako typ **int**. Obdobně vypíše „ANO“, přeložíme-li jej jako konzolovou aplikaci pro Win32 (zde mají jak ukazatele, tak i typ **int** rozsah 4 B). Přeložíme-li jej ale např. ve velkém modelu pro DOS, vypíše „NE“, neboť vzdálený ukazatel se do typu **int** již nevejde celý. Pokusíme-li se tento program přeložit v Borland C++ 5.0, ohlásí ve velkém modelu chybu – ukazatel nelze převést na typ **int** (nevejde se do něj).

Podobně i ostatní převody, které tento operátor umožňuje, jsou obvykle závislé na implementaci a jiných okolnostech.

Operátor **reinterpret_cast** umožňuje přetypovat ukazatel na jednu třídu na ukazatel na jinou, naprosto nesouvisející třídu, a to dokonce i v případě, že tyto třídy nebyly dosud plně definovány. Takovéhoto přetypování jsme využili v příkladu C11 – 3 v definici operátoru, který přetypovával instanci třídy *A* na referenci na třídu *B*:

```
A::operator B&() {
  return *reinterpret_cast<B*>(this);
}
```

Protože se jedná o nestatickou metodu třídy *A*, je v ní k dispozici ukazatel na aktuální instanci **this**. Ten jsme přetypovali na ukazatel na *B*, dereferencovali a vrátili.

Operátor **reinterpret_cast** umožňuje dokonce přetypovat jednu třídu na jinou i v případě, že neexistuje metoda, která by takový převod prováděla. Podívejme se na příklad:

```
/* Příklad C11 – 5 */
#include <iostream.h>

struct Alfa {
  int i;
  Alfa(int ii): i(ii){}
  void f(){cout << "Třída Alfa " << i;}
};

struct Beta {
  int i;
  Beta(int ii): i(ii){}
  void g(){cout << " Třída Beta " << i;}
};

int main(){
  Alfa a(1);
  reinterpret_cast<Beta&> (a).g(); // ***
  return 0;
}
```

Struktury *Alfa* a *Beta* mají naprosto stejné uspořádání, a proto má smysl uvažovat o přetypování *Alfa* na *Beta*. V řádku, označeném hvězdičkami, přikážeme chápat instanci *a* třídy *Alfa* jako referenci na instanci třídy *Beta* a zavoláme pro ni metodu *g()*.

V tomto okamžiku se mohou začít vkrádat pochybnosti. Proč bychom měli v programu definovat dva naprosto identické typy? Ale i kdybychom něco takového potřebovali, proč bychom pro ně neměli definovat konverzní operátory? Jenže při skutečném programování se může stát ledacos, a proto je rozumné mít i takovouto možnost. Nicméně je asi jasné, že tento operátor umožňuje operace, ve kterých si musíme dávat opravdu pozor, co vlastně děláme.

Operátor const_cast

Posledním z operátorů, o kterých budeme v podkapitole o přetypování v C++ hovořit, je operátor **const_cast**. Tento operátor umožňuje jako jediný udělat z nekonstanty konstan-

tu nebo naopak. **To je něco, co nemohl žádný z předcházejících tří**. Podobně umožňuje „přidávat“ nebo „odebírat“ i modifikátor **volatile**. Žádné jiné konverze však tento operátor neumí.

Ve výrazu

```
const_cast<T>(V)
```

se typ *T* od typu výrazu *V* smí lišit pouze v modifikátorech **const** nebo **volatile**. Toto přetypování se provede již v době kompilace a výsledek bude typu *T*.

Příklady

Tento operátor se používá nejčastěji pro přetypování konstantní instance objektového typu na nekonstantní a naopak. To sice potřebujeme málokdy, nicméně může se např. stát, že chceme zavolat konstantní metodu pro nekonstantní objekt. Ukážeme si jednoduchý příklad:

```
/* Příklad C11 – 6 */
#include <iostream.h>

struct Alfa{
  void f(){
       cout << "nekonstantní metoda" << endl;
  }

  void f() const {
       cout << "konstantní metoda" << endl;
  }
};

int main(){
  Alfa t;
  t.f();                                                  // 1
// Voláme konstantní metodu pro nekonstantní objekt
  const_cast<const Alfa&>(t).f();                         // 2
  return 0;
}
```

V řádku, označeném v komentáři číslem 1, se bude volat metoda *Alfa::f()*, zatímco v řádku, označeném 2, se bude volat metoda *Alfa:: f() const*, přesto, že *t* není konstantní instance.

Operátor **const_cast** lze použít i k přeměně konstant základních typů na nekonstanty a naopak. Něco takového ovšem ve skutečnosti zpravidla postrádá smysl.

Co s tím

Pravidla pro používání nových přetypovacích operátorů se zdají na první (ale i na druhý a třetí) pohled složitá. Ve skutečnosti jde však spíše o nezvyk – seznámili jsme se s těmito operátory v době, kdy jsme si už zvykli používat „klasické“ přetypování. To je ale situace, ve které jsou dnes prakticky všichni uživatelé C++, neboť tyto operátory nacházíme jen v nejnovějších překladačích.

Chceme-li se je naučit používat, můžeme postupovat asi takto:

1. Jestliže potřebujeme udělat z konstanty nekonstantu nebo naopak, příp. jestliže potřebujeme ubrat nebo připojit modifikátor **volatile**, použijeme operátor **const_cast**. To jiný operátor nezvládne.
2. Jestliže potřebujeme přetypování objektů nebo ukazatelů v rámci dědické hierarchie a chceme přitom za běhu programu (dynamicky) kontrolovat správnost této operace, použijeme operátor **dynamic_cast**.
3. Jinak zkusíme operátor **static_cast**, a pokud nám překladač oznámí, že takovéto přetypování neumí, použijeme **reinterpret_cast**.

Nikdo vás samozřejmě nenutí tyto operátory používat; klasický operátor přetypování ve tvaru **(***typ***)** měl také svůj půvab a dokázal v podstatě totéž – až na dynamickou kontrolu správnosti a několik dalších drobností, jako je přetypování virtuálního předka na potomka. Myslíme si ale, že nové operátory mohou přece jen zpřehlednit program a usnadnit vám tak život.

12.2 Nový přetypovací operátor v Object Pascalu

V Object Pascalu, implementovaném v Delphi, můžeme samozřejmě používat přetypování hodnot nebo proměnných stejně jako v Turbo Pascalu, jak jsme se s ním seznámili již dříve. Vedle toho ale nabízí Object Pascal nový operátor **as**, který slouží k přetypování s dynamickou kontrolou správnosti.

Syntax použití tohoto operátoru je

reference_na_objekt **as** *reference_na_třídu*

Výsledkem je pravý operand, chápaný ovšem jako hodnota typu, představovaného levým operandem. Levý operand, *reference_na_objekt*, musí být **nil** nebo instance třídy označované pravým operandem (*reference_na_třídu*) nebo instance potomka této třídy. Pokud není splněna ani jedna z těchto podmínek, vznikne výjimka.

Dokáže-li překladač určit, že levý a pravý operand spolu nesouvisejí, tj. že objekt na levé straně není instancí pravého operandu ani jeho předka nebo potomka, ohlásí chybu již při překladu.

Následující příklad pochází z manuálu k Delphi:

```
with Sender as TButton do
begin
  Caption := '&OK';
  OnClick := OkClick;
end;
```

Operátor **as** má prioritu 2, tj. stejnou, jako multiplikativní operátory *, /, **div**, **mod** atd. Z toho plyne, že pokud jej chceme použít na levé straně přiřazovacího příkazu pro přístup ke složkám objektu, musíme jej uzavřít do závorek:

```
(Sender as TButton).Caption := '&Ok';
```

13. Prostory jmen

Prostory jmen jsou součástí návrhu jazyka C++ již poměrně dlouho – jejich popis najdeme např. již ve vydání knihy *The Annotated C++ Reference Manual* [4], která se stala podkladem pro návrh normy ANSI. Setkáváme se však s nimi až v nejnovějších překladačích, např. v Borland C++ 5.0. V Turbo Pascalu, ani v Object Pascalu nic podobného není.

13.1 O co vlastně jde

Převážná většina profesionálních programů se skládá z většího počtu souborů a vyvíjí je tým programátorů. Při týmové práci připravuje každý z vývojářů nějakou skupinu zdrojových souborů. Přitom se mohou snadno vyskytnout konflikty jmen. Stačí aby dva programátoři použili ve svých souborech globální nestatickou proměnnou jménem *i*, *N* apod., a přesto, že jednotlivé díly programu fungovaly bezvadně, začnou se po jejich spojení dít neuvěřitelné věci.

Na podobné problémy můžeme narazit i při používání knihoven a hlavičkových souborů, dodávaných s překladačem.

Lze samozřejmě namítnout, že slušný programátor se vyhýbá globálním proměnným. Koneckonců jedna z nezanedbatelných výhod objektového programování spočívá v tom, že můžeme proměnné zapouzdřit do instancí objektových typů a pracovat s nimi výhradně pomocí přístupových funkcí[25].

Ovšem teoretické námitky jsou jedna věc a praktické programování věc jiná. Za prvé, zdaleka ne všichni programátoři jsou – pokud jde o způsob psaní programů – slušní. Kromě toho profesionální programátoři mají obvykle velice málo času na to, aby vymýšleli elegantní a teoreticky čistá řešení, a tak často použijí první nápad, který jim přijde na mysl – a podle toho to pak v programech vypadá.

Konflikty jmen mezi soubory jsou velice nepříjemná věc. Proto se programátoři občas uchylovali k různým trikům, které jim měly zabránit. Jedním z nich například bylo, se všechny proměnné, které by jinak byly globální, definovali jako složky nějaké globální struktury. Znamenalo to sice trochu více psaní, ale při týmové práci to stálo zato.

Na podobné myšlence jsou založeny prostory jmen v C++.

Prostor jmen opravdu připomíná po formální stránce strukturu nebo třídu: jména, definovaná uvnitř prostoru jmen, můžeme používat i mimo něj, musíme ale vždy říci, do kterého prostoru jmen patří. To znamená, že je musíme kvalifikovat identifikátorem prostoru, ve kterém byly deklarovány, nebo jinak zpřístupnit.

[25] To vlastně není žádná novinka – s touto myšlenkou přišlo již dříve modulární programování. Z tohoto hlediska se můžeme na OOP dívat jako na logické pokračování modulárního programování, ve kterém se z modulu stal datový typ.

13.2 Deklarace prostoru jmen

Syntax deklarace prostoru jmen je

namespace *identifikátor*$_{opt}$ **{***deklarace* **}**

V tomto popisu *identifikátor* představuje jméno deklarovaného prostoru jmen. Index $_{opt}$ naznačuje, že je můžeme vynechat – tak vznikne anonymní prostor jmen.

Deklarace jsou deklarace proměnných, funkcí, typů atd., které leží v deklarovaném prostoru jmen. V těchto deklaracích můžeme na rozdíl od deklarací složek struktur nebo tříd používat i modifikátorů, určujících paměťové třídy (**register**, **extern** atd.).

Deklaraci prostoru jmen můžeme rozdělit do několika částí. Kromě toho můžeme uvnitř jednoho prostoru jmen deklarovat další, vnořený prostor jmen.

Prostor jmen nepředstavuje obor viditelnosti. Vše, co v něm deklarujeme, můžeme v programu používat všude tam, kde to dovolí obvyklá pravidla viditelnosti identifikátorů, jako kdyby žádné prostory jmen neexistovaly. Pokud ale identifikátor, deklarovaný v prostoru jmen, použijeme mimo něj, musíme jej kvalifikovat, tj. musíme k němu operátorem „::“ připojit identifikátor prostoru jmen, podobně jako kvalifikujeme např. jména typů, deklarovaných uvnitř objektových typů.

Poznamenejme, že pokud by se kvalifikace identifikátorem prostoru jmen často opakovaly, můžeme se jim vyhnout pomocí direktivy nebo deklarace **using**. O tom si ale povíme později.

Nejprve si ukážeme příklad deklarace prostoru jmen:

```
/*  Příklad C12 – 1   */
#include <iostream.h>

// Globální proměnná mimo prostory jmen
int n = 11;

namespace Nas_Prvni_Prostor_Jmen {
  int n = 22;
  void f();
}

void Nas_Prvni_Prostor_Jmen::f(){
  cout << "ve funkci f: n = "
       << n << endl;
}

int main(){
  int n = 555;
  cout << "lokalni n = " << n << endl;
  cout << "Nas_Prvni_Prostor_Jmen::n = "
       << Nas_Prvni_Prostor_Jmen::n << endl;          // *
  cout << "globalni n = " << ::n << endl;             // **
  Nas_Prvni_Prostor_Jmen::f();
  return 0;
}
```

V příkladu C12 – 1 jsme deklarovali globální proměnnou jménem *n*, která nepatří do žádného prostoru jmen, a jeden globální prostor jmen s názvem *Nas_Prvni_Prostor_Jmen*. V něm jsme deklarovali opět proměnnou *n* a funkci *f()*.

Definiční deklaraci funkce *f()* z prostoru *Nas_Prvni_Prostor_Jmen* můžeme zapsat mimo deklaraci tohoto prostoru tak, jak jsme to udělali v našem příkladu. Pak ale musíme jméno funkce v této deklaraci kvalifikovat identifikátorem prostoru jmen, musíme tedy psát *Nas_Prvni_Prostor_Jmen::f()*.

Tělo funkce *Nas_Prvni_Prostor_Jmen::f()* je součástí prostoru jmen *Nas_Prvni_Prostor_Jmen* bez ohledu na to, kde leží definiční deklarace této funkce. To znamená, že v těle funkce *Nas_Prvni_Prostor_Jmen::f()* znamená identifikátor *n* odkaz na proměnnou Na*s_Prvni_Prostor_Jmen::n*. (To je podobné jako u metod objektových typů. I když zapíšeme definici metody mimo deklaraci třídy, můžeme v ní používat jména složek třídy bez kvalifikace.)

Pokud lokální jméno zastíní jméno globálního objektu, který jsme nedeklarovali uvnitř žádného prostoru jmen, můžeme se tohoto globálního objektu dovolat pomocí unárního operátoru „::“. V příkladu C12 – 1 jsme tak v řádku, označeném dvěma hvězdičkami, vypsali hodnotu globální proměnné *n*.

V následujícím příkladu si ukážeme použití vnořených prostorů jmen:

```
/*  Příklad C12 – 2   */
#include <iostream.h>

namespace Prvni {
   char *Text = "Prvni";
   void Tiskni(){cout << Text << endl;}
}

namespace Druhy {
   char *Text = "Druhy";
   void Tiskni(){cout << Text << endl;}
   namespace Treti{
        char *Text = "Druhy::Treti";
        void Tiskni(){cout << Text << endl;}
   }
}

int main(){
   Prvni::Tiskni();
   Druhy::Tiskni();
   Druhy::Treti::Tiskni();                    // ***
   return 0;
}
```

Zde máme dva globální prostory jmen, které jsme vtipně pojmenovali *Prvni* a *Druhy*. V prostoru *Druhy* jsme deklarovali ještě vnořený prostor s neméně výstižným názvem *Treti*. V každém z nich jsme deklarovali proměnnou *Text* a funkci *Tiskni()*, která vypíše znakový řetězec, na který ukazuje *Text*. (To vypadá, že nám naprosto došla fantazie. Berte to lehce, to se stává.)

Ve funkcích *Tiskni()* používáme proměnnou *Text* bez kvalifikace. To znamená, že každá z funkcí *Tiskni()* použije proměnnou, definovanou ve „svém“ prostoru jmen. Ve

funkci *main()* ovšem musíme při volání funkcí *Tiskni()* vždy specifikovat, kterou že chceme. V případě funkce, deklarované ve vnořeném prostoru jmen, musíme uvést jména obou prostorů.

Přezdívky prostoru jmen – alias

Volání funkce *Nas_Prvni_Prostor_Jmen::f()* z příkladu C12 – 1 nebo *Druhy::Treti::Tiskni()* z příkladu C12 – 2 ukazuje, že kvalifikovaná jména mohou být zejména u vnořených prostorů jmen nepříjemně dlouhá a brzy by nás odnaučila prostory jmen používat. Pokud s nějakým takovým prostorem jmen pracujeme častěji, vyplatí se ho přejmenovat – přesněji řečeno, dát mu přezdívku neboli alias. K tomu poslouží deklarace tvaru

namespace *alias* = *jméno*;

kde *alias* je nově zaváděné označení prostoru jmen (přezdívka) a *jméno* je jméno existujícího prostoru jmen (případně včetně kvalifikace, jde-li o vnořený prostor jmen). Pokud nám tedy v předchozím příkladu připadá zdlouhavé psát *Druhy::Treti* nebo *Nas_Prvni_Prostor_Jmen*, můžeme si tyto prostory jmen překřtít:

```
namespace DT = Druhy::Treti;
namespace NPPJ = Nas_Prvni_Prostor_Jmen;
```

Příkaz, označený ve funkci *main()* v příkladu C12 – 1 jednou hvězdičkou, pak můžeme přepsat do tvaru

```
cout << "Nas_Prvni_Prostor_Jmen::n = " << NPPJ::n << endl;    // *
```

a příkaz, označený v příkladu C12 – 2 třemi hvězdičkami, do tvaru

```
DT::Tiskni();                                                  // ***
```

Deklarace po částech

Deklaraci prostoru jmen můžeme rozdělit na několik částí. Např. takto:

```
namespace A{ int a; }

namespace B{
  int g() {/* ...*/
}

namespace A{ int b; }
```

Proměnné *a* a *b* zde budou ležet v témže prostoru jmen *A*. Pro jejich používání budou samozřejmě platit obvyklá pravidla: jméno musíme nejprve deklarovat, pak je můžeme použít. To znamená, že např. ve funkci *B::g()* můžeme použít proměnnou *A::a*, nikoli však *A::b*. Z toho plyne, že kdybychom doplnili definici funkce *B::g()* takto:

```
namespace B{
  int g() {                  // NELZE
       return A::a + A::b;
  }
```

```
}
```

zlobil by se překladač, že proměnnou *A::b* nezná, neboť ji deklarujeme až za místem použití.

Stačí ovšem ponechat v deklaraci prostoru *B* pouze prototyp funkce *g()* a definiční deklaraci uvést až za deklarací druhé části prostoru *A* a překladač bude spokojen:

```
/*  Příklad C12 — 3   */
namespace A {int a = 999;}
namespace B {int g();}
namespace A {int b = 1212;}

// OK, teď už překladač zná A::a i A::b
int B::g(){
  return A::a+A::b;
}
```

Anonymní prostor jmen

V deklaraci prostoru jmen můžeme vynechat identifikátor. Pokud to uděláme, vznikne anonymní prostor jmen. Podívejme se na příklad:

```
namespace {
  class X {
       int i;
    public:
       X();
  };
}
```

Na jména, deklarovaná v anonymním prostoru jmen, se odvoláváme bez kvalifikace, podobně jako na globální proměnné, které neleží v žádném prostoru jmen.

Všechny anonymní globální prostory jmen ve stejném souboru (tj. všechny anonymní prostory, deklarované na úrovni souboru) spojí překladač v jeden prostor a přidělí mu jakési jednoznačné vnitřní jméno, v každém souboru jiné. To znamená, že proměnné a funkce, které v něm deklarujeme, se budou chovat jako statické – nebudeme je moci používat v jiných samostatně překládaných souborech.

13.3 using

Kdybychom opravdu museli explicitně kvalifikovat každý identifikátor, který chceme použít mimo jeho vlastní prostor jmen, nepochybně bychom brzy prostorů jmen přestali používat (a nezachránily by to ani přezdívky). Naštěstí máme dvě možnosti, jak kvalifikaci obejít. Jednu z nich představuje direktiva **using** a druhou je deklarace **using**.

Deklarace using

Tato deklarace má tvar

using *prostor_jmen*::*identifikátor;*

kde *prostor_jmen* je identifikátor prostoru jmen, v případě potřeby kvalifikovaný jménem nadřízeného prostoru, a *identifikátor* je jméno, které chceme používat. Kdybychom např. v příkladu C12 – 2 uvedli na počátku funkce *main()* deklarace

```
using Prvni::Tiskni;
using DT:Text;
```

bude v oboru viditelnosti této deklarace zápis *Tiskni()* znamenat vždy volání funkce *Prvni::Tiskni()* a zápis *Text* proměnnou *Druhy::Treti::Text*.

Jedna deklarace **using** může zpřístupňovat pouze jedno jméno. Ovšem v jednom oboru viditelnosti může být deklarací **using** více, pokud nezpůsobí nejednoznačnost. Kdybychom např. ve funkci *main()* v příkladu C12 – 2 deklarovali

```
using Prvni::Tiskni;
using BG::Tiskni;
```

způsobilo by volání

```
Tiskni();
```

chybu, neboť překladač by neuměl rozhodnout, kterou z funkcí *Tiskni()* má volat.

Direktiva using

Jestliže chceme v jednom místě programu používat více identifikátorů ze stejného prostoru jmen, použijeme místo deklarace **using** direktivu **using**. Ta má tvar

using namespace *prostor_jmen***;**

kde *prostor_jmen* je jméno zpřístupňovaného prostoru. Direktiva **using** umožňuje používat všechny identifikátory z daného prostoru jmen bez kvalifikace (opět pokud nedojde k nejednoznačnosti). Zůstaneme ještě u příkladu C12 – 2. Jestliže ve funkci *main()* napíšeme

```
int main(){
   using namespace Prvni;
// Zde se použije Prvni::Text
   Text = "Nazdar";
   Tiskni();
// ...
}
```

použije překladač funkci proměnnou *Prvni::Text* a funkci *Prvni::Tiskni()*, takže přeložený program vypíše *Nazdar*.

Prostory jmen a třídy

Třídy a struktury jsou ze syntaktického hlediska pokládány za prostory jmen. Mají ovšem své zvláštnosti. Uvnitř deklarace objektového typu nelze použít direktivu **using**, která zpřístupňuje celý prostor jmen. Můžeme tam však použít deklaraci **using** a s její pomocí zpřístupnit veřejně přístupné jméno, deklarované ve veřejně přístupném předkovi.

Ukážeme si opět jednoduchý příklad:

```
/*  Příklad C12 – 4   */
#include <iostream.h>

// Předek: třída je prostor jmen
struct Alfa {
int a;
  Alfa(int i=0):a(i){}
  void Tisk(){
       cout << "Trida Alfa: " << a << endl;
  }
};

// Potomek, ve kterém přetížíme
// zděděnou metodu tisk
struct Beta: public Alfa {
  int a;
  Beta(int i=0): Alfa(i), a(i) {}
  using Alfa::Tisk;
  void Tisk(int i){ cout << "Trida Beta: " << a*i << endl;
  }
};

int main(){
  Beta b(11);
  b.Tisk();          // Alfa::pis
  b.Tisk(3);         // Beta::pis
  return 0;
}
```

Ve třídě *Alfa* deklarujeme veřejně přístupnou metodu *Tisk(void)* bez parametrů, která vypíše hodnotu atributu *a*. Od této třídy odvodíme veřejného potomka, třídu *Beta*, ve které z nedostatku fantazie deklarujeme opět atribut *a* a veřejně přístupnou metodu *Tisk(int)*, která má tentokrát jeden parametr typu **int**.

Za normálních okolností by nyní byla zděděná metoda *Tisk(void)* bez parametrů nepřístupná bez explicitní kvalifikace a museli bychom ji volat zápisem

```
b.Alfa::Tisk();
```

protože jsme však ve třídě *Beta* použili deklaraci **using**, zpřístupnili jsme si tím zděděnou metodu, takže překladač přijme zápis

```
b.Tisk();
```

bez námitek.

13.4 Vyhledávání operátorů

Při použití operátoru nelze uplatnit kvalifikaci (pokud nechceme operátory volat jako obyčejné funkce nebo metody, ale to bychom je nemuseli deklarovat jako operátory).

Operátory se vyhledávají jak v kontextu jejich použití, tak i v kontextu jejich operandů. Přitom „kontext“ operandu se skládá z prostoru jmen, ve kterém je deklarován,

a kontext použití znamená prostor jmen, do kterého patří funkce, v jejímž těle operátor použijeme. Na množinu nalezených operátorů se pak uplatní standardní pravidla pro rozlišování přetížených funkcí. Podívejme se na dva příklady:

```
/*  Příklad C12 – 5   */
#include <iostream.h>

namespace Alfa {
  class X{
       int a;
   public:
       X():a(0){}
       X& operator++(){a++; return *this;}
  };
}

// Operátor ++ deklarován v kontextu operandu
namespace Beta {
  void f(Alfa::X &aa){
       ++aa;
  }
}
```

V příkladu C12 – 5 jsme operátor „++“, deklarovaný v prostoru jmen *Alfa*, použili na operand typu *Alfa::X*. Operátor je definován v kontextu operandu, takže jej překladač bez problémů najde, i když jej použijeme v prostoru *Beta*.

Druhý příklad:

```
/*  Příklad C12 – 6   */
namespace Alfa {
  struct X;
}

namespace Beta {
  void f(Alfa::X &aa);
  Alfa::X& operator++(Alfa::X&);
}

struct Alfa::X{
  int a;
 public:
  X():a(0){}
};

Alfa::X& Beta::operator++(Alfa::X& a){
  a.a++;
  return a;
}

// Operátor ++ deklarován v kontextu použití
void Beta::f(Alfa::X &aa){
  ++aa;
}
```

Také příklad C12 – 6 se přeloží bez problémů, neboť operátor „++“ je deklarován v prostoru jmen *Beta*, tedy v kontextu, kde jsme jej také použili.

14. Dodatek

Tato kapitola obsahuje informace, které se jinam nehodily, ale které jsme se cítili povinni do knihy zařadit.

14.1 _CLASSDEF a makra, která s ním souvisí

V praktickém programování se často setkáváme se situacemi, kdy potřebujeme spolu s právě definovaným typem definovat i ukazatel na tento typ, referenci na něj a případně i některé další odvozené typy. Borlandské překladače proto nabízejí v hlavičkovém souboru *_defs.h* makro, které nám umožní jedním příkazem definovat celou skupinu typů souvisejících se zadaným typem.

Pokud v této definici vynecháme některé pomocné konstrukce, které mají zaručit, aby byla definice korektní ve všech paměťových modelech a v některých dalších speciálních situacích (jedná se o specifika implementací jazyka C++ na mikroprocesorech řady Intel 80x86), budou mít tyto definice následující tvar:

```
// Zjednodušená verze některých definic ze souboru _DEFS.H
// v Borland C++
#define _PTRDEF(name)       typedef name * P##name;
#define _REFDEF(name)       typedef name & R##name;
#define _REFPTRDEF(name)    typedef name *& RP##name;
#define _PTRCONSTDEF(name)  typedef const name * PC##name;
#define _REFCONSTDEF(name)  typedef const name & RC##name;
#define _CLASSDEF(name)     class name ; \
                            _PTRDEF ( name ) \
                            _REFDEF ( name ) \
                            _REFPTRDEF ( name ) \
                            _PTRCONSTDEF( name ) \
                            _REFCONSTDEF( name )
```

Jak vidíte, prvních pět maker definuje k danému datovému typu pět různých spřažených datových typů. Identifikátory těchto odvozených typů jsou složeny z identifikátoru původního typu, ke kterému je přidána předpona, charakterizující nový typ.

Pokud bude mít původní datový typ identifikátor *Typ*, pak první makro definuje ukazatel na tento typ, který bude mít identifikátor *PTyp*, druhé makro definuje referenci na tento typ a přiřadí ji identifikátor *RTyp*, třetí makro definuje referenci na ukazatel na náš typ a přiřadí jí identifikátor *PRTyp*, čtvrté makro definuje ukazatel na konstantu daného typu s identifikátorem *PCTyp* a páté makro definuje odkaz (referenci) na konstantu daného typu s identifikátorem *RCTyp*.

Všech těchto pět maker je vlastně pouze pomocných, protože v programech se většinou používá až šesté makro, *_CLASSDEF*, které výše zmíněných pět maker zavolá postupně jedno po druhém. (Připomínáme, že obrácené lomítko na konci řádku oznamuje, že makro na dalším řádku pokračuje.)

Nyní nás nepřekvapí, že se (zejména starší) borlandské zdrojové texty hemží makry *_CLASSDEF*. Zopakujme si, že nezjednodušené verze výše popisovaných maker najdeme v souboru *_DEFS.H*. Používá je však i řada dalších hlavičkových souborů, takže pokud např. do svého programu vložíme hlavičkový soubor *iostream.h*, máme tato makra k dispozici.

Podívejme se, jak by mohla vypadat definice třídy *cObject*, se kterou jsme se setkali ve 2. kapitole, a některých „spřízněných“ typů:

```
#include <stddef.h>
#include <iostream.h>

typedef unsigned int classType;

_CLASSDEF( cObject )
class cObject
{
  public:
       virtual ~cObject() {}
       virtual classType isA() const = 0;
       virtual char * nameOf() const = 0;
       virtual int isEqual( const cObject& ) const = 0;
       virtual void printOn( ostream & ) const = 0;
       friend ostream& operator << ( ostream&, const cObject& );
};

inline ostream& operator <<
( ostream& out, const cObject& obj )
{
  obj.printOn( out );
  return out;
}

inline int operator ==
( const cObject& test1, const cObject& test2 )
{
  return (test1.isA() == test2.isA()) && test1.isEqual( test2 );
}

inline int operator !=
( const cObject& test1, const cObject& test2 )
{
  return !( test1 == test2 );
}
```

14.2 O borlandské grafice

Předpokládáme, že se základy používání borlandské grafiky (BGI) jste dostatečně obeznámeni. Zde si povíme pouze o tzv. „registraci“ grafického ovladače a o problémech, které s tím souvisí.

Jak jistě víte, je při inicializaci borlandské grafiky nutno zadat funkci *initgraph()* jako jeden z parametrů řetězec, obsahující cestu ke grafickému ovladači (driveru). Ovla-

dač (např. soubor *EGAVGA.BGI*) je zpravidla umístěn v poddresáři *\BGI* v domovském adresáři borlandského Céčka nebo Pascalu.

Problém ovšem je, že soubory s ovladači a s vektorovými fonty nelze dále distribuovat. Jestliže tedy chceme svůj program dále šířit, musíme potřebné soubory konvertovat do formátu .OBJ a nechat překladač resp. linker, aby je připojil ke spustitelnému souboru.

K tomu slouží pomocný program *BGIOBJ.EXE*, který najdete v adresáři *\BGI*. Příkaz pro převod ovladače do formátu relativního souboru má tvar

```
BGIOBJ [/F] OVLADAČ [REL_SOUBOR] [JM_FUNKCE] [JM_SEG] [TŘ_SEG]
```

Všechny parametry kromě jména ovladače jsou nepovinné, ale vyplatí se vědět o nich a občas je použít.

Program *BGIOBJ* vytvoří z ovladače – např. z *EGAVGA.BGI* – funkci jménem *JM_FUNKCE* typu **void** (tedy proceduru) bez parametrů. Tuto funkci uloží do souboru *REL_SOUBOR.OBJ*. Parametr /F určuje, zda se vytvoří blízká nebo vzdálená funkce.

Poslední dva parametry určují jméno a třídu kódového segmentu, ve kterém bude tato funkce ležet.

Ovšem pozor: zde můžeme narazit na první problém. *JM_FUNKCE* **je jméno ve tvaru, jak je má vidět linker**, a ten se nemusí shodovat – obvykle **neshoduje** – **s tvarem, v jakém je vidí programátor ve zdrojovém textu**.

Pokud chceme např. takto konvertovat ovladač *EGAVGA.BGI*, vytvořit z něj blízkou funkci *void ev_driver()*, uložit ji v souboru *EVDR.OBJ*, a tuto funkci chceme používat v programu v C++ v malém paměťovém modelu, zadáme příkaz

```
BGIOBJ EGAVGA.BGI EVDR.OBJ @ev_driver$qv _TEXT
```

„Vnitřní“ jméno funkce typu **void** bez parametrů vznikne v Borland C++ připojením znaku „@“ (zavináč) před identifikátor a řetězce „$qv“ za něj. *_TEXT* je standardní jméno kódového segmentu v modelech, které používají blízké ukazatele na kód.

Pokud bychom tutéž funkci potřebovali v programu v jazyku C, museli bychom uvést vnitřní jméno *_ev_driver* (tj. připojit před identifikátor podtržítko).

Podobně postupujeme i při programování v Turbo Pascalu. „Vnitřní“ jméno ovšem tentokrát bude *EV_DRIVER* (všechna písmena se převedou na velká, jinak se identifikátor neupravuje):

```
BGIOBJ EGAVGA.BGI GDRIVER.OBJ EV_DRIVER _TEXT
```

V programu pak tuto funkci musíme deklarovat. To znamená, že v C/C++ zapíšeme její prototyp (a soubor .OBJ, který ji obsahuje, připojíme k projektu). V Pascalu ji deklarujeme s direktivou *external* a direktivou v komentáři („dolarovou poznámkou“) sdělíme překladači jméno souboru, ve kterém ji najde. Např. takto:

```
procedure ev_driver; external;
{$L gdriver.obj}
```

Potom musíme tuto funkci zaregistrovat pomocí knihovní funkce *registerbgidriver* nebo (pokud jsme konverzí ovladače vytvořili vzdálenou funkci) *registerfarbgidriver*:

```
void ev_driver(void);
int i = registerbgidriver(ev_driver);
if (i < 0) chyba(reg);
```

resp.

```
vysl := registerbgidriver(@ev_driver);
if(vysl < 1) then chyba_proc(reg);
```

Funkce *registerbgidriver* má jako parametr ukazatel na grafický ovladač (v Pascalu *pointer*). Pokud tato funkce vrátí nulu nebo zápornou hodnotu, došlo při registraci k chybě. Podrobnější informace o ní najdete v souboru *UTILS.TXT*, který je součástí instalace, a v nápovědě.

14.3 O myši a jiných hlodavcích

Používání myši nebo jiného ukazovátka v dosovských programech je ve skutečnosti velice jednoduché. Většinu práce obstará rezidentní ovladač, instalovaný zpravidla již při spuštění počítače. Tento ovladač reaguje na pohyby myši, na stisknutí tlačítek apod. S okolím spolupracuje prostřednictvím jakýchsi datových struktur, uložených v paměti.

Programátor může s ovladačem myši komunikovat prostřednictvím služeb, napojených na přerušení 0x33 (desítkově 51). Přitom postupuje podobně jako např. při volání služeb jádra DOSu: do registru *AX* uloží číslo funkce (tedy číslo, kterým říká, kterou službu požaduje), do ostatních registrů případné parametry a vyvolá uvedené přerušení. Vracené hodnoty najde opět v registrech procesoru.

Přehled vybraných služeb ovladače myši najdete v následující tabulce.

Služba	Význam
0	naváže spojení s ovladačem (resetuje jej)
1	povolí zobrazení kurzoru
2	zakáže zobrazení kurzoru
3	čte polohu myši a stav tlačítek
4	nastaví polohu myši
5	čte počet stisknutí tlačítka a polohu posledního stisknutí
6	čte počet puštění tlačítka a polohu posledního puštění

Tab. 13.1 Vybrané služby ovladače myši

Další služby umožňují omezit pohyb myši na určitou oblast obrazovky, nastavit citlivost, změnit tvar kurzoru atd.

V příkladu v kapitole 5 jsme potřebovali služby 0, 1, 2 a 6. Funkce pro práci s myší se obvykle píší v asembleru. Lze je ale napsat i v Borland C++, použijeme-li pseudoproměnných *_AX*, *_BX* atd. pro práci s registry a funkci *geninterrupt()* pro vyvolání pře-

rušení (její prototyp je v *dos.h*). V Pascalu můžeme pro práci s registry procesoru použít předdefinovanou strukturu *registers*.

Podívejme se na funkci, která navazuje spojení s ovladačem. V C++ ji můžeme napsat takto:

```
#include <dos.h>

const int MysInt = 0x33;    // Číslo přerušení pro spolupráci s myší

int mys::Zjisti(){
   _AX=0;                   // Číslo služby: 0, bez parametrů
   geninterrupt(MysInt);    // Vyvoláme přerušení
   if(!_AX) return 0;       // V AX je informace o přítomnosti ovladače
   else return _BX;         // V BX je počet tlačítek
}
```

Táž metoda má v Turbo Pascalu tvar

```
const
  MysInt=$33;              { Číslo přerušení pro spolupráci s myší}

function tmys.Zjisti: integer;
var regs: registers;
begin
   regs.ax := 0;           { Číslo služby: 0, bez parametrů }
   intr(MysInt, regs);     { Vyvoláme přerušení }
   if(regs.ax = 0) then Zjisti := 0     { Není-li instalována, vrať 0}
 else Zjisti := regs.bx;   { jinak vrať počet tlačítek – je v BX }
end;
```

Služba 0 vrátí v registru *AX* hodnotu -1, je-li ovladač myši instalován, a 0, jestliže instalován není. Pokud je ovladač instalován, vrátí v registru *BX* počet tlačítek. Funkce *Zjisti()* tedy vrátí 0 nebo počet tlačítek.

Funkce pro zobrazení a skrytí kurzoru jsou mimořádně jednoduché. Uvedeme si pouze jednu z nich. V C++ bude mít tvar

```
// Zobrazí kurzor myši
void mys::ZobrazKurzor(){
   _AX = 1;                      // Číslo služby
   geninterrupt(MysInt);
}
```

a v Pascalu

```
{ Zobrazí kurzor myši }
procedure tmys.ZobrazKurzor;
var regs: registers;
begin
   regs.ax := 1;                 { Číslo služby }
   intr(MysInt, regs);
end;
```

Poznámka:

Před každým kreslením nebo mazáním na obrazovce je třeba skrýt kurzor myši, neboť jinak mohou na obrazovce zůstávat „zbytky“ – kousky čar nebo barevné skvrny v místech, která kurzor zakrýval.

Chceme-li zjistit, kolikrát bylo puštěno levé tlačítko a kde bylo puštěno naposledy, použijeme službu 6. (Zavoláme-li tuto službu, začnou se uvolnění tlačítka počítat znovu od hodnoty 0.)

Služba 6 očekává v registru *BX* číslo určující, zda se zajímáme o levé (0) nebo pravé (1) tlačítko. V *BX* vrátí počet uvolnění levého tlačítka, v *CX* resp. *DX* vrátí obrazovkovou souřadnici *x* resp. *y* kurzoru v okamžiku posledního puštění. Navíc vrací tato služba v registru *AX* okamžitý stav zadaného tlačítka myši.

Metoda *PusteniLeveho* má tedy parametry *x* a *y*, předávané odkazem, ve kterých se vrátí souřadnice místa, kde bylo levé tlačítko myši uvolněno. Funkční hodnota obsahuje počet puštění tlačítka od posledního volání této metody. V C++ má tvar

```
int mys::PusteniLeveho(int &x, int &y){
  _AX = 6;                  // Číslo služby
  _BX = 0;                  // Zajímá nás levé tlačítko
  geninterrupt(MysInt);
  if(_BX ){                 // Pokud bylo alespoň jednou stisknuto
       _AX = _BX;
       x = _CX;             // vrať souřadnice v parametrech
       y = _DX;
       return _AX;          // a vrať počet stisknutí
  }
  else return 0;            // jinak vrať 0
}
```

a v Turbo Pascalu

```
function tmys.PusteniLeveho(var x, y: integer): integer;
var regs: registers;
begin
  regs.ax := 6;                  { Číslo služby }
  regs.bx := 0;                  { Zajímá nás levé tlačítko }
  intr(MysInt, regs);
  if(regs.BX <> 0 ) then begin   { Pokud bylo alespoň jednou stisknuto }
       regs.AX := regs.BX;
       x := regs.CX;             { vrať v parametrech souřadnice }
       y := regs.DX;
       PusteniLeveho := regs.AX; { a vrať počet stisknutí }
  end
  else PusteniLeveho := 0;       { jinak vrať 0 }
end;
```

Další podrobnosti lze najít v dokumentaci. My jsme např. čerpali informace ze souboru *GMOUSE.DOC*, který je součástí dodávky myši *Genius*.

Pseudoproměnné

Rádi bych také upozornili čtenáře na jedno nebezpečí, se kterým se mohou setkat při používání pseudoproměnných *_AX, _BX* atd. v Borland C/C++.

Při překladu přiřazovacích příkazů používá překladač registrů a přitom nám může pokazit hodnoty, které v nich máme. Kdybychom např. přepsali funkci *mys::PusteniLeveho()* do (zdánlivě logičtějšího) tvaru

```
int mys::PusteniLeveho(int &x, int &y){
   _AX = 6;
   _BX = 0;
   geninterrupt(MysInt);
   if(_BX ){
        x = _CX;
        y = _DX;
        return _BX;
   }
   else return 0;
}
```

nefungovala by. Při přiřazení

```
x = _CX;
```

se totiž použije registr *BX* a jeho hodnota se přitom přemaže.

Při používání pseudoproměnných je tedy nezbytná maximální opatrnost; rozumnější je často použít příkazu **asm** a naprogramovat takovéto jednoduché operace přímo v asembleru.

14.4 Dlouhý skok

V kapitole o výjimkách jsme se také zmínili o tzv. „dlouhém skoku“. Je to pravděpodobně nejjednodušší možnost mimořádného přenosu řízení mezi funkcemi, kterou najdeme již v ANSI C. Umožňuje je dvojice standardních funkcí *setjmp()* a *longjmp()*, jejichž prototypy jsou v hlavičkovém souboru *setjmp.h*. I když dlouhé skoky nepředstavují skutečné ošetřování výjimek, zastavíme se u nich, neboť při programování v ANSI C vlastně jinou možnost nemáme.

Představte si, že v těle funkce *main()* chceme volat funkci *f()*, v *f()* budeme volat *g()* a v *g()* budeme volat *h()*. Pokud ale zjistíme v kterékoli z těchto funkcí chybu, potřebujeme se vrátit do funkce *main()* těsně za volání *f()*. Musíme si tedy nejprve zaznamenat stav programu v místě, do kterého se budeme vracet, a pak se můžeme vydat na cestu do nebezpečných míst, ze kterých budeme možná muset utíkat.

K záznamu okamžitého stavu programu (tj. k uložení všech registrů procesoru) slouží standardní funkce *setjmp()*. Pro uložení stavu registrů používá proměnnou (pole) typu *jmp_buf*. Její prototyp je *int setjmp(jmp_buf bafr)*.

Po zaznamenání stavu procesu (tedy po prvním průchodu tímto místem) vrátí funkce *setjmp()* hodnotu 0.

Jestliže později usoudíme, že je na čase z místa výpočtu zbaběle utéci a vrátit se do funkce *main()*, použijeme funkci *jongjmp()*. Její prototyp je *void longjmp(jmp_buf bafr, int kód)*.

Parametr *bafr* je proměnná, obsahující zaznamenaný stav programu v místě, do kterého se chceme vrátit. (Můžeme si zaznamenat stav programu v několika místech do několika proměnných typu *jmp_buf* a podle okolností se rozhodnout, kam se vrátíme.) Parametr *kód* může nést dodatečné informace – o místě, odkud se vracíme, o druhu chyby apod.

Volání funkce *longjmp()* způsobí návrat do místa, kde jsme si zaznamenali stav programu v parametru *bafr*, to znamená do místa, kde jsme zavolali funkci *setjmp()*. Program se tedy bude chovat, jako by se právě vrátil z funkce *setjmp()*; vrácená hodnota bude *kód*, druhý parametr, použitý při volání *longjmp()*.

Podívejme se na příklad:

```
/*   Příklad CD – 1   */
#include <setjmp.h>
#include <iostream.h>

void f(int), g(int), h(int);
jmp_buf bafr1;                    // Proměnná pro záznam stavu

int main(){
   int i, kod = -1;
   for(i = 1; i < 4; i++){
        kod = setjmp(bafr1);      // Sem se budeme vracet
        if(!kod) f(i);            // Po návratu volání f() přeskočíme
        cout << "Funkce main, kód = " << kod << endl;
   }
   return 0;
}

void f(int j){
   cout << "funkce f při j = " << j << endl;
   if(j == 1)longjmp(bafr1, 1);   // Návrat dlouhým skokem
   else g(j);
}

void g(int j){
   cout << "funkce g při j = " << j << endl;
   if(j == 2)longjmp(bafr1, 2);
   else h(j);
}

void h(int j){
   cout << "funkce h při j = " << j << endl;
   longjmp(bafr1, 3);
}
```

Ve funkci *main()* jsme si zaznamenali stav programu a uložili jsme si jej do proměnné *bafr1*. Funkce *setjmp()* při záznamu vrátila 0, takže následující příkaz zavolal funkci *f()*.

Je-li ve funkci *f()* parametr *j* roven 1, zavolá se funkce *longjmp()*. Ta způsobí návrat na místo, kde jsme si zaznamenali stav do proměnné *bafr1*, tedy do funkce *main()*. Ve funkci *main()* se tento návrat bude jevit jako návrat z funkce *setjmp()*; do proměnné *kokd* se uloží návratový kód, (druhý parametr funkce *longmp()*), tedy 1.

Poznámka:
Kdybychom nekontrolovali vracenou hodnotu, mohli bychom uváznout v nekonečném cyklu, neboť funkce longjmp() nás vrátí před volání funkce f().

14.5 Standardní knihovna jazyka C++

Povídání o standardní knihovně jazyka C++ by samo o sobě vystačilo na několikadílnou knihu. My si zde o ní povíme alespoň nezbytné minimum, ve kterém se pokusíme ukázat, co všechno v ní můžeme najít.

Standardní knihovna jazyka C++ je součástí nové normy tohoto jazyka. S její první implementací se setkáme v Borland C++ 5.0, starší překladače ji neobsahují. Tato implementace pochází od softwarové firmy *Rogue Wave*.

Hlavičkové soubory standardní knihovny C++ se direktivách **#include** uvádějí bez přípony *.h*. Jejich jména mohou být i delší než 8 znaků.

Standardní knihovna C++ obsahuje mj. následující části:

- ✧ řadu předdefinovaných datových struktur a algoritmů pro práci s nimi, implementovaných pomocí šablon (tato část bývá obvykle označována jako standardní šablonová knihovna – *standard template library*, STL),
- ✧ vstupní a výstupní datové proudy,
- ✧ prostředky pro nastavení lokálních zvyklostí,
- ✧ šablonu třídy *string* pro práci se znakovými řetězci,
- ✧ šablony třídy pro reprezentaci komplexních čísel,
- ✧ jednotné nástroje pro popis prostředí, ve kterém probíhá výpočet, zprostředkovaný šablonou *numeric_limits*,
- ✧ nástroje pro správu paměti,
- ✧ nástroje pro ošetřování výjimek.

Tato knihovna není implementována důsledně objektově. Téměř nepoužívá dědičnost a také zapouzdření nepoužívá důsledně. Většinu datových struktur definuje jako šablony tříd; tam, kde to bylo výhodné s ohledem na efektivnost, zůstala však data oddělena od operace s nimi. Díky tomu můžeme stejné algoritmy používat např. na seznamy, s jejichž prvky pracujeme pomocí iterátorů, stejně jako na pole, s jejichž prvky pracujeme pomocí konvenčních ukazatelů. (Podobně jsme postupovali v kapitole 7 (*Šablony*) v příkladu C7 – 8.)

Součásti standardní knihovny sdílí společný prostor jmen *std*; obvykle jej zpřístupňujeme pomocí deklarace

```
using namespace std;
```

Pokud se nám to z nějakých důvodů nehodí nebo nelíbí, můžeme si poručit, aby součásti standardní knihovny ležely mimo jakýkoli prostor jmen tím, že na počátku svého programu před vložením prvního hlavičkového souboru této knihovny #definujeme makro

RWSTD_NO_NAMESPACE. (Pak ale může zase dojít ke konfliktům se jmény z ostatních hlavičkových souborů.)

Kontejnery a iterátory

Ve standardní knihovně najdeme šablony pro vektory (jednorozměrná pole), seznamy, dvoustranné fronty, množiny, mapy, zásobníky, fronty a fronty s předbíháním a další. Pro přístup k uloženým datům ve většině případů používají iterátory.

Poznámka:
Kontejnery, ve kterých lze prvky uspořádat podle velikosti a v takovémto pořadí pak snadno procházet, budeme v této kapitole označovat také jako ***posloupnosti****. Jde především o pole, zásobníky apod.*

Iterátory

Iterátory ve standardní knihovně se velice podobají ukazatelům. Např. k hodnotě, na kterou iterátor ukazuje, přistupujeme pomocí operátoru „*“; dva iterátory se sobě rovnají, ukazují-li na týž prvek apod. Jestliže určíme pro některý z knihovních algoritmů pomocí dvou iterátorů *iter1* a *iter2* nějaké rozmezí hodnot ke zpracování, bude počáteční hodnota, **iter1*, součástí tohoto rozmezí, zatímco koncová hodnota, **iter2*, nikoli – podobně jako u prvků pole. To znamená, že musíme mít k dispozici také iterátor, který se chová, jako by ukazovat na neexistující prvek za posledním prvkem kontejneru.

Iterátory můžeme „posunout“ na následující prvek pomocí operátoru „++“, na předchozí pomocí „--“.

Vzhledem k rozdílným vlastnostem kontejnerů se ovšem rozlišují i různé druhy iterátorů: iterátory s náhodným přístupem, jež mohou libovolně „přebíhat“ z prvku na prvek, iterátory dopředné, které se mohou „posunovat“ po kontejneru pouze vpřed atd.

Kontejnery

Kontejnery jsou ve standardní knihovně definovány pomocí šablon. Podívejme se podrobněji na některé z nich.

Pole

Třída *vector<T>*, jejíž šablona je definována v hlavičkovém souboru *vector*, je zobecněním klasického céčkovského pole. Do prvků objektu typu *vector<T>* lze ukládat hodnoty typu *T*. S prvky lze zacházet také pomocí operátoru indexování (prvky jsou indexovány od 0). Na rozdíl od klasického pole se velikost vektoru může dynamicky měnit.

Vektor *v* s 10 prvky typu **int** deklarujeme zápisem

```
#include <vector>
vector<int> v(10);
```

Pomocí metod třídy *vector* lze zjistit velikost vektoru a počet prvků, které vektor obsahuje. Dále tu najdeme také přiřazovací operátor, metody pro vkládání a odstraňování prvků v určitém rozsahu, pro změnu velikosti vektoru, prohození prvků mezi dvěma

vektory apod. Součástí implementace je také operátor pro lexikografické porovnávání dvou vektorů.

Seznam
Šablona seznamu, do kterého lze ukládat hodnoty typu *T*, tedy třídy *list*<*T*>, je v hlavičkovém souboru *list*. Tato třída implementuje dvousměrný seznam, do jehož prvků lze ukládat hodnoty typu *T*.

Pro seznamy je definován mimo jiné přiřazovací operátor, který umožňuje přenesení obsahu jednoho seznamu do jiného. Dále máme k dispozici procedury na zjišťování velikosti, na třídění a slučování seznamů, na vkládání a odstraňování prvku ze seznamu atd. Seznamy lze také porovnávat pomocí operátorů „==“ a „<“ (porovnání je lexikografické).

Fronta
Fronta je implementována v STL jako adaptér – tedy třída, která obsahuje nějaký kontejner a využívá jeho služeb. Při deklaraci fronty musíme proto vedle typu ukládaných dat určit, jaký kontejner bude základem fronty. Implicitně použije překladač dvoustrannou frontu (*deque*). Deklarace fronty celých čísel, založené na seznamu, může mít tvar

```
#include <queue>
#include <list>
queue<int, list<int> > Fronta;
```

Metody *front()*, resp. *back()* umožňují zjistit prvek na počátku, resp. na konci fronty, aniž by jej odstranily. Metoda *pop()* vyjme prvek z čela fronty, metoda *push()* vloží prvek na konec fronty. Další metody umožňují zjistit, zda je fronta prázdná, příp. její velikost.

Dvoustranná fronta
Připomeňme si, že dvoustranná fronta (šablonu najdeme v hlavičkovém souboru *deque*) je struktura podobná frontě, která umožňuje efektivně přidávat a odebírat prvky na obou koncích. S prvky dvoustranné fronty typu *deque*<*T*> lze pracovat také pomocí indexů. V mnoha ohledech se chová jako vektor a seznam dohromady.

Metody pro dvoustranné fronty umožňují vložit nebo vyjmout jeden nebo několik prvků doprostřed nebo na jeden z konců, zjistit, zda je prázdná, změnit její velikost, prohodit prvky s jinou dvoustrannou frontou atd.

Zásobník
Šablona zásobníku je definována v hlavičkovém souboru *stack*. Zásobník ve standardní knihovně C++ je opět adaptér, implementovaný pomocí vektoru, seznamu či jiného kontejneru. Kromě souboru *stack* musíme tedy do programu vložit i hlavičkový soubor pro kontejner, ze kterého si chceme zásobník vytvořit, a v deklaraci zásobníku musíme specifikovat vedle typu ukládaných dat i typ kontejneru, na němž bude zásobník postaven. Neuvedeme-li jej, použije překladač dvoustrannou frontu. Chceme-li např. vytvořit zásobník, založený na seznamu, a ukládat do něj reálná čísla, deklarujeme jej takto[26]:

[26] V zápisu

```
#include <stack>
#include <list>
stack <double, list<double> > dZasob;
```

Mezi metodami zásobníku najdeme *push()* pro vložení prvku na vrchol zásobníku, *pop()* pro vyjmutí prvku z vrcholu zásobníku, *top()*, která vrátí prvek z vrcholu zásobníku, ale nevyjme jej, a metodu pro zjištění počtu prvků v zásobníku.

Generické algoritmy

Součástí standardní knihovny jsou také tzv. generické algoritmy, tedy algoritmy, které lze použít pro libovolný kontejner (případně pro libovolný uspořádaný kontejner). Jsou implementovány jako řadové funkce (nikoli jako metody objektových typů) a s daty pracují pomocí iterátorů. Většinu jejich šablon najdeme v hlavičkovém souboru *algorithm*, několik jich je v souboru *numeric*. Jsou rozděleny do několika skupin.

Inicializační algoritmy

Jde o funkce, které vyplní celou posloupnost – tj. zásobník, pole (vektor) atd. – nebo její část zadanou hodnotou, které okopírují jednu posloupnost do druhé, inicializují posloupnost pomocí zadaného generátoru (např. náhodných čísel) apod.

Vyhledávací algoritmy

Tyto funkce umožňují vyhledat prvek, který splňuje určitou podmínku, opakující se prvky, maximální nebo minimální prvek v posloupnosti, první rozdílný prvek apod.

Transformace na místě

Jde o transformace, které sice posloupnost změní, avšak nepotřebují k tomu dodatečnou paměť. Do této skupiny patří např. převrácení pořadí všech prvků, nahrazení hodnot určitých prvků novými atd.

Počítací algoritmy

Tyto funkce umožňují zjistit počet prvků v posloupnosti, počet prvků, vyhovujících dané podmínce, spočítat součet prvků, nebo zjistit, zda dvě posloupnosti obsahují tytéž prvky apod. Do této skupiny jsou zařazeny i algoritmy pro lexikografické porovnávání.

Další algoritmy

Tyto algoritmy umožňují aplikovat na každý prvek danou funkci, vytvořit posloupnost částečných součtů, vytvořit posloupnost rozdílů následujících prvků apod.

`stack <double, list<double> > dZasob`
je nutná mezera mezi lomenými závorkami na konci, jinak bude překladač hlásit podivné chyby.

Příklad

V následujícím příkladu vytvoříme pomocí šablony *list<T>* ze standardní knihovny seznam celých čísel, který zaplníme 0 – 99. Potom nahradíme všechna čísla dělitelná 5 číslem -10, všechny hodnoty, uložené v seznamu, změníme, a nakonec seznam setřídíme. Po jednotlivých operacích si jej vytiskneme na obrazovku. Program může vypadat takto:

```
/*   Příklad CD – 2   */
// Použití standardní knihovny – seznam a operace s ním
// Pozor, v BC++ 5.0 nutno překládat jako konzolovou aplikaci pro Win32

#include <iomanip>
#include <list>
#include <algorith>
#include <math.h>

const int N = 100;

using namespace std;

void tisk(list<int>& L){                    // Výpis seznamu
   int i = 0;
   cout << "----------------" << endl;
   for
    (list<int>::iterator kde = L.begin();    // ***
     kde != L.end();
     kde++) {
        cout << setw(5) << *kde;
        if (++i == 10 ){ i = 0; cout << endl; }
   }
}
bool Delitelne5(int n){               // Je n dělitelné pěti?
   return (n/5)*5 == n;
}

void zmen(int &m){                    // Změní hodnotu čísla m
   m = sin(m)*100;
}

int main(){
   list<int> num;                                          // 1
   for(int i = 0; i < N; i++) {
        num.push_front(i);                                 // 2
   }
   replace_if(num.begin(), num.end(), Delitelne5, -10); // 3
   tisk(num);
   for_each(num.begin(), num.end(), zmen);                 // 4
   tisk(num);
   num.sort();                                             // 5
   tisk(num);
   return 0;
}
```

Číslem 1 jsme v komentáři označili deklaraci seznamu celých čísel *num*. V následujícím cyklu, označeném číslem 2, vložíme do tohoto seznamu čísla 0 – 99. Protože metoda *push_front()* je vkládá na počátek, budou v seznamu v obráceném pořadí.

V příkazu 3 voláme funkci *replace_if()*, což je generický algoritmus pro nahrazení určitých hodnot hodnotou jinou. První dva parametry této funkce jsou iterátory a určují rozmezí, ve kterém chceme náhrady provést. My jsme použili metody *list<T>::begin()* a *list<T>:: end()*, které vrátí iterátory, ukazující na první a za poslední prvek seznamu, takže se náhrada provede v celém seznamu. Třetím parametrem je ukazatel na *predikát*, funkci, kterou lze aplikovat na hodnoty, uložené v seznamu a jež určuje prvky, které chceme nahrazovat.

V příkazu, označeném 4, voláme jiný generický algoritmus, funkci *for_each()*, jejímž třetím parametrem je funkce, která se zavolá pro každou z hodnot v seznamu. Funkce *zmen()* jako vedlejší efekt uložené hodnoty změní.

V příkazu, označeném 5, seznam setřídíme pomocí metody *sort()*. (Ve standardní knihovně existuje i generický algoritmus pro třídění, ten ale nelze použít pro seznam.)

Podívejme se ještě na funkci *tisk()*, která má za úkol vytisknout seznam na obrazovku (vždy 10 prvků na řádek). Prvky seznamu vypisujeme v cyklu, označeném třemi hvězdičkami. K výpisu používáme iterátor *kde*, který inicializujeme pomocí iterátoru, odkazujícího na první prvek seznamu; operace *kde*++ způsobí „přesun“ iterátoru vždy na následující prvek.

Algoritmy pro setříděné kontejnery

Rozsáhlou skupinu tvoří algoritmy, které lze použít pro uspořádané (setříděné) kontejnery nebo pro kontejnery, jejichž prvky lze setřídit.

Najdeme zde algoritmy pro třídění, pro stabilní třídění, pro třídění části posloupnosti a pro třídění, jehož výsledek se ukládá do kopie původního kontejneru. Dále tu najdeme algoritmy pro vyhledání n-tého největšího prvku, pro binární vyhledávání a pro slučování setříděných posloupností.

Do této skupiny patří také množinové operace – sjednocení, průnik, rozdíl, symetrická diference a test, zda je jedna posloupnost podmnožinou jiné.

Komplexní čísla

V hlavičkovém souboru *complex* jsou definována komplexní čísla pomocí šablony. Díky tomu máme k dispozici komplexní čísla, tvořená dvojicí čísel typu **float**, **double**, **long double**, **int** atd.

Jsou pro ně definovány obvyklé aritmetické operace, operace pro převod komplexních čísel do goniometrického tvaru a zpět, vstupní a výstupní operátory „>>“ a „<<“ atd. Pro komplexní čísla jsou také přetíženy některé běžné matematické funkce jako *cos()*, *sin()*, *asin()*, *acos()*, *atan()*, *sinh()*, *cosh()*, *exp()*, *log()*, *sqrt()* atd.

Řetězce

K práci s řetězci je určena třídy *string*, kterou najdeme ve stejnojmenném hlavičkovém souboru. Je vytvořena pomocí šablony *basic_string*, jež může mít jako parametry různé znakové typy – **char**, **unsigned char** atd. Třída *string* je založena na typu **char**.

Třída *string* je vlastně indexovaný kontejner, jehož délka se může dynamicky měnit. Můžeme pro ni použít výstupní operátor „<<“, operátory „+“ a „+=“ pro spojování řetězců a řadu metod a generických algoritmů. S jednotlivými znaky lze pracovat pomocí operátoru indexování nebo pomocí iterátorů. Řetězce lze také lexikograficky porovnávat pomocí obvyklých relačních operátorů.

Systémové konstanty

V programu potřebujeme občas zjistit různé systémové konstanty, jako největší celé číslo, které lze v daném systému použít, nejmenší nenulové reálné číslo atd. Tyto hodnoty byly v předchozích verzích C++ popisovány pomocí symbolických konstant, #definovaných v souborech *limits.h* a *float.h*.

Standardní knihovna umožňuje jednotný přístup k těmto údajům pomocí šablonové třídy *numeric_limits*, definované v hlavičkovém souboru *limits*. Například nejmenší hodnotu daného typu zjistíme voláním metody *min()*, největší voláním *max()*.

Třída *numeric_limits* musí poskytovat informace o vestavěných číselných typech; různé implementace C++ mohou ovšem informovat i o dalších typech. Pro jakýkoli datový typ *T* můžeme pomocí hodnoty *numeric_limits<T>::is_specialized* zjistit, zda o něm daná implementace informuje.

Pro reálné typy je navíc k dispozic řada metod a složek, které poskytují informace o nejmenším nebo největším možném exponentu, o mezi rozdílu 1 a nejmenším větším číslem atd.

Jako příklad si vypíšeme některé údaje o typu **long double**:

```
/*   Příklad CD – 3   */
#include <limits>
#include <iostream.h>

int main(){
  cout << "TYP long double: " << endl;
  cout << "Nejmenší hodnota: "
       << std::numeric_limits<long double>::min()<< endl;
  cout << "Nejmenší e takové, že 1+e != 1: "
       << std::numeric_limits<long double>::epsilon()<< endl;
  cout << "Má nekonečno? "
       << (std::numeric_limits<long double>::has_infinity
                ? "Ano" : "Ne") << endl;
  cout << "Splňuje standard 559? "
       << (std::numeric_limits<long double>::is_iec559? "Ano" : "Ne");
  return 0;
}
```

Zde jsme se mimo jiné dotazovali, zda tento typ umožňuje v Borland C++ pracovat se strojovým nekonečnem a zda vyhovuje standardu IEC 559 pro výpočty v pohyblivé řádové čárce (jde o normu, která je totožná se standardem IEEE 754). Když jsme tento program přeložili jako konzolovou aplikaci pro Win32, vypsal po spuštění

```
TYP long double:
Nejmenší hodnota: 3.3621e-4932
Nejmenší e takové, že 1+e != 1: 1.0842e-19
Má nekonečno? Ano
Splňuje standard 559? Ano
```

Poznamenejme, že v Borland C++ 5.0 se tento program podaří přeložit jako 32bitovou aplikaci. Budeme-li jej však překládat jako 16bitovou aplikaci (pro DOS nebo pro 16bitová Windows), bude linker hlásit podivné chyby. Jde o jakési nedopatření v překladači.

Automatické ukazatele

Chytré (automatické) ukazatele, *smart pointers*, jsou objekty, které se chovají jako ukazatele, ale při zániku zničí objekt, na který ukazují. Ve standardní knihovně je v hlavičkovém souboru *memory* definována šablona *auto_ptr<X>*, jejíž instance jsou automatické ukazatele na typ *X*.

Třída *auto_ptr<X>* obsahuje mj. přetížené operátory „*“, „–>“, „=“. Destruktor třídy *auto_ptr<X>* volá na alokovaný objekt operátor **delete** (a tím také případně jeho destruktor).

Automatický ukazatel, který ukazuje na nějaký objekt, jej „vlastní“. Při přiřazení mezi dvěma automatickými ukazateli se přenáší i vlastnictví objektu; automatický ukazatel, který vlastnictví ztratí, neukazuje nikam (obsahuje 0).

Podívejme se na příklad, který nám ukáže, k čemu jsou automatické ukazatele dobré. Nejprve bez chytrých ukazatelů:

```
void g(void);

class Kuku {
  public:
       Kuku();
       ~Kuku();
       // ...a další složky
};

void f(){
  Kuku* ukuk = new Kuku;
  g();
  delete ukuk;
}
```

Takto napsaná funkce *f()* má jednu nevýhodu: pokud vznikne ve funkci *g()* výjimka a rozšíří se z ní, skončí *f()* předčasně a operátor **delete**, která má uvolnit instanci třídy *Kuku*, se nezavolá. Použijeme-li automatické ukazatele, bude vše jednodušší:

```
#include <memory>
void g(void);
```

```
class Kuku {
  public:
       Kuku();
       ~Kuku();
       // ...a další složky
};

void f(){
  auto_ptr<Kuku> ukuk = new Kuku;
  g();
}
```

Ve funkci *f()* zde operátor **delete** vůbec explicitně nevoláme, neboť o to se postará automaticky destruktor automatického ukazatele *ukuk*. Navíc máme jistotu, že se operátor **delete** zavolá i v případě, že se z *g()* rozšíří výjimka .

Přitom s instancí *ukuk* můžeme zacházet téměř stejně jako s obyčejným ukazatelem – můžeme např. používat operátory „*“ nebo „–>“. Napíšeme-li ale přiřazení

```
auto_ptr<Kuku> uxux;
uxux = ukuk;
```

bude *ux* obsahovat adresu instance třídy *Kuku*, na kterou předtím ukazovala proměnná *ukuk*, ale *uxux* bude obsahovat 0 – to znamená, že se přiřazením přeneslo vlastnictví objektu. Na danou instanci bude stále ukazovat jediný automatický ukazatel. (K podobnému přenosu vlastnictví dojde i při deklaraci s inicializací, tedy při použití kopírovacího konstruktoru).

14.6 Přehled novinek v C++

Současná verze normy jazyka C++ přinesla řadu změn a nové překladače je implementují. Většina z nich neovlivňuje zpětnou kompatibilitu. To znamená, že starší programy by měly jít bez problémů překládat novými překladači a měly by mít i stejný význam. Existuje ale několik výjimek, které v následujícím přehledu zvýrazníme.

Výčet změn, který v této podkapitole najdete, není (bohužel) konečný, neboť vývoj jazyka ještě neskončil.

Souhrn

- ♦ mění se oblast platnosti deklarace v inicializačním výrazu příkazu **for**,
- ✧ v podmínce příkazů **if**, **for**, **while** a **switch** můžeme nyní deklarovat proměnou,
- ✧ zavádějí se tzv. explicitní konstruktory objektových typů,
- ✧ můžeme deklarovat měnitelné složky konstantních objektů,
- ✧ statické konstantní atributy objektových typů lze inicializovat přímo v deklaraci třídy,
- ✧ pro práci s logickými hodnotami máme k dispozici nový typ **bool**,

- rozlišují se 3 různé znakové typy (**char**, **unsigned char**, **signed char**).

Na změny, které se týkají šablon, jsme upozorňovali průběžně již v kapitole 7. I tam lze ale očekávat ještě další změny – např. pokud jde o možnosti hodnotových parametrů u šablon funkcí. Budoucí změny mohou také poněkud upravit vlastnosti operátorů **dynamic_cast**, **static_cast**, **reinterpret_cast** a **const_cast** (zejména pokud jde o zacházení s konstantami).

Podívejme se nyní na změny, o nichž jsme se zmínili v úvodu této podkapitoly.

Deklarace v příkazech

První z novinek se týká možnosti deklarovat proměnnou v řídicích výrazech některých příkazů.

Deklarace v inicializaci cyklu for

Na to, že v inicializačním výrazu příkazu **for** můžeme deklarovat proměnnou, jsme si již dávno zvykli. Ale pozor**: V nejnovějších překladačích se mění rozsah platnosti této deklarace.** Proměnná, deklarovaná v inicializačním výrazu, je k dispozici pouze v těle cyklu. Nyní tedy příkaz

```
for(int i = 0; i < N; i++) cout << i;
```

znamená totéž co

```
{
  int i;
  for(i = 0; i < N; i++) cout << i;
}
```

To ovšem může občas způsobit problémy. Podívejme se na jednoduchý příklad, který bude znamenat něco jiného ve staré a něco jiného v nové verzi jazyka:

```
/* Příklad CD – 4 */
#include <iostream.h>

int i = 50;                                    // Globální proměnná

int main(){
 int j = 0 ;
 for(int i = 0; i < 10; i++)j += i;            // Deklarace v příkazu for
 cout << i;                                    // *** Které i se vypíše?
 return 0;
}
```

Přeložíme-li tento program např. pomocí Borland C++ 4.x nebo 3.1, vypíše hodnotu 10, neboť *i* v příkazu, označeném třemi hvězdičkami, bude znamenat proměnnou, deklarovanou v příkazu **for**. Pokud použijeme Borland C++ 5.0, vypíše 50, neboť *i* ve zmíněném příkazu bude znamenat globální proměnnou.

Nové překladače by ovšem měly nabízet přepínače pro zpětnou kompatibilitu. „Klasické“ zacházení s oborem viditelnosti deklarace v příkazu **for** předepíšeme v Borland C++ v příkazové řádce samostatného překladače volbou **-Vd** a v IDE zaškrtnutím pole

Do not restrict scope of 'for' loop expression variables v okně *Options* | *Project* | *C++ Options* | *C++ Compatibility*.

Deklarace v podmínce

V nové verzi jazyka C++ lze deklarovat proměnnou také v podmínce příkazu **if**, **while**, **switch** a **for**. Pro viditelnost takto deklarovaných proměnných platí totéž, co pro proměnné, deklarované v inicializačním výrazu příkazu **for**. Můžeme je tedy používat pouze v příkazech, které dané **if**, **while**, **for** nebo **switch** „řídí".

Proměnnou, deklarovanou v podmínce příkazu **if**, můžeme používat jak v části za **if,** tak i v části za **else** (pokud ji příkaz **if** obsahuje).

Proměnnou, deklarovanou v inicializačním výrazu příkazu **for**, lze použít kromě těla cyklu také v podmínce a v reinicializačním výrazu (tedy ve druhém a třetím výrazu v příkazu **for**). Proměnou, deklarovanou v podmínce opakování příkazu **for**, můžeme použít v těle cyklu a v reinicializačním (třetím) výrazu.

Poznamenejme, že v deklaracích v příkazech **for**, **if**, **switch** a **while** smíme deklarovat proměnné jakýchkoli typů kromě polí. Nesmíme zde deklarovat funkci.

Ukážeme si několik jednoduchých příkladů. Nejde o konstrukce nijak převratné, ale mohou zjednodušit zápis a tak nám usnadnit život.

```
if(long lp = f()) Zpracuj(lp);   // Deklarace v podmínce příkazu if
else Zahod(lp);

while(int n = F()) Vypis(n*n);   // Deklarace v podmínce příkazu while

switch(char cti = getch()){      // Deklarace ve výrazu v příkazu switch
  case 'A': DelejNecoSA(cti); break;
  case 'N': DelejNecoSN(cti); break;
  default: NedelejNic(cti); break;
}

for(int m = F();       // Deklarace v inicializačním výrazu příkazu for
    int n = G(i);      // Deklarace v podmínce opakování příkazu for
    m += n){
  Zacatek(m, n);
  Konec(n, m);
}
```

Deklarace třídy

Další novinky se týkají deklarace objektových typů.

Explicitní konstruktory

Konstruktory může překladač volat automaticky, aniž bychom to explicitně předepsali. Velmi často se to stává v případě jednoparametrických (konverzních) konstruktorů, které se používají také k implicitním konverzím.

Vezměme třídu *Haha*, která má konstruktor s jedním parametrem typu **int**. Abychom ji mohli použít v jednoduchém příkladu, deklarujeme v ní ještě výstupní operátor „<<":

```
/* Příklad CD - 5 */
#include <iostream.h>
```

```
class Haha {
       int y;
  public:
       Haha(int i): y(i){};
       friend ostream& operator <<(ostream& p, Haha y);
};

// výstupní operátor
ostream& operator <<(ostream& proud, Haha z)
{
  proud << z.y;
  return proud;
}

void f(Haha){/* ...*/}

int main(){
  Haha Ya(1);
  void f(Haha);
  Ya = 65;                        // 1
  f(5);                           // 2
  Haha Cha = 111;                 // 3
  return 0;
}
```

V příkazu, označeném v komentáři číslem 1, se zavolá konstruktor *Haha(int)* a vytvoří se pomocná instance, která se přiřadí proměnné *Ya*. V příkazu, označeném číslem 2, se opět automaticky zavolá konstruktor a vytvořená pomocná instance se předá funkci *f()* jako pomocný parametr. V příkazu 3 se nejprve zavolá konverzní konstruktor, jenž vytvoří pomocnou instanci, a tu pak překopíruje kopírovací konstruktor do instance *Cha*.

Takováto automatická přetypování nemusí být vždy vítaná, neboť mohou zabránit překladači v odhalení překlepů. Proto zavádí norma ANSI modifikátor **explicit**, který se používá podobně jako modifikátory **static** nebo **inline**. Konstruktor s tímto modifikátorem, tzv. **explicitní konstruktor**, nesmí překladač použít k implicitní konverzi.

Pokud tedy použijeme například překladač Borland C++ 5.0 a konstruktor *Haha::Haha(int)* označíme jako explicitní

```
class Haha {
       int y;
  public:
       explicit Haha(int i): y(i){};
       friend ostream& operator <<(ostream& p, Haha y);
};
```

bude překladač považovat příkazy, označené 1 – 3, za chybné, neboť se v nich snažíme použít tento konstruktor k implicitnímu přetypování.

Explicitní konstruktor lze samozřejmě použít k explicitnímu přetypování, neboť v něm jednoznačně vyjadřujeme své přání použít daný konstruktor ke konverzi. Následující příkazy se proto přeloží i v případě, že konstruktor *Haha::Haha(int)* deklarujeme jako explicitní:

```
 Ya = (Haha)65;             // OK
```

```
 f((Haha)5);                // OK
 Haha Cha = (Haha)111;      // OK
```

Přitom je samozřejmě jedno, zda pro přetypování proměnné *z* použijeme tradičního céčkovského zápisu *(Haha)z* nebo novějšího *Haha(z)*.

Měnitelné složky konstant

Pod konstantou si obvykle představujeme něco, co se opravdu nemění. Přesto se občas může stát, že potřebujeme objektový typ, ve kterém se jedna složka může měnit za všech okolností. Představme si např. model systému, který se skládá z řady objektů, z nichž některé jsou – z hlediska modelovaného systému – neměnné. U všech objektů však sledujeme počet použití, tedy počet volání určitých metod. Pro takovéto účely je rozumné mít v každé instanci složku, která se může měnit, a to i v konstantní instanci.

Norma ANSI C++ proto zavádí modifikátor **mutable**, který označuje právě takové vždy měnitelné složky. Podívejme se na jednoduchý příklad:

```
class slozka{
       mutable unsigned pocet_pristupu;
       DATA data;                // data, popisující modelovaný systém
  public:
       slozka();
       void zpracuj() const;
  // ... a další metody
};

slozka::slozka()                 // konstruktor
:pocet_pristupu(0){
  // ...
}

// použití systému
void slozka::zpracuj() const
{
  pocet_uziti++;
  // ...
}
```

Všimněte si, že funkci *slozka::zpracuj() const* jsme deklarovali jako metodu pro konstantní objekty, tj. jako metodu, která nemění atributy instance (hlavička končí klíčovým slovem **const**). Atribut *pocet_pristupu* ovšem v této metodě měnit můžeme, neboť jsme jej deklarovali jako měnitelný – s paměťovou třídou **mutable**.

Inicializace statických konstantních atributů

Statické konstantní atributy můžeme v ANSI C++ inicializovat přímo v definici třídy. Můžeme proto napsat

```
class Q {
  static const int q = 68;
  // ...
};
```

I pro takto inicializovaný atribut musíme uvést někde v programu definiční deklaraci

```
const int Q::q;
```

Nic nám ovšem nebrání zapisovati inicializaci statických konstantních atributů tak, jak jsme byli zvyklí, tedy až v definici

```
class Q {
  static const int q;
  // ...
};

const int Q::q = 68;
```

Zdůrazňujeme ale, že tento způsob inicializace se týká pouze atributů, které jsou **zároveň statické a konstantní.**

Datové typy

Dvě novinky se také týkají datových typů.

Typ bool

V ANSI C++ se setkáváme s typem **bool**, určeným pro práci s logickými hodnotami. Tento typ nabývá dvou možných hodnot, vyjádřených vyhrazenými slovy **true** a **false** (pravda, nepravda). Typ **bool** je samostatný celočíselný typ se znaménkem.

Spolu se zavedením typu **bool** se mění i definice některých operátorů a příkazů. Operátory „>“, „<“, >=“, „<=“, „!=“ a „==“ vracejí v ANSI C++ hodnoty typu **bool**, nikoli **int**. To se může projevit při rozlišování přetížených funkcí.

Pro typ **bool** jsou definovány automatické konverze na celá čísla: **false** se konvertuje na 0 a **true** na 1. Podobně mohou být číselné hodnoty, ukazatele a ukazatele do tříd automaticky konvertovány na typ **bool**: hodnota 0 resp. ukazatele, které neukazují nikam, se konvertují na **false**, ostatní pak na **true**.

V souvislosti se zavedením typu **bool** se i lehce mění syntax příkazů **if**, **while**, **do–while** a **for**: výraz, který řídí opakování cyklu nebo výběr větve příkazu **if**, je nyní *podmínka*, tedy výraz typu **bool**. Vzhledem k automatické konverzi čísel a ukazatelů na typ **bool** je však praktický dosah této změny zanedbatelný.

Tři znakové typy

ANSI C++ rozlišuje 3 znakové typy: **char**, **unsigned char** a **signed char**. Typ **char** je sice implementován jako jeden ze zbývajících dvou, ze syntaktického hlediska je ale považován za samostatný typ. To se opět může odrazit při rozlišování přetížených funkcí:

```
void F(char){/* ...*/}
void F(signed char){/* ...*/}
void F(unsigned char){/* ...*/}
```

Zde mohou opět nastat problémy se zpětnou kompatibilitou: např. překladače Borland C++ 3.1 a starší uznávaly pouze dva znakové typy a význam typu **char** bez modifikátoru **signed** nebo **unsigned** závisel na nastavení přepínačů.

Také tyto problémy lze v Borland C++ 4.x a 5.0 vyřešit pomocí přepínačů pro zpětnou kompatibilitu, a to buď v IDE zaškrtnutím pole *Do not treat 'char' as distinct type* v okně *Options* | *Project* | *C++ Options* | *C++ Compatibility* nebo v příkazové řádce pomocí přepínače **-K2**.

Literatura

1. Pecinovský, R. – Virius, M.: *Objektové programování 1. Učebnice s příklady v Turbo Pascalu a Borland C++*. Grada, Praha 1996
2. *Slovník spisovného jazyka českého*. Academia, Praha 1989
3. Stroustrup, B.: *The Design and Evolution of C++*. AT&T Bell Labs, 1994
4. Stroustrup, B. - Ellis, M. A.: *The Annotated C++ Reference Manual*. Addison - Wesley, 1994, 1996.
5. Arthur Bloch: *Murphyho zákon*. Svoboda – Libertas, Praha 1993.
6. *Software Fault Tolerance*. Editor M. R. Lyu. J. Wiley & Sons, 1995.
7. K. Goodman: *Clearer, More Comprehensive Error Processing with Win32 Structured Exception Handling*. Microsoft Systems Journal, leden 1994, s. 29
8. *Microsoft Win32 Programmer's Reference*, Vol. 2. Microsoft Press, 1994
9. *Working Paper for Deaft proposed International Standard for Information Systems – Programming Language C++*. Doc. No. X3J16/96-0108 WG21/N0926
10. Borland C++ 5.0. C++ Programmer's Guide. Borland international Inc., Scotts Valley 1996

Rejstřík

Q

R

Ř

S

Š

T

U

V

W

X

Z