Grada

Vakát

Rudolf Pecinovský, Miroslav Virius

Objektové programování I

Učebnice s příklady
v Turbo Pascalu a Borland C++

Ing. Rudolf Pecinovský, CSc., Ing. Miroslav Virius, CSc.

Objektové programování I

Učebnice s příklady v Turbo Pascalu a Borland C++

ISBN 80-7169-366-9

Obsah

Předmluva

Otevíráte další díl kursu programování. V této knize se seznámíte se základy objektově orientovaného programování v jazycích C++ a Pascal. Vznikla přepracováním a doplněním závěrečné části úspěšného seriálu „Cesta k profesionalitě“, který vycházel v letech 1992 – 1994 v časopisu ComputerWorld. Obsahuje výklad o zapouzdření a příklady na skládání tříd. Další věci, které k objektově orientovanému programování neodmyslitelně patří, tj. dědičnost a polymorfismus, najdete v příštím dílu. V něm byste také měli najít povídání o dalších pokročilých programovacích nástrojích, na které v časopisecké verzi vzhledem k jeho předčasnému ukončení již nezbylo místo – máme na mysli šablony a prostory jmen v C++ a výjimky v C++ a v Pascalu.

Náš výklad v tomto 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. Nevyhýbáme se ovšem ani těm vlastnostem jazyků C++ a Pascal, které implementují až pokročilejší překladače z poslední doby (Borland C++ 4.x a 5.0, Microsoft Visual C++ 2.0, Watcom C++ 10.5 a pozdější, Borland Pascal 7.0, Delphi). O Delphi se zmiňujeme ovšem pouze okrajově, neboť jde o nástroj pro profesionální práci, určený zejména k vývoji databázových aplikací pro Windows 3.1 resp. Windows 95. Systematičtější informace o novinkách, které v Delphi najdete a které se týkají látky, probírané v tomto dílu, najdete v kapitole Dodatek.

Předpoklady

Od čtenářů 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 typu a umějí definovat vlastní (neobjektové) datové typy,
- ✧ znají základy práce s ukazateli a umějí používat dynamické proměnné.
- ✧ umějí zacházet s některým z vývojových prostředí pro tyto jazyky, dodávaných firmou Borland, Microsoft, Watcom nebo jinou.

Na druhé straně nepředpokládáme žádné předběžné znalosti o objektech a objektově orientovaném programování.

Terminologie

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

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

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“, borlandský“, „pascalista“, „céčkař“, „pluskař“ apod., přesto, že proti nim lze mít výhrady – alespoň podle mínění jazykových korektorů. Pro Windows si dovolíme používat občas označení „Wokna“.

Typografické konvence

V textu této knihy používáme následující konvence:

while	Tučně píšeme klíčová slova.
třída	Tučně píšeme rovněž 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 jazyka Pascal, jsou po straně označeny jednoduchou svislou čarou.

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

K této knize lze zakoupit doprovodnou disketu, která obsahuje zdrojové texty příkladů uvedených v knize a řadu dalších programů, které vám mohou sloužit k inspiraci i testování alternativních možností.

1. Začínáme s objektovým programováním

Lidé se často na přednáškách ptají na vzájemný vztah strukturovaného a objektově orientovaného programování (OOP). Dovolíme si začít tvrzením, že kdo doposud nepřijal za své zásady strukturovaného programování, neměl by se vůbec snažit o programování objektově orientované. Dodejme, že čím bylo a je strukturované programování v oblasti algoritmů, tím je objektově orientované programování v oblasti datových struktur. Možná v některých oblastech snižuje efektivitu výsledného programu, ale na druhou stranu velice výrazně zvyšuje produktivitu programátorské práce.

1.1 Trochu historie

Historie programování je proložena neustálým rozporem mezi rychle narůstající výkonností počítačů a pomalým nárůstem produktivity programátorské práce. Tato produktivita však většinou nevzrůstala lineárně. Její nárůst probíhal ve skocích způsobených novými prostředky a novými metodami práce.

První programy na prvních počítačích byly bezezbytku programovány ve strojovém kódu. V polovině padesátých let však již byly schopnosti počítačů a jim odpovídající požadavky zákazníků natolik vysoké, že jim programátoři přestávali být schopni vyhovět. Začaly vznikat první programovací jazyky a jejich příchod znamenal velký skok v produktivitě programátorské práce.

V polovině šedesátých let se začala vyhrocovat druhá krize. Opět se zdálo, že pro uspokojení potřeb zákazníků bude muset být zanedlouho každý druhý člověk na zeměkouli programátorem. Objevily se však nové programovací jazyky a zejména nová metodika: strukturované programování. Tato metodika přinesla zásadní zvýšení produktivity práce, přičemž žádala po programátorech jediné: psát programy maximálně přehledné a srozumitelné. Strukturované programy byly sice většinou nepatrně delší a pomalejší, avšak zato byly mnohem bezpečnější (tj. méně chybové) a hlavně byly mnohem dříve hotové.

Jedním z velkých přínosů strukturovaného programování bylo prosazení návrhu programů metodou shora dolů a s ním spojené abstrahování od detailů. Programátor se má podle této metodiky v každou chvíli soustředit pouze na základní požadované funkce programu a nezatěžovat se podružnými detaily. Dokud podprogramy pracují spolehlivě a přijatelně efektivně, nemusí nás vůbec zajímat, jakým způsobem svůj úkol plní.

Jak se však domyslíte, složitost programů stále rostla a programátorské týmy řešící složité úlohy se opět dostávaly stále častěji do problémů vyvolaných již samotnou složitostí úlohy. Klíčovou se začínala stávat otázka datových struktur. Složité datové struktury byly zpracovávány v mnoha podprogramech a jakákoliv změna jejich definice vedla k nutnosti přepracování rozsáhlých částí programů.

Častým případem byly např. části programů, které řešily nové úlohy, nicméně úlohy velice blízké úlohám již vyřešeným, pro něž bylo možno najít odpovídající podprogramy v knihovně. Vzhledem k drobným odlišnostem však bylo nutno naprogramovat všechny podprogramy znovu, protože modifikace původních podprogramů nepřicházela v úvahu – bylo by totiž nutno znovu otestovat všechny starší programy, které modifikované podprogramy využívaly, a to bylo většinou mnohem nákladnější než napsat všechny podprogramy znovu.

Tyto všechny rozpory se snaží řešit objektově orientované programování. Jeho nosnou ideou je zpřehlednění práce s datovými strukturami obdobně, jako strukturované programování zpřehlednilo algoritmy. I pro objektově orientované programování platí, že jeho hlavní zbraní je abstrakce: nepiď se po tom, jak je daná datová struktura implementována, a zaměř se pouze na to, zda ti vyhovuje či nevyhovuje.

Hlavním zlomem v myšlení, který OOP přináší, je ale změna nositele aktivity. Na rozdíl od dosavadního přístupu, kdy nositelem veškeré činnosti byly algoritmy, v objektově orientovaném programování se těžiště aktivity přenáší na data. Pokud jsme např. doposud chtěli otevřít nějaký soubor, zavolali jsme patřičný podprogram na otevírání souborů a označili mu soubor, který má otevřít (tj. předali jej jako parametr). V OOP je tomu naopak: tam požádáme soubor, aby se otevřel. Jinými slovy, aktivní je soubor a jedna z činností, které pro nás umí udělat, je otevřít se.

Myšlenka objektově orientovaného programování se objevila na přelomu šedesátých a sedmdesátých let. Za předchůdce a prarodiče dnešních objektově orientovaných programovacích jazyků je všeobecně vydáván jazyk Simula 67, který přinesl koncepci třídy, což bylo zobecnění tehdejšího způsobu implementace datových typů. Třídy totiž, na rozdíl od tehdejších zvyků, zahrnovaly definice obou složek definovaného datového typu, tj. jak množiny přípustných hodnot (tu najdeme i v klasických implementacích datových typů), tak i množiny přípustných operací (tu tam již nenajdeme).

Simula 67 navíc zavedla mechanismus dědičnosti datových typů a řadu dalších rysů, které pozdější objektově orientované jazyky převzaly. Některými jejími rysy se nechali inspirovat vývojáři firmy Xerox, kteří si vytkli za cíl vytvoření jednotného a přitom maximálně komfortního vývojového a uživatelského prostředí. Výsledkem jejich snažený byl systém Smalltalk, který v průběhu sedmdesátých let dále vyvíjeli až do verze Smalltak 80, která se stala základem většiny současných implementací tohoto jazyka.

Smalltalk se sice prozatím vzhledem ke svým enormním nárokům na výkon počítače příliš nerozšířil, avšak další vývoj programování ovlivnil mnohem více, než řada jazyků běžně používaných. Programátoři mu vděčí za to, že přivedl na svět objekty a objektově orientované programování (program ve Smalltalku je tvořen objekty, které si navzájem posílají zprávy), uživatelé by mu měli být nekonečně vděčni za myšlenku komfortní práce s okny a zejména za neodmyslitelný doplněk většiny dnešních počítačů – za myš.

V průběhu sedmdesátých a osmdesátých let začala vznikat řada dalších objektově orientovaných jazyků. Programátorský svět z nich nejvíce ovlivnil jazyk C++, který je objektově orientovaným rozšířením nejpopulárnějšího jazyka profesionálních programátorů osmdesátých let – jazyka C – a který se stále pronikavěji prosazuje jako hlavní programovací jazyk let devadesátých. Jeho autorem je Bjarne Stroustrup, pracující u firmy

AT&T Bell Laboratories, která dala mimo jiné vzniknout operačnímu systému UNIX a s ním i jazyku C.

Bjarne Stroustrup potřeboval simulovat rozdělení jádra operačního systému UNIX a žádný z dostupných jazyků mu nepřipadal dostatečně mocný, tvárný, výkonný a přitom přenositelný. Rozhodl se, že si potřebný nástroj vytvoří „objektovým zorientováním" jazyka C. V první etapě (1982) to vyřešil cestou nejmenšího odporu – preprocesorem (připomínáme: preprocesor je program, který upravuje zdrojový text před tím, než je předán vlastnímu překladači), který převáděl zdrojový text v jazyku, jejž Bjarne Stroustrup nazval „C s třídami" (*C with classes*), do jazyka C.

C s třídami, které bylo v podstatě pouze rozšířením jazyka C o práci s objekty, se ukázalo jako velmi dobrá cesta, kterou mohou programátoři přejít ze světa klasického programování do světa programování objektově orientovaného. To mělo samozřejmě vliv na další vývoj jazyka. V dalších letech proto vznikla nová verze, tentokráte již také jako plnohodnotný překladač, která dostala i nové jméno: C++. První verze jazyka C++ byly všeobecně dostupné od roku 1985.

Jazyk rychle získával na popularitě a jeho autor jej na základě připomínek uživatelů dále vylepšoval. Postupně se objevily verze, označované Cfront[1] 1.0, 1.1, 1.2, 2.0, 2.1, 3.0, avšak v současné době by se již měla objevit společná norma ANSI/ISO. Z překladačů, které jsou na trhu, implementuje Borland C++ 3.0 a 3.1 verzi Cfront 3.0, starší borlandské překladače implementují Cfornt 2.1 resp. 2.0. Překladače Borland C++ 4.x a 5.0 odpovídají stavu návrhu normy ANSI v době přípravy těchto překladačů. Microsoft C++ 7.0 a Visual C++ 1.5 implementuje C++ podle specifikace Cfront 2.1, Visual C++ 2.0 a 4.0 se již orientují na ANSI C++ (implementují však vždy jen ty rysy jazyka, které byly v době přípravy překladače již jednoznačně odsouhlaseny standardizační komisí).

Myšlenkou nadstavby objektové orientace nad doposud objektově „neorientovaný" jazyk se nechali inspirovat i vývojáři firmy Borland a po vzoru jazyka C++ vytvořili objektově orientovanou verzi jazyka Pascal (dokonce ji uvedli na trh dříve, než vlastní verzi překladače jazyka C++). Příjemné na ní je to, že je jazyku C++ syntakticky blízká, takže potíže programátorů, kteří musí mezi těmito dvěma jazyky přebíhat, příliš nezvyšuje. Nepříjemné je naopak to, že její autoři zůstali na polovině cesty a implementovali pouze nejzákladnější objektově orientovaná rozšíření a jejich rozsah podstatně nezvětšili ani ve verzi 7.0.

Objektové programování v Turbo Pascalu může trochu připomínat strukturované programování v klasickém Basicu. Nástroje jazyka sice umožňují realizaci většiny konstrukcí, ale programátor si musí sám ohlídat, aby na nic nezapomněl a aby jeho program zůstal korektní. Stejně jako Basic nabízel některé možnosti, které byly v Pascalu a dalších strukturovaných jazycích těžko dosažitelné (např. prolínání procedur), tak i Pascal nabízí konstrukce, které např. v C++ nenajdete (např. destruktory s parametry). A stejně jako v klasickém Basicu některé důležité konstrukce chyběly (např. procedury s parame-

[1] Označení Cfront se obecně užívá pro překladače, které překládají do jazyka C. V našem případě tedy jde o překladače z C++ do C.

try), stejně chybí některé konstrukce i v Pascalu a musíme je tu více, tu méně složitě obcházet (násobnou dědičnost, šablony, přetěžování operátorů atd.).

Chtěli bychom proto předem varovat všechny programátory v Pascalu, kteří ještě nepřesedlali na C++, že to budou mít oproti programátorům v C++ o něco těžší.

Při výkladů konstrukcí, u nichž se filozofie implementace v Pascalu a C++ výrazněji liší, budeme nejprve ve společné části výkladu hovořit o významu této konstrukce obecně s přihlédnutím k implementaci v C++, která většinou poměrně přesně odráží původní důvod zavedení vysvětlované konstrukce. V následující části, věnované pouze C++, se pak již budeme většinou zabývat pouze některými detaily implementace. Po části věnované specifikům C++, bude většinou následovat část věnovaná specifikům Pascalu a v ní si vysvětlíme veškeré odchylky implementace Pascalu od obecného popisu v úvodní společné části.

Víme, že tento způsob výkladu bude pro pascalisty náročnější, ale domníváme se, že je důležité, aby hned zpočátku věděli „co je co v OOP“ a neměli své představy pokrouceny specifiky implementace té které konstrukce v Pascalu.

1.2 Naše cesta k OOP

K objektově orientovanému programování (OOP) vedou tři schody, představované jeho třemi hlavními rysy: **zapouzdřením** (*encapsulation*), **dědičností** (*inheritance*) a **polymorfismem** (mnohotvárností, *polymorphism*). V učebnicích jazyků podporujících OOP, které nám doposud prošly rukama, většinou autoři na počátku vysvětlili všechny tři hlavní rysy OOP a pak začali probírat jednotlivé konstrukce jazyka s promítnutím všech tří rysů. Tento postup může být pro čtenáře zbytečně náročný, protože musí neustále operovat s velkým množstvím zatím nezažitých pojmů. V této knize se proto pokusíme jít na věc opačně: budeme na „schody“ vystupovat postupně a na každém z nich si ukážeme, jak se dotyčný rys promítá do konstrukcí jazyka.

Než vystoupíme na první schod, proberme si nejprve trochu terminologie.

Klíčovým pojmem, kolem nějž se v OOP vše točí, je **třída**, o níž jsme si již řekli, že je vlastně zobecněnou podobou klasického datového typu. Ve škole nás učili, že datový typ je určen množinou hodnot, kterou mohou **instance** (konkrétní konstanty a proměnné) tohoto typu nabývat, a množinou operací, které můžeme s instancemi (objekty) daného datového typu provádět.

Všechny „objektově neorientované“ jazyky zůstávaly při deklaracích datových typů pouze u první poloviny definice. Definujeme-li v jazycích Pascal nebo C datový typ, definujeme pouze strukturu vnitřní reprezentace instancí daného typu, tedy množinu hodnot, kterých mohou konstanty a proměnné tohoto typu nabývat. Jakékoliv definice operací nad instancemi tohoto typu mají v programu naprosto autonomní postavení a jsou s typem zpracovávaných objektů svázány pouze deklaracemi v hlavičce funkce nebo procedury.

Naproti tomu v deklaracích tříd v objektově orientovaných jazycích deklarujeme jak vnitřní strukturu objektů (instancí) dané třídy, tak i příslušné operace, které je možno

nad objekty (instancemi) dané třídy provádět. Deklarace tříd tedy obsahuje jak **datové složky** (budeme jim říkat **atributy**), tak i **funkční složky** (budeme jim říkat **metody**). Toto „svaření“ deklarace struktury vnitřní reprezentace a patřičných metod do společné deklarace třídy označujeme jako **zapouzdření** (encapsulation).

Třídám budeme někdy říkat **objektové** (**datové**) **typy**. Považujte proto tyto termíny za synonyma.

Poznámka:
V anglické literatuře o jazyku C++ se pro metody používá termín member function*, který se překládá jako „členské funkce“. Termín metoda (*method*), používaný v klasickém OOP i v příručkách Pascalu, se nám zdá výhodnější.*

V textu této knihy budeme občas potřebovat odlišit „normální“ funkce od metod objektových typů. V takovém případě budeme pro „obyčejné“ funkce používat označení **řadové funkce**.

2. Zapouzdření

Naše povídání o OOP začne na prvním schodě, tedy u zapouzdření. Z hlediska probíraných programovacích jazyků to znamená, že začneme u deklarace objektového typu.

2.1 Deklarace třídy

Jako příklad si ukážeme v obou jazycích deklaraci

1. třídy *cBod*, která bude mít dvě celočíselné datové složky (atributy) *_x* a *_y* a pět funkčních složek (metod), jimiž budou funkce *x* a *y* a procedury *Nastavx*, *Nastavy* a *Nastav*,
2. třídy *cOkno*, jež bude mít datové složky (atributy) *LH* (levý horní), *PD* (pravý dolní) a *Kurzor*, které budou všechny typu *cBod*, a jejímiž funkčními složkami (metodami) budou procedury *Otevři* a *Nastav*.

V C++ se třídy definují stejně jako struktury a unie, pouze v jejich definicích přibudou deklarace nebo definice metod. Mohli bychom tedy říci, že struktury a unie jsou vlastně degenerované třídy, které neobsahují žádné metody.

```
/* Příklad C5 - 1  */
struct cBod {
   int _x;                        // souřadnice x
   int _y;                        // souřadnice y
   int x();                       // Vrací hodnotu x
   int y();                       // Vrací hodnotu y
   void Nastavx( int _x );        // Nastaví x
   void Nastavy( int _y );        // Nastaví y
   void Nastav( int _x, int _y ); // Nastaví x a y
};

struct cOkno {
   cBod LH;                       //Levý horní roh okna
   cBod PD;                       //Pravý dolní roh okna
   cBod Kurzor;                   //Souřadnice kurzoru
   void Otevri( int l=1, int h=1, int p=80, int d=25 );
   void Nastav();                 //Nastav okno jako aktuální
};
```

Všimněte si, že v obou třídách je definována metoda *Nastav*. Ke kolizi však nemůže dojít ani v případě, že budou mít obě dvě stejné parametry. Můžete si to zdůvodnit např. tak, že pro překladač není identifikátorem dané metody identifikátor uvedený v deklaraci, ale že jím je identifikátor vytvořený spojením identifikátorů třídy a metody oddělených čtyřtečkou. V prvém případě by si tedy překladač pojmenoval metodu *cBod::Nastav* a v druhém *cOkno::Nastav*. Pak už samozřejmě k žádné kolizi dojít nemůže, protože se jedná o dva různé identifikátory.

V Pascalu je definice třídy syntakticky téměř totožná s definicí záznamu, až na to, že klíčové slovo **record** nahradíme v deklaracích tříd klíčovým slovem **object**. V deklaraci musíme uvést nejprve atributy a pak teprve metody. Naše deklarace by tedy měly tvar:

```
(*  Příklad P5 - 1  *)
type
  cBod = object
       _x: integer;                          (*souřadnice x*)
       _y: integer;                          (*souřadnice y*)
       function x:integer;                   (* Vrátí hodnotu _x*)
       function y:integer;                   (* Vrátí hodnotu _y*)
       procedure Nastavx( ix:integer );      (* Nastaví x *)
       procedure Nastavy( iy:integer );      (* Nastaví y *)
       procedure Nastav(ix,iy:integer);      (* Nastaví x a y *)
  end;
  cOkno = object
       LH: cBod;                  (* Levý horní roh okna *)
       PD: cBod;                  (* pravý dolní roh okna *)
       Kurzor: cBod;              (* Souřadnice kurzoru *)
       procedure Otevri( l, h, p, d: integer );
       procedure Nastav;          (* Nastav okno jako aktuální *)
  end;
```

Všimněte si, že v obou třídách je definována metoda *Nastav*. Ke kolizi však nemůže dojít, a to právě proto, že tato procedura je definována jako metoda. Můžete si to zdůvodnit např. tak, že pro překladač není identifikátorem dané metody identifikátor uvedený v deklaraci, ale že jím je identifikátor vytvořený spojením identifikátorů třídy a metody oddělených tečkou. V prvém případě by si tedy překladač pojmenoval metodu *cBod.Nastav* a v druhém *cOkno.Nastav*. Pak už samozřejmě k žádné kolizi dojít nemůže, protože se jedná o dva různé identifikátory.

Deklarované metody lze ihned používat, protože deklarace metody v rámci deklarace třídy má stejný účinek, jako deklarace kdekoliv jinde. Nicméně v dalším textu programu (nejlépe ještě v rámci téhož modulu) je pak nutno jednotlivé metody také definovat. Ale o tom až za chvíli.

2.2 Instance třídy – objekty

Konstantám a proměnným daného typu – třídy – říkáme **instance dané třídy** nebo **objekty** (odtud také objektově orientované programování). I tyto dva termíny budeme používat jako synonyma.

Poznámka:
Turbo Pascal bohužel používá klíčové slovo **object** *pro definici třídy a v manuálech pak tímto termínem autoři střídavě označují tu třídu a tu její instance. Nedejte se tím zmást. Doufáme, že v naší knize bude vždy jasné, kdy budeme hovořit o třídě, tj. o objektovém datovém typu, a kdy o objektu, tj. o instanci této třídy.*

Instance objektových datových typů deklarujeme a definujeme stejně jako konstanty a proměnné jiných datových typů. V následujících ukázkách jsou deklarace a definice instancí dříve definovaných tříd.

```
/*  Příklad C5 - 2  */
//Deklarace
extern cOkno oo;                 //Deklarace globální proměnné
extern const cBod Poc;           //Deklarace globální konstanty
//Definice
cBod b;
cOkno o1, o2;
cBod bb = {1, 2 };               //Inicializovaná proměnná
cOkno oo = {{1, 1}, {80, 25}, {1, 1} };
//Inicializovaná globální proměnná
const cBod Poc = {1, 1};         //Definice konstanty
```

```
(*  Příklad P5 - 2  *)
{
  V Pascalu nelze instance pouze deklarovat, každá deklarace instance
  třídy je zároveň její definicí. Zároveň není v Pascalu možno definovat
  konstanty objektových datových typů.
}
var              {Neinicializované objektové proměnné}
  b: cBod;
  o1, o2: cOkno;
const            {Inicializované objektové proměnné }
  bb: cBod = ( _x:1; _y:2 );
  oo: cOkno = ( LH: ( _x: 1; _y: 1 );
  PD: ( _x:80; _y:25 );
  Kurzor: ( _x: 1; _y: 1 ) );
```

V obou jazycích pak používáme metody jako složky třídy (v C++ jsme se s tím již setkali při práci s datovými proudy), tzn. že při vyvolání dané metody napíšeme vždy identifikátor proměnné, jejíž metodu chceme vyvolat, tečku a identifikátor volané metody. Budeme-li tedy např. chtít, aby se před chvílí deklarované okno *o1* otevřelo přes celou obrazovku, požádáme je o to C++ příkazem (implicitní hodnoty parametrů nám vyhovují):

```
o1.Otevri();
```

a v Pascalu příkazem:

```
o1.Otevri( 1, 1, 80, 25 );
```

Budeme-li chtít, aby se okno *o2* otevřelo uprostřed obrazovky a pokrývalo ji asi z jedné čtvrtiny, požádáme je o to v obou jazycích příkazem:

```
o2.Otevri( 20, 6, 60, 18 );
```

Poznámka:
V literatuře o OOP se často dočtete o tom, že „objekty si posílají zprávy“. Poslání zprávy objektu je v obou probíraných jazycích realizováno jako vyvolání některé z jeho me-

tod. Předchozí příkazy bychom tedy mohli interpretovat i tak, že jsme objektům o1 a o2 poslali zprávu, v níž jsme je žádali, aby se otevřely (zprávu Otevri).

2.3 Definice metod

Metody již tedy umíme deklarovat a používat. Nyní se je naučíme definovat.

První, na co nesmíme při definici zapomenout je, že identifikátor metody pro překladač vytvoříme spojením identifikátoru třídy a deklarovaného identifikátoru metody, přičemž tyto dva identifikátory od sebe v Pascalu oddělujeme tečkou a v C++ čtyřtečkou.

Druhou věcí, kterou bychom měli mít na paměti, je, že metody mají ještě jeden skrytý parametr, a tímto parametrem je odkaz na objekt (Pascal), resp. konstantní adresa objektu (C++), jehož metodu jsme volali – tj. objektu, který jsme požádali o danou službu (jemuž jsme poslali zprávu) a nad jehož složkami má daná metoda pracovat[2]. Tento parametr se jmenuje v C++ **this** a v Pascalu *self* (v C++ je to klíčové slovo, v Pascalu nikoli). Přestože se v seznamu parametrů explicitně neuvádí, můžeme se na něj kdykoliv odvolat. Použití identifikátorů **this** a *self* však většinou není potřeba (zejména v Pascalu), protože identifikátory složek třídy bez kvalifikace chápe překladač jako identifikátory složek daného objektu.

V ukázce demonstrující definice metod v programech si všimněte následujících skutečností:

1. Jazyk C++ na rozdíl od Pascalu umožňuje, aby identifikátory parametrů a lokálních proměnných byly shodné s identifikátory složek dané třídy. V takovém případě tyto nové identifikátory zakryjí identifikátory objektu a k těm je pak nutno přistupovat právě prostřednictvím explicitní kvalifikace pomocí **this**.
2. C++ navíc dovoluje definovat těla metod hned uvnitř deklarace třídy. Stačí pouze před středníkem ukončujícím deklaraci vložit složené závorky a do nich zapsat tělo definované metody. Takto definované metody pak překladač přeloží jako vložené (inline) – samozřejmě pokud jsme generování vložených metod nepotlačili.
3. Pokud nechceme kazit jednotnou úpravu deklarací, můžeme vloženou metodu označit klasicky pomocí klíčového slova **inline** v definici metody.

```
/*  Příklad C5 - 3  */
struct cBod2
{
```

[2] V Pascalu je situace jednoduchá: **self** je instance, pro kterou metodu voláme, předaná odkazem. V C++ se situace liší podle verze jazyka. Ve starších verzích představoval **this** ukazatel na instanci, pro kterou danou metodu voláme (tj. v metodě třídy *X* představoval hodnotu typu *X**). Počínaje verzí Cfront 2.1 představuje **this** konstantní ukazatel na instanci, pro kterou danou metodu voláme, tj. v metodě třídy *X* představuje hodnotu typu **const** *X**.

```
  int x;
  int y;
  void Nastavx( int ix )         //Vložená funkce – přímá
   {x = ix; };                   //definice v deklaraci třídy
  void Nastavy( int );
//Připomínáme, že v C++ na identifikátoru
//parametru v deklaraci nezáleží. Tento identifikátor se nemusí shodovat
//s identifikátorem v definici (viz procedura Nacti) a dokonce nemusí
//být v deklaraci uveden vůbec. Stačí identifikátor
//datového typu – viz poslední deklaraci.
  void Nastav( int x, int y );
  void Nacti( cBod2 Bod );
  cBod2& Predej();
//Funkce Predej je zde sice deklarována, ale není nikde definována.
//Překladač a sestavovací program to však nebudou považovat za chybu,
//protože funkce není nikde použita.
};

inline void /*****/ cBod2::Nastavy /*****/
//Samostatná definice vložené metody
( int iy )
{
  y = iy;                  //y označuje this->y
}
/********** cBod2::Nastavy **********/

void /*****/ cBod2::Nastav /*****/
( int x, int y )
/*
  Identifikátory parametrů překryjí identifikátory složek instance, a
  proto je třeba složky tohoto objektu kvalifikovat pomocí ukazatele
  this.

*/
{
  this->x = x;
  this->y = y;
}
/********** cBod2::Nastav **********/

void /*****/ cBod2::Nacti /*****/
( cBod2 b )
/*
  Tuto funkci odkrokujte a přitom mějte ve sledovacím okně (Watch)
  nastaveno sledování parametrů this a b.
*/
{
  x = 2 * b.x;
  y = 2 * b.y;
  this->x = 100 * b.x;
  this->y = 100 * b.y;
  this->x = 10 * x;
  this->y = 10 * y;
}
/********** cBod2::Nacti **********/

void /*****/ Test3 /*****/ ()
```

```
/*
  Testovací procedura pro ověření funkce parametru this prostřednictvím
  krokování procedury cBod::Nacti.
*/
{
  cBod2 aa, bb;
  aa.Nastav( 1, 2 );
  bb.Nacti( aa );
}
/********** Test3 **********/
```

Pascalistům můžeme důvod, proč není explicitně třeba používat parametr *self*, přiblížit jinými slovy: představte si, že celé tělo metody je uvnitř bloku

```
with self do
begin
...
end
```

Pascalský překladač nedovolí, aby některý parametr či lokální proměnná měly stejný identifikátor jako některá ze složek třídy. Pokud by se však uvnitř bloku objevil např. příkaz **with**, jehož kvalifikační proměnná by byla téhož typu, vyvstala by potřeba složky rozlišit. Naprogramujte si následující program, zadejte ve sledovacím okně příkazy

```
aa,r
bb,r
```

odkrokujte si metodu *Kopiruj* a sledujte přitom změny hodnot složek objektů *aa* a *bb*.

```
(*  Příklad P5 - 3  *)
type
cBod2 = object
  x: integer;
  y: integer;
  procedure Nacti( Bod : cBod2 );
end;
procedure (*****) cBod2.Nacti (******)
( Bod : cBod2 );
(*
  Tuto funkci odkrokujte a přitom mějte ve sledovacím
  okně (Watch) nastaveno sledování parametrů self a Bod.
*)
begin
  x := 2 * Bod.x;
  y := 2 * Bod.y;
  with Bod do
  begin
       x := 10 * Bod.x;
       y := 10 * Bod.y;
       self.x := 100 * Bod.x;
       self.y := 100 * Bod.y;
  end;
end;
(********** cBod2.Nacti **********)
```

```
procedure (*****) Test3 (*****);
{
  Testovací procedura pro ověření funkce parametru self
  prostřednictvím krokování procedury cBod.Nacti.
}
const
  aa: cBod2 = ( x:1; y:2 );
  var
  bb: cBod2;
begin
  bb.Nacti( aa );
end;
(********** Test3 **********)
```

2.4 Konstruktor

Datový typ se stane doopravdy objektovým až od chvíle, kdy mezi jeho metodami bude tzv. **konstruktor**, což je speciální metoda, sloužící pouze k vytváření – zřizování objektů dané třídy. (Mohli bychom říci, že volání konstruktoru je ekvivalentní zaslání zprávy „vytvoř se“.) Konstruktory mají za úkol vyhradit pro vytvářenou instanci místo v paměti a uvést ji do takového stavu, aby ji pak bylo možno plnohodnotně používat.

U doposud probíraných (tj. neobjektových) skalárních typů i u námi definovaných vektorových či strukturových typů stačilo k „zprovoznění“ dané instance (konstanty či proměnné) vyhradit potřebnou paměť a popřípadě přiřadit počáteční hodnotu. Pokud si však vzpomenete na kapitoly o práci se soubory a proudy, tak tam jsme již potřebovali pro plnohodnotné používání souborů a proudů navíc ještě tyto soubory a proudy také otevřít. Dokud jsme datový soubor nebo proud neotevřeli, nemohli jsme s ním začít pracovat.

Konstruktory instancí objektových datových typů slouží právě k tomu, aby bylo možno definovat instance objektových datových typů ve vší potřebné komplexnosti. Zabezpečují, že se při definici daného objektu nejen vyhradí potřebný paměťový prostor, ale že se navíc provedou i nutné inicializační akce. (V Pascalu je to trochu jinak – viz dále.) Vyhrazení potřebného místa si bere na starost překladač (i když také v tom jej můžeme usměrnit, ale o tom až později), na programátora zbývá pouze požadovaná inicializace[3].

[3] Podíváme-li se na problém z druhé strany, mohli bychom říci, že překladači dali jeho tvůrci k dispozici prostředky pro vytváření, inicializaci a přetypování instancí vestavěných datových typů: číselných, ukazatelových, a v Pascalu i řetězcových a množinových. Definicí konstruktorů poskytujeme překladači obdobné prostředky i pro vytváření, inicializaci a přetypování námi definovaných objektových datových typů - prostředky potřebné k tomu, abychom s instancemi těchto objektových typů mohli pracovat stejně přirozeně, jako s instancemi typů zabudovaných. Tedy přesněji: poskytujeme je v C++, v Pascalu to nejde.

Konstruktorů může být ve třídě definováno několik a mohou se lišit počtem a typem parametrů nebo pouze požadovanou činností. Mezi nimi mají dva konstruktory výjimečné postavení. Jde o **bezparametrický konstruktor** a **kopírovací konstruktor**.

Bezparametrický konstruktor je konstruktor bez parametrů. (V C++ jej může zastupovat i konstruktor, jehož všechny parametry mají definované implicitní hodnoty.) Implicitní verze bezparametrického konstruktoru, kterou je za jistých podmínek překladač ochoten vytvořit automaticky, pouze vyhradí místo pro definovanou instanci, popřípadě provede ještě některé další akce, aby se vytvořila plnohodnotná instance – ale o tom si povíme později.

Poznámka:
Bezparametrický konstruktor nazývají někteří autoři implicitní konstruktor, neboť jej překladač volá „implicitně". Tento termín se však také používá pro konstruktor, vytvořený implicitně překladačem. Abychom se vyhnuli zbytečným zmatkům, budeme označení implicitní konstruktor používat pouze pro konstruktor vytvořený překladačem.

Kopírovací konstruktor je konstruktor, jehož jediným parametrem je odkaz na proměnnou (v C++ je možno použít i odkaz na konstantu) téhož typu, kterého je on konstruktorem. (V C++ jej může zastupovat i konstruktor, jehož všechny ostatní parametry mají definované implicitní hodnoty.) Také kopírovací konstruktor umí v případě potřeby vytvořit překladač. Tato implicitní verze kopírovacího konstruktoru, vytvořená překladačem, vyhradí potřebnou paměť pro nový objekt a zkopíruje složku po složce obsah svého parametru do konstruovaného objektu.

Jak jste již asi odhadli, význam a použití konstruktorů se v obou probíraných jazycích výrazně liší. V C++ se konstruktor volá automaticky při definici objektu, kdežto v Pascalu překladač při definici pouze vyhradí potřebnou paměť a popřípadě ji zaplní definovanými počátečními hodnotami, zatímco vlastní konstruktor musí programátor zavolat někde na vhodném místě sám. Liší se i v dalších charakteristických rysech, a proto si povíme o každém jazyku zvlášť.

V C++ se konstruktory definují jako metody, jejichž identifikátor je shodný s identifikátorem třídy, jejíž instance konstruují. Ovšem pozor, **v deklaraci konstruktoru nesmíme uvádět typ vracené hodnoty** (**ani void**). Pokud tedy chceme definovat několik různých konstruktorů, musí se lišit počtem nebo typem použitých parametrů.

V definicích konstruktorů je samozřejmě možné použít i implicitních hodnot parametrů, avšak stejně jako u přetížených řadových funkcí musíme dát pozor na případné nejednoznačnosti – např. na to, že volané funkce nelze rozlišit pouze podle předávání parametrů odkazem nebo hodnotou. Ale nebojte se, kdybyste na něco zapomněli, překladač vás na vaše opomenutí nezapomene upozornit.

Na jednu věc bychom však měli pamatovat: **jakmile třída obsahuje nějaký explicitní** (tj. námi definovaný) **konstruktor, negeneruje překladač implicitní bezparametrický konstruktor**. Pokud tedy chceme definovat instance daného typu způsobem uvedeným pod bodem 1, který vede zákonitě k použití bezparametrického konstruktoru, **musíme bezparametrický konstruktor definovat sami** – stačí vložený konstruktor s prázdným tělem. Připomínáme, že vlastní vyhrazení paměti je i nadále věcí překlada-

če, v těle konstruktoru se uvádějí pouze následné činnosti. (O situacích, při nichž je bezparametrický konstruktor potřeba, pohovoříme za chvíli.)

Logika předchozího pravidla je vcelku jasná: Dokud není ve třídě definován žádný explicitní konstruktor, může překladač pokládat danou třídu za klasický strukturový datový typ rozšířený o metody. Objekty takovýchto typů konstruovat umí a podobně zkonstruuje i instanci objektového typu.

Jakmile však definujete jakýkoliv vlastní konstruktor, překladač, který neumí zjistit, jaké dodatečné akce tento konstruktor provádí, si přestane být jist, že by jeho verze bezparametrického konstruktoru udělala vše, co je pro vytvoření daného objektu potřeba. Proto raději ponechá definici bezparametrického konstruktoru na programátorovi.

V opačném případě by se totiž mohlo stát, že by programátor na definici bezparametrického konstruktoru zapomněl, překladač by v potřebném okamžiku nasadil svůj, tj. implicitní bezparametrický konstruktor, a chování tímto implicitním konstruktorem vytvořených proměnných by nemuselo být ekvivalentní chování konstant a proměnných vytvořených ostatními konstruktory.

S kopírovacím konstruktorem je to jinak. Ten si je překladač ochoten vygenerovat vždy, tedy i tehdy, když jsme již nějaké konstruktory definovali. Kopírovací konstruktor tedy explicitně definujeme pouze v případě, kdy chceme, aby dělal něco jiného než jeho implicitní verze generovaná překladačem.

Příkladem třídy, pro níž je nutno definovat vlastní kopírovací konstruktor, je např. třída *B* z naší následující ukázky, protože tato třída počítá své doposud vytvořené instance a to by samozřejmě implicitní kopírovací konstruktor nedělal.

Poznámka:
Aby nemusely být komentáře v doprovodných ukázkách příliš dlouhé, budeme v nich občas používat několik zkratek:

BK - bezparametrický konstruktor

IBK - implicitní bezparametrický konstruktor

KK - kopírovací konstruktor

IKK - implicitní kopírovací konstruktor

k-r - konstruktor (k-r je „těsnopisná“ zkratka, kterou budeme také skloňovat: k-ru znamená konstruktoru atd.)

```
/*  Příklad C5 - 4  */
struct A              //Třída bez explicitních k-rů. Akce nutné
{                     //pro vytvoření jejich instancí se neliší od
  int a1;             //akcí při definici struktur a obnášejí akce
  int a2;             //realizované IBK
  void Metoda() {};
};
unsigned nB=0;        //Počet vytvořených instancí třídy B

struct B              //Třída s explicitně definovanými k-ry
```

```
{
  int bi;
  A bA;
  B( A& a, int i=0 )        //K-r s jedním parametrem typu A&
  {                         //a druhým celočíselným s implicitní
      bA = a;               //počáteční hodnotou. K-r je definován
      bi = i;               //přímo, tj. jako vložená funkce.
      nB++;                 //Může sloužit jako přetypovací k-r
  };
// Je-li definován jakýkoliv konstruktor, překladač bezparametrický
// konstruktor negeneruje a v případě potřeby jej musí definovat
// programátor sám – např.takto:
//
// B() {};
//
// tj. jako vloženou funkci s prázdným tělem. V našem případě bude
// funkci BK plnit konstruktor:
  B( int=0, int=0, int=0 );
//B( int& i );              //CHYBA: Takto definovaný konstruktor by
                            //nebylo možno odlišit od předchozího.
  B( B& b );                //KK musíme definovat, protože IKK nevyhovuje
};

inline /*****/ B::B /*****/
//Konstruktor definujeme jako vložený, protože při jeho
//jednoduchosti by předávání prametrů pouze zdržovalo
( int i1, int i2, int i3 )
{
  bi = i1;
  bA.a1 = i2;
  bA.a2 = i3;
  nB++;                     //Zvyšuje se počet vytvořených instancí
}
/********** B::B **********/

/*****/ B::B /*****/
( B& b )
{
  bi = b.bi;
  bA.a1 = b.bA.a1;
  bA.a2 = b.bA.a2;
  nB++;
}
/********** B::B **********/
```

Než se pustíme do výkladu o používání konstruktorů, měli bychom si nejprve říct něco o možných způsobech definic objektů.

1. Nechceme-li přiřazovat definovanému objektu žádné speciální počáteční hodnoty ani s ním provádět nějaké parametrizované akce, můžeme jej definovat klasicky, tj. tak, že napíšeme identifikátor příslušného datového typu a za něj identifikátor definovaného objektu. Překladač v tomto případě použije **bezparametrický konstruktor**. Pokud ve třídě není žádný explicitně (tj. programátorem) definovaný konstruktor, generuje překladač implicitní verzi bezparametrického konstruktoru sám. Tímto způsobem jsou v příští ukázce definovány proměnné *as1*, *bs1* a *ba1*.

2. Pokud potřebujeme přiřadit definovanému objektu nějaké speciální počáteční hodnoty nebo s ním provádět akce blíže specifikované hodnotami nějakých parametrů, musíme zvolit jednu z následujících tří variant definic:

   a) Pokud **nejsou** ve třídě **explicitně** definovány žádné konstruktory a pokud chceme atributům definovaného objektu pouze přiřadit dané počáteční hodnoty, můžeme použít stejný způsob, jaký známe z definic strukturových datových typů, tj. seznam hodnot jednotlivých datových složek (atributů) uvedený ve složených závorkách. Takto je v příští ukázce definována proměnná *as2*.

   b) Pokud chceme definovanému objektu přiřadit hodnotu konstanty nebo proměnné nějakého typu **T** (budeme jí říkat inicializační hodnota) a ve třídě je definován konstruktor, jehož jediný parametr je typu **T** nebo je jím odkaz na objekt typu **T** (stačí konstruktor, u nějž mají ostatní parametry definovány implicitní hodnoty), pak je možno definovanou instanci inicializovat tak, že za její identifikátor napíšeme v definici rovnítko a za něj onu inicializační hodnotu. V příští ukázce jsou tímto způsobem definovány instance *as3*, *bs2* až *bs5*, *bls1* a *ba2* až *ba5*.

   c) Pokud je parametrizace potřebného konstruktoru složitější, napíšeme v definici instance jednotlivé skutečné parametry konstruktoru do kulatých závorek za identifikátor instance. (Tuto syntax můžeme samozřejmě použít i v předchozím případě, avšak řešení s rovnítkem nám připadá elegantnější.) Popsaným způsobem jsou v příští ukázce definovány instance *bs6* až *bsB*, *bls2* a *ba6* až *baB*, avšak instance *bs6* až *bs9* a *ba6* až *ba9* bylo možno definovat i způsobem uvedeným v předchozím bodě, tj. s rovnítkem.

Konstruktory se volají ve chvíli, kdy jsou konstruované objekty vytvářeny. Definujeme-li proto globální nebo statický objekt, (objekt definovaný vně jakékoliv funkce, nebo definovaný uvnitř funkce, ale se specifikátorem **static**), provede se konstruktor ještě před inicializačními procedurami, tj. před procedurami uváděnými v direktivách **#pragma startup**, a tím samozřejmě i před funkcí *main*. Inicializační procedury proto mohou počítat s tím, že při jejich zahájení jsou již všechny statické objekty připraveny k použití. Naproti tomu automatické objekty jsou vytvářeny při každém vstupu do programu a stejně tak jsou volány i jejich konstruktory.

Podívejte se nyní na následující ukázku, v níž jsme se snažili předvést všechny probrané možnosti. Nejprve ovšem ještě dvě poznámky:

Poznámka 1:
Konstruktor B::B(a&, int) *je definován přímo v deklaraci a překladač jej tedy bude chtít přeložit jako vložený. Jako vložený je definován i konstruktor*

```
B::B(int,int,int).
```

Vložené podprogramy však nejdou za normálních okolností krokovat. Abyste při krokování dokázali zjistit, který z konstruktorů byl použit, musí být při překladu v BC++ 3.1 nastavena volba [Options | Compiler | C++ Options | Out-of-line inline functions]*, v BC++ 4.x volba* [Options | Project | Compiler | Debugging | Out-of-line inline functi-

ons]*, při níž překladač překládá i vložené funkce jako normální. Ve Visual C++ 1.5 zakážeme rozvoj vložených funkcí tak, že v dialogovém okně* Options | Project *stiskneme tlačítko* Compiler*...; objeví se další dialogové okno, ve kterém v seznamu* Category *zvolíme možnost* Optimizations*. V rozbalovacím seznamu* Inline Expansion of Functions *pak vybereme možnost* Disable*.*

Poznámka 2:
Chceme-li tedy zjistit, jak probíhá volání konstruktorů při definicích statických instancí, nestačí použít pouhé krokování, protože na počátku krokování, tj. při vstupu do funkce main*, jsou již statické instance definovány, a činnost odpovídajících konstruktorů je tedy skončena. Jedinou možností jak krokovat konstruktory statických instancí je proto umístění zarážek („breakpointů" –* CTRL+F8*) do konstruktorů, které chceme krokovat. (Doporučujeme umísťovat zarážky na uzavírací závorky jejich těla, kdy už můžete zkontrolovat, co konstruktor udělal.)*

To, kterou instanci právě vytváříte, zjistíte snadno, jestliže si do sledovacího okna necháte vypisovat hodnotu proměnné **this**, která (jak víme) obsahuje adresu instance, jež danou metodu volala – v našem případě vytvářené proměnné či konstanty. Protože u ukazatelů na statické instance debugger nevypisuje adresu číselně, ale uvede identifikátor instance, na níž ukazatel ukazuje, je lokalizace vytvářené instance více než snadná.

U automatických proměnných debugger uvádí obsah parametru **this** číselně, a to nám moc neřekne. Tyto proměnné však vytváříme až v průběhu programu, a pokud „opouštíme" konstruktor pomocí klávesy F7, přesune nás program hned za příkaz, který konstruktor zavolal – a opět jsme doma.

```
/*  Příklad C5 - 5  */
A as1;                      // Použije se implicitní PK
A as2 = {0, 2 };            // Protože třída A nemá definovány žádné
//explicitní k-ry, je tato syntaxe
//inicializace přípustná
const A as3 = as2;          // Použije se implicitní KK
A as4[ 10 ];                // Vektor deseti objektů typu A

//B bs0 = {10 };            // CHYBA: Tato syntax je u tříd
                            // s explicitními konstruktory nepoužitelná.
//Globální statické instance – odpovídající konstruktory se zavolají
// PŘED incializačními procedurami. Použije se:
B bs1;                 //PK – u nás B( 0, 0, 0 )
B bs2 = 12;                 //B( int, 0, 0 )
const B bs3 = bs2.bi;       //B( int, 0, 0 )
B bs4 = bs2;                //KK B( B& )
B bs5 = as2;                //B( A&, 0 )
B bs6( 16 );                //B( int, 0, 0 )
B bs7( bs2.bA.a1 );         //B( int, 0, 0 )
B bs8( bs2 );          //KK B( B& )
const B bs9( as2 );         //B( A&, 0 )
B bsA( as2, 3 );            //B( A&, int )
B bsB( 10, 20, 30 );        //B( int, int, int )
B bsC[ 3 ];                 //3 x PK, tj. 3 x B( 0, 0, 0 )
```

```
#pragma warn -use           //Aby překladač "nebublal" kvůli
#pragma warn -aus           //nepoužitým proměnným

void /*****/ Test4_2 /*****/ ()
{
//Lokální statické instance – pro volání jejich k-rů
// platí totéž, co pro globální statické instance. Použije se:
  B bls1 = bsB              //KK
  B bls2( 1, 2, 3 );        //B( int int int );
//Automatické instance: konstruktor se volá při každém vstupu do
// podprogramu – pro ověření volejte tento podprogram dvakrát za sebou.
// Použije se:
  B ba1;                    //PK, tj. B( 0, 0, 0 )
  B ba2 = 12;               //B( int, 0, 0 )
  B ba3 = ba2.bA.a1;        //B( int, 0, 0 )
  B ba4 = ba2;              //KK B( B& )
  const B ba5 = as2;        //B( A&, 0 )
  B ba6( 16 );              //B( int, 0, 0 )
  B ba7( ba2.bi );          //B( int, 0, 0 )
  B ba8( ba2 );             //KK B( B& )
  const B ba9( as2 );       //B( A&, 0 )
  B baA( as2, 10 );         //B( A&, int )
  B baB( 7, 8, 9 );         //B( int, int, int )
  B baC[ 5 ];               //5 x PK, tj. 5 x B( 0, 0, 0 )
}
/********** Test4_2 **********/
```

V předchozích ukázkách měly obě třídy své bezparametrické konstruktory. Třída *B* dostala do vínku explicitní konstruktor, jehož všechny parametry měly přiřazeny své implicitní hodnoty, a díky tomu jej mohl překladač použít v situacích, v nichž bývá volán bezparametrický konstruktor. Třída *A* naproti tomu žádné explicitní konstruktory neměla, a proto byl překladač ochoten její bezparametrický konstruktor (přesněji IBK) vytvořit sám.

Co by se stalo, kdyby třída neměla definován konstruktor, který by šel použít jako bezparametrický, a měla definovány explicitní konstruktory, takže by překladač odmítal vytvořit IBK?

1. Její instance by nemohly být definovány způsobem uvedeným v bodu 1. (V předchozí ukázce tak byly definovány proměnné *as1*, *bs1* a *ba1*.)
2. Nebylo by možno definovat vektory daného typu. (Později si povíme, jak je možno toto omezení obejít.)
3. Její instance by nesměly být složkami jiných strukturových a objektových typů – k tomuto omezení se ještě vrátíme.

Prozatím jsme hovořili o třídách vytvořených ze struktur. Jak jsme si ale řekli, třídy mohou být vytvořeny i z unií. V definicích těchto tříd však musíme dbát na některá omezení. Při našich současných znalostech se na nás vztahuje pouze jedno, a to že **složky unie nesmějí mít konstruktory**.

Naproti tomu **unie samy své konstruktory mít mohou**. Jistě si vzpomenete, že v definicích instancí (konstant a proměnných) unií můžeme inicializovat pouze jejich pr-

vou složku. Definicí konstruktorů můžeme toto omezení obejít. Nic nám totiž nebrání definovat tolik různých konstruktorů, kolik má unie složek, a volat při definici objektu konstruktor inicializující odpovídající složku – viz následující ukázka:

```
/*  Příklad C5 - 6  */
union C          //Unie s explicitně definovanými konstruktory
{                //NEDEFINOVALI jsme BK!
  char c;
  int i;
  A a;           //Strukturový typ bez explicitních konstruktorů
//B b;           //==> NELZE, třída B má explicitní konstruktor,
                 // a proto nemůže být složkou unie
  C( char cc ) {c = cc; };
  C( int ii ) {i = ii; };
  C( A aa ) {a = aa; };
  C( int i1, int i2 ) {a.a1 = i1; a.a2 = i2; };
};
// Ve třídě C byl explicitně definován aspoň jeden k-r, a proto
// překladač IPK nevytvoří. Protože není definován PK, nejsou možné ani
// některé konstrukce:
void /*****/ Test4_3 /*****/ ()
{
//C c;                //=> NELZE – není definován BK
//Použije se:
  C ca = 'a';         //C( char )
  C c1 = 1;           //C( int );
  C c2( 10, 20 );     //C( int, int );
  C cA = as2;         //C( A ) – Proměnná as2 je z předchozí ukázky
//C CX[ 10 ];         //==>NELZE – není definován BK
}
/********** Test4_3 **********/
```

Konstruktory se volají vždy, když je třeba vytvořit nějaký objekt daného typu, a to může být i jindy, než při explicitní definici instance. Konstruktory mohou být volány buď implicitně překladačem nebo explicitně programátorem.

Podívejme se nejprve na implicitně volané bezparametrické konstruktory. Před minulou ukázkou jsme vám říkali, že instance třídy, která nemá a nemůže mít bezparametrický konstruktor, nemohou být složkami jiných tříd. (Třídám, které mají složky objektových typů, říkáme **složené třídy** – *composed classes*.) Je to proto, že při konstruování instancí složených tříd překladač nejprve zavolá bezparametrické konstruktory všech jejich složek. A proto také ono výše uvedené omezení. (Později se naučíme obejít i toto omezení.)

Další třídou implicitně volaných konstruktorů jsou tzv. **konverzní** neboli **jednoparametrické konstruktory** (*conversion constructors*), což jsou konstruktory, které lze volat pouze s jedním parametrem. (Mohli bychom mezi ně svým způsobem zařadit i kopírovací konstruktor.) Po těchto konstruktorech sáhne překladač ve chvíli, kdy potřebuje inicializovat instanci daného objektového typu a má k dispozici instanci typu, pro nějž je definován potřebný konverzní konstruktor. Typickým příkladem je např. předávání

parametrů procedurám a funkcím. Jedná se tedy vlastně o ekvivalent implicitního přetypování (typových konverzí), s kterým jsme se již setkali.

Explicitně voláme konstruktory obdobně, jako bychom volali funkce vracející hodnotu daného typu. Explicitní volání konstruktorů používáme při vytváření nových (většinou přechodných) objektů v průběhu programu.

Všechny výše vyjmenované případy najdete v následující ukázce:

```
/*  Příklad C5 - 7  */
//Předpokládáme platnost všech definic z předchozích ukázek v této
  kapitole
struct D
{
  B b;
  //C c;                    //==> NELZE, třída C nemá BK
  D( int i, int j, int k )
//Před vlastním vstupem do funkce se nejprve automaticky
//volá bezparametrický konstruktor B::B
  {
       b = B( i, j, k );    //Explicitní volání k-ru
  };
       void d( B& bb )      //Toto není konstruktor, ale
  {b = bb; };               //obyčejná vložená metoda
};

//Obyčejná funkce s hodnotovým parametrem typu B,
//vracející hodnotu typu B
B /*****/ Pokus /*****/
( B b )                     //Při předávání parametrů hodnotou
{                           //se volá kopírovací konstruktor
  return b;                 //Při předávání vracené hodnoty se volá KK
}

void /*****/ Test4_4 /*****/ ()
{
  D d1( 100, 200, 300 );
  d1.d( B( as2 ) );         //Explicitní volání B::B( A& ) pro
                            // přetypování parametru
  for( int i=0; ++i <= 3; )
       bsC[ i ] = B( i, i*i );
  bs1 = Pokus( bsC[ 2 ] ); //Protože se parametr předává
                 //hodnotou, musí se vytvořit nová instance -
                 //automaticky se proto zavolá KK
  bs2 = Pokus( 11 );        //Automaticky se volá konverzní
                            //konstruktor B::B( int, 0, 0 )
}
/********** Test4_4 **********/
```

V Turbo Pascalu probíhá definice instancí objektových datových typů dvoufázově. Při vlastní definici překladač provede pouze ty nejzákladnější akce potřebné pro vytvoření definované proměnné: vyhradí pro danou proměnnou paměť a popřípadě ji přiřadí počáteční hodnoty. Při těchto akcích vystačí s implicitním (tj. překladačem definovaným) bezparametrickým, resp. kopírovacím konstruktorem. Pokud akce těchto konstruktorů

programátorovi nestačí, definuje pro kompletaci definice dané proměnné zvláštní metody, které označí jako konstruktory. Správně bychom je však měli označit jako semikonstruktory, protože v okamžiku jejich volání je řada akcí spojených se vytvořením dotyčné proměnné hotova, a na ně zbývají pouze dodělávky, na které implicitní verze konstruktorů nestačí.

Poznámka:
Pokud budeme v dalším textu hovořit o pascalských konstruktorech, budeme mít na mysli právě tyto semikonstruktory.

Pascalské konstruktory se deklarují obdobně jako ostatní metody. Jediný rozdíl je v tom, že místo klíčového slova **procedure** jsou jejich deklarace uvozeny klíčovým slovem **constructor**. Pro tyto konstruktory překladač žádné předem definované jméno nevyhrazuje, ale je dobrým zvykem používat pro ně identifikátor *Init*. Pokud však potřebujete těchto konstruktorů více, musí se jejich identifikátory navzájem lišit.

Nutnost dvoufázovosti pascalských konstruktorů vyplývá ze samotné koncepce Turbo Pascalu – vždyť i skalární automatické proměnné musíte definovat v sekci proměnných, ale počáteční hodnoty jim můžete přiřadit až v těle procedury. Samotná dvoufázovost by však ještě nemusela být tak nepříjemná. Možná, že je to věc teoreticky nečistá, ale dá se na ni zvyknout. Obrovskou nevýhodou této koncepce je však to, že volání či nevolání konstruktorů ponechává Pascal zcela v rukou programátora a jeho zapomnětlivosti. Tato skutečnost je tím nepříjemnější, že definice proměnných, jejichž dodatečné konstruktory budeme ještě potřebovat volat, zapisujeme ve zdrojovém textu na jiné místo, než kam píšeme příkazy k volání těchto konstruktorů. (Viz předchozí zmínka o definici a inicializaci automatických skalárních proměnných.)

Velice nepříjemně se tato vlastnost projevuje zejména při definicích statických proměnných, tj. externích proměnných (proměnných definovaných vně všech podprogramů) a lokálních proměnných definovaných v sekci **const**, jimž jsme byli zvyklí přiřazovat v rámci definice zároveň počáteční hodnotu. Jakmile totiž musí konstruktor provádět nějakou netriviální činnost (např. v naší ukázce inkrementuje proměnnou *nB*), nemá vůbec smysl vytvářené proměnné inicializovat v rámci definice, protože vzhledem k netriviálnosti konstruktoru musíme jejich konstruktor stejně volat, a ten nám většinou jejich přednastavené hodnoty přepíše.

Tuto situaci lze řešit dvěma způsoby: buď definujeme další verze konstruktorů, které tyto počáteční hodnoty přepisovat nebudou, nebo se prostě smíříme s tím, že i statické proměnné budeme inicializovat až při volání jejich konstruktoru. My se spíše přikláníme ke druhému řešení, protože vede k jednodušším programům a hlavně koncentruje všechny inicializační operace na jedno místo. Lze si však představit situace, v nichž bude druhé řešení výhodnější. Pro úsporu místa jsme jej však již v následující ukázce neuváděli.

Zamysleme se nyní nad otázkou, kde a kdy konstruktory jednotlivých proměnných volat. Na otázku *Kde* byste nám jistě dokázali odpovědět sami: konstruktory externích proměnných bychom měli uvést v inicializační části daného modulu a konstruktory lokálních statických proměnných v těle příslušného podprogramu. Nesmíme však zapomenout zajistit, aby se nám tyto lokální statické proměnné inicializovaly pouze jednou

a aby se jejich konstruktory nevolaly při každém vstupu do podprogramu znovu. Klasický způsob vyřešení tohoto úkolu je předveden v ukázce.

```
(*  Příklad P5 - 4  *)
type
  A = object               {Třída bez explicitních konstruktorů}
  a1 : integer;
  a2 : integer;
  end;

var
  av:A;

const
  ac:A = ( a1:0; a2:2 );
  nB:integer = 0;

type
  B = object               {Třída s explicitními konstruktory}
       bi : integer;
       bA : A;
       constructor Init;
       constructor Init3( i1, i2, i3:integer );
       constructor InitA1( var a:A; i:integer );
       constructor InitB( var b:B );
{
  Následující tři konstruktory nebudeme pro úsporu místa deklarovat a
  implicitní hodnoty jejich parametrů doplníme v programu vždy sami:
  constructor Init3( i1: integer );
  constructor Init3( i1, i2: integer );
  constructor InitA( var a: A );
}
  end;

constructor (*****) B.Init (******)
;
begin
  B.Init3( 0, 0, 0 );
end;
(********** B.Init **********)

constructor (*****) B.Init3 (******)
( i1, i2, i3:integer );
begin
  bi := i1;
  bA.a1 := i2;
  bA.a2 := i3;
  Inc( nB );
end;
(********** B.Init3 **********)

constructor (*****) B.InitA1 (******)
( var a:A; i:integer );
begin
  bi := i;
  bA := a;
```

```
  Inc( nB );
end;
(********** B.InitA1 **********)

constructor (*****) B.InitB (******)
( var b:B );
begin
  self := b;
  Inc( nB );
end;
(********** B.InitB **********)

procedure (*****) pokus (*****)
( a:A );
begin
  a.a1 := 7;
end;
(********** pokus **********)

const      {===== Globální inicializované proměnné =====}
  bs0:B = ( bi:10 );
  {bsi:B = 22;  {CHYBA - nutno inicializovat jinak}
  {bsA:B = ac;  {CHYBA - nutno inicializovat jinak}
  {bsB:B = bs0; {CHYBA - nutno inicializovat jinak}
  bs3:B = ( bi:bs0.bi);

{bA:( a1:bs0.bA.a1; a2:bs0.bA.a2 ) );}
{I kdyby to bylo povoleno, přiřadilo by se "smetí", protože hodnota
  proměnné bs0 není
 ještě v době překladu připravena }
{Globální data => Jinak přiřazení k-rem v inicializační proceduře daného
 modulu, kde by mělo být:
  bsi.Init3( 22, 0, 0 );
  bsA.InitA1( ac, 0 );
  bsB.InitB( bs0 );
  Ale vzhledem k tomu, že v proměnné bs1 je "smetí" a při definici
  proměnných bs0 a bs1 se neikrementovala proměnná nB, musí se v
  inicializační proceduře objevit:
  bs0.Init3( 10 );
  bs3.Init3( bs0.bi, bs0.bA.a1, bs0.bA.a2 );
  proměnnou bs1 by však bylo asi jednodušší konstruovat:
  bs3.InitB( bs0 );
}

procedure (*****) Test4_1; (*****)
const {Lokální statické proměnné}
bls0:B = (bi:100);
{blsi:B = 22;              {CHYBA - nutno inicializovat jinak}
{blsA:B = ac;              {CHYBA - nutno inicializovat jinak}
{blsB:B = bs2;             {CHYBA - nutno inicializovat jinak}
{bls3:B = ( bi:bs0.bi; bA:( a1:bs0.bA.a1; a2:bs0.bA.a2 ) );}
{ I kdyby to bylo povoleno, přiřadilo by se "smetí", protože hodnota
  proměnné bs0 není
 ještě v době překladu připravena}
{Lokální data => Jinak přiřazení konstruktorem v těle podprogramu. Pozor
 na možnost násobné inicializace. Pro její vyloučení je třeba
 deklarovat: }
```

```
  POPRVE: boolean=FALSE;
  blsi : B = ();           {Statickým proměnným, které nelze }
  blsA : B = ();           {přímo incializovat, nepřiřadíme }
  blsB : B = ();           {v jejich definici nejprve nic. }
var       {Automatické proměnné}
  ba1, ba2, baA:B;
begin
  if( POPRVE )             {Abychom zabránili násobné inicializaci}
  then begin               {statických proměnných}
       blsi.Init3 ( 22, 0, 0 );
       blsA.InitA1( ac, 0 );
       blsB.InitB( bs2 );
       bls0.Init( ba0.bi );      {V definici byla sice počáteční
           hodnota přiřazena správně, ale neikrementovalo se
           nB. Protože nám však konstruktor přepíše původně
           přiřazenou hodnotu, musíme ji přiřadit znovu }
       bls3.InitB( bs0 );   {Opravujeme "smetí" přiřazené v def.}
  end;
{Inicializace automatických proměnných}
  ba1.Init3( 11, 0, 0 );
  ba2.Init2( 2, 22, 0 );
  baA.InitA1( ac, 0 );
  pokus( av );
end;
(********** Test4_1 **********)
```

Po shlédnutí předchozí ukázky možná některé z vás napadlo, proč je třeba vytvářet pro inicializaci objektů zvláštní třídu metod (tj. konstruktory) a zda by nestačilo použít běžné metody. Souhlasíme s vámi, že v předchozím programu by to stačilo. Časem si však povíme o programových konstrukcích, v nichž bychom s běžnými metodami nevystačili.

Nepříjemnou vlastností pascalské koncepce konstruktorů je to, že překladač není schopen konstruktory automaticky volat. Tím nám ovšem velice komplikuje život. Představte si, že potřebujeme nějakému podprogramu předat parametr daného objektového typu a tento parametr chcete inicializovat hodnotou typu, který je na daný typ převeditelný (tj. existuje konverzní konstruktor, jehož jediným parametrem je objekt onoho převeditelného typu). To, že překladač tyto konverzní konstruktory automaticky volat neumí, nás v tuto chvíli nutí vytvořit (a po návratu z volaného podprogramu zase zrušit) pomocnou proměnnou, do níž předávanou hodnotu nejprve přetransformujeme, a teprve tuto pomocnou proměnnou volané proceduře či funkci předáme (viz volání procedury *Proc4_2* v následující ukázce).

Nesmíme zapomenout ani na případ, kdy podprogramy přebírají své parametry hodnotou. Programátor totiž musí zabezpečit, aby se v těle podprogramu volaly konstruktory všech parametrů předaných hodnotou – nebo alespoň ty netriviální z nich, tj. ty, které provádějí některé akce, které implicitní konstruktory nezabezpečí. Je to proto, že při předávání parametru hodnotou se pomocí implicitního kopírovacího konstruktoru vytvoří na zásobníku nový objekt daného typu, který je kopií objektu původního. Pokud je kopírovací konstruktor netriviální, tj. pokud zabezpečuje více věcí než pouhé okopírování složky po složce, musí se v těle podprogramu pro tuto kopii volat námi definovaný konstruktor, aby i pro ni byly ony netriviální akce vykonány. Protože však tato kopie má

již přiřazeny počáteční hodnoty svých atributů, musí se pro odpovídající třídu definovat speciální bezparametrický konstruktor, jenž ponechá hodnoty všech složek, které nepodléhají onomu výše zmíněnému netriviálnímu zpracování.

Samostatnou kapitolou jsou složené třídy, tj. třídy, jejichž některé atributy (alespoň jeden) jsou objektových typů. Při definici konstruktorů těchto tříd nesmíte zapomenout volat konstruktory všech jejich objektových atributů, protože – jak již víme – dokud pro kteroukoliv instanci objektového typu, tedy i pro složku složené třídy, není zavolán konstruktor, nemůžeme tuto instanci považovat za plnohodnotně zkonstruovanou.

Všechny tyto otázky opět demonstruje následující ukázka:

```
(*  Příklad P5 - 5  *)
const
  nC : integer = 0;
type
  C = object
       ci : integer;  {Pořadí konstruovaného objektu - tato
                       složka bude u každé instance jiná}
       cA : A;
       cB1 : B;
       cB2 : B;
       constructor Init0;
       constructor Init;
       constructor InitB( var b : B );      {Předáváme odkazem, abychom
                                  neměli potíže při předávání hodnotou}
  end;

constructor (*****) C.Init; (*****)
{Neinicializující PK - k-r, který nemění hodnoty složek}
begin
  Inc( nC );
  ci := nC;     {Atribut cA nemusíme konstruovat, protože třída A
                 neobsahuje netriviální kopírovací konstruktor.}
  cB1.InitB( cB1 );
  cB2.InitB( cB2 );
{ V předchozích dvou příkladech předpokládáme, že jsme již obdrželi
  hotovou třídu B a do její definice nemůžeme zasahovat.
  Jinak by bylo lepší, kdyby i třída B obsahovala konstruktor, jenž by
  složky konstruované instance neinicializoval
}
end;
(********** C.Init **********)

constructor (*****) C.Init0; (*****)
{Inicializující BK}
begin
  Inc( nC );
  ci := nC;
  cA.a1 := 0;   {Třída A nemá svůj inicializační k-r}
  cA.A2 := 0;
  cB1.Init;
  cB2.Init;
end;
(********** C.Init0 **********)
```

```
constructor (*****) C.InitB (*****)
( var b : B ); {Konverzní konstruktor z B na C }
begin
   Inc( nC );
   ci := nC;
   cA := b.bA;
   cB1.InitB( b );
   cB2.InitB( b );
end;
(********** C.InitB **********)

procedure (*****) Proc4_2 (*****)
( b : B );
begin
{ Všimněte si, že adresa parametru b a adresa skutečného
   parametru, jehož hodnotu parametr b obsahuje, jsou různé – jedná se o
   dvě různé instance.
   Novou instanci – parametr b – je nutno zkonstruovat }
b.Init;
{Vlastní tělo procedury}
end;
(********** Proc4_2 **********)

procedure (*****) Test4_2; (*****)
var
   bp : B;
begin
   bp.InitA1( ac, nB+1 );
   Proc4_2( bp );
end;
(********** Test4_2 **********)
```

Vrátíme se ještě ke konstruktorům.

V předchozím oddílu jsme si řekli, že složky složených tříd musí mít bezparametrické konstruktory, aby je bylo možno vytvořit, a na příkladu konstruktoru *D(int, int, int)* jste měli příležitost si celý proces vytváření nové instance složené třídy odkrokovat. Jistě vás při tom zamrzelo, že tento proces je trochu neefektivní, protože se konstruktor třídy *B* musel volat ještě jednou, abychom mohli složku *b* inicializovat.

Nyní si ukážeme, jak je možné objektové složky složených tříd inicializovat přímo. C++ totiž umožňuje přímo v hlavičce konstruktoru složené třídy určit konstruktor, který budeme chtít pro vytvoření dané složky použít. Dosáhneme toho tak, že za seznamem parametrů konstruktoru složené třídy napíšeme dvojtečku a za ní uvedeme seznam identifikátorů složek, které chceme vytvářet nestandardně. Pokud bychom naše dosavadní vědomosti o syntaxi konstruktorů zapsali jako syntaktická pravidla, vypadala by tato definice následovně:

Definice_konstruktoru:
Ident_třídy :: Ident_třídy (seznam parametrů)
+ [: K_inic [, K_inic]opak]
+ {Seznam_příkazů }

K_inic: *//Inicializátor konstruktoru*
Identifikátor_složky ([Seznam_parametrů])

Na této definici je příjemné, že umožňuje tímto způsobem určit způsob inicializace nejen pro objektové, ale i pro neobjektové složky třídy. To znamená, že pokud bude složkou definované třídy např. číslo *i,* které bude typu **int** a které budeme chtít vynulovat, můžeme klidně v seznamu inicializátorů konstruktoru uvést položku *i(0).*

Poznámka:
Tento způsob zápisu je možno použít i v běžném programu, avšak nebývá to zvykem, neboť připomíná volání funkce. Nicméně pro ilustraci jsme tuto možnost v následující ukázce předvedli ve funkci Test4_5 *na proměnné* ***i****.*

Při vytváření instance se její jednotlivé složky vytvářejí v tom pořadí, v jakém byly deklarovány v definici třídy. Pořadí uvedení inicializátorů složek v hlavičce konstruktoru proto nemá na pořadí jejich inicializace žádný vliv. (V následující ukázce je uvedeno schválně prohozeně, abyste se o tom mohli přesvědčit.) Doporučujeme proto zachovávat v hlavičce pořadí složek uvedené v definici třídy.

Složky, jejichž inicializátory nebudou uvedeny v hlavičce konstruktoru, budou vytvořeny bezparametrickým konstruktorem.

Složky, které již byly inicializovány (tj. složky, které v definici třídy předcházejí složce právě inicializované), je možné použít jako parametry v inicializátorech dalších složek. (Viz následující ukázku, kde je použita složka *i* v definici složky *d* a složka *a* v definicích složky *c.*) Teoreticky je samozřejmě možné použít v parametrech také složky, které dosud inicializovány nebyly, ale není to moudré, protože v nich bude v dané chvíli pouze nějaké blíže nedefinované „smetí“.

Abyste si mohli vše náležitě vyzkoušet, doplňte si k předchozím ukázkám následující ukázku a pokuste se odkrokovat (F7 – TRACE) činnost funkce *Test4_5*.

```
/*  Příklad C5 - 8  */
//Předpokládáme platnost všech definic z předchozích
//ukázek v této kapitole, tj. z příkladů C2 - 1 až C2 - 7.
struct E
{
  int i;
  A a;
  B b1;
  B b2;
  C c;                                        //Už to umíme i bez BK
  D d;
  E( B bb, A aa, int id2, int id3 )
//Před vlastním vstupem do funkce se nejprve automaticky volají
// konstruktory jednotlivých složek – protože je v definici uvádíme,
// nepoužije se BK, ale použijí se konstruktory námi definované
  : i( 842 ),
  d( i/2, id2, id3 ),
  a( aa ),                 //Použije se IKK => nelze krokovat
  c( a ),                  //Třída C nemá BK – nevadí, protože se stejně
                           // použije konstruktor C::C( A )
```

```
  b1( bb )                  //Složka b2 není v seznamu uvedena,
                            // a proto bude inicializována BK
//Pořadí uvedení konstruktorů neodpovídá pořadí deklarací
// jednotlivých složek – ověřte si krokováním, že překladač
// volá k-ry v pořadí určeném pořadím deklarací složek
//;
  {
  };
};

void /*****/ Test4_5 /*****/ ()
{
//Před vlastním vstupem do následujícího k-ru třídy E se nejprve volají
//kopírovací k-ry třídy B a A. Při krokování však zaznamenáme pouze
//volání KK třídy B, protože ve třídě A jsme jej explicitně
//nedefinovali. Proto se použije IKK,který krokovat nelze.
  E e( bsB, as2, 123, 456 );
  int i( 7 ); //Tento zápis lze použít i pro
//instance neobjektových typů
}
/********** Test4_5 **********/
```

Druhá situace, o níž jsme si zatím říkali, že potřebuje pro zdárný průběh bezparametrický konstruktor, je deklarace polí objektových typů. I zde si však můžeme pomoci.

Pole instancí objektových typů můžeme inicializovat **naprosto stejně** jako pole instancí neobjektových typů. To znamená, že v definici daného pole napíšeme za hranaté závorky obsahující počet prvků rovnítko, otevírací složenou závorku, seznam čárkami inicializačních hodnot a zavírací složenou závorku. Jediný drobný rozdíl je v tom, že pokud není typ oné inicializační hodnoty mezi typy, pro něž je definován konverzní konstruktor nebo pokud potřebujeme přeřadit danému prvku pole hodnotu, pro jejíž vytvoření je třeba konstruktor s více parametry, uvedeme místo dané hodnoty přímo konstruktor se seznamem potřebných parametrů tak, jak je tomu v inicializaci třetí až šesté složky pole *b9* v následující ukázce.

Stejně jako u klasických polí, i u polí instancí objektových typů však platí, že inicializačních hodnot nesmí být více než je prvků pole a že stejně jako u klasických polí nemusíme inicializovat všechny prvky daného pole. Jak si asi jistě sami domyslíte, ty prvky, na něž se inicializátor nedostane, budou inicializovány bezparametrickým konstruktorem.

Také inicializaci prvků objektových polí je možno krokovat, a to jak hrubě (F8 – STEP), tak jemně (F7 – TRACE).

V následující ukázce jsou inicializovaná pole definována tak, aby bylo možno jejich inicializace snadno krokovat. Přesvědčte se, že u pole *i* to samozřejmě nejde.

```
/*  Příklad C5 - 9  */
//Předpokládáme platnost všech definic z předchozích ukázek v této
//kapitole
void /*****/ Test4_6 /*****/ ()
{
  int i [ 3 ] =
 {
```

```
      1,
      2,
      3
 };
  B b2[ 2 ];
  B b6[ 9 ] =
 {
      as2,                      //[0]
      111,                      //[1]
      bsA,                      //[2]
      B( 133, 233 ),            //[3]
      B( as2, 244 ),            //[4]
      B( bsA ),                 //[5]
      B( 166, 266, 366 )        //[6]
                    //[7], [8] – na poslední dva prvky se použije PK
};
C c3[ 3 ] =
 {
      C( as2 ),
      c3[ 0 ],
      C( 1, 2 )
 };
}
/********** Test4_6 **********/
```

Třetí věcí, o níž se chceme nyní zmínit, jsou **konstantní atributy**, tj. datové složky, které jsou v definici třídy deklarovány se specifikátorem **const**. Konstantním atributům nelze, stejně jako klasickým konstantám, přiřadit v průběhu programu žádnou novou hodnotu, a to ani v běžných funkcích (na to jsme již zvyklí), ani v metodách příslušné třídy, ale dokonce ani v tělech konstruktorů této třídy. Jediný způsob korektní inicializace konstantních atributů je přiřazení počáteční hodnoty prostřednictvím inicializátoru uvedeného v hlavičce konstruktoru.

Konstantní atributy tedy uchovávají hodnoty, které se nastaví při vzniku instance a v průběhu jejího života se již nemění. Může to být např. čas vzniku dané instance, její pořadí mezi instancemi nebo jakýkoliv jiný údaj, u nějž chcete zaručit, abyste jej nemohli v průběhu života instance nedopatřením změnit.

V následující ukázce obsahuje třída *Počítaná* konstantní atribut *Pořadí*, v němž je uloženo pořadí dané instance mezi všemi instancemi třídy *Počítaná* vytvořenými od počátku běhu programu.

```
/*  Příklad C5 - 10  */
//Předpokládáme platnost všech definic z předchozích ukázek v této
//kapitole
#include <iostream.h>
struct Pocitana
{
  const int Poradi;
  int Obsah;
  Pocitana( int=0 );
};
int nP = 0;
```

```
/*****/ Pocitana::Pocitana /*****/
( int i )
: Poradi( ++nP ), Obsah( i )
{
  cout  << "Zřídil jsem " << Poradi
        << ". počítanou instanci – má hodnotu "
        << Obsah << '\n';
}
/********** Pocitana::Pocitana **********/

void /*****/ Test4_7 /*****/ ()
{
  Pocitana Prvni;
  Pocitana Pole[ 4 ] = {10, 20, 30 };
  Pocitana Sesta = 66;
}
/********** Test4_7 **********/
```

2.5 Destruktor

Protipólem konstruktorů jsou destruktory. Tak, jako měly konstruktory za úkol připravit vytvářené instance do takového stavu, aby s nimi bylo možno pracovat, tak úkolem destruktorů je uklidit vše do stavu, v němž je možno bez nebezpečí pokračovat v dalším plnění programu.

Jistě si vzpomínáte na přirovnání k práci s datovými soubory, se kterým jsme se setkali na počátku kapitoly o konstruktorech. Tam jsme si říkali, že jednou z činností, kterou by mohl konstruktor dělat, je otevření souboru. (Připomeňme si, že některé „pluskové“ konstruktory datové proudy v rámci definice také zároveň otevírají.) Kdybychom ve stejném duchu uvažovali o destruktorech, tak by nám naprosto logicky a přirozeně muselo dojít, že součástí akcí, které by měl mít na starosti destruktor datového souboru nebo proudu, by mělo být jeho zavření. Dokud totiž soubor nebo proud neuzavřeme, nemáme jistotu, že po ukončení programu o svá data nepřijdeme, ani jistotu, že zbytečně otevřené soubory, resp. proudy nebudou blokovat prostředky operačního systému souborům, resp. proudům, které se teprve otevřít chystáme.

U konstruktorů jsme si řekli, že jich můžeme definovat více, přičemž se budou navzájem lišit počtem a typy svých parametrů (u Pascalu se mohou lišit i svými identifikátory). Naproti tomu destruktor bývá v každé třídě většinou nejvýše jeden a bez parametrů. C++ vám dokonce ani jiný destruktor deklarovat nedovolí. Pascal je v tomto smyslu benevolentnější a dovoluje definovat více různých destruktorů a dovoluje navíc definovat destruktory s parametry. V praxi se však této možnosti téměř nevyužívá. (Tím ovšem nechceme říct, že by se tato možnost nemohla ve speciálních případech ukázat jako užitečná.) Abychom zachovali konzistentnost terminologie, budeme destruktor bez parametrů označovat jako **bezparametrický destruktor**. Jeho implicitní verze neudělá v obou jazycích téměř nic jiného, než to, že uvolní paměť přidělenou destruované instanci.

S destruktory je to v obou jazycích obdobné jako s konstruktory – takže si budeme povídat zvlášť o destruktorech v C++ a zvlášť o destruktorech v Pascalu.

V C++ můžeme definovat pouze bezparametrický destruktor, tj. destruktor bez parametrů. Jiný destruktor vám překladač definovat ani nedovolí. Destruktor má stejně jako konstruktor předem definovaný identifikátor: je jím identifikátor třídy, před nímž je znak tilda (~). Pokud nedefinujete bezparametrický destruktor sami, definuje překladač implicitní bezparametrický destruktor.

Překladač volá destruktor automaticky, a to ve chvíli, kdy končí doba života odpovídající proměnné – u automatických proměnných při opuštění příslušného bloku, u statických proměnných při ukončení programu, tj. po opuštění funkce *main*, resp. při volání procedury *exit*.

Destruktory se volají v obráceném pořadí než konstruktory – tedy v obráceném pořadí deklarací. To platí jak pro statické proměnné, tak pro automatické proměnné, tak pro objektové složky složených tříd.

```
/*  Příklad C5 – 11  */
#include <iostream.H>
int Pocet;                  //Celkový počet živých instancí
int Poradi;                 //Pořadí právě vytvořené instance
struct T
{
  int Hodnota;
  int Por;                  //Pořadí vzniku dané instance
  T( int=999 );             //Bezparametrický a konverzní konstruktor
  T( const T& t );          //Kopírovací konstruktor
  ~T();                     //Destruktor
};

ostream& /*****/ operator << /*****/
( ostream& o, T& t )
//Parametr t nelze předávat hodnotou, protože bychom tiskli
//hodnotu kopie skutečného parametru vytvořené na zásobníku
{
  o    << "Pořadí=" << t.Por
       << ", Hodnota=" << t.Hodnota
       << ", Živých=" << Pocet << '\n';
  return o;
}
/********** operator << **********/

/*****/ T::T /*****/
( int h )                   //Bezparametrický konstruktor +
{                           //konverzní k-r pro celá čísla
  Hodnota = h;
  Por = ++Poradi;           //Vznikla další instance daného typu
  Pocet++;                  //Přibylo instancí daného typu
  cout << „TP - Vytvořeno: „ << *this;
}
/********** T::T **********/

/*****/ T::T /*****/
( const T& t )   //Kopírovací konstruktor
{
  Hodnota = t.Hodnota;
```

```
  Por = ++Poradi;          //Pořadí nepřebírá, generuje vlastní
                           //protože vzniká další instance
  Pocet++;                 //Přibylo instancí daného typu
  cout << "TK - Vytvořeno: " << *this;
}
/********** T::T **********/

/*****/ T::~T /*****/
()         //Destruktor
{
  Pocet--;                 //Ubylo instancí daného typu
  cout << "DD - Zrušeno: " << *this;
  if( !Pocet )  //Po zrušení poslední instance
  cout << "############################\n\n";
}
/********** T::~T **********/

T t1 = 10;

T /*****/ T5 /*****/
( T t=888 )
{
  cout << "...................\n";
  static T t4 = 40;        //Konstruuje se při prvním volání
                           //destruuje se po skončení programu
  cout << "T5 - lokální proměnné: \n"
       << " Parametr t: " << t
       << " Statická t4: " << t4;
 {
  T t01 = 100;        //Konstruuje se při každém vstupu,
                      //destruuje se před opuštěním bloku
  cout << " Automatická t01: " << t01;
 }
  t4.Hodnota++;
  cout << "::::::::::::::::::::\n";
  return 5555;
}
/********** T5 **********/

T t2 = 20;

void /*****/ Test5a /*****/
()
{
  T t02 = 200;             //Konstruuje se při každém volání,
                           //destruuje se před opuštěním fce
  cout << "Test5a - lokální proměnné: \n"
       << " Automatická t01: " << t02;
 {
  static T t3 = 30;        //Konstruuje se při prvním vstupu,
                           //destruuje se po skončení programu
  cout << " Statická t3: " << t3;
 }
  cout << "\nVolám T5( t02 ) - funkce vrací hodnotu:\n";
  cout << T5( t02 );
  cout << "\nVolám T5() - funkce vrací hodnotu:\n";
  cout << T5();
```

```
  cout << "\nVolám T5( 111 ) - funkce vrací hodnotu:\n";
  cout << T5( 111 );
}
/********** Test5a **********/
```

V Pascalu se destruktory definují stejně jako procedurální metody, pouze se místo klíčového slova **procedure** napíše klíčové slovo **destructor**. Destruktory si můžete označit libovolně, ale bývá zvykem je označovat identifikátorem *Done* (uděláno)**.**

Jak si jistě sami domyslíte, v Pascalu si musí programátor všechny netriviální destruktory (tj. ty, které jsou definovány, protože je není možno nahradit implicitními) zavolat sám. Volat by je měl správně v těch okamžicích, kdy by je v ekvivalentním programu v C++ volal překladač, tj. destruktory automatických proměnných při opouštění podprogramu, v němž byly definovány, destruktory externích proměnných ve finalizační proceduře modulu.

Samostatnou kapitolou jsou lokální statické proměnné, tj. lokální proměnné definované v sekci **const**. Ty by se totiž měly destruovat po posledním opuštění daného podprogramu. Jak ale poznáte, které volání podprogramu je to poslední? Jedno rozumné řešení, jak se z dané situace vylhat: Definujte si nějaký vektor, do nějž budete ukládat ukazatele na tyto proměnné (nejlépe pomocí nějaké procedury), a proceduru, kterou zavoláte těsně před ukončením programu a která zdestruuje všechny proměnné, na něž prvky daného vektoru ukazují.

Jiná možnost: nepoužívat lokální statické proměnné objektových typů s netriviálními destruktory a definovat tyto proměnné raději jako externí, tj. mimo všechny podprogramy. V tu chvíli ale ztrácíte neovlivnitelnost této proměnné v důsledku chyb v jiných částech programu a to by bylo proti zásadám moderního programování. Proto jsme v následující ukázce použili první možnost.

```
(*  Příklad P5 - 6  *)
{Program realizuje stejné funkce, jako odpovídající ukázka v C++}
const
  Pocet : integer = 0;
  Poradi: integer = 0;
type
  T = object
       Hodnota : integer;
       Por : integer;
       constructor Init;
       constructor InitI( i:integer );
       constructor InitT( t:t );
       constructor Init_;
       destructor Done;
       procedure Write;
  end;
  pT = ^T;                          {Ukazatel na objekt typu T}

procedure (*****) T.Write (*****)
;
begin
  Writeln( 'Pořadí=', Por, ', Hodnota=', Hodnota,
```

```
  ', Živých=', Pocet );
end;
(********** T.Write **********)

constructor (*****) T.Init (*****)
;                                {Bezparametrický konstruktor}
begin
  Hodnota := 999;
  system.Write('T. - ' );        {Kdybychom napsali samotné write,
               překladač by to chápal jako volání metody write.
               Proto musíme uvést i jednotku System}
  T.Init_;
end;
(********** T.Init **********)

constructor (*****) T.InitI (*****)
( i:integer );              {Konverzní konstruktor pro celá čísla}
begin
  Hodnota := 999;
  system.Write( 'TI - ' );
  self.Init_;
end;
(********** T.InitI **********)

constructor (*****) T.InitT (*****)
( t:T );                    {Kopírovací konstruktor}
begin
  Hodnota := t.Hodnota;
  system.Write( 'TT - ' );
  T.Init_;
end;
(********** T.InitT **********)

constructor (*****) T.Init_ (*****)
;                           {Neinicializující konstruktor}
begin
  Inc( Poradi );
  Por := Poradi;
  Inc( Pocet );
  system.Write( 'Vytvořeno: ' );            {Voláme systémovou funkci}
  write;                                    {Voláme metodu }
end;
(********** T.Init_ **********)

destructor (*****) T.Done (*****)
;
begin
  Dec( Pocet );
  system.Write( 'DD - Zrušeno: ' );         {Volám syst. funkci}
  write;                                    {Volám metodu }
  if( Pocet = 0 )then
       writeln( '##########################' );
end;
(********** T.Done **********)

const nst : integer = 0;    {Počet lokálních statických
                             objektů určených k závěrečné destrukci}
var vst : array[ 1..5 ] of ^T;   {Vektor ukazatelů na lokální statické
```

```
                              objekty určené k závěrečné destrukci}
procedure (*****) StaticT (*****)
{Procedura zařadí svůj parametr do vektoru objektů
 destruovaných až při ukončení programu }
( t : pT );
begin
  Inc( nst );
  vst[ nst ] := t;
end;
(********** StaticT **********)

procedure (*****) DoneStaticT (*****)
{Tuto proceduru je třeba volat před ukončením programu, aby zdestruovala
 lokální statické objekty zařazené do vektoru vst}
;
begin
  while( nst > 0 )do begin
       vst[ nst ]^.Done;
       Dec( nst );
  end;
end;
(********** DoneStaticT **********)

var t1 : T;
{
  Z nejasných důvodů odmítá překladač TP 6.0 v následující proceduře u
  proměnných t4 a t01 typ T, přestože v proceduře Test5, která za ní
  vzápětí následuje, mu tento typ u proměnných t3 a t01 nevadí.
  Experimentálně jsme zjistili, že následující finta zabírá, přestože
  nevíme proč.
}
type Q=T;
procedure (*****) T5 (*****)
{ Pascal nedovoluje, aby funkce vracely hodnoty objektových typů. Jednou
  z možností řešení je přidání výstupního parametru.
}
( t : T; var ret : T );
const t4 : Q=();      {Výše zmiňovaná finta - myslete si,
                  že je typu T}
POPRVE : boolean = TRUE;
var t01 : Q;          {Opět finta na odmítání typu T}
begin
  writeln( '......................' );
  t.Init_;            {Konstruktor nepřepisující předávané hodnoty}
  if( POPRVE )then
  begin
  t4.InitI( 40 );
  StaticT( @t4 );
  POPRVE := FALSE;
  end;
  writeln( 'T5 - lokální proměnné: ' );
  write( ' Parametr t: ' ); t.write;
  write( ' Statická t4: ' ); t4.write;
  begin
       t01.InitI(100);      {Konstruuje se při každém vstupu, }
                            {destruuje se před opuštěním bloku }
```

```
      write( ' Automatická t01: ' ); t01.write;
      t01.Done;
  end;
  Inc( t4.Hodnota );
  ret.Hodnota := 5555;
  writeln( '::::::::::::::::::::::' );
end;
(********** T5 **********)

const t2 : T = ();

procedure (*****) Test5 (*****)
;
const
  t3 : T=();
  POPRVE : boolean = TRUE;
var
  t02 : T;
  tpom: T;                {Pomocná proměnná pro předání parametru
                           funkci T5 a pro převzetí funkční hodnoty}
begin
  t02.InitI(200);         {Konstruuje se při každém volání, }
                          {destruuje se před opuštěním funkce }
  writeln( 'Test5 - lokální proměnné: ' );
write( ' Automatická t02: ' ); t02.write;
  begin
      if( POPRVE )then begin
        t3.InitI( 30 );
        StaticT( @t3 );
        POPRVE := FALSE;
      end;
      write('Statická t3: ' ); t3.write;
  end;
  Writeln( 'Voláme T5( t02 ) - funkce vrací hodnotu:' );
  tpom.Init;              {Konstrukce pomocné proměnné}
  T5( t02, tpom );        {Převzetí návratové hodnoty}
  tpom.write;             {Zpracování návratové hodnoty}
  Writeln( 'Volám T5() - funkce vrací hodnotu:' );
{ Protože jsme nedefinovali funkci pro inicializaci instancí typu T
  prostřednictvím celočíselné hodnoty a nechceme používat nečisté
  praktiky přímého přístupu k atributům, pomůžeme si následující fintou:
}
  tpom.Done;              {Nejprve musíme proměnnou destruovat}
  tpom.InitI( 999 );      {abychom ji mohli zkonstruovat s novou
                           počáteční hodnotou}
  T5( tpom, tpom );       {První parametr předáváme hodnotou,
                       takže můžeme tutéž pomocnou proměnnou
                       použít pro převzetí funkční hodnoty}
  tpom.write;         {Zpracování návratové hodnoty}
  tpom.Done;          {Pokud budeme proměnnou konstruovat
                       hned po použití, snížíme pravděpodobnost,
                       že na destruktor zapomeneme}
  Writeln( 'Volám T5( 111 ) - funkce vrací hodnotu:' );
  tpom.InitI( 111 );  {Při vícenásobném použití dané proměnné se její
                       opakované konstruování a destruování nevyplácí,
                       také bývá jednodušší definovat nějakou
```

```
                        jednoduchou kpírovací funkci }
  T5( t02, tpom );
  tpom.Write;
  tpom.Done;
  t02.Done;
end;
(********** Test5 **********)
begin
  t1.InitI( 10 );          {Předpokládáme, že funkce Test5 slouží}
  t2.InitI( 20 );          {i jako inicializační rutina modulu }
  writeln( '--------------------' );
  Test5;
  writeln( '======================' );
  DoneStaticT;
  t2.Done;
  t1.Done;
  writeln( '======================' );
end.
```

Pascal, na rozdíl od C++, umožňuje definovat několik destruktorů, lišících se navzájem svými identifikátory. Dokonce umožňuje definovat i destruktory s parametry (to C++ také neumí). V běžné programátorské praxi ale této možnosti většinou nevyužijeme.

V souvislosti s pascalskými destruktory je třeba se ještě zmínit o destruktorech složených tříd, tj. tříd, jejichž některé složky jsou objektových typů. **Destruktory složených tříd musí ve svém těle obsahovat i volání destruktorů všech objektových složek destruované instance**, přičemž tyto destruktory by měly být volány v obráceném pořadí k pořadí volání jejich konstruktorů v konstruktoru složené třídy. Bohužel, pokud na to zapomenete, překladač vás na vaše opomenutí neupozorní. Nezbývá vám tedy, než se ohlídat sami.

2.6 Přístupová práva – třídní oblast platnosti

Zapouzdření nám slouží ke dvěma účelům. Za prvé k tomu, abychom mohli sdružit objekty definovaného typu s příslušnými operacemi, a za druhé k tomu, abychom jasně odlišili ty rysy definovaného typu – třídy, které mohou používat i části programu stojící mimo definici dané třídy, od rysů, do nichž okolnímu programu nic není. Tohoto rozlišení dosahujeme definicí přístupových práv k jednotlivým deklarovaným složkám.

V rámci tohoto dílu budeme rozlišovat dva typy složek (v příští knize přidáme ještě třetí): složky, které zpřístupníme každému, budeme označovat jako **veřejné** (*public*), a naopak složky, které chceme před ostatními částmi programu skrýt, budeme označovat jako **soukromé** (*private*). Můžeme si to představit tak, že třída o těchto složkách prohlašuje, že jsou její soukromou věcí a že do nich nikomu nic není. K soukromým složkám, a to jak atributům, tak metodám, pak mají přístup pouze metody této třídy. (Výjimky z tohoto pravidla poznáme později.)

Podívejme se nyní, jak se výše uvedené konstrukce realizují v obou probíraných jazycích:

Deklarace třídy může v C++ sestávat ze sekcí, které obsahují složky se stejnými přístupovými právy. Implicitní přístupová práva jsou pro třídy založené na uniích a strukturách veřejná. Tato přístupová práva platí i pro složky mezi hlavičkou třídy a první specifikací přístupových práv.

Novou sekci s odlišnými přístupovými právy deklarovaných složek otevřeme jedním z klíčových slov **private**, **protected** nebo **public**, za které napíšeme dvojtečku. (Formálně to pak vypadá jako návěští v deklaraci.) Klíčovým slovem **private** tak uvádíme sekci soukromých složek a klíčovým slovem **public** sekci složek veřejných. O významu klíčového slova **protected** si povíme v dalším dílu („na příštím schodě“), až se dozvíme něco o dědičnosti.

Syntaktický popis definice třídy bychom mohli při našich současných znalostech zapsat asi takto:

Definice_třídy:
Druh_třídy Ident_třídy {[Sekce]opak };

Druh_třídy:
1 z: **class struct union**

Sekce:
[Specifikátor_přístupu :] [deklarace]opak

Ze syntaktické definice vyplývá, že sekcí může být v definici libovolný počet, tj. že každá sekce se může v definici vyskytnout i několikrát, a že není žádným způsobem předepsáno jejich pořadí.

Jak jste si jistě všimli, mezi druhy třídy bylo spolu se známými klíčovými slovy **struct** a **union** uvedeno i klíčové slovo **class** (třída). Třída deklarovaná pomocí tohoto klíčového slova má naprosto shodné vlastnosti s třídou definovanou pomocí klíčového slova **struct**. Jedinou výjimkou, o které si zatím řekneme, je to, že složky třídy definované pomocí klíčového slova **class** jsou implicitně soukromé.

Domníváme se, že klíčové slovo **class** bylo zařazeno do jazyka C++ z jediného důvodu: aby bylo možno jednoduchým způsobem vizuálně odlišit definice objektových datových typů od těch, které se chystáme definovat klasicky, tj. neobjektově. Pokud se podíváte na publikované programy, tak se v nich používá pro definice objektových typů takřka výhradě klíčové slovo **class**. Budeme je proto od této chvíle používat i v naší knize.

Ze syntaktického popisu definice třídy a z toho, co jsme si řekli o implicitních přístupových právech, je vám asi zřejmé, že definice objektových typů, používající klíčové slovo **class**, by měly obsahovat alespoň dvě sekce, protože složky deklarované v nulté sekci jsou soukromé a objekty, jejichž všechny atributy i metody jsou soukromé, budeme těžko používat. (O tom, k čemu mohou být takové třídy dobré, si povíme ještě v této kapitole, až se dostaneme ke spřáteleným funkcím a třídám v podkapitole *Přátelé*.)

V následující ukázce si definujeme datový typ *cDatum* s využitím všeho, co jsme se doposud dozvěděli:

```
/*  Příklad C5 - 12  */
#include <dos.h>
typedef unsigned word;      //Pro sjednocení s Pascalem
/*****/ class cDatum /*****/
{
  public:
       word Rok() {return rok; };
       word Mesic() {return mesic; };
       word Den() {return den; };
       cDatum( word Den=0, word Mesic=0, word Rok=0 );
//Předchozí k-r slouží i jako prázdný k-r
//Kopírovací konstruktor nedefinujeme, protože nám
//funkce implicitního kopírovacího k-ru vyhovuje
       Nastav( word Den=0, word Mesic=0, word Rok=0 );
       void NastavRok( word=0 );
       void NastavMesic( word=0 );
       void NastavDen( word=0 );
  private:
       word rok;
       word mesic;
       word den;
}
/********** class cDatum **********/

/*****/ cDatum::cDatum /*****/
( word Den, word Mesic, word Rok )
{
  NastavDatum( Den, Mesic, Rok );
}
/********** cDatum::cDatum **********/

void /*****/ cDatum::Nastav /*****/
( word Den, word Mesic, word Rok )
{
  struct date SysD;
  if( (Den & Mesic & Rok) != 0 )
  getdate( &SysD );
  if( Den )
       NastavDen( Den )
  else
       den = SysD.da_day;
  if( Mesic )
       NastavMesic( Mesic )
  else
       mesic = SysD.da_mon;
  if( Rok )
       NastavDen( Rok )
  else
       rok = SysD.da_year;
}
/********** cDatum::Nastav **********/

void /*****/ cDatum::NastavMesic /*****/
( word Mesic )
{
  if( (Mesic==0) || (Mesic > 12)
```

```
  {
       struct date SysD;
       getdate( &SysD );
       mesic = SysD.da_mon;
  }
  else
       mesic = Mesic;
}
/********** cDatum::NastavMesic **********/
//Další definice si již dodělejte sami nebo si
//je zkopírujte z doprovodné diskety

ostream& /*****/ operator << /*****/
( ostream& o, cDatum& D )
//Tělo tohoto operátoru je podrobně okomentováno, abyste
//se naučili definovat obdobné operátory i pro své vlastní
//datové typy.
{

//Zapamatuj si původní nastavení formátovacích příznaků
//a nastav tisk v desítkové soustavě zarovnaný doprava.
//Hodnota ostatních příznaků pro nás nemá význam.

  long flags = o.flags( ios::right + ios::dec );

//Zjisti rozdíl mezi požadovanou a potřebnou velikostí
//oblasti pro tisk

  int width = o.width() - 8;

//Zapamatuj si nastavený výplňový znak -
//my budeme plnit nulou - např. 02.07.92

  char fill = o.fill( '0' );

//Pokud je požadována větší oblast než je nezbytně třeba
//a pokud nebylo nastaveno zarovnávání vlevo, zaplň
//úvodní přebytek výplňovými znaky

  if( (width > 0) && !(flags & ios::left) )
  {

//Příkaz je v příkazových závorkách proto,
//aby proměnná i zůstala v tomto bloku lokální

       for( int i=0; i < width; i++, o << fill );
  }

//Vytiskni požadovaná data

  o << setw( 2 ) << D.Den() << "."
    << setw( 2 ) << D.Mesic() << "."
    << setw( 2 ) << (D.Rok() % 100) ;       //Jen poslední dvojčíslí

//Obnov původní nastavení formátovacíh příznaků
//a výplňového znaku
  o.fill( fill );
  o.flags( flags );

//Pokud byla požadována větší oblast než potřebná a
//pokud bylo nastaveno zarovnávání vlevo, zaplň
```

```
//závěrečný přebytek výplňovými znaky
   if( (width > 0) && (flags & ios::left) )
   {
        for( int i=0; i < width; i++, o << fill );
   }
   return o;
}
/********** operator << **********/

void /*****/ Test6_1 /*****/
()
{
   cDatum Dnes, Zitra, Loni;
   Zitra.NastavDen();
}
/********** Test6_1**********/
```

V předchozí ukázce jste se možná zarazili nad tím, že konstruktor a metoda *Nastav* dělají vlastně totéž, a možná vás napadlo, zda by nebylo možno definovat pouze jednu z těchto metod. Nebylo. Jak si jistě pamatujete, jednou ze základních vlastností konstruktorů je to, že je překladač automaticky volá ve chvílích, v nichž to považuje za potřebné. K tomu bychom jej u obyčejné metody nepřiměli. Konstruktor je prostě obyčejnou metodou nezastupitelný.

Obdobně je tomu i naopak. Stejně, jako není možné nahradit konstruktor obyčejnou metodou, není možné ani nahradit obyčejnou metodu konstruktorem. Metoda *Nastav* nám totiž umožní přiřadit dané instanci, zkonstruované již dříve, novou hodnotu. Naproti tomu konstruktor není možno volat jako obyčejnou metodu dříve zkonstruované instance. Konstruktor vždy vytváří instanci novou – a to by se nám v dané situaci asi nehodilo.

Turbo Pascal zavedl od veze 7.0 možnost omezení přístupu k některým složkám třídy. Bohužel konstrukce, s níž autoři Turbo Pascalu přišli, se dotýká řízení přístupových práv složek třídy pouze nepřímo. Turbo Pascal verze 7.0 totiž zavedl možnost vymezit pomocí klíčových slov **private** a **public** definici třídy na několik částí. Syntaktický popis definice objektového typu by tedy s tímto rozšířením mohla vypadat takto:

Definice_třídy:
object *Ident_třídy = [spec_příst_práv sekce] end*

Sekce:
[Deklarace_atributu]opak [Deklarace_metody]opak

Specif_příst_práv:
1 z: **public private**

Z této definice je zřejmé, že v TP musíme v každé sekci deklarovat nejprve datové složky (atributy), a teprve pak funkční složky (metody).

Vraťme se ale k přístupovým právům. Nejprve upozornění: TP verze 6.0 obsahuje pouze klíčové slovo **private** a TP verze 5.5 neumožňuje omezit přístupová práva vůbec.

Složky, které deklarujeme za hlavičkou objektu před první specifikací přístupových práv, jsou veřejné. Podobně jsou veřejné složky, deklarované v sekci za klíčovým slovem **public** v TP 7.0.

Složky, deklarované za klíčovým slovem **private**, jsou sice také veřejné, ale navíc jsou lokální v daném modulu. Pro daný modul vystupují tedy jako veřejné a pro ostatní moduly jako soukromé.

Pascalská koncepce předpokládá, že každá třída deklarována bude v samostatném modulu. Pokud toto pravidlo nedodržíme, vystavujeme se poněkud většímu riziku než v C++. Tato koncepce totiž dostatečně nebrání tomu, aby se „v útrobách" instancí dané třídy „přehrabovaly" i procedury a funkce, které by k tomu neměly mít žádné oprávnění. Vzniklá rizika jsou navíc umocněna tím, že pascalští programátoři inklinují k tvorbě obrovských modulů, které mají i několik tisíc řádek. V takových modulech prudce roste pravděpodobnost neúmyslných porušení různých konvencí – např. právě pokud jde o přístupová práva ke složkám programů. Musíte se proto naučit dodržovat dobrovolně zásady, k jejichž dodržování nás v C++ nutí překladač.

V následující ukázce jsme se pokusili definovat třídu *cDatum*, která je ekvivalentní stejnojmenné třídě z ukázky pro jazyk C++.

```
 (*  Příklad P5 – 7  *)
type
  (*****)cDatum (*****) = object
       function Rok : word;
       function Mesic: word;
       function Den : word;
       constructor Init ( Den, Mesic, Rok : word );
       procedure NastavDatum( Den, Mesic, Rok : word );
       procedure NastavRok ( R : word );
       procedure NastavMesic( M : word );
       procedure NastavDen ( D : word );
       procedure write;
       procedura writef( var f:text; n:integer );
  private
       rr : word;
       mm : word;
       dd : word;
  end;
(********** object cDatum **********)

constructor (*****) cDatum.Init (*****)
( Den, Mesic, Rok : word );
begin
  NastavDatum( Den, Mesic, Rok );
end
(********** cDatum::cDatum **********)

procedure (*****) cDatum.Nastav (*****)
( Den, Mesic, Rok : word );
var
  d, m, r, t : word;
begin
  if( (Den and Mesic and Rok) <> 0 )then
```

```
  getdate( r, m, d, t );
  if( Den <> 0 )then
       NastavDen( Den )
  else
       dd := d;
  if( Mesic <> 0 )then
       NastavMesic( Mesic )
  else
       mm := m;
  if( Rok )then
       NastavDen( Rok )
  else
       rt := r;
end;
(********** cDatum::Nastav **********)

procedure (*****) cDatum::NastavMesic (*****)
( Mesic : word )
var
  d, m, r, t : word;
begin
  if( (Mesic=0) or (Mesic > 12) )
  then begin
       getdate( r, m, d, t );
       mm := m;
  end else
       mm := Mesic;
end;
(********** cDatum::NastavMesic **********)
{ Další definice si již dodělejte sami nebo si
  je zkopírujte z doprovodné diskety }

procedure (*****) cDatum.write (*****)
;
begin
  writef( output, 0 );
end;
(********** cDatum.write **********)

procedure (*****) cDatum.writef (*****)
( var f:text; n:integer );
{ Metoda umožňuje tisknout datum ve stejném formátu jako operátor <<
  jazyka C++, s výjimkou možnosti zarovnání výsledku vlevo, a definice
  výplňového znaku, které Pascal neposkytuje ani ve standardní proceduře
  write
}
procedure T( w:word );
{ Lokální procedura pro tisk dvojciferných čísel
  s případnou vedoucí nulou
}
begin
  if( w < 10 )then
       write( f, 0 );
  write( f, w );
end;
begin
```

```
  if( n > 8 )then
  write( f, ' ':n-8 );
  T( dd ); write( f, '.' );
  T( mm ); write( f, '.' );
  T( rr mod 100 );    {Pouze poslední dvojčíslí}
end;
(********** cDatum.writef **********)

procedure (*****) Test6_1 (*****)
;
cDatum Dnes, Zitra, Loni;
begin
 Zitra.nastavDen(0);
end;
(********** Test6_1**********)
```

Jak jsme si již několikrát řekli, objektové datové typy nám oproti klasickým datovým typům poskytují při programování v řadě směrů daleko větší možnosti. Abychom je proto na první pohled poznali i v našich programech, dohodneme se, že identifikátory všech objektových typů (tříd) budou začínat malým **c** (jako **c**lass), za nímž bude následovat vlastní jméno daného typu, začínající velkým písmenem.

V předchozích ukázkách si všimněte metod *Den*, *Měsíc* a *Rok*, které nedělají nic jiného, než že předávají volajícímu programu hodnotu daného atributu. Mohlo by se zdát, že jsou zbytečné, když si můžeme hodnotu daného atributu zjistit přímo z obsahu patřičné instance.

Trochu méně pochybností budou vyvolávat metody *NastavDen*, *NastavMěsíc* a *NastavRok*, protože jejich těla jsou již komplikovanější, ale mnohé z vás jistě napadlo, že pro řadu situací jsou takto definovaná nastavení hodnot zbytečně složitá a že byste jim v řadě situací dokázali přiřadit jejich hodnoty efektivněji – přímo.

Souhlasíme s vámi, že přímý přístup je v řadě případů efektivnější (nepočítáme-li vložené – *inline* – funkce jazyka C++, které programu na efektivitě nic neubírají), ale v žádném případě není bezpečnější. Přímým přístupem k atributům zvyšujete riziko chybného nastavení a stavíte se tak do opozice zásadám, které stály u zrodu objektově orientovaného programování.

Připomeňme si, že jedním z hlavních cílů OOP je další zvýšení produktivity programátorské práce. Jednou z metod, kterými toho chce dosáhnout, je, že nabízí programátorům prostředky k tomu, aby se části programu, které jsou již odladěné, nestaly zdrojem chyb v důsledku špatné interakce se zbytkem programu. Jinými slovy, když jednou odladíme modul ošetřující práci s daty, chceme mít jistotu, že se na odpovídající datový typ můžeme vždy spolehnout a že se nám např. v jeho „útrobách“ nebude „přehrabovat“ nikdo nepovolaný.

Pro ty, které jsme dosud nepřesvědčili, máme ještě druhý argument. Představte si, že při ladění programu zjistíte, že vám data zabírají neúměrně velký paměťový prostor, a rozhodnete se, že vám datum nemusí zabírat 6 bajtů paměti, když všichni vědí, že se pohodlně vejde do dvou (5 bitů pro den, 4 bity pro měsíc a 7 bitů pro 128 roků rozumně zvoleného období – tak to má např. DOS). Všichni ti, kteří nastavovali hodnoty atributů instancí typu *cDatum* přímo, se musí po této změně ihned ponořit do svých programů

(a to mohou být desetitisíce řádek) a najít v nich všechna místa, kde četli nebo přiřazovali hodnoty složkám instancí typu *cDatum*, a na všech těchto místech program změnit. Je více než pravděpodobné, že někde některé přiřazení přehlédnou, nebo naopak, někde jinde změní něco, co původně vlastně vůbec měnit nechtěli.

Pokud budete celý program vytvářet v duchu OOP a budete respektovat to, že se ke složkám instancí objektových typů dostanete pouze zprostředkovaně, budou pro vás všechny požadavky na takové změny definic tříd, které nevedou ke změnám jejich rozhraní s okolním světem, znamenat pouze nutnost přepracování té části kódu, která s danou změnou definice přímo souvisí, a zbytku programu si vůbec nemusíte všímat – zůstane takový, jaký byl.

Zprostředkovaný přístup ke složkám s sebou nese mnoho výhod i pro proces ladění. Při jeho respektování je snadné modifikovat čtecí funkci či zapisovací proceduru tak, aby prováděla v průběhu ladění přísnější kontroly (a v ostré verzi pak tyto kontroly v zájmu maximální efektivity vynechat), aby v průběhu ladění pořizovala záznam o každém přiřazení do definovaného souboru, aby přerušila běh programu v případě, že někdo chce přiřazovat nekorektní hodnoty, aby, aby, aby…

2.7 Statické atributy a metody

V minulé podkapitole jsme brojili proti neřízenému přístupu ke složkám instancí objektových typů, ale pokud si prohlédnete předchozí příklady, najdete tam několik definic tříd, které si počítají své prvky – např. třída *Počítaná* v části věnované C++ a před tím třída *B* v ukázkách pro oba jazyky. Všechny tyto třídy trpěly stejným neduhem: proměnná, která obsahovala počet vytvořených instancí, a proměnná obsahující pořadí vytvářené instance, byly obyčejné globální proměnné. To znamená, že k nim měl přístup úplně každý podprogram v daném modulu. To je ale právě to, proti čemu jsme v minulé kapitole vedli plamennou řeč.

Brojit proti něčemu je jedna věc, a vyřešit daný problém je věc druhá. Problém počítání instancí, a mnoho problémů podobných, nelze bohužel řešit doposud probranými prostředky čistě, tj. tak, aby byly dodrženy nejdůležitější principy objektově orientovaného programování. Proměnná, obsahující počet vytvořených instancí, totiž nemůže být složkou instancí dané třídy, protože pak bychom měli po paměti roztroušeno tolik proměnných, kolik by bylo vytvořeno instancí, a my bychom nemuseli vždy poznat, která z nich je ta pravá (pokud bychom danou informaci nechtěli udržovat ve všech instancích, ale to je ještě větší nesmysl).

Jazyk C++ řeší tyto problémy zavedením atributů, které jsou pro všechny instance společné, a metod, které nepoužívají skrytý parametr **this** (ekvivalent pascalského *self*)**.**

Poznámka:
V originálních manuálech pro C++ se tyto atributy a metody nazývají statické (static members). Týž termín se však zároveň používá i pro metody, které nejsou virtuální, což je zcela jiná třída metod. V překladech manuálů, vydávaných firmou Apro, se používá

mnohem lepší termín „atributy a metody třídy“, zatímco atributy a metody, o nichž jsme hovořili doposud, jsou označovány jako „atributy a metody instancí“.

Tyto termíny jsou sice výstižné, ale občas se „fackují“ s češtinou, protože při jejich používání přestanou být některé věty jednoznačné. Napíšu-li např., že někde použiji „atribut třídy A“, není z toho jasné, zda se jedná o „<atribut třídy> A“ (tj. o atribut s identifikátorem A, který je atributem třídy) nebo o „atribut <třídy A>“ (tj. o atribut, který je datovou složkou třídy s identifikátorem A, přičemž se nic neříká o tom, zda se jedná o atribut třídy nebo o atribut instancí).

*V naší knize se proto přikloníme k termínu **statický** s tím, že pro označení metod, které nejsou virtuální, zavedeme jiný termín. Atributům a metodám, které jsme používali doposud, většinou žádný přívlastek dávat nebudeme. Pokud bychom někdy potřebovali jejich nestatičnost zdůraznit, budeme je označovat jako **běžné**.*

Poznámka:
Pokud jsou mezi čtenáři-pascalisty takoví, které nezajímají konstrukce, jež Pascal neumí, mohou klidně zbytek kapitoly přeskočit. Poznamenáváme ale, že se statickými metodami se můžeme setkat v Pascalu, implementovaném v Delphi – viz Dodatek.

Statické atributy

Statické atributy (tj. atributy tříd) se chovají obdobně jako běžné statické proměnné deklarované na dané úrovni. Je-li jejich třída definována externě (tj. je-li definována mimo těla funkcí), chovají se jako globální proměnné, je-li jejich třída definována uvnitř funkce či vnořeného bloku, chovají se jako lokální statické proměnné. Od těchto proměnných se odlišují pouze omezením přístupových práv definovaným v definici třídy – veřejný (public) statický atribut vám oproti odpovídající proměnné přináší pouze jediné rozšíření, a to možnost omezení počtu použitých identifikátorů, a tím i zpřehlednění programu (např. statické atributy uchovávající počet vytvořených instancí dané třídy se mohou ve všech třídách jmenovat *Pocet*).

Statické atributy tedy použijeme všude tam, kde potřebujeme sdílet nějakou informaci všemi instancemi dané třídy, avšak nechceme tuto informaci zpřístupňovat každému, ale chceme mít naopak možnost řídit k ní přístupová práva stejně, jako řídíme přístupová práva k běžným, nestatickým atributům.

V C++ musíme statické atributy **deklarovat dvakrát** (tj. obdobně jako metody). Nejprve musíme v definici třídy uvést deklaraci daného atributu a označit jej pomocí specifikátoru **static** jako statický, a pak jej na vhodném místě definovat jako odpovídající proměnnou nebo konstantu. V této definici pak uvedeme identifikátor definované konstanty či proměnné ve tvaru:

<identifikátor_třídy> :: <identifikátor_atributu>

Statickým atributům můžeme přiřadit počáteční hodnotu pouze v této definici.

Pokud jsou statické atributy objektových datových typů, platí pro volání jejich konstruktorů stejná pravidla jako pro volání konstruktorů externích proměnných, což znamená, že konstruktory statických atributů jsou volány ještě před procedurami uvedený-

mi v direktivě **#pragma startup**, a tím i před vstupem do funkce *main*, přičemž pořadí jejich volání odpovídá pořadí jejich definice ve zdrojovém textu programu.

Z předchozího odstavce ovšem vyplývá jedna závažná věc: pokud se budou definice instancí dané třídy vyskytovat ve zdrojovém programu před definicemi jejich objektových statických atributů, nebudou při konstrukci těchto instancí odpovídající statické atributy ještě zkonstruovány, což může ve svých důsledcích vést k záhadnému chování vytvořeného programu.

U neobjektových atributů a atributů objektových typů, které nemají explicitní konstruktory, toto nebezpečí nehrozí, protože jejich definitivní podoba je připravena již v EXE souboru. Přesto bychom měli i definice těchto atributů umístit ve zdrojovém textu před definice instancí jejich definiční třídy (tj. třídy, jejímiž jsou atributy).

Se statickým atributem můžeme v programu pracovat stejně jako s běžným atributem. Přestože se totiž nezávisle na počtu instancí vyskytuje v paměti pouze v jednom exempláři, je dostupný všem instancím, a jediný rozdíl je, že se při práci s ním všechny instance odvolávají na stejné místo v paměti. Můžeme jej tedy kvalifikovat identifikátorem kterékoliv instance dané třídy, tj. identifikovat jej zápisem

<ident_instance>.<ident_atributu>

Abychom mohli pracovat se statickými atributy i tehdy, když ještě žádné instance dané třídy nevznikly, zavádí C++ ještě druhou možnost, a to kvalifikaci identifikátorem třídy. Identifikátor třídy však při tomto typu kvalifikace neoddělujeme tečkou, ale čtyřtečkou, tj. identifikujeme atribut zápisem

<indent_třídy>::<ident_atributu>

```
/*  Příklad C5 – 13  */
#include <iostream.h>
class Trida
{
  public:
       char* t;
       Trida()
       {t = "1234 třída";
        cout << "+++ Vytvořena Třída +++\n"; };
       ~Trida()
       {cout << „--- Zrušena Třída ---\n"; };
};

class Pocitana
{
       static int Zrizeno;       //Implicitně private
       static int Zivych;
       static const Trida T;     //Objektový atribut
       const int Poradi;
       const char * Text;
  public:
       Pocitana( char*s="" );
  ~Pocitana();
};
```

```
Pocitana Prvni = "Prvni";   //CHYBA = Instance je
                            //definována před svými statickými atributy!
int Pocitana::Zrizeno = 0;  //Statické atributy musí
int Pocitana::Zivych = 0;   //být definovány před
const Trida Pocitana::T;    //všemi instancemi třídy, jejímiž jsou
  atributy
Pocitana Druha = "Druha";

/*****/ Pocitana::Pocitana /*****/
( char* Jmeno )
: Poradi( ++Zrizeno ), Text( Jmeno )
{
  Zivych++;
  cout << "Zřídil jsem " << Poradi
       << ". instanci se jménem " << Text << (T.t+4)
       << "; celkem je \"živých\" " << Zivych
       << " instancí\n";
 }
/********** Pocitana::Pocitana **********/

/*****/ Pocitana::~Pocitana /*****/
()
{
  Zivych--;
  cout << "Zrušil jsem " << Poradi
       << ". instanci se jmenem " << Text << (T.t+4)
       << "; celkem je \"živých\" " << Zivych
       << " instancí\n";
}
/********** Pocitana::~Pocitana **********/

Pocitana Treti = "Třetí";

void /*****/ Test7_1 /*****/ ()
{
  Pocitana Ctvrta = "Čtvrtá";
  Pocitana Pole[ 3 ] = {"Pátá", "Šest" };
  for( int i=0; i < 3; i++ )
       Pocitana Osma = "Osmá";
}
/********** Test7_1 **********/
```

Pascal statické atributy nezavádí, a musíme v něm proto místo nich i nadále používat lokální proměnné modulu, a v případě veřejných atributů globální proměnné. Pokud se nad problémem trochu zamyslíte, brzy zjistíte, že vlastně pro jejich zavedení ani není důvod, protože Pascal neumí řídit přístupová práva k jednotlivým složkám třídy a jediné, co umožňuje, je lokalizovat v rámci daného modulu atributy a metody uvedené v sekci **private**. Pak je samozřejmě téměř jedno, jestli je dotyčná informace uchovávána ve statickém atributu nebo v lokální proměnné – jedinou přetrvávající nevýhodou je nemožnost omezení počtu identifikátorů, tj. nemožnost označení informací stejného druhu stejným identifikátorem.

Statické metody (metody tříd)

Statické metody se obvykle chovají stejně jako řadové procedury a funkce a liší se od nich zpravidla opět pouze přístupovými právy. Na rozdíl od atributů jsou však „přístupové odchylky“ statických metod dvojí. Od běžných procedur a funkcí se liší za prvé tím, že můžeme v definici třídy ovlivnit, kdo všechno smí danou metodu zavolat (to mají metody s atributy společné), a za druhé tím, že na rozdíl od běžných funkcí mohou (stejně jako ostatní metody) používat ve svém těle i soukromé složky dané třídy, tj. volat soukromé metody a měnit hodnoty soukromých atributů. Mají zkrátka vůči složkám dané třídy stejná práva, jako kterákoliv její jiná metoda.

Statické metody má smysl použít tam, kde:

1. potřebujeme pracovat pouze se statickými atributy (např. v inicializačních podprogramech, které musí připravit vše potřebné proto, aby mohla být zkonstruována první instance), přičemž tyto atributy nemusí být vždy veřejně přístupné,
2. chceme omezit přístupová práva k dané proceduře nebo funkci pouze na metody dané třídy,
3. potřebujeme podprogramu, který z nějakých důvodů nemůže být běžnou metodou, umožnit přístup k soukromým složkám některého ze svých atributů. Tento problém se však většinou řeší pomocí označení daného podprogramu za přítele, což je postup, který bude podrobněji vysvětlen v příští podkapitole.

Turbo Pascal 7.0 metody tříd nezavádí. S těmi se setkáme až v Delphi – viz *Dodatek*.

Stejně jako statické atributy, i statické metody se v definici procedury označují specifikátorem **static**. Definují se pak stejně jako běžné metody, pouze musíme mít při jejich definici na paměti to, že vzhledem k tomu, že nejsou vázány na žádnou instanci, která by je volala, nemůžeme v nich samozřejmě používat identifikátor **this**.

Statické metody můžeme kvalifikovat stejně jako statické atributy, tj. buď prostřednictvím identifikátoru nějaké instance (ten oddělujeme tečkou), nebo prostřednictvím identifikátoru třídy, jejímž je atributem (ten oddělujeme čtyřtečkou).

```
/*  Příklad C5 – 14  */
class Pocitana2
{
   static int Zrizeno;            //Implicitně private
   static int Zivych;
   const int Poradi;
   const char * Text;
public:
   Pocitana2( char*s="" );
   ~Pocitana2();
   static void Stav();
};

void /*****/ Pocitana2::Stav /*****/
()
{
   cout << "Stav instancí třídy »Pocitana2«:\n"
```

```
        <<  " Vytvořeno instancí: " << Zrizeno <<'\n'
        <<  " Živých instancí: " << Zivych << '\n';
}
/********** Pocitana2::Stav **********/

int Pocitana2::Zrizeno = 0;
int Pocitana2::Zivych = 0;
Pocitana2 Prvni = "Prvni";

/*****/ Pocitana2::Pocitana2 /*****/
( char* Jmeno )
: Poradi( ++Zrizeno ), Text( Jmeno )
{
   Zivych++;
   cout << "Zřídil jsem " << Poradi
        << ". instanci se jménem " << Text
        << "; celkem je \"živých\" " << Zivych
        << " instancí\n";
}
/********** Pocitana2::Pocitana2 **********/

/*****/ Pocitana2::~Pocitana2 /*****/
()
{
   Zivych--;
   cout << "Zrušil jsem " << Poradi
        << ". instanci se jmenem " << Text
        << "; celkem je \"živých\" " << Zivych
        << " instancí\n";
}
/********** Pocitana2::~Pocitana2 **********/

void /*****/ Test7_2 /*****/ ()
{
   Prvni.Stav();                  //Kvalifikace instancí
   Pocitana2 Druha = "Druha";
   Pocitana2 Pole[ 3 ] = {"Třetí", "Čtvrtá" };
   Pocitana2::Stav();             //Kvalifikace třídou
   for( int i=0; i < 3; i++ )
   {
        static char* j[3] =
            {"Šestá (6)", "Šestá (7)", "Šestá (8)" };
        Pocitana2 Sesta = j[ i ];
        Pocitana2::Stav();
   }
}
/********** Test7_2 **********/
```

2.8 Přátelé

To, že mohu zakázat přístup k některým složkám všem, kdo stojí mimo třídu, je sice krásné, ale občas je třeba rozlišovat. Lidé také zamykají své příbytky, ale dobrým přátelům jsou ochotni klíče od bytu půjčit – např. proto, aby jim během dovolené zalili kytič-

ky a postarali se o zvířata. Stejně i třídy si mohou vybrat ve svém okolí své přátele, kterým přístup do svého soukromí umožní – většinou proto, že od nich něco potřebují.

Poznámka:
Protože Pascal nezavádí možnost řízení přístupových práv k jednotlivým složkám tříd (jak jsme si již řekli, sekce private hovoří tak trochu o něčem jiném), nepotřebuje zavádět ani institut přátelství. Pokud jsou mezi čtenáři-pascalisty takoví, které konstrukce, jež Pascal neumí, nezajímají, mohou klidně zbytek kapitoly přeskočit.

Přáteli se mohou stávat jak jednotlivé procedury nebo funkce, tak i celé třídy. Pokud je přítelem deklarované třídy celá třída, vztahuje se možnost přístupu ke všem složkám deklarované třídy na všechny metody spřátelené třídy.

Stejně jako v běžném životě, i v programování platí, že přátelství není tranzitivní. To znamená, že pokud je třída *B* přítelem třídy *A* a pokud je třída *C* přítelem třídy *B*, vůbec z toho ještě nevyplývá, zda je třída *C* také přítelem třídy *A* či nikoliv.

To, kdo patří mezi její přátele, musí deklarovat každá třída sama. Nikdo sám o sobě nemůže prohlásit: „Já jsem kamarád toho a toho, a proto chci to a to.“ Je tomu právě naopak. Nositelem sdělení o přátelství může být jedině třída, o jejichž soukromých složkách se jedná. Jedině ta může ve své deklaraci překladači oznámit: „Toto jsou moji přátelé, tak jim nebraň v přístupu k mým soukromým složkám.“

Z předchozího odstavce je asi zřejmé, že přátelství není ani reflexivní, což znamená, že je-li *A* přítelem *B* (tzn., že třída *B* prohlásí třídu *A* za svého přítele), ještě neznamená, že *B* je také přítelem *A*. Může být, ale také nemusí. Vše záleží na tom, zda i třída *A* uvede třídu *B* mezi svými přáteli či nikoliv.

Možnost označení některých programů za přátele se využívá hlavně v případech, které byly v předchozí podkapitole uvedeny v sekci o statických funkcích jako třetí možný důvod jejich použití. Hlavní výhodou spřátelených podprogramů oproti statickým metodám je, že je při jejich použití nemusíme kvalifikovat, a programy, které je využívají, jsou proto přehlednější.

V praktických programech jsme se doposud setkali se spřátelenými podprogramy nejčastěji při rozšiřování definic různých operátorů, o němž budeme hovořit v příští podkapitole, v níž si také ukážeme, proč tomu tak je. Se spřátelenými třídami jsme se pak setkali nejčastěji při definici tzv. iterátorů, o nichž si budeme povídat, jakmile si ukážeme možnosti dynamické alokace instancí objektových datových typů.

V C++ se přátelé označují pomocí klíčového slova **friend**, které je následováno buď prototypem podprogramu nebo identifikátorem třídy, kterou právě deklarovaná třída označuje za svého přítele. Pokud chcete zdůraznit, že deklarovaným přítelem je třída, můžete před identifikátor spřátelené třídy uvést příslušné klíčové slovo z trojice **class**, **struct**, **union**.

```
/*  Příklad C5 - 15  */
#include <iostream.h>
class cTajnaSchranka
{
  char * Obsah;
```

```
//Protože doposud nebylo změněno implicitní nastavení přístupových práv,
//je následující konstruktor soukromý, a proto je z programu
//nedosažitelný.
//Instanci typu cTAjnaSchranka proto mohou zkonstruovat pouze přátelé.
   cTajnaSchranka( char* o = „“ ) {Obsah = o; };
   friend class cAgent;

//Abychom mohli při ladění programu používat kontrolní tisky, musíme si
//zajistit přístup i k soukromým složkám
   friend ostream& operator<< ( ostream& o, cTajnaSchranka ts )
   {
        o << ts.Obsah << „\n“;
        return o;
   };
};

class cAgent
{
   int Cislo;
   char * KryciJmeno;
   cAgent( int=0, char* = "" );
   void ZridSchranku(){};
   void VyberSchranku( cTajnaSchranka& TS ){};
   void PredejZpravu( char* Text, cTajnaSchranka& TS ){};
   void UkazSchranku( cAgent A, cTajnaSchranka* TS ){};
   friend ostream& operator<< ( ostream&, cAgent);
   friend cRidiciOrgan;
};

class cRidiciOrgan
{
public:
   void ZridAgenta( int Cislo ) {};
   void ZrusAgenta( int Cislo ) {};
};

/*****/ cAgent::cAgent /*****/
( int c, char* j )
{
   if( c && *j )
   {
        Cislo = c;
        KryciJmeno = j;
   }
   else
        cout << "\n\n\7AGENT MUSÍ MÍT PŘIDĚLENO ČÍSLO "
             "A KRYCÍ JMÉNO!\7\n\n";
}
/********** cAgent::cAgent **********/

ostream& /*****/ operator << /*****/
( ostream& o, cAgent a)
{
   o << "Agent číslo " << a.Cislo
     << " má krycí jméno " << a.KryciJmeno << "\n";
   return o;
}
```

```
/********** operator << **********/
```

K tomu, v kterých situacích je vhodné přátelství deklarovat, se ještě vrátíme v pasážích věnovaných odpovídajícím tématům.

2.9 Metody aplikovatelné na konstantní instance

V podkapitole o instancích objektových datových typů jsme hovořili o tom, že v C++ můžeme definovat nejen proměnné, ale i konstanty objektových datových typů. Zároveň jsme si říkali, že překladač „dohlédne“ na to, abychom hodnotu těchto konstant v průběhu programu nezměnili.

Toto jeho snažení však má jeden háček: překladač neumí v obecném případě posoudit, zda nějaká metoda ovlivní hodnotu své instance či nikoliv, a tudíž vám pro jistotu nedovolí pro konstantu použít žádnou metodu. Jediné, co s ní můžete dělat, je předávat ji jako parametr procedurám a funkcím, které mají ve svém prototypu u příslušného parametru specifikátor **const**.

Aby bylo možno aplikovat na konstantní objekty metody, zavádí C++ možnost označit metody, které nemění hodnoty instance, pro kterou je voláme (tj. nemění ***this**), takže je lze bezpečně použít i na konstanty. Tyto metody se v deklaraci (v definici i v prototypu) označují klíčovým slovem **const**, které se uvede za závorkou, ukončující seznam parametrů.

```
/*  Příklad C5 – 16  */
#include <dos.h>
typedef unsigned word; //Pro sjednocení s Pascalem
/*****/ class cDatum /*****/
{
  public:
//Prvé tři metody je možno použít i na konstanty
       word Rok() const {return rok; };
       word Mesic() const {return mesic; };
       word Den() const;
       cDatum( word Den=0, word Mesic=0, word Rok=0 );
       Nastav( word Den=0, word Mesic=0, word Rok=0 );
       void NastavRok( word=0 );
       void NastavMesic( word=0 );
       void NastavDen( word=0 );
  private:
       word rok;
       word mesic;
       word den;
};
/********** class cDatum **********/

word /*****/ cDatum::Den /*****/
() const
//Klíčové slovo const NESTAČÍ uvést pouze v prototypu
//v definici příslušné třídy
 {
```

```
  return den;
 }
/********** cDatum::Den **********/
```

Poznámka:

Vedle metod, které lze používat pro konstantní instance, lze v C++ deklarovat a používat také metody pro nestálé instance, tj. instance, deklarované s modifikátorem ***volatile****. Syntax těchto metod je podobná syntaxi metod pro konstantní instance, pouze klíčové slovo* **const** *nahradíme klíčovým slovem* **volatile**.

3. Přetěžování operátorů

Otevíráte kapitolu, v níž se budeme zabývat jednou z nejpříjemnějších novinek, s nimiž se setkáte „na prvém schodu“, a to možností rozšíření definic operátorů. V předchozím dílu jste se zatím naučili rozšiřovat pouze definice vstupního operátoru >> a výstupního operátoru <<. Nyní si možnosti rozšíření definic operátorů probereme v plné šíři.

Jazyk Pascal definice operátorů bohužel rozšiřovat neumí. Přesto pascalistům doporučujeme, aby tuto kapitolku nepřeskakovali a pokusili se ji alespoň zběžně prolistovat; vždy se hodí vědět o možnostech, které nabízí konkurence – a navíc nelze např. vyloučit, že se s ní v některé z příštích verzí Turbo Pascalu setkáte. Kromě toho, syntaktická zjednodušení nejsou všechno. Problémy, které řešíme v C++ pomocí přetížených operátorů, můžeme v Pascalu vyřešit jinak – sice někdy na pohled méně elegantně, nicméně stejně účinně.

Poznámky k terminologii:
V časopisecké verzi kursu jsme místo o přetěžování funkcí hovořili o homonymech. Tento termín naprosto přesně vystihuje, o co jde: Dvě různé operace se jmenují stejně. V české literatuře se ovšem zpravidla používá termín přetěžování, což je doslovný překlad anglického výrazu overloading. V této knize budeme podle okolností používat oba názvy.

Homonyma, probíraná v této podkapitole, budeme také nazývat rozšiřující definice operátorů, protože rozšiřují množinu datových typů, na něž jsou rozšiřované operátory aplikovatelné. Časem si povíme ještě o druhé možné podobě homonymních definic.

Jednou z příjemností, které nám jazyk C++ nabízí, je možnost přetěžovat nejen funkce, ale i převážnou většinu operátorů. Přesněji řečeno, můžeme **přetěžovat všechny unární a binární operátory kromě operátoru . (tečka), .* (tečka–hvězdička) a :: (čtyřtečka) a operátoru sizeof**. Nelze také přetěžovat ternární operátor **?:** (podmíněný výraz) a operátory **typeid**, **dynamic_cast**, **static_cast**, **reinterpret_cast** a **const_cast**, které zavedla norma ANSI a se kterými se setkáme v BC++ počínaje verzí 4.0. Nelze pochopitelně přetěžovat ani operátory preprocesoru #, ## a **defined**. To ale vlastně nejsou operátory jazyka C++; ty zpracovává preprocesor, nikoli překladač.

Jak vidíte, je tedy možno rozšířit téměř všechny operátory. Dále si povíme, jak se operátory přetěžují a k čemu by nám mohlo být takové přetížení užitečné. Nejprve se ale seznámíme s obecnými pravidly.

1. Z hlediska přetěžování se operátory v C++ obvykle rozdělují na 4 skupiny.
   V první z nich jsou operátory, které nelze přetěžovat vůbec – o těch jsme si již pověděli.
   Druhou skupinu tvoří operátory, které můžeme přetěžovat pouze jako nestatické metody objektových typů. Sem patří operátory **( )** (volání funkce), **[ ]** (indexování), **->** (nepřímého přístupu), = (prosté přiřazení) a operátor přetypování **(***typ***)**.

Třetí, nejrozsáhlejší skupinu tvoří operátory, které lze přetěžovat jako nestatické metody objektových typů nebo jako řadové funkce. Jejich výčet nebudeme uvádět – sem patří prostě všechny operátory, které nejsou v ostatních skupinách. Tyto operátory musí mít alespoň jeden operand objektového nebo výčtového typu.

Čtvrtou skupinu tvoří operátory pro správu paměti **new** a **delete**. Pro ně platí zvláštní pravidla, a proto o nich budeme hovořit později samostatně.

Homonyma operátorů druhé skupiny musí být metodami objektových typů. Homonyma operátorů třetí skupiny musí být buď metodami objektových typů nebo musí mít alespoň jeden parametr objektového nebo výčtového typu[4] (předávaný hodnotou nebo odkazem).

To znamená, že nelze změnit chování operátorů nad argumenty standardních datových typů.

2. Homonyma operátorů mají stejnou prioritu jako odpovídající standardní operátory. To znamená, např. binární operátor ***** (hvězdička – standardně násobení) bude mít vždy vyšší prioritu než operátor **+** (plus – standardně sčítání) a jazyk C++ neumožňuje tuto prioritu změnit.
3. Homonyma operátorů mají stejnou asociativitu jako odpovídající standardní operátory. Připomeňme si, že asociativita určuje pořadí vyhodnocování shodných operátorů ve výrazech. Např. operátor **+** je asociativní zleva doprava, to znamená, že výraz

```
1 + 2 + 3 + 4
```

se interpretuje jako

```
((1 + 2) + 3) + 4
```

a tak tomu bude i pro všechna homonyma tohoto operátoru.

4. Homonyma operátorů zachovávají aritu (počet operandů) odpovídajících standardních operátorů. Např. operátor **!** (vykřičník) je unární (má jeden operand) a operátor **/** (lomítko) binární (má dva operandy) a tuto aritu musí zachovat i všechna jejich homonyma.

   Operátory **&** (ampersand), ***** (hvězdička), **+** (plus) a **-** (minus) mohou být jak unární, tak binární. Při různých aritách však mají různý význam – např. unární hvězdička označuje operaci dereference (získání hodnoty na zadané adrese), kdežto binární hvězdička operaci násobení.

   Výjimečné postavení má z tohoto hlediska operátor volání funkce **()** (závorky), který může mít jakoukoliv aritu větší nebo rovnu jedné. (Prvním operandem je volaná procedura nebo funkce, dalšími jsou pak její parametry.)

[4] Možnost definovat homonyma některých operátorů pro výčtové typy zavedla až norma ANSI. Setkáme se s ní v překladačích firmy Microsoft počínaje verzí 7.0 a v borlandských překladačích počínaje verzí 4.0. Starší překladače povolují přetěžování operátorů druhé skupiny pouze pro objektové typy.

5. Pro přetížené operátory nesmíme definovat implicitní hodnoty parametrů – to je však zřejmé z předchozího bodu, protože pak bychom měnili jejich aritu.
6. Pokud přetížíme operátor jako nestatickou metodu, bude mít o jeden parametr méně, protože prvým parametrem bude instance dané třídy (tj. ***this**).
7. Pokud nám nevyhovuje definice, která má jako prvý parametr objekt dané třídy, nemůžeme definovat operátor jako nestatickou metodu. V takovém případě jej zpravidla definujeme jako spřátelenou funkci.
8. Definici jazyka C++ nelze rozšiřovat zaváděním nových operátorů. Není tedy možno zavést dosud neexistující operátor – nelze např. přidat fortranský operátor **** (dvě hvězdičky) a definovat pro něj operaci umocňování.

Kromě výše uvedených zásad, které jsou dány definicí jazyka, je vhodné při definicích homonym operátorů dodržovat následující doporučení:

1. Nesnažte se využívat možnosti definice homonymních operátorů za každou cenu. Nevhodné rozšíření definic operátorů může sice ušetřit trochu práce při psaní programu, ale na druhou stranu velice výrazně ztíží porozumění napsanému programu. V mnoha případech může být použití operátorů zavádějící a matoucí, takže se jako daleko výhodnější může ukázat použití klasických podprogramů.
   Představte si např., že budeme chtít v nějakém programu vypočítat datum, které leží uprostřed mezi dvěma zadanými daty. Asi bude pro získání výsledku výhodnější zvolit klasickou funkci než např. homonymum operátoru dělení.
2. Pokud se rozhodnete pro řešení založené na rozšíření definice některého operátoru, pečlivě vybírejte, který operátor je svojí přirozeností zadané úloze nejblíže. Volba vhodného operátoru je vlastně volbou identifikátoru, kterým budeme danou operaci v programu označovat. U tříd numerického charakteru (komplexní čísla, zlomky, pole atd.) je volba většinou poměrně jasná, a tudíž bezproblémová. Nejasnosti však mohou vzniknout při volbě operátorů pro reprezentaci nenumerických operací.
3. Pamatujte na to, že rozšíření definice jednoho operátoru neznamená automatické rozšíření definic operátorů, které s ním nějak souvisejí. Pro standardní operátory sice platí, že příkazy

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

   znamenají totéž, pro homonyma to ovšem neplatí – alespoň ne automaticky. Rozšíření definice operátoru + ještě žádným způsobem nepředurčuje chování operátoru += a prefixové a postfixové verze operátoru ++. Vše záleží na tom, jak budou tyto tři operátory rozšířeny.

3.1 Přiřazovací operátory

Mezi přiřazovací operátory řadíme jak operátor prostého přiřazení (operátor =), tak i operátory, které ve své klasické podobě provedou před vlastním přiřazením ještě nějaký další výpočet. Jedná se tedy o tyto operátory:

```
=    !=    %=    &=    *=    +=    -=    <<=    >>=    ^=    |=
```

Obecné poznámky

Operátor prostého přiřazení definuje překladač implicitně kdykoli je třeba – stejně, jako např. kopírovací konstruktor (mají k sobě ostatně velice blízko).

Připomeňme si, že prostý přiřazovací operátor můžeme přetěžovat pouze jako nestatickou metodu objektového typu, zatímco složené operátory můžeme přetěžovat i jako řadové funkce.

Zpravidla rozlišujeme dva druhy prostých přiřazovacích operátorů: **kopírovací operátor** a **vkládací operátor**. Kopírovací operátor ve třídě *X* je metoda s prototypem

```
X& X::operator=(X&);
```

popřípadě

```
X& X::operator=(const X&);
```

Jako vkládací operátor označujeme metodu, která má parametry jiného typu než *X*.
Je-li *a* instance třídy *X*, znamená zápis

```
a = b;
```

totéž jako

```
a.operator=(b);
```

Musíme také ještě jednou zdůraznit, že přetížením operátoru „=“ nebo „*“ jsme ještě nepřetížili složený operátor „*=“ a můžeme jej definovat jakkoli. Totéž platí pochopitelně i pro ostatní složené přiřazovací operátory. Je ovšem zpravidla rozumné definovat je tak, aby opravdu dělaly to, co jejich symbol naznačuje – tedy aby např. přetížený operátor „*=“ používal služeb operátorů „=“ a „*“.

Použití přiřazovacího operátoru

První věc, kterou si musíme při rozšiřování přiřazovacího operátoru rozmyslet, je otázka, zda opravdu chceme přiřazovací operátor explicitně rozšířit, a v případě že ano, tak proč.

Implicitní verze přiřazovacího operátoru řeší přiřazování hodnot objektů prostým okopírováním jednoho objektu do druhého. To nám přestane vyhovovat např. ve chvíli, kdy naše objekty budou rozděleny na několik částí svázaných navzájem ukazateli. Typickým příkladem takto konstruované třídy by mohla být třída *cStrPas* z následující ukázky, v níž je částečně definován datový typ s obdobnými možnostmi, jaké má pas-

calský datový typ *String*. (V příštích podkapitolách budeme definici postupně rozšiřovat.)

Pascalské řetězce jsou pole znaků předem známé délky. Ve své nulté položce obsahují skutečnou délku textu, který je v nich uložen – z toho také automaticky vyplývá omezení na maximální možnou délku řetězce, která činí 255 znaků.

Instance našeho typu budou obsahovat pouze ukazatel na pole znaků a ve zvláštním atributu si budou pamatovat velikost vyhrazené paměti. Aby však bylo možno využívat doposud používané řetězcové operace a aby se usnadnila koexistence instancí tohoto typu s řetězci v klasické podobě, budou texty v naší třídě ukončeny prázdným znakem, tj. znakem s kódem 0.

```
/* Příklad C6 – 1 */
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class /*****/ cStrPas /*****/
//Třída implementující pascalské řetězce
{
  public:
       cStrPas( int md=255 );
       cStrPas( const char * );
       cStrPas( const cStrPas & );
       ~cStrPas() {if( Text ) delete Text; };
 // vkládací operátor
       cStrPas& operator= ( const char * );
 // kopírovací operátor
       cStrPas& operator= ( const cStrPas& );
       cStrPas& operator+= ( const char * );
       cStrPas& operator+= ( const cStrPas& );
  private:
       int MaxD;      //Maximální délka řetězce
       char * Text;
       void Prirad( const char*, int );
       void Pridej( const char*, int );
       friend ostream& operator<< ( ostream&, cStrPas& );
};
/********** cStrPas **********/

/*****/ cStrPas::cStrPas /*****/
( int md )
{
  MaxD = md;
  Text = new char[ md+2 ];       //Vyhraď v paměti místo
  *Text = 0;                     //Nultý znak obsahuje délku
  Text[1] = 0;                   //Prázdný řetězec
}
/********** cStrPas::cStrPas **********/

/*****/ cStrPas::cStrPas /*****/
( const char * s )
 {
  MaxD = strlen(s);
  if( MaxD > 255 ) MaxD = 255;
```

```
  Text = new char[ MaxD+2 ];                //Vyhraď v paměti místo
  *Text = MaxD;                             //Nultý znak obsahuje délku
  strncpy( Text+1, s, MaxD );               //Zkopíruj do něj text
  Text[ MaxD+1] = 0;                        //Aby text končil prázdným
}                                           //znakem, i kdyby byl delší.
/********** cStrPas::cStrPas **********/

/*****/ cStrPas::cStrPas /*****/
( const cStrPas& S )
//Kopírovací konstruktor - implicitní verze je nepřijatelná, protože by
//nový objekt ukazoval na tentýž textový řetězec, jako objekt, který
//kopírujeme.
{
  Text = new char[ S.MaxD+2 ];              //Vyhraď místo v paměti
  strcpy( Text, S.Text );                   //Zkopíruj tam řetězec
  if( *Text==0 )                            //Kdyby byl řetězec prázdný,
      Text[1] = 0;                          //nezkopíroval by se
}
/********** cStrPas::cStrPas **********/

void /*****/ cStrPas::Prirad /*****/
( const char * s, int i )
//Pomocná metoda, která přiřadí volající instanci hodnotu
//textového řetězce, který je na adrese s a má i znaků.
{
  if( i > MaxD ) i = MaxD;                  //Aby nám "nevytekl"
  *Text = i;                                //Nultý znak obsahuje délku
  strncpy( Text+1, s, i );                  //Zkopíruj do něj text
  Text[ i+1] = 0;                           //Aby text končil prázdným
}                                           //znakem, i kdyby byl delší.
/********** cStrPas::Prirad **********/

inline cStrPas& /*****/ cStrPas::operator= /*****/
( const char * s )
{
  Prirad( s, strlen( s ) );
  return *this;
}
/********** cStrPas::operator= **********/

inline cStrPas& /*****/ cStrPas::operator= /*****/
( const cStrPas& S )
//Implicitní verze přiřazovacího operátoru je nepřijatelná, protože
//zkopíruje pouze hodnotu ukazatele a vlastního textu si nevšímá -
//ukazatele by pak v obou objektech ukazovaly na tentýž textový řetězec,
//a to je špatně.
{
  Prirad( S.Text+1, *S.Text );
  return *this;
}

/********** cStrPas::operator= **********/

void /*****/ cStrPas::Pridej /*****/
( const char * s, int i )
//Pomocná metoda, která přiřadí volající instanci hodnotu
//textového řetězce, který je na adrese s a má i znaků.
{
```

```
   if( (*Text+i) > MaxD )
        i = MaxD = *Text;          //Aby nám "nevytekl"
   strncpy( Text + (*Text + 1), s, i );
   *Text += i;          //Připočteme délku
   Text[ *Text+1] = 0;            //Aby text končil prázdným
}               //znakem, i kdyby byl delší.
/********** cStrPas::Pridej **********/

inline cStrPas& /*****/ cStrPas::operator+= /*****/
( const char * s )
{
   Pridej( s, strlen( s ) );
   return *this;
}
/********** cStrPas::operator+= **********/

inline cStrPas& /*****/ cStrPas::operator+= /*****/
( const cStrPas& S )
{
   Pridej( S.Text+1, *S.Text );
   return *this;
}
/********** cStrPas::operator+= **********/

inline ostream& /*****/ operator << /*****/
( ostream& o, cStrPas& s )
{
   o << (void*) (s.Text+1)                    //Adresa textu
    << ": " << int(*s.Text)       //Délka textu
    << " »" << (s.Text+1) << "«\n";          //Vlastní text
   return o;
}
/********** operator << **********/

void /*****/ Test_1 /*****/ ()
{
   cStrPas a = "Adam";
   cStrPas b( "Božena" );
   cStrPas bb( b );
   cStrPas c;
   cout << "a =" << a << "b =" << b
        << "bb =" << bb << "c =" << c;
   // zde použijeme kopírovací operátor
   c = a;
   cout << "c2 =" << c;
   // zde použijeme vkládací operátor
   cout << "c3 =" << (c = "Cyril");
   c += b;
   cout << "c4 =" << c;
   cout << "c5 =" << (c += " Alfons");
}
/********** Test_1 **********/
```

V předchozí ukázce jsme definovali řetězce přesně podle pascalského vzoru, tj. staticky: jakmile jsme přidělili našemu řetězci místo v paměti, již mu zůstalo až do doby, než jsme instanci destruovali. To s sebou ovšem přináší dvě nevýhody: za prvé můžeme

blokovat zbytečně mnoho místa v paměti a za druhé nemůžeme vyloučit situaci, kdy nám předem vyhrazené místo nebude stačit.

V klasickém jazyku C se s řetězci pracuje většinou dynamicky: pro řetězce se vyhradí pouze tolik paměti, kolik nezbytně potřebuje. Pokud v pozdější době vyvstane potřeba řetězec prodloužit, vyhrazená paměť se vrátí systému a pro řetězec se vyhradí paměť nová. Totéž se však udělá i v případě, kdy je potřeba řetězec zkrátit. Nevýhodou daného řešení oproti statickému je větší režie spojená s přidělováním a uvolňováním paměti, a navíc i možnost rozdrobení volné paměti v haldě.

Zkusme si nyní definovat takovouto třídu v obou probíraných jazycích. Opustíme však pascalskou koncepci délky uložené na počátku řetězce a přidělíme délce vlastní atribut. Naproti tomu si již nemusíme pamatovat délku přidělené paměti, protože víme, že tato délka je shodná s délkou řetězce.

Domníváme se, že předchozí příklad by mohl být dostatečným vodítkem pro to, abyste se pokusili definovat tuto třídu (nazvěme ji *cStrDyn*) sami a svá řešení pak porovnali s řešením ukázkovým.

```
/*  Příklad C6 – 2  */
typedef unsigned word;
class /*****/ cStrDyn /*****/
//Třída implementující řetězce dynamicky
{
  public:
       cStrDyn() {Text = NULL; };
//Prázdný ukazatel symbolizuje to, že daný řetězec
// ještě nemá přiřazenu žádnou hodnotu
       cStrDyn( const char * );
       cStrDyn( const cStrDyn & );
       cStrDyn( int, const char * s )       //Konstruktor konstant
       {Delka = strlen( s );                //- podrobnosti viz
        Text = (char*)s;
       };                                   //text za příkladem
       ~cStrDyn()
       {if( Text ) delete Text; };
       cStrDyn& operator= ( const char * s )
       {return Prirad( s, strlen(s) ); };
       cStrDyn& operator= ( const cStrDyn& S )
       {return Prirad( S.Text, S.Delka ); };
       cStrDyn& operator+= ( const char * s )
       {return Pridej( s, strlen(s) ); };
       cStrDyn& operator+= ( const cStrDyn& S )
       {return Pridej( S.Text, S.Delka ); };
  private:
       word Delka;
       char * Text;
       void Platny( const char* = (const char*)2, int=0 )const;
       cStrDyn& Prirad( const char*, int );
       cStrDyn& Pridej( const char*, int );
       friend ostream& operator<< ( ostream&, const cStrDyn& );
};
/********** cStrDyn **********/

/*****/ cStrDyn::cStrDyn /*****/
```

```
( const char * s )
{
  Platny( s, 1 );
  Delka = strlen( s );
  Text = new char[Delka+1];      //Vyhraď v paměti místo
  strcpy( Text, s );             //Zkopíruj inicializační text
}

/********** cStrDyn::cStrDyn **********/

/*****/ cStrDyn::cStrDyn /*****/
( const cStrDyn& S )
{
  S.Platny();               //Přiřazovaný řetězec musí mít hodnotu
  Delka = S.Delka;
  Text = new char[ Delka+1 ];    //Vyhraď místo v paměti
  strcpy( Text, S.Text );        //Zkopíruj tam řetězec
}

/********** cStrDyn::cStrDyn **********/

void /*****/ cStrDyn::Platny /*****/
( const char * s, int i )const
//Pomocná metoda pro usnadnění kontroly korektnosti operací.
//Není-li ukazatel s i ukazatel this->Text nenulový,
//vypíše zprávu a přeruší další běh programu.
//Je-li parametr i nenulový, netestuje se volající instance.
{
  if( s && (i || Text) ) return;
  cerr << "\n\n\7\7Použití řetězce bez hodnoty\7\7\n\n";
  abort();
}
/********** cStrDyn::Platny **********/

cStrDyn& /*****/ cStrDyn::Prirad /*****/
( const char * s, int i )
//Pomocná metoda, která přiřadí volající instanci hodnotu
//textového řetězce, který je na adrese s a má i znaků.
{
  Platny( s, 1 );           //Test ukazatelů Text a s
  if( Text )                //Ukazatel někam ukazuje => řetězec měl
  delete Text;              //původně jinou hodnotu - smazat!
  Delka = i;
  Text = new char[ Delka+1 ];    //Vyhraď místo
  strcpy( Text, s );             //Zkopíruj text řetězce
  return *this;
}

/********** cStrDyn::Prirad **********/
cStrDyn& /*****/ cStrDyn::Pridej /*****/
( const char * s, int i )
{
  Platny( s );              //Test ukazatelů Text a s
  if( *s )                  //Přidáváme neprázdný řetězec
  {                         //- jinak není co řešit
       char* T = new char[ Delka + i + 1 ];
       strcpy( T, Text );   //Přesuň text na nové místo
```

```
      strcpy( T+Delka, s );
      Delka += i;
      delete Text;         //Smaž původní hodnotu řetězce
      Text = T;
  }
  return *this;
}
/********** cStrDyn::Pridej **********/

inline ostream& /*****/ operator << /*****/
( ostream& o, const cStrDyn& s )
{
  o << (void*) s.Text            //Adresa textu
    << ": " << s.Delka           //Délka textu
    << " »" << s.Text << "«\n";  //Vlastní text
  return o;
}
/********** operator << **********/

void /*****/ Test_2 /*****/ ()
{
  cStrDyn a = "Adam";
  cStrDyn b( "Božena" );
  cStrDyn bb( b );
  cStrDyn c;
  const cStrDyn d( 1, "David" );
  cout << "a =" << a << "b =" << b
       << "bb =" << bb << "c =" << c << "d =" << d;
  c = a;
  cout << "\nc2 =" << c;
  cout << "bb2 =" << (bb = "Bohoušek ");
  bb += d;
  cout << "bb3 =" << bb;
  cout << "bb4 =" << (bb += " Josefka");
  a = bb = NULL;           //Nepovolená operace - měl by křičet
}
/********** Test_2 **********/
```

Povězme si nyní o některých obratech, které by vám mohly připadat podivné.

Prvním z nich by mohlo být to, že jsme prázdným ukazatelem na text označovali neexistenci hodnoty dané instance. Je asi jasné, že po definici automatické instance některého zabudovaného datového typu bez inicializace je v dané proměnné nějaké blíže nedefinované „smetí“. Méně jasné je však to, zda daná definice instance objektového datového typu zahrnuje inicializaci nebo ne. To překladač z podoby konstruktoru odvodit neumí, a je na programátorovi, aby určil, kdy je již daná proměnná inicializovaná – a tedy použitelná – a kdy ještě ne.

Je výhodné, pokud můžeme definovat neinicializující konstruktor tak, aby operátory, které chtějí danou proměnnou použít, uměly poznat, že dotyčná proměnná ještě není použitelná, a na tuto skutečnost nás nějakým způsobem upozornily. Rozšiřujeme tak možnosti překladače, který se nám snaží tuto službu poskytnout u proměnných zabudovaných skalárních typů.

Druhým „podezřelým“ obratem by mohl být dvouparametrický konstruktor, u jehož prvního (celočíselného) parametru není uveden žádný identifikátor. Tento parametr nám slouží pouze k tomu, abychom odlišili daný konstruktor od druhého konstruktoru, jehož parametrem je také textový řetězec. Tím, že jsme neuvedli u parametru jméno, jsme překladači předem naznačili, že dotyčný parametr nehodláme používat a že nás proto nemá upozorňovat, že jsme jej nepoužili.

Můžete namítnout, že by téhož bylo možno dosáhnout v jediné definici, která by měla celočíselný parametr s implicitní hodnotou a dvěma větvemi, které by řešily obě varianty konstrukce. Je to sice pravda, ale takto jsou obě varianty lépe odděleny, a kromě toho jsme vás chtěli seznámit s často používaným obratem.

Rozebíraný konstruktor má ale ještě další podezřelé vlastnosti. Konstruovanému objektu nealokuje místo v haldě, kam by si mohl uložit inicializační text, ale místo toho nasměruje odpovídající ukazatel přímo na tento text. To je sice možné, ale nese to s sebou dvě rizika: nikdo nesmí změnit inicializační text, protože by se tím automaticky změnila i hodnota řetězce konstruovaného, a nikdo nesmí změnit hodnotu konstruované instance, protože tím by se automaticky změnila i hodnota řetězce inicializačního.

Tento konstruktor bude výhodné použít v případě, kdy zřizujeme konstantu (nechtěná změna její hodnoty je tak téměř vyloučena) a inicializačním řetězcem je literál, u nějž změna hodnoty také nehrozí (nesmíte však mít povoleno slučování řetězců).

Součástí předchozího příkladu nejsou jen konstruktory a destruktory, ale i operátory přiřazení. Protože jsme chtěli, aby je bylo možno použít jak s klasickými textovými řetězci, tak s řetězcem právě deklarovaného typu, jsou ve dvou verzích. Protože by si však jejich těla byla velice podobná, vytvořili jsme pomocnou funkci, která realizuje společnou část algoritmu, a oba operátory jí pouze předávají vhodné parametry.

Všimněte si i obratu s příkazem **return**, u nějž jsme využili toho, že pomocné funkce vracejí odkaz na **this**, a tím jsme jej oproti předchozímu programu ještě dále zjednodušili (i když můžete namítnout „jak pro koho“; to je ale do značné míry věc zvyku).

Dovolíme si vám nyní doporučit, abyste si opravdu pozorně prošli všechny tři soukromé metody a pokusili se zdůvodnit, proč jsme je naprogramovali právě takto.

Než budete číst dále, pokusíme se vám nasadit brouka do hlavy.

S výjimkou specifického konstruktoru konstant inicializovaných literály jsme v každém konstruktoru i při každém přiřazení vždy alokovali potřebnou paměť a do ní okopírovali potřebný text. Této alokaci a kopírování se však můžeme vyhnout, a to tak, že dovolíme, aby několik instancí, které obsahují společný text, mohlo sdílet i odpovídající společnou paměť s oním sdíleným textem. Než budete číst dál, pokuste se vymyslet, jak byste tento problém řešili.

Podívejme se nyní, jak bychom třídu dynamických řetězců naprogramovali v Turbo Pascalu.

```
 (*  Příklad P6 – 1  *)
type
  charray = array[0 ..MaxInt] of Char;
  ucha = ^charray;
  cStrDyn = object
```

```
      Delka : word;
      Text : ucha;
      constructor InitKop( var S : cStrDyn );
      constructor InitStr( S : String );
      destructor Done;
      procedure Prirad( var S : cStrDyn );
      procedure Pripoj( var S : cStrDyn );
      procedure PrirStr( S : String );
      procedure PripStr( S : String );
      procedure Write;
      procedure WriteF( var F : text );
  private
      procedure PomPrip( u : ucha; i : integer );
  end;
(********** cStrDyn **********)

constructor (*****) cStrDyn.InitStr (*****)
( S : String );
begin
  Delka := Length( S );
  GetMem( Text, Delka );
  Move( S[1], Text^[0], Delka );            {Inicializační text}
end;
(********** cStrDyn.InitStr **********)

constructor (*****) cStrDyn.InitKop (*****)
( var S : cStrDyn );
begin
  Delka := S.Delka;
  GetMem( Text, Delka );
  Move( S.Text^[0], Text^[0], Delka );      {Zkopíruj řetězec}
end;
(********** cStrDyn.InitKop **********)

destructor (*****) cStrDyn.Done (*****)
;
begin
  FreeMem( Text, Delka );
end;
(********** cStrDyn.Done **********)

procedure (*****) cStrDyn.Prirad (*****)
( var S : cStrDyn );
{Přiřazení dynamického řetězce S }
begin
  FreeMem( Text, Delka );                   {Smaž původní hodnotu}
  Delka := S.Delka;
  GetMem( Text, Delka );                    {Vyhraď místo }
  Move( S.Text^[0], Text^[0], Delka );      {Zkopíruj text}
end;
(********** cStrDyn.Prirad **********)

procedure (*****) cStrDyn.PrirStr (*****)
( S : String );
{Přiřazení textového řetězce S}
begin
  FreeMem( Text, Delka );                   {Smaž původní hodnotu}
```

```
  Delka := Length( S );
  GetMem( Text, Delka );                    {Vyhraď místo}
  Move( S[1], Text^[0], Delka );            {Zkopíruj text řetězce}
end;
(********** cStrDyn.PrirStr **********)

procedure (*****) cStrDyn.Pripoj (*****)
( var S : cStrDyn );
{Připojení dynamického řetězce na konec stávajícího }
begin
  PomPrip( S.Text, S.Delka );
end;
(********** cStrDyn.Pripoj **********)

procedure (*****) cStrDyn.PripStr (*****)
( S : String );
{Přidá na konec dynamického řetězce text z klasického řetězce }
begin
  PomPrip( Pointer( @S[1] ), Length( S ) );
end;
(********** cStrDyn.PripStr **********)

procedure (*****) cStrDyn.PomPrip (*****)
( u : ucha; i : Integer );
{Pomocná metoda pro připojení textu k dynam. řetězci }
var T : ^charray;
begin
  if( i <> 0 )then         {Přidáváme neprázdný řetězec}
  begin         {- jinak není co řešit }
       GetMem( T, Delka+i );
       Move( Text^[0], T^[0], Delka );      {Text na nové místo}
       Move( u^[0], T^[Delka], i );
       FreeMem( Text, Delka );   {Smaž původní hodnotu řetězce}
       inc( Delka, i );
       Text := Pointer( T );
  end;
end;
(********** cStrDyn.PripStr **********)

procedure (*****) cStrDyn.WriteF (*****)
( var F : text );
var i : Integer;
begin
  System.Write( F,    {Musíme označit, o které WRITE nám jde}
       seg( Text ), ':', ofs( Text ),                  {Adresa textu}
       ': ', Delka,                                    {Délka textu }
       ' »' );                   {Uvozovky před vlastním textem}
  for i:=0 to Delka-1 do         {Vlastní text}
       System.Write( F, Text^[i] );
  System.WriteLn( F, '«' );      {Uvozovky za vlastním textem}
end;
(********** cStrDyn.WriteF **********)
```

```
procedure (*****) cStrDyn.Write (*****) ;
begin
  WriteF( OutPut );
end;
```

```
 (********** cStrDyn.Write **********)
var a, b, c: cStrDyn;
procedure (*****) Test_1 (*****)
;
begin
  a.InitStr( 'Adam ' );
  b.InitStr( 'Božena ' );
  c.InitKop( a );
  Write( 'a = ' ); a.Write;
  Write( 'b = ' ); b.Write;
  Write( 'c = ' ); c.Write;
  c.Pripoj( b ); Write( 'c2 = ' ); c.Write;
  c.PripStr( 'Cyril ' ); Write( 'c3 = ' ); c.Write;
  a.Prirad( c ); Write( 'a2 = ' ); a.Write;
  c.PrirStr( 'Cvalda ' ); Write( 'c4 = ' ); c.Write;
  c.Done;
  b.Done;
  a.Done;
end;
(********** Test_1 **********)
```

Srovnáte-li oba programy, zjistíte některé odlišnosti. Povězme si nyní alespoň o těch nejmarkantnějších.

Především nenajdete v definici dva konstruktory: prázdný konstruktor a konstruktor konstant. Důvod, proč jsme nezavedli konstruktor konstant, je jasný: Turbo Pascal skutečné konstanty objektových typů nezná a kromě toho z něj „nevydolujete“ ani adresu literálu.

Prázdný konstruktor také ztrácí v pascalské verzi svůj smysl. V C++ jsme jej zavedli proto, aby bylo možno definovat instance, aniž by se jim přiřazovala počáteční hodnota, a aby bylo možno ohlídat, že používáme proměnnou, která ještě nemá žádnou hodnotu. V Pascalu však budeme vždy definovat instance tohoto typu bez přiřazení počáteční hodnoty a tuto hodnotu jim můžeme korektně přiřadit až voláním konstruktoru. Nic bychom tedy zavedením tohoto konstruktoru nezískali a neušetřili.

Nepřímo jsme ušetřili naopak tím, že jsme tento konstruktor nezavedli, protože nyní nemá smysl kontrolovat nulovost ukazatele na text. Samotná nenulovost tohoto ukazatele totiž ještě vůbec nic neříká o tom, zda v něm není nějaké „smetí“. Toto zjednodušení je však vykoupeno tím, že musíme sami zajistit, aby byla v ukazateli vždy smysluplná hodnota, protože to po nás již nemá kdo zkontrolovat.

Další změna vychází z toho, že Pascal až po verzi 6.0 včetně neumí pracovat s poli předem neznámé délky[5].

Toto omezení obejdeme tak, že použijeme ukazatel na pole maximální možné délky a při alokaci paměti užijeme místo procedury *New* proceduru *GetMem*, které můžeme

[5] Od verze 7.0 umí Turbo Pascal předávat jako parametr pole, jehož délka není známa v době kompilace (viz kap. 7). To nám zde ovšem příliš nepomůže, takže následující výklad platí i pro verzi 7.0.

předepsat, kolik paměti má vlastně pro vytvářený objekt vyhradit. Tak jsme postupovali také v našem programu.

V důsledku všech výše uvedených změn se nám algoritmus operátoru přiřazení trochu zjednodušil, takže jsme ani nevytvářeli společnou pomocnou funkci. Tu jsme v Pascalu vytvořili pouze pro metody připojující další text k textu své instance.

Poslední změnou jsou dvě verze metody *Write* - jedna pro tisk do obecného souboru a druhá zjednodušená pro tisk do standardního výstupního souboru.

Podívejme se nyní na výhody a nevýhody dynamicky implementovaných řetězců a porovnejme je s řetězci implementovanými staticky, tj. např. podle koncepce přijaté v jazyku Pascal.

O jedné výhodě dynamických řetězců jsme již hovořili: Nezabírají žádnou nadbytečnou paměť, protože si pro sebe vyhradí vždy právě tolik paměti, kolik potřebují. Tato úspora paměti je však vykoupena zdrženími při neustálé alokaci a dealokaci paměťového prostoru, Nepříjemným důsledkem může být, že se nám „nešikovnou“ posloupností alokací a dealokací podaří rozdrobit volnou paměť natolik, že v ní již nebude dostatečně velké místo pro alokaci nového objektu, přestože součet velikostí jednotlivých roztroušených volných míst požadovanou velikost mnohonásobně převýší.

Naproti tomu při statické implementaci vyhradíme pro každý řetězec tolik místa, kolik si myslíme, že bude v průběhu programu potřebovat. Tím odpadá nutnost neustálé alokace a dealokace paměti, protože všechny změny mohou probíhat v rámci přiděleného paměťového prostoru. Za to však platíme tu větším, tu menším množstvím zablokované nevyužité paměti. Kromě toho existuje řada aplikací, u nichž paměťové nároky jednotlivých řetězcových proměnných dopředu neznáme, a pokud bychom pro všechny zvolili maximální očekávanou velikost, nevešli bychom se do paměti počítače.

Asi vás napadlo, zda by nebylo možno výše uvedené přístupy nějakým způsobem zkombinovat a vytvořit třídu, kde by byly řetězce implementovány částečně staticky a částečně dynamicky. Mohli bychom to udělat např. tak, že bychom potřebnou paměť alokovali po nějakých rozumně velkých kvantech. Dokud by se řetězec i po úpravách vešel do vyhrazeného prostoru, ponechali bychom jej tam, kde je. Pokud by se do něj nevešel, alokovali bychom pro něj prostor o nějaké to kvantum větší, a původní prostor uvolnili.

Pokud by byly změny velikostí našich řetězců velké, mohli bychom hlídat i to, zda některý řetězec nemá po zkrácení vyhrazený příliš velký prostor, a v případě, že vyhrazený prostor je o několik kvant větší než aktuální délka řetězce, bychom nepotřebné přebytky opět uvolnili.

Předchozí koncepci bychom mohli ještě posílit tím, že bychom netrvali na tom, aby byl paměťový prostor vyhrazený našim řetězcům souvislý. To znamená, že jednotlivé části řetězce by mohly být na různých místech paměti (okolnímu programu se však každý řetězec musí jevit jako jednolitý celek). Tento přístup má tu výhodu, že pak jsou přidělované úseky paměti stejně veliké, a v paměti se proto nebudou objevovat nepoužitelně malé „drobky“, které jinak vznikají tím, že se na místo uvolněné velkým objektem umístí objekt o trochu menší.

Datová struktura, pomocí níž se takovéto „roztroušené“ objekty implementují, se nazývá seznam. Budeme si o ní podrobněji povídat, až probereme běžné operátory a až se budeme zabývat vlastnostmi a možnostmi využití operátorů **new** a **delete**. Pak si také ukážeme, jak je možno realizovat implementaci řetězců, o níž jsme hovořili v minulých odstavcích.

Vraťme se ale k implementacím, na něž nám naše současné znalosti stačí. Před chvílí jsme vám navrhli, abyste se pokusili definovat třídu řetězců tak, že několik řetězců bude moci sdílet společný text v paměti[6].

Pokud totiž v naší aplikaci pracuje více instancí s jedním a týmž textem, je hloupé, aby tento text byl v paměti tolikrát, kolik instancí se na něj odkazuje. Pokud však necháme všechny instance, jejichž hodnotou je daný text, aby sdílely stejný prostor v paměti, vyvstane před námi řada otázek, které budeme muset řešit.

Jednou z prvních a zároveň nejdůležitějších je otázka, co s řetězcem, pokud potřebujeme dané proměnné přiřadit novou hodnotu. Nemůžeme totiž uvolnit paměť blokovanou textem, protože nevíme, zda se na daný text neodkazují ještě jiné instance. Na druhou stranu ale nemůžeme text v paměti jen tak ponechat, protože není vyloučeno, že se na něj již nikdo neodvolává, a že pak bude v paměti zbytečně překážet. Tak bychom si mohli velice rychle „zaplácat“ celou paměť, a to by se nám asi vůbec nelíbilo.

Jak asi sami odhadnete, budeme muset někde udržovat informaci o tom, kolik instancí sdílí daný text. První, co asi mnohé z vás napadne, je definovat atribut, ve kterém si budeme počet „držitelů“ daného textu pamatovat. To znamená, že bychom mohli deklarovat atributy naší třídy v C++:

```
private:
int Pocet;       //Počet držitelů stejného textu
int Delka;       //Počet znaků odkazovaného řetězce
char *Text;      //Ukazatel na vlastní řetězec
```

a v Pascalu

```
Pocet : integer;      {Počet držitelů stejného textu }
Delka : integer;      {Počet znaků odkazovaného řetězce }
Text : ucha;          {Ukazatel na vlastní řetězec }
```

Jak byste ale velice záhy zjistili, takto postupovat nelze. Problém je totiž v tom, že jak délka, tak počet abonentů, jsou vlastnosti vlastního řetězce, a nikoli vlastnosti instance, která se na daný řetězec odkazuje. Pokud by byl daný řetězec např. hodnotou instance *a* a my bychom hodnotu této instance přiřadili instanci *b*, museli bychom upravit atribut *Pocet* u obou instancí. To by nebyl problém. Problém by nastal ve chvíli, kdy bychom hodnotu jedné z našich dvou instancí chtěli přiřadit instanci třetí – nazvěme si ji třeba *c*. Pak bychom museli upravovat hodnotu atributu *Pocet* již u tří instancí. To však znamená, že by o sobě musely jednotlivé instance vědět a nebo mít nějakou jinou možnost jak zařídit, aby se dotyčný atribut aktualizoval u všech instancí, které se odkazují na daný řetězec.

[6] Tento nápad pochází z knížky „Turbo C++ Disk Tutor“.

Autoři citované knížky vyřešili problém tak, že nahradili celočíselný atribut *Pocet* odkazem na číslo, v němž byl počet držitelů uchováván. Upravili tedy výše uvedenou část deklarace třídy na tvar:

```
private:
int *Pocet;      //Počet držitelů stejného textu
int Delka;       //Počet znaků odkazovaného řetězce
char *Text;      //Ukazatel na vlastní řetězec
```

resp.

```
Pocet : ^integer;     {Počet držitelů stejného textu }
Delka : integer;      {Počet znaků odkazovaného řetězce }
Text : ucha;          {Ukazatel na vlastní řetězec }
```

Instance, která se přihlásila k danému řetězci jako první, zřídila v paměti buňku, do níž uložila jedničku (byla prvním a zároveň jediným držitelem). Při přiřazování hodnoty jiné instanci se pak předala pouze hodnota ukazatele na tuto buňku, takže se všechny instance odkazovaly na stejnou buňku v paměti, a jakmile kterákoliv z instancí hodnotu této proměnné změnila, věděly o dané změně i všechny ostatní instance.

Možná, že se nyní budete ptát, proč se neupravil na ukazatel i atribut *Delka*, protože i u něj se jedná o hodnotu, kterou všechny instance sdílejí. Důvody jsou nejméně dva: za prvé se hodnota atributu *Delka* mění pouze při výměně odkazovaného řetězce, takže to není nezbytně potřeba, a za druhé je u některých paměťových modelů výhodnější pamatovat si 2bajtové číslo než 4bajtový ukazatel.

Jedna věc se nám ale na této koncepci nelíbí: myslíme si, že je to sice vykročení správným směrem, avšak nedotažené. Aby totiž bylo možno řetězec při odpoutání se posledního držitele uvolnit z paměti, je třeba v konstruktorech a operátorech přiřazení, jejichž parametrem je klasický textový řetězec používaný v daném jazyku, vyhradit místo v paměti a do něj tento řetězec zkopírovat. Uvolnit totiž můžeme pouze tu paměť, kterou jsme někdy před tím alokovali.

Pokud se zamyslíme nad předchozí námitkou, vidíme, že takto koncipovaná třída odstraňuje problém násobného výskytu řetězců pouze do té doby, dokud navzájem přiřazujeme hodnoty jejich instancí. Pokud však chceme přiřadit několika instancím hodnotu téhož klasického řetězce, objeví se v paměti vedle originálu tohoto řetězce ještě odpovídající počet jeho kopií.

Tento problém můžeme vyřešit poměrně jednoduchou úpravou. Rozdělme si odkazované řetězce na dvě skupiny: na klasické řetězce a na řetězce alokované metodami naší třídy v paměti. Instance může mezi těmito dvěma druhy řetězců rozlišovat podle hodnoty atributu *Delka*. Bude-li délka kladná, půjde o klasický řetězec, bude-li záporná, bude se jednat o řetězec alokovaný metodami této třídy.

Vzhledem k tomu, že řetězce alokované metodami této třídy nemusí přesně kopírovat strukturu klasických řetězců, můžeme společně alokovat místo pro vlastní textový řetězec i pro údaj o počtu jeho držitelů, a popřípadě i další údaje, které uznáme za vhodné.

Jediné, na co musíme při takto definovaných řetězcích myslet, je to, abychom „objektovému řetězci“ nepřiřadili klasický řetězec alokovaný na haldě, který bychom se později rozhodli smazat. Pokud však budeme používat klasické řetězce pouze jako tex-

tové konstanty a ve všech ostatních případech použijeme řetězce objektové, nemělo by nám toto nebezpečí hrozit.

```
/*  Příklad C6 – 3  */
typedef unsigned word;
class /*****/ cStrT /*****/
//Třída implementující řetězce sdílením
{
  public:
       cStrT() {Text = NULL; };
//Prázdný ukazatel symbolizuje to, že daný řetězec
// ještě nemá přiřazenu žádnou hodnotu
       cStrT( const char * );
       cStrT( const cStrT & );
       ~cStrT() {Odhlas(); };
       cStrT& operator= ( const char * );
       cStrT& operator= ( const cStrT& );
       cStrT& operator+= ( const char * );
       cStrT& operator+= ( const cStrT& );
  private:
       int Delka;
       union {              //Anonymní unie představuje
         char * Text;       //položku, která vystupuje pod
         word * Param;      //dvěma jmény a dvěma typy.
       };
       static const word Pocet; //Pomocné konstanty, které pojmenovávají
       static const word Bytu;  // posunutí položek obsahujících počet
                      //napojených instancí a počet vyhrazených bajtů.
       static const word Pridat; //Počet bajtů, o něž je velikost
                 // vyhrazované paměti větší než délka alokovaného textu.
       void Platny ( const char* = „", int=0 )const;
       void Napoj ( const char * );
       void Prihlas( const cStrT& );
       void Odhlas ();
       cStrT& Pridej ( const char*, int );
       friend ostream& operator<< ( ostream&, const cStrT& );
};
/********** cStrT **********/
const word cStrT::Pocet = -2;
const word cStrT::Bytu = -1;
const word cStrT::Pridat= 2*sizeof( word );
inline void /*****/ cStrT::Napoj /*****/
( const char * s )
//Napojení instance na klasický řetězec
{
  Delka = strlen( s );      //Kladná délka symbolizuje
  Text = (char *)s;         //klasický řetězec, který
}                           //je nedotknutelný.
/********** cStrT::Napoj **********/
void /*****/ cStrT::Prihlas /*****/
( const cStrT& S )
//Připojení instance k držitelům objektového řetězce
{
  Text = S.Text;
  Delka = S.Delka;
```

```
  if( Delka < 0 )          //Záporná délka symbolizuje
  Param[ Pocet ] ++;       //kombinovaný řetězec alokovaný
}                          //některou z ostatních metod.
/********** cStrT::Prihlas **********/
void /*****/ cStrT::Odhlas /*****/
()
//Instance se vzdává drženého řetězce
{
  if( Delka < 0 )          //Je námi alokovaný?
       if( --Param[ Pocet ] == 0 )          //Drží se ještě někdo?
        delete (char*)(Text-Pridat);        //Nikdo - smazat!
                       //Počátek alokované paměti je o Pridat
}                      //bajtů před počátkem vlastního textu.
/********** cStrT::Odhlas **********/
void /*****/ cStrT::Platny /*****/
( const char * s, int i )const
//Pomocná metoda pro usnadnění kontroly korektnosti operací.
//Není-li ukazatel s i ukazatel this->Text nenulový,
//vypíše zprávu a přeruší další běh programu.
//Je-li parametr i nenulový, netestuje se volající instance.
{
  if( s && (i || Text) ) return;
  cerr << "\n\n\7\7Použití řetězce bez hodnoty\7\7\n\n";
//Před vypsáníM zprávy a po něm vždy 2x pípne
  abort(); //Abnormální ukončení programu
}
/********** cStrT::Platny **********/
/*****/ cStrT::cStrT /*****/
( const char * s )
{
  Platny( s, 1 );     //Ukazatel nesmí být prázdný
  Napoj( s );
}
/********** cStrT::cStrT **********/
/*****/ cStrT::cStrT /*****/
( const cStrT& S )
 {
  S.Platny();         //Přiřazovaný řetězec musí mít hodnotu
  Prihlas( S );
}
/********** cStrT::cStrT **********/
cStrT& /*****/ cStrT::operator= /*****/
( const char * s )
{
  Platny( s, 1 );     //Ukazatel nesmí být prázdný
  Odhlas();           //Vzdej se drženého řetězce
  Napoj( s );         //a napoj se na přiřazovaný
  return *this;
}
/********** cStrT::operator= **********/
cStrT& /*****/ cStrT::operator= /*****/
( const cStrT& S )
{
  S.Platny();         //Přiřazovaný řetězec musí mít hodnotu
  Odhlas();           //Vzdej se drženého řetězce
```

```
  Prihlas( S );       //a napoj se na přiřazovaný
  return *this;
}
/********** cStrT::operator= **********/
cStrT& /*****/ cStrT::Pridej /*****/
( const char * s, int i )
//Pomocná metoda realizující společnou část operací
//pro spojení dvou řetězců operátory +=
{
  int D = abs(Delka);
  Platny( s );                  //Oba řetězce musí mít hodnotu
  if( *s )                      //Přidávám neprázdný řetězec
  {                             // - jinak není co řešit
  word B = D + i + 1;           //Počet bajtů pro vlastní text
  char* T = new char[ B + Pridat ];
  *((word*)T)++ = 1;            //Držím se jej zatím sám
  *((word*)T)++ = B;            //Počet bajtů pro vlastní text
  strcpy( T, Text );            //Přesuň text na nové místo
  strcpy( T+D, s );             //Přihraj přidávaný text
  Odhlas();                     //Vzdej se původního textu
                      //Odhlas musíme volat před změnou Delky!
  Delka = -(D + i);             //Nastav délku nového textu -
                      //je záporná, protože jsme jej sami alokovali
  Text = T;                     //Nastav novou adresu textu
 }
  return *this;
}
/********** cStrT::Pridej **********/

inline cStrT& /*****/ cStrT::operator+= /*****/
( const char * s )
{
  return Pridej( s, strlen( s ) );
}
/********** cStrT::operator+= **********/

inline cStrT& /*****/ cStrT::operator+= /*****/
( const cStrT& S )
{
  return Pridej( S.Text, abs(S.Delka) );
}
/********** cStrT::operator+= **********/

ostream& /*****/ operator << /*****/
( ostream& o, const cStrT& s )
{
  o << (void*) s.Text           //Adresa textu
    << ": " << s.Delka          //Délka textu
    << " »" << s.Text << "«\n";  //Vlastní text
  return o;
}
/********** operator << **********/

//Deklarujete-li proměnné jako externí, tj. mimo těla funkcí, bude je
//degugger vždy znát a vy se jej můžete kdykoliv zeptat na jejich
//hodnotu.
//Při ladění je nejlepší mít instance nejprve deklarovány uvnitř funkcí
```

```
//a odladit konstruktory. Pak je můžete vyjmout a odladit ostatní
//metody.
cStrT a = "Adam ";
cStrT b( "Božena " );
cStrT c = a;
cStrT d( b );
cStrT e;

void /*****/ Test_3 /*****/ ()
{
  cout << "\na =" << a << "b =" << b
       << "c =" << c << "d =" << d << "e =" << e;
  d = a;
  cout << "\nd2 =" << d;
  cout << "d3 =" << (d = "Bohoušek ");
  c += b;
  cout << "\nc2 =" << c;
  cout << "c3 =" << (c += "Cyril ");
  a = e;   //Nepovolená operace - měl by křičet
}
/********** Test_3 **********/
```

Podívejme se nyní na některé obraty, které by nemusely být zcela jasné.

Prvním obratem, vyžadujícím pravděpodobně podrobnější vysvětlení, je anonymní unie, deklarovaná v soukromé části definice třídy *cStrT*:

```
union {
  char * Text;
  word * Param;
};
```

Unie slouží k tomu, abychom mohli k určité oblasti paměti přistupovat několika způsoby. Anonymní unie nám pak umožňují vyhnout se kvalifikaci složek pomocí identifikátoru unie a používat identifikátory složek unie stejně jako identifikátory obyčejných proměnných. Jinými slovy, anonymní unie vlastně zavádí množinu proměnných, které spolu sdílejí stejné místo v paměti (přesněji – začínají na stejném místě, protože každá může být jinak dlouhá)[7].

V našem programu jsme takto definovali dvojici proměnných – ukazatel na vektor znaků a ukazatel na vektor slov – které leží na stejném místě v paměti. Tak vznikla vlastně proměnná, která má dva identifikátory. Prostřednictvím identifikátoru *Text* ji můžeme používat jako ukazatel na vektor znaků a prostřednictvím identifikátoru *Param* jako ukazatel na vektor dvoubajtových slov (přesněji celých čísel bez znaménka).

Tato konstrukce má jediný účel: umožnit, aby ve vyhrazené části paměti byla jak slova tak znaky a abychom nemuseli při práci s některými z nich používat operátor přetypování.

Při alokaci paměti pro uchovávaný řetězec program vyhrazoval o 5 bajtů více, než kolik měl ukládaný řetězec znaků. Bylo to proto, že spolu se znaky textového řetězce

[7] Doplníme ještě jedno pravidlo: anonymní unie, deklarovaná na úrovni souboru, musí být statická.

ukládal i závěrečný prázdný znak (1 bajt), délku řetězce (2 bajty) a počet jeho držitelů (2 bajty). V paměti pak byly tyto údaje uspořádány tak, že první bylo slovo obsahující počet držitelů řetězce, za ním následovalo slovo s počtem znaků v řetězci včetně závěrečného prázdného znaku a za ním pak vlastní textový řetězec zakončený prázdným znakem.

Ukazatel v naší unii byl nastaven tak, že ukazoval na počátek vlastního textového řetězce. Pokud jej budeme interpretovat jako ukazatel na vektor znaků, můžeme s ním pracovat stejně jako s klasickým textovým řetězcem jazyka C. Pokud jej interpretujeme jako ukazatel na vektor slov, tak víme, že ukazuje těsně za pomocné údaje a že tedy tyto údaje jsou jeho -1. a -2. položkou.

Aby byl program přehlednější a aby bylo možno někdy v budoucnu podobu této datové struktury změnit, aniž bychom pak museli procházet celý program a upravovat jej, zavedli jsme tři pomocné konstanty: *Pocet*, *Bytu* a *Pridat*. První dvě z nich obsahují indexy odpovídajících položek v poli *Param* a třetí pak počet bajtů, které se vyhrazují navíc oproti bajtům nutným k uložení céčkovského řetězce (tj. včetně závěrečného prázdného znaku).

Tyto konstanty jsme definovali jako statické, neboť jinak by pro ně překladač vyhrazoval místo v každé instanci. Takto nejen ušetříme paměť vyhrazovanou pro instance, ale navíc umožníme překladači nevyhrazovat pro tyto konstanty místo vůbec a pracovat s nimi jako s pojmenovanými literály (pokud to uzná za vhodné).

Dalším obratem, o kterém bychom se měli zmínit, je definice implicitní hodnoty prvního parametru soukromé metody *Platny*:

```
void Platny( const char* = "", int=0 ) const;
```

Překladač vytvoří někde v paměti literál "" jako jednobajtový vektor, obsahující znak s kódem 0. Implicitní hodnotou příslušného parametru je pak adresa tohoto „řetězce“. Pro vlastní algoritmus není důležité, jaká konkrétně tato hodnota je. Důležitá je pouze skutečnost, že tato hodnota musí být nenulová: **prázdný, tj. nulový ukazatel není totéž jako ukazatel na prázdný řetězec!**

Také následující kód z metody *Pridej* mohl méně pozorného čtenáře zmást:

```
char* T = new char[ B + Pridat ];
*((word*)T)++ = 1;
*((word*)T)++ = B;
```

Zde alokujeme vektor znaků dané délky. Poté je třeba na počátek alokované oblasti paměti uložit dvě celočíselné hodnoty. Přetypovali jsme tedy ukazatel *T* na ukazatel na vektor slov (**(***word****)***T*). Ten jsme dereferencovali a do daného místa jsme uložili potřebnou hodnotu. Tento přetypovaný ukazatel jsme také hned po použití inkrementovali. Vzhledem k tomu, že dereferencovaný a inkrementovaný ukazatel byl typu *word******, zvětšila se jeho hodnota (tj. adresa buňky, na kterou ukazuje) o délku objektu typu *word*.

V našem případě se odkazovaná buňka paměti posunula o 2 bajty, ale kdybychom tento program přeložili na jiném překladači, nebo dokonce na počítači jiného typu, mohla by to být zcela jiná velikost. (Přestože bereme starost o uspořádání paměti na sebe, snažíme se, aby byl program stále přenositelný.)

Použití přetypovaného ukazatele nebylo v metodě *Pridej* nezbytně nutné. Mohli jsme si stejně jako v definici třídy *cStrT* pomoci zavedením anonymní unie. Program by pak dostal tvar:

```
union {
   char * T;
   word * W;
};

T = new char[ B + Pridat ];
*W++ = 1;
*W++ = B;
 ...
Text = T;  //Nebo Param = W;
```

Programátorští puristé asi označí předchozí obraty za nečisté. Souhlasíme s nimi a uznáváme, že do školního kursu jejich výklad nepatří. Domníváme se však, že na rozdíl od školního kursu je tato kniha určena lidem, kteří se programováním chtějí živit, nebo se jím dokonce již živí. Uvedené obraty můžeme prohlásit za nečisté, avšak musíme jim přiznat, že jsou poměrně bezpečné a v praxi používané. A proto do naší knihy patří.

Podívejme se nyní, jak vypadá týž program v Pascalu:

```
(*  Příklad P6 – 2  *)
const POSUN = 2*SizeOf( Word );  {Počet bajtů, o něž je
velikost vyhrazované paměti větší než délka alokovaného textu. }
type
   echarray = array[-POSUN ..MaxInt] of Char;
   charray = array[ 0 ..MaxInt] of Char;
   warray = array[-POSUN div 2 ..0] of Word;
   uecha = ^echarray;
   ucha = ^charray;
   uwo = ^warray;
   cStrT = object
   {Třída implementující řetězce sdílením }
        Delka : Integer;
        Data : uecha;
        constructor InitKop( var S : cStrT );
        constructor InitStr( var s : String );
        constructor InitLit( s : String );
        destructor Done;

        procedure Prirad( var S : cStrT );
        procedure Pripoj( var S : cStrT );
        procedure PrirStr( var S : String );
        procedure PripStr( var S : String );
        procedure PrirLit( S : String );
        procedure PripLit( S : String );
        procedure Write;
        procedure WriteF( var F : text );
   private
        procedure Napoj ( var s : String );
        procedure Prihlas( var S : cStrT );
```

```
      procedure Odhlas;
      procedure PomPrip( u : ucha; i : Integer );
  end;
(********** cStrT **********)

procedure (*****) cStrT.Napoj (*****)
( var s : String );
{Kopie klasického řetězce }
var i : Integer;
begin
  i := -4;
  Delka := Length(s);
  Data := Pointer(@s[i]);
end;
(********** cStrT.Napoj **********)

procedure (*****) cStrT.Prihlas (*****)
( var S : cStrT );
{Připojení instance k držitelům objektového řetězce }
begin
  Data := S.Data;
  Delka := S.Delka;
  if( Delka<0 ) then             {Záporná délka symbolizuje }
  inc(uwo(Data)^[ -2 ]);         {kombinovaný řetězec alokovaný}
end;       {některou z ostatních metod }
(********** cStrT.Prihlas **********)

procedure (*****) cStrT.Odhlas (*****)
;
{Instance se vzdává drženého řetězce }
begin
  if( Delka<0 ) then       {Je námi alokovaný? }
  begin
       dec(uwo(Data)^[ -2 ]);
       if( uwo(Data)^[ -2 ]=0 ) then        {Drží se ještě někdo?}
        FreeMem(Data,-Delka+POSUN+1);       {Nikdo - smazat! }
                      {Počátek alokované paměti je o "POSUN" }
  end;                {bajtů před počátkem vlastního textu. }
end;
(********** cStrT.Odhlas **********)

constructor (*****) cStrT.InitStr (*****)
( var s : String );
begin
  Napoj( s );
end;
(********** cStrT.InitStr **********)

constructor (*****) cStrT.InitKop (*****)
( var S : cStrT );
begin
  Prihlas( S );
end;
(********** cStrT.cStrT **********)

constructor (*****) cStrT.InitLit (*****)
( s : String );
begin
```

```
  Delka := 0;                    {Aby se nic nedealokovalo }
  PomPrip( @s[1], Length(s) );
end;
(********** cStrT.Init **********)

destructor (*****) cStrT.Done (*****)
;
begin
  Odhlas;
end;
(********** cStrT.Done **********)

procedure (*****) cStrT.Prirad (*****)
( var S : cStrT );
begin
  Odhlas;                        {Vzdej se drženého řetězce }
  Prihlas( S );                  {a napoj se na přiřazovaný }
end;
(********** cStrT.Prirad **********)

procedure (*****) cStrT.PrirStr (*****)
( var s : String );
begin
  Odhlas;                        {Vzdej se drženého řetězce }
  Napoj( s );                    {a napoj se na přiřazovaný }
end;
(********** cStrT.PrirStr **********)

procedure (*****) cStrT.PrirLit (*****)
( s : String );
begin
  Odhlas;                        {Vzdej se drženého řetězce}
  Delka := 0;                    {- aby se nic nedealokovalo }
  PomPrip( @s[1], Length(s) );   {a napoj se na přiřazovaný }
end;
(********** cStrT.PrirLit **********)

procedure (*****) cStrT.PomPrip (*****)
( u : ucha; i : Integer );
{Pomocná metoda pro připojení textu k dynam. řetězci }
var T : uecha;
  D : Integer;
  B : Word;
begin
  D := abs(Delka);
  if( i <> 0 )then               {Přidávám neprázdný řetězec }
  begin                          {- jinak není co řešit }
       B := D + i + 1;           {Počet bajtů pro vlastní text}
       GetMem( T, B+POSUN );
       uwo(T)^[ -2 ] := 1;  {Držím se jej zatím sám }
       uwo(T)^[ -1 ] := B;  {Počet bajtů pro vlastní text}
       Move( Data^[1], T^[1], D );          {Přesuň text na nové místo}
       Move( u^[0], T^[D+1], i );           {Přihraj přidávaný text }
       Odhlas;                   {Vzdej se původního textu }
                          {Odhlas musíme volat před změnou Delky! }
       Delka := -(D + i);
       Data := Pointer( T );
```

```
      if( -Delka>255 )then      {Nastavíme délku pascalského}
        Data^[0] := Char(255)   {řetězce - zde neplatná, }
      else
       Data^[0] := Char(-Delka);           {- ale zde platná }
  end;
end;
(********** cStrDyn.PripStr **********)

procedure (*****) cStrT.Pripoj (*****)
( var S : cStrT );
begin
  PomPrip( @S.Data^[1], abs(S.Delka) );
end;
(********** cStrT.Pripoj **********)

procedure (*****) cStrT.PripStr (*****)
( var s : String );
begin
  PomPrip( @s[1], Length(s) );
end;
(********** cStrT.PripStr **********)

procedure (*****) cStrT.PripLit (*****)
( s : String );
begin
  PomPrip( @s[1], Length(s) );
end;
(********** cStrT.PripStr **********)

procedure (*****) cStrT.WriteF (*****)
( var F : text );
var i : Integer;
begin
  System.Write( F,          {Musíme označit o které WRITE nám jde}
      seg( Data ), ':', ofs( Data ),        {Adresa textu}
      ': ', Delka,              {Délka textu }
      ' »' );                   {Uvozovky před vlastním textem}
  for i:=1 to abs(Delka) do                 {Vlastní text}
       System.Write( F, Data^[i] );
  System.WriteLn( F, '«' );     {Uvozovky za vlastním textem }
end;
(********** cStrT.WriteF **********)

procedure (*****) cStrT.Write (*****)
;
begin
  WriteF( OutPut );
end;
(********** cStrT.Write **********)

{Deklarujete-li proměnné jako externí, tj. mimo těla funkcí, bude je
 degugger pořád znát a vy se jej můžete kdykoliv zeptat na jejich
 hodnotu.
 Při ladění je nejlepší mít instance nejprve deklarovány uvnitř funkcí a
 odladit konstruktory. Pak je můžete vyjmout a odladit ostatní metody.
}
var a,b,c,d,e : cStrT;
var s,u,v : String;
```

```
procedure (*****) Test_3 (*****)
;
begin
   s := 'Božena ';
   u := 'Evička ';
   v := 'Ferda ';
   a.InitLit('Adam ');
   b.InitStr( s );
   c.InitKop( a );
   d.InitKop( b );
   Write('a ='); a.Write;
   Write('b ='); b.Write;
   Write('c ='); c.Write;
   Write('d ='); d.Write;
   d.Prirad( a ); Write('d2 ='); d.Write;
   d.PrirLit( 'Bohoušek ' ); Write('d3 ='); d.Write;
   d.PrirStr( u ); Write('d4 ='); d.Write;
   c.Pripoj( b ); Write('c2 ='); c.Write;
   c.PripLit('Cyril '); Write('c3 ='); c.Write;
   c.PripStr( v ); Write('c4 ='); c.Write;
   d.Done;
   c.Done;
   b.Done;
   a.Done;
end;
(********** Test_3 **********)
```

3.2 Základní binární operátory

V této podkapitole se budeme zabývat přetěžováním základních binárních operátorů. O binárních operátorech, které se používají méně a které jsou nějakým způsobem specifické, si povíme později.

Za základní binární operátory budeme považovat operátory

```
+  -  *  /  %  >  <  >=  <= ==  !=
      &&  ||  |  &  ^  <<  >>
```

Stejně jako tomu bylo u operátorů přiřazení, i v tomto případě je identifikátor operátoru tvořen klíčovým slovem **operator**, za nímž následuje značka daného operátoru, která může být od slova **operator** oddělena libovolným počtem mezer. Většinou je ale zvykem psát identifikátor operátoru kompaktně, tedy například **operator+**, **operator<<** atd.

Pokud binární operátor definujeme jako řadovou funkci (tj. nedefinujeme jej jako metodu), musí mít dva parametry a alespoň jeden z nich musí být objektového nebo výčtového typu. U operátoru definovaného jako metoda je jeho levým argumentem instance, jejíž metodou operátor je (ta je samozřejmě objektového typu), takže v definici již deklarujeme pouze jeden parametr – pravý operand.

Funkční hodnota operátoru může být teoreticky libovolného typu. Pokud však budeme chtít používat operátor ve výrazech, měl by vracet nějakou hodnotu. Možná, že

někteří z vás namítnou, že výhodnější by mohlo být vracet referenci. Nikoli – a hned si povíme proč.

Představte si, že bychom si definovali třídu zlomků a chtěli pro ni rozšířit i definice aritmetických binárních operátorů. A protože se nám zdá, že předávat 8bajtovou hodnotu je neefektivní, rozhodneme se, že naše operátory budou vracet reference. No jo, ale reference na co?

Budete se mnou asi souhlasit v tom, že binární aritmetické operátory by neměly měnit hodnoty svých operandů – od toho jsou složené přiřazovací operátory typu += apod. Nemůžeme tedy uložit hodnotu výsledku do žádného z operandů a vrátit referenci na tento operand, protože tím bychom změnili jeho hodnotu. (Samozřejmě je možné definovat **operator+** tak, že mění hodnoty svých operandů, ale slušný člověk to neudělá, protože tím zmate nejen všechny ty, kteří po něm program čtou, ale brzy i sám sebe.)

Pro předání výsledku nemůžeme použít ani žádnou lokální proměnnou. Po opuštění těla operátoru se totiž všechny v něm definované automatické lokální proměnné automaticky destruují, a my bychom pak vraceli referenci na něco, co už vlastně neexistuje (navíc slušné překladače dodržování tohoto zákazu kontrolují).

Také použití lokální statické proměnné není nejlepší nápad. Vezměme výraz

```
(a + b) * (c + d)
```

Jak by jej překladač vyhodnotil? Nejprve by vyhodnotil podvýraz (*a* + *b*) a náš **operator+** by uložil výsledek do statické proměnné. Pak by vyhodnotil výraz (*c* + *d*) a náš **operator+** by uložil výsledek do **téže** statické proměnné. Je asi jasné, že jako součin bychom určitě obdrželi něco úplně jiného, než co bychom obdržet chtěli[8].

Poslední stéblo, kterého by se mohli zastánci předávání reference zachytit, je možnost alokace odpovídající proměnné v haldě a předání reference na tuto proměnnou. To vypadá jako akceptovatelné řešení až do chvíle, než si položíme otázku, kdo tuto proměnnou zruší, až nebude potřeba. Nikdo. A proto musíme předávat výsledek hodnotou.

Známe dvě cesty, jak tato omezení trochu obejít. První z nich je tvorba co nejmenších vlastních objektů, obsahujících ukazatele na své rozsáhlejší části umístěné na haldě (obdobně, jako je tomu u třídy *cStrT*) a druhou je využití metody alokace, které říkáme **bazén** (v literatuře se můžete setkat také s označením **aréna**) a ke které se později vrátíme.

Zkuste si nyní rozšířit program z minulého pokračování o funkce z následující ukázky a vyzkoušejte si, že binární operátory fungují opravdu tak, jak očekáváte.

```
/*  Příklad C6 – 4  */
class /*****/ cStrT /*****/
//Třída implementující řetězce sdílením
{
  public:
       cStrT() {Text = NULL; };
```

[8] Rafinovanější překladače tuto konkrétní situaci dokáží zvládnout, ale mohou selhat u komplikovanějších konstrukcí.

```
      cStrT( const char * );
      cStrT( const cStrT & );
      ~cStrT() {Odhlas(); };
      cStrT& operator= ( const char * );
      cStrT& operator= ( const cStrT& );
      cStrT& operator+= ( const char * );
      cStrT& operator+= ( const cStrT& );
      friend ostream& operator<< ( ostream&, const cStrT& );
      friend cStrT operator+( const cStrT&, const cStrT& );
//Následující dvě deklarace již nejsou nezbytně nutné -
// viz text za ukázkou
//friend cStrT operator+( const char*, const cStrT& );
//friend cStrT operator+( const cStrT&, const char* );
  private:
      in Delka;
      union {              //Anonymní unie představuje jednu
       char * Text;        //položku, která vystupuje pod
       word * Param;       //dvěma jmény a dvěma typy.
      };

      static const word Pocet;
      static const word Bytu;
      static const word Pridat;
      void Platny ( const char* = "", int=0 )const;
      void Napoj ( const char * );
      void Prihlas( const cStrT& );
      void Odhlas ();
      cStrT& Pridej ( const char*, int );
};
/********** cStrT **********/

inline cStrT /*****/ operator+ /*****/
( const cStrT& a, const cStrT& b )
{
  cStrT pom = a;           //pom = pomocná proměnná pro součet
  return pom += b;
}
/********** cStrT::operator+ **********/

cStrT A = "Adam";
cStrT E = "Eva";
cStrT Vyrok;

void /*****/ Test_1 /*****/ ()
{
  cout << "\nA =" << A << "E =" << E;
  Vyrok = A + " a " + E + " snědli jablko.";
  cout << "Výrok = " << Vyrok;
  cout << ("Hříšníci " + Vyrok + "!!!");
//Závorky jsou nutné, protože jinak vydá překladač
//varování: "Dvojznačný operátor vyžaduje závorky".
// cout << "Součet = " << ("Spojujeme " + "text.");
//Předchozí příkaz by neprošel, ale projde ve tvaru
  cout << "Součet = " << (cStrT("Spojujeme ") + "text.");
}
/********** Test_1 **********/
```

V předchozí ukázce jsme v komentáři v definici třídy uvedli, že operátory se smíšenými parametry není nezbytně nutné definovat. To proto, že jsme ve třídě definovali konverzní konstruktor s parametrem typu **char***. Překladač díky tomu umí zkonstruovat pomocný objekt, kterému přiřadí hodnotu předávaného řetězce a který pak předá operátoru jako skutečný parametr. O tom se můžete sami přesvědčit tím, že si nastavíte zarážku do zmíněného konverzního konstruktoru a přesvědčíte se, že je opravdu volán před voláním operátoru.

Odstraněním komentářových závorek a dodefinováním zbývajících dvou homonym operátoru sčítání tedy nepřidáváte programu žádné nové, dříve neexistující funkce, ale pouze program zkracujete a zrychlujete, protože odpadá nutnost konstruování pomocné proměnné.

3.3 Unární operátory !, ~, + a -

S přetěžováním těchto unárních operátorů byste ve svých programech neměli narazit na žádné problémy. Pamatujte pouze na to, že pokud chcete definovat homonyma těchto operátorů jako řadové funkce, musíte je definovat jako funkce s jedním parametrem a tento parametr musí být objektového nebo výčtového typu. Pokud je definujete jako metody, pak budou bez parametrů (operand bude instancí, pro kterou tuto metodu voláme).

Unární operátory se používají v nejrůznějších významech. Z vyjmenovaných bývá nejčastěji rozšiřován operátor **!**, který většinou testuje jakousi „nenulovost“ svého operandu (např. neprázdnost textového řetězce), ale setkali jsme se již např. s jeho použitím pro výpočet absolutní hodnoty komplexního čísla. S přetíženým operátorem bitové negace ~ jsme se setkali opět u jedné definice třídy komplexních čísel, kde sloužil jako operátor výpočtu čísla komplexně sdruženého. Unární + a – jsme zase použili ve třídě *cStrT*, v níž převáděly řetězec na velká, resp. malá písmena.

To byly jen velice stručné a náznakové příklady možných aplikací. Předpokládáme, že vás při tvorbě vlastních programů napadnou ještě mnohé další.

3.4 Operátory inkrementace a dekrementace

Ti, kdo ovládají alespoň pasivně C++ (a mezi ně jistě patří většina čtenářů této knihy), vědí, že standardní inkrementační a dekrementační operátory vystupují v tomto jazyce v párech: jeden je prefixový (tj. píšeme jej před operand) a druhý postfixový (píšeme jej za operand). Prefixový operátor provede žádanou operaci a vrátí novou hodnotu svého operandu, tj. jeho hodnotu po provedení operace. Postfixový operátor sice provede stejnou operaci jako prefixový kolega, ale na rozdíl od něj vrací hodnotu svého operandu před provedením požadované operace.

Poznámka:

Jak jistě víte, hovoříme-li o prefixovosti a postfixovosti operátorů, hovoříme pouze o tom, kam ve výrazu zapisujeme znak operátoru vůči jeho operandu. V zájmu stručnosti a přehlednosti však budeme v dalším textu používat termíny „prefixové" a „postfixové" chování, čímž budeme rozumět to, zda operátory vracejí hodnotu, odvozenou z původní hodnoty svého argumentu, nebo hodnotu odvozenou od jeho hodnoty konečné. Abychom však upozornili na přenesený význam těchto označení, budeme je psát v uvozovkách.

V C++ je v případě homonym operátoru ++ a -- situace trochu zamotaná. Možnosti přetěžování operátorů inkrementace a dekrementace se totiž v jednotlivých verzích překladačů liší. Definice jazyka do verze 2.0 (tedy např. ještě BC++ 2.0) nerozlišovaly mezi prefixovými a postfixovými verzemi homonym těchto operátorů. Umožňovaly definovat pro každý typ argumentu pouze jedno homonymum a toto homonymum pak bylo možno v programu zapisovat jak prefixově tak i postfixově. Činnost operátoru však způsob jeho použití nijak neovlivnila. Ať jste přetížený operátor použili prefixově nebo postfixově, udělal vždy totéž. To, zda se bude chovat podle svého prefixového nebo postfixového vzoru, jste mohli ovlivnit pouze tím, jak jste jej naprogramovali.

V následující ukázce je operátor ++ přetížen „prefixově" a operátor -- „postfixově". Čtenáři, kteří snad ještě používají starší verze překladačů, si na této ukázce mohou vyzkoušet, že se činnost operátorů při prefixovém a postfixovém použití neliší, a čtenáři, kteří používají překladače verze 3.0 a vyšší, si naopak mohou vyzkoušet, že uvedená homonyma překladač chápe jako prefixové operátory a že vás na jejich postfixové použití, které je podle něj chybné, upozorní varováním.

Společně si pak můžete ověřit, že se operátory nechovají podle toho, jak jsou použity, ale podle toho, jak jsou naprogramovány.

```
/*  Příklad C6 – 5  */
class IIDD
{
  public:
       IIDD( int I=0, int i=1 )
        {Index = I; Inkrement = i; }
       int operator ++();
       int operator --();
       friend ostream& operator<<( ostream& o, IIDD& x )
        {o << '[' << x.Index << "; " << x.Inkrement << ']';
         return o; }
  private:
       int Index;
       int Inkrement;
};

int /*****/ IIDD::operator ++ /*****/ ()
{                                 //"Prefixové" chování
  Index += Inkrement;             //Provede operaci
  return Index;                   //Vrací novou hodnotu
}
/********** IIDD::operator ++ **********/
int /*****/ IIDD::operator -- /*****/ ()
{                                 //"Postfixové" chování
  int i=Index;                    //Zapamatuje si původní hodnotu
```

```
  Index -= Inkrement;            //Provede operaci
  return i;                      //Vrátí původní hodnotu
}
/********** IIDD::operator ++ **********/

void /***** Test_1 /*****/ ()
{
  IIDD a;
  cout << "\n\n===== Test_1 =====\n";
  cout << "Počátek: " << a << endl;
  cout << "Preinkrement: " << ++a;
  cout << " - Výsledek: " << a << endl;
  cout << "Postinkrement: " << a++;
  cout << " - Výsledek: " << a << endl;
  cout << "Predekrement: " << --a;
  cout << " - Výsledek: " << a << endl;
  cout << "Postdekrement: " << a--;
  cout << " - Výsledek: " << a << endl;
 }
/********** Test_1 **********/
```

Jak jste si mohli všimnout v předchozí ukázce, přetížené operátory se od svých „standardních“ protějšků mohou odlišovat nejen některými detaily činnosti, ale také tím, že hodnota vrácená operátorem může být jiného typu než operand.

Jak jsme již řekli, novější verze překladačů respektují i novější normu jazyka, která umožňuje přetěžovat zvlášť prefixové a zvlášť postfixové verze výše zmíněných operátorů. Podle nové specifikace jazyka jsou definice z předchozí ukázky považovány za definice homonym prefixových verzí operátorů. Pokud chcete definovat postfixovou verzi operátoru, musíte přidat ještě jeden parametr typu **int**.

Chceme-li tedy podle nové definice jazyka přetížit prefixový operátor ++ nebo --, musíme jej definovat buď jako metodu bez parametrů, nebo jako řadovou funkci s jedním parametrem objektového nebo výčtového typu.

Chceme-li pak přetížit postfixový operátor ++ nebo --, musíme jej deklarovat buď jako metodu s jedním parametrem typu **int**, nebo jako řadovou funkci se dvěma parametry, z nichž první je objektového nebo výčtového typu a druhý je typu **int**.

Ani v nové verzi jazyka však nebude pro chování přetíženého operátoru rozhodující, zda jsme rozšířili prefixovou či postfixovou verzi operátoru, ale pouze a jedině to, jak jsme toto rozšíření naprogramovali.

V následující ukázce jsme definovali rozšíření obou verzí inkrementačního operátoru jako metody a obě verze dekrementačního operátoru jako řadové funkce. „Prefixové“ a „postfixové“ chování je naprogramováno tak, aby chování operátorů odpovídalo jejich fixaci.

```
/*  Příklad C6 – 6  */
class ID
{
  public:
       ID() {i = 1; d = 1.1; }
//Prefixová verze operátoru ++ definovaná jako metoda
```

```
       ID& operator++()
        {i++; return *this; }
//Postfixová verze operátoru ++ definovaná jako metoda
       ID operator ++ ( int )
        {ID id = *this; d+=0.1; return id; }
//Prefixová verze operátoru-definovaná jako funkce
       friend ID& operator --( ID& x )
        {x.i--; return x; }
//Postfixová verze operátoru-definovaná jako funkce
       friend ID operator -- ( ID& x, int )
        {ID id = x; x.d-=0.1; return id; }
//Operátor výstupu
       friend ostream& operator << ( ostream& o, ID& x )
        {o << "[ " << x.i << "; " << x.d << " ]";
       return o; }
  private:
       int i;
       double d;
};

void /*****/ Test_2 /*****/ ()
{
  ID a;
  cout << "\n\n===== Test_2 =====\n";
  cout << "Počátek: " << a << endl;
  cout << "Preinkrement: " << ++a;
  cout << " - Výsledek: " << a << endl;
  cout << "Postinkrement: " << a++;
  cout << " - Výsledek: " << a << endl;
  cout << "Predekrement: " << --a;
  cout << " - Výsledek: " << a << endl;
  cout << "Postdekrement: " << a--;
  cout << " - Výsledek: " << a << endl;
}
/********** Test_2 **********/
```

Rozeberme si tuto ukázku podrobněji.

Především bychom chtěli připomenout, že v seznamu formálních parametrů nemusí být identifikátor parametru bezpodmínečně uveden. Takovýto **anonymní parametr** pak sice není možno v definici použít, ale pokud je v seznamu parametrů pouze „do počtu“ (jako např. celočíselný parametr v postfixových operátorech ++ a --), nebude nám to vadit. Tímto způsobem sdělíme překladači, že tento parametr nepotřebujeme, a překladač si na oplátku odpustí varování, že jsme jej nepoužili.

Dalším trikem, na který bychom chtěli upozornit, je postup, jak donutíme operátor, aby se choval „postfixově“, tj. aby vracel výsledek odvozený z hodnoty parametru před aplikací operátoru. Tuto původní hodnotu si musíme nejprve někde odložit, pak provést vlastní činnost operátoru a nakonec odloženou hodnotu vrátit.

Důležité je, že „prefixovost“ či „postfixovost“ chování přetížených operátorů je plně v našich rukou a že překladač nás v tomto ohledu nezastoupí. Naprogramujeme-li postfixovou verzi operátoru „prefixově“, bude se i „prefixově“ chovat (a naopak).

V souvislosti s „prefixovým“ a „postfixovým“ chováním homonym musíme myslet i na typ vracené hodnoty. U „prefixově“ se chovajících operátorů je v podstatě jedno, zda budeme vracet hodnotu, referenci či ukazatel. Pokud operátor přebírá parametr odkazem, ví, že daný parametr existoval již před voláním operátoru, a může proto vrátit odkaz na něj či na nějakou hodnotu z něj odvozenou – např. jeho složku. Nelze však vracet odkaz ani ukazatel na parametr předávaný hodnotou, protože bychom předali adresu pomocné proměnné, která při opuštění operátoru zanikne!

„Postfixově“ se chovající operátory nemají na vybranou a svůj výsledek musí vrátit hodnotou. Nevracejí totiž hodnotu objektu, který existoval již před jejich voláním, ale hodnotu pomocné proměnné, do níž si potřebné údaje odložily a která při návratu do volajícího programu zanikne. Při pokusu o vracení odkazu bychom se pak dostali do stejných potíží, o kterých jsme si vyprávěli v souvislosti s binárními operátory.

Poslední věc, o níž bychom se chtěli zmínit, s našimi operátory souvisí pouze nepřímo. Občas se setkáme s úvahami na téma, zda by nebylo možno v našich testovacích programech (obvykle se jmenují asi tak *Test_x*) tisknout všechny výsledky jedním příkazem.

Typickým příkladem může být následující úsek programu:

```
int i = 0;
cout << ++i << ", " << i++ << ", " << i;
```

Pokud něco podobného napíšete, může se stát, že počítač místo očekávaného

```
1, 1, 2
```

vytiskne

```
2, 0, 0
```

Co se stalo? Jistě si vzpomínáte, že jazyk C++ při vyhodnocování výrazů nepředepisuje pořadí, ve kterém budou vyhodnoceny operandy jednotlivých operátorů[9]. V našem případě si tedy překladač nejprve připravil hodnotu *i*, pak si připravil hodnotu podvýrazu *i*++ a nakonec si připravil hodnotu podvýrazu ++*i*. Tyto hodnoty pak v požadovaném pořadí vytiskl. Překladač tedy vygeneroval kód, který odpovídá programu:

```
int i = 0;
int a = i;
int b = i++;
inc c = ++i;
cout << c << ", " << b << ", " << a;
```

V Pascalu nemá smysl uvažovat o prefixovosti či postfixovosti operátorů *Inc* a *Dec*, protože to jsou obyčejné procedury, které žádnou hodnotu nevracejí. Pokud však budete definovat podobné podprogramy sami, nic vám nebrání je definovat jako funkce. Pak se ovšem objeví otázka, co vlastně budou tyto funkce vracet.

[9] Připomeňme si, že výjimkou jsou operátory &&, ||, :? a operátor , (čárka).

Víme, že v Pascalu nemohou funkce vracet hodnoty objektových typů, ale pouze ukazatele na ně. Při tom vyvstávají stejné problémy, o jakých jsme mluvili při rozboru binárních operátorů vracejících referenci. Funkce může vracet ukazatel na svůj parametr, avšak to má smysl pouze u funkcí s „prefixovým“ chováním. Z důvodů rozebíraných v kapitole o binárních operátorech však není dost dobře možné vracet ukazatel na lokální proměnnou (a to jak statickou, tak automatickou) ani na dynamickou proměnnou zřízenou na haldě.

Někdy však může být výhodné, aby funkce vracela hodnotu některého ze standardních typů. Pak samozřejmě začne být otázka „prefixového“ či „postfixového“ chování operátoru na místě. V případě definice „postfixového“ chování budete mít dokonce oproti svým „plusovým“ kolegům drobnou výhodu v tom, že můžete opravdu explicitně předat návratovou hodnotu na počátku funkce, a pak provést vlastní požadovanou akci. Tím se program samozřejmě zpřehlední a zprůzračí.

Pokud bychom převedli do Pascalu předchozí příklad, získali bychom takovýto program:

```
(*  Příklad P6 – 3 *)
type
  IIDD = object
       Index : integer;
       Inkrement : integer;
       constructor Init;
       function Inc : integer;
       function Dec : integer;
       procedure Write;
  end;

constructor (*****) IIDD.Init; (******)
begin
  Index := 0;
  Inkrement := 1;
end;
(********** IIDD.Init **********)

function (*****) IIDD.Inc (*****) : integer;
begin                                        {"Prefixové" chování}
  System.Inc( Index, Inkrement );            {Provede operaci }
  Inc := Index;       {Vrátí hodnotu }
end;
(********** IIDD.Inc **********)

function (*****) IIDD.Dec; (*****)
begin                                        {"Postfixové" chování }
  Dec := Index;                              {Připraví vracenou hodnotu}
  System.Dec( Index, Inkrement );            {provede operaci }
end;
(********** IIDD.Dec **********)

procedure (*****) IIDD.Write; (******)
begin
  System.Write( '[', Index, ', ', Inkrement, ']' );
end;
(********** Write **********)
```

```
procedure (*****) Test; (******)
var a : IIDD;
begin
  a.Init;
  writeln( #10#13#10#13'===== Test_1 =====' );
  write( 'Počátek: ' ); a.write; writeln;
  write( 'Preinkrement: ' ); write( a.Inc );
  write( ' - Výsledek: ' ); a.write; writeln;
  write( 'Postdekrement: ' ); write( a.Dec );
  write( ' - Výsledek: ' ); a.write; writeln;
end;
(********** Test **********)
```

3.5 Operátor indexování [ ]

V této podkapitole budeme postupovat trochu jinak než dosud. Nejprve si ukážeme použití nového operátoru a teprve pak si o něm povíme něco bližšího.

Jednou z nejčastějších pascalských výhrad proti Céčku (a řadě dalších jazyků) byla jeho neschopnost kontrolovat překročení mezí při práci s poli. Jazyk C nabízel sice možnost definovat makro, které by se podle hodnoty nějaké preprocesorové proměnné rozvíjelo buď do volání funkce, kterou bychom používali místo klasického indexovacího operátoru a která by tyto meze kontrolovala, nebo do podoby klasického indexovacího operátoru. Například takto:

```
/*  Příklad C6 – 7  */
#define LADIM 1             //Po odladění programu zaměním 1 za 0
#if LADIM                   //+++++ Větev aktivovaná ve fázi ladění
# define ix( Pole, Index ) *INDEX( &Pole, Index )
#else                       //===== Větev aktivovaná po odladění
# define ix( Pole, Index ) Pole.p[ index ]
#endif

#define PRVKU 2
/*****/ struct sPole /*****/
{
  int n;
  int p[PRVKU];
};
/********** struct sPole **********/

 //Následující makro nám umožňuje bezpečně definovat strukturu Pole
 //typu sPole, která obsahuje pole o PRVKU prvcích. Při jeho používání
 //máme zaručeno, že se při definici přiřadí prvé složce struktury hod-
 //nota rovná počtu prvků pole. Vlastní pole zůstává neinicializováno
 #define sPole( Pole ) struct sPole Pole; Pole.n=PRVKU;

int* /*****/ INDEX /*****/
( struct sPole *P, int i )
{
  assert( ((i >= 0) && (i < P->n)) );
  return &P->p[ i ];
```

```
}
/********** INDEX **********/

int /*****/ Test_61 /*****/
()
{
  int i, s;
  DsPole( P );              //Definujeme pole P o PRVKU prvcích
  for( s=0, i=0; i <= P.n; i++ )
//Cyklus je definován tak, aby při posledním průchodu
//byly meze pole překročeny.
  {
       ix(P, i) = i;
       s += ix( P, i );
  }
  return s;
}
/********** Test_61 **********/
```

Předchozí ukázku jsme naprogramovali tak, aby byla pokud možno opravdu bezpečná, tj. aby byla maximálně omezena možnost chyb, zaviněných nějakým opomenutím – např. nepřiřazení velikosti pole prvé složce struktury. Z uvedené ukázky je ale jistě zřejmé, proč se programátoři takovýmto konstrukcím vyhýbají. Za možnost efektivní a přitom bezpečné kontroly překročení mezí platí sníženou přehledností programu – např. v naší ukázce jsme místo definice volali „funkci“ (expandovali jsme makro, ale lexikální podoba je shodná s voláním funkce) a stejně tak jsme „funkci“ volali místo indexování. Jednu příčinu chyb jsme se snažili odstranit a přitom jsme připravili živnou půdu pro několik jiných. Je proto jasné, že se programátoři o podobné konstrukce nepokoušejí a spoléhjí spíše na to, že ve svých programech meze nepřekročí.

C++ přináší prostředky, kterými je možno problémy obdobného typu řešit poměrně elegantně. Pokud bychom měli předešlou ukázku naprogramovat s využitím prostředků jazyka C++, zvolili bychom asi následující řešení:

```
/* Příklad C6 - 8  */
#define LADIM 1  //Po odladění programu zaměním 1 za 0
/*****/ class cPole /*****/
{
  public:
       cPole( int Mez )          //Konstruktor vytvoří vlastní pole
       : n( Mez ),               //na haldě
       p( new int[ Mez ] ) {};
       ~cPole()
       {delete [] p; }
       int& operator[]( int );   //Homonymum operátoru indexace
       int Prvku()               //Dotaz na počet prvků pole
       {return n; }
  private:
       int const n;              //Počet prvků pole
       int *const p;             //Ukazatel na počátek tohoto pole
};
/********** class cPole **********/
```

```
#if LADIM  // Větev aktivovaná ve fázi ladění
int& /*****/ cPole::operator [] /*****/
( int i )
{
  assert( (i >= 0) && (i < n) );
  return p[ i ];
}
/********** cPole::operator [] **********/
#else                              // Větev aktivovaná po odladění
inline int& /*****/ cPole::operator [] /*****/
( int i )
{
  return p[ i ];
}
/********** cPole::operator [] **********/
#endif                        // Konec sekce podmíněných překladů

int /*****/ Test_62 /*****/
()
{
  cPole P( 2 );
  for( int s=0, i=0; i <= P.Prvku(); i++ )
//Cyklus je definován tak, aby při posledním průchodu
//byly meze pole překročeny.
  {
       P[ i ] = i;
       s += P[ i ];
  }
  return s;
 }
/********** Test_62 **********/
```

Možná, že se některým z vás bude zdát objektová podoba našeho příkladu složitější, protože jsme navíc museli definovat konstruktor a destruktor nově definovaného objektového datového typu *cPole*. Ano, vlastní definice objektového typu trochu složitější je, ale ve chvíli, kdy ji jednou dokončíme a odladíme, tak na ni a na její „složitost“ můžeme zapomenout a využíváme pak již pouze jejích výhod.

Při přetěžování operátoru indexování nesmíme zapomenout, že tento operátor smíme přetížit pouze jako nestatickou metodu objektového typu s jedním parametrem.

Možnost přetěžovat operátor indexování před námi otevírá netušené perspektivy. Jen namátkou nastíníme některé z nich:

- ✧ Možnost indexovat strukturovanými datovými typy – např. textovými řetězci nebo zlomky.
- ✧ Možnost definovat pole, která se dynamicky zvětšují tak, aby se do nich vešly všechny požadované položky. Pokud bychom tedy chtěli do pole uložit položku, jejíž index odkazuje mimo aktuální meze pole, pole se automaticky zvětší tak, aby se do něj vešla i právě indexovaná položka.
- ✧ Možnost definovat rozsáhlá pole, která se nevejdou do paměti. Operátor indexace si bude udržovat přehled o tom, které části pole jsou v dané chvíli v paměti a které jsou

právě odložené na disku, a podle potřeby by tyto části operativně přesouvat mezi pamětí a diskem.

✧ Možnost elegantní a efektivní práce s řídkými maticemi a vícerozměrnými poli, která jsou sice teoreticky velice rozsáhlá, avšak mají převážnou většinu prvků nulových. Z těchto polí bývají v paměti uloženy pouze nenulové prvky a práce s nimi je proto zákonitě komplikovanější. Vhodně definovaný operátor indexování nám umožní pracovat s takovýmito řídkými poli stejně jako s poli „hustými“.

Pascal sice neumožňuje přetěžovat operátory tak elegantně jako C++, ale jinak jsou jeho možnosti vcelku obdobné. Jedinou nevýhodou (vynecháme-li samozřejmě dříve zmiňované problémy s konstruktory, destruktory, přístupovými právy apod.) je, že v Pascalu nemůžeme definovat skutečná homonyma, ale pouze metody, vykonávající obdobné funkce, a že v něm proto není možné používat vžitý způsob zápisu.

Naprogramujme nyní v obou jazycích úlohu vyhledání vzdálenosti mezi dvěma městy. Začneme tím, že si rozmyslíme, jak bychom mohli tuto úlohu vyřešit, a teprve pak si ukážeme jeden návrh řešení.

Nejprve bychom si měli definovat, co všechno má náš program umět. Vzhledem k tomu, že se bude jednat o „mikroprográmek“, který by se měl vejít do jedné podkapitoly, nesmíme toho po něm chtít příliš.

Dohodněme se, že se nás program nejprve zeptá, ve kterém městě se právě nacházíme, a pak se nás bude ptát na názvy měst, jejichž vzdálenost od našeho momentálního působiště nás zajímá. Abychom uživateli práci s programem co nejvíce zjednodušili, umožníme mu nezadávat celé názvy měst, ale pouze jejich počátky. Aby se zadávání názvů měst ještě více zjednodušilo, měli bychom uživatelům umožnit, aby zadávané názvy mohly začínat malými písmeny a nemusely obsahovat žádné háčky a čárky. To by však řešení zbytečně zkomplikovalo – kdo chce, může se o takovou úpravu pokusit sám.

Abychom udělali náš program trochu univerzálnější, nebudou jména měst ani vzdálenosti mezi nimi součástí zdrojového textu programu, ale budou se načítat z diskových souborů. Jakákoliv změna v tabulce (přejmenování města, dokončení dálnice mezi dvěma městy), stejně jako jakékoli rozšíření tabulky, bude znamenat pouze modifikaci odpovídajícího souboru a nezpůsobí žádný zásah do programu.

Pro začátek si vezmeme jen krátký soubor: budeme vyhledávat pouze vzdálenosti mezi Prahou, Plzní, Ústím nad Labem, Hradcem Králové, Ostravou, Brnem a Českými Budějovicemi.

Ve druhé etapě bychom si měli navrhnout základní datové struktury, na kterých pak bude postavena celá následující koncepce řešení. Zde se nám nabízí nepřeberné množství variant. Z těch, které vzhledem k naší dosavadní úrovni znalostí přicházejí v úvahu, vybíráme jednu, která možná bude některým z vás připadat trochu exotická, ale to je jenom první zdání.

Pokud vám budou připadat některé navrhované obraty a konstrukce neefektivní, může to být ze dvou důvodů: buď se tímto způsobem snažíme připravit půdu pro některá chystaná rozšíření, nebo nás prostě efektivnější řešení nenapadlo. Než je však zcela zavrhnete, počkejte si chvíli na kapitolu o dynamických datových strukturách, kde se

k tomuto příkladu vrátíme a kdy vám ukážeme, proč jsme některé věci řešili právě takhle.

Předkládané řešení jsme vybrali mimo jiné proto, abyste viděli, že se není vždy nutno úzkostlivě držet klasických řešení, a abychom vás „postrčili“ k tomu, že ve svých projektech dokážete v případě potřeby navrhnout obdobná „méně klasická“ či zcela neklasická řešení sami. Klasickým řešením by asi v tomto případě bylo pole struktur, obsahujících text s názvem města (příp. ukazatel na něj) a řádek s hodnotami jednotlivých vzdáleností. Takto navržená datová struktura však neumožňuje ušetřit paměť za položky, které se opakují (cesta z Prahy do Brna je stejně dlouhá, jako cesta z Brna do Prahy).

Podle našeho návrhu si definujeme třídu *cTabulka*, jejímiž atributy budou pole řetězců a matice s hodnotami kilometrových vzdáleností mezi jednotlivými městy. V této matici bude *r*-tém řádku a *s*-tém sloupci uložena vzdálenost z města *r* do města *s*. Tato třída bude mít jediný konstruktor, jehož parametrem bude název souboru se jmény měst v tabulce a s jejich vzájemnými vzdálenostmi.

Protože vzdálenost z města *r* do města *s* je stejná jako vzdálenost z města *r* do města *s* (možnost různě dlouhých jednosměrných objížděk velkoryse pomineme), bude mít v této matici prvek *[r,s]* stejnou hodnotu jako prvek *[s,r]*. Matice, které mají tyto vlastnosti, nazýváme **symetrické**. Jejich symetrie můžeme využít k tomu, abychom je v paměti uložili úsporně: každou hodnotou pouze jednou. Navíc hodnoty položek *[r,r]* nemusíme ukládat vůbec, protože o nich předem víme, že jsou nulové.

Jak si asi pamatujete z různých atlasů, tabulky vzdáleností různých míst se většinou nekreslí čtvercové, ale trojúhelníkové. Stejně to uděláme i my. Z naší původní čtvercové tabulky si zapamatujeme jen její levou dolní nebo pravou horní polovinu, a tuto trojúhelníkovou tabulku si uložíme do paměti do pole pěkně jednu položku za druhou.

V tuto chvíli však před námi vyvstane problém, jak v tomto vektoru najít hodnotu položku s indexy *[r,s]*. Sami si asi lehce odvodíte, že pro položky z levého dolního rohu tabulky musí platit, že jejich řádkový index je vždy větší než sloupcový, tzn. že platí $r > s$, a že pro položky z pravého horního rohu tabulky musí naopak platit, že jejich řádkový index je vždy menší než sloupcový, tzn. že platí $r < s$.

Odtud si jistě dokážete odvodit i to, že pokud si označíme písmenem *N* je počet řádků, resp. sloupců naší původní čtvercové tabulky, pak při uložení levé dolní poloviny bude hodnota položky v našem vektoru uložena v položce s indexem

```
r * (r-1) / 2 + s
```

a při uložení pravého horního rohu bude hodnota položky *[r,s]* uložena v položce s indexem

```
((2*N - r - 1) * r) / 2 + (s - r - 1)
```

Pokud požadovaná nerovnost mezi *r* a *s* neplatí, tj. pokud hledáme položky z opačného rohu původní čtvercové tabulky, pak v odpovídajícím vzorečku pouze zaměníme hodnoty *r* a *s*. Pokud se náhodou stane, že $r = s$, pak v tabulce nic hledat nebudeme, protože víme, že hledaná vzdálenost je nulová.

Než se pustíme do dalších problémů, měli bychom si ještě ujasnit, kolik místa vlastně budeme pro uložení takové symetrické matice o *N* řádcích a sloupcích potřebovat.

Sami si můžete ověřit, že pokud neukládáme diagonálu (a to my neukládáme), potřebujeme uložit

```
N * (N - 1) / 2
```

položek.

S uložením hodnot jsme se tedy vypořádali. Problémem ovšem zůstává, jak se k těmto hodnotám elegantně dostat.

Programátoři v Pascalu mají rozhodování jednodušší. Pascal jim moc volnosti neposkytuje, takže nejvhodnějším řešením bude definice metody – funkce se dvěma parametry, která vrátí požadovanou hodnotu.

Programátoři v C++ mohou pascalské řešení implementovat také, ale připravili by se tak o možnost zachování způsobu zápisu, na který jsou zvyklí. Pokud bychom však chtěli v C++ použít indexových operátorů, vyvstane před námi jeden problém: C++ umí indexovat pouze v polích (vektorech). Pascalskou maticovou indexaci typu *[i,j]* nezná (dá se sice dodefinovat, ale o tom až někdy jindy) a používá místo ní dvojitou vektorovou indexaci, tj. indexaci typu *[i][j]*.

Problém můžeme obejít tak, že indexový operátor, aplikovaný na objekt typu *cTabulka*, bude vracet hodnotu nějakého pomocného typu. Pokud bude tento pomocný typ objektový, můžeme pro něj opět definovat indexový operátor, a ten bude vracet požadovanou hodnotu z tabulky. (Je snad jasné, že indexový operátor ve třídě *cTabulka* musí vracet opravdu hodnotu tohoto pomocného typu, a nikoli ukazatel nebo odkaz na něj.)

Definujeme si tedy třídu *cMisto*, kterou však použijeme v našem programu i jinak.

Jako nejvhodnější řešení se nám jeví definice funkční metody *Vzdalenost*, jejímiž parametry budou názvy měst a jejichž vzdálenost pak funkce vrátí.

Protože pascalisté nemusí přetěžovat indexovací operátor, mohli by třídu *cMísto* ignorovat jako nepotřebnou. Doporučujeme jim však, aby tak nečinili, protože objekty této třídy bychom rádi v budoucnu použili pro některá rozšíření naší úlohy.

Třída *cMísto* bude obsahovat dva atributy: adresu tabulky, jejíž hodnoty nás zajímají, a index daného města v tabulce. Kromě bezparametrického konstruktoru bude mít ještě dva dvouparametrické konstruktory. Oba budou mít první parametr typu *cTabulka* a budou se lišit typem parametru druhého. Jeden bude očekávat celé číslo, druhý textový řetězec s názvem města (v C++ objekt typu *cStrT*).

Pro obě třídy přetížíme operátor indexování. Prvním parametrem (tj. parametrem, uzavíraným do indexových závorek) bude objekt typu *cStrT*, tj. objektově „zabalený“ textový řetězec, označující město, na které se ptáme. Operátor indexování pro třídu *cTabulka* bude vracet hodnotu typu *cMisto* a operátor pro třídu *cMisto* bude vracet celé číslo, udávající počet kilometrů z daného místa do místa označeného indexem.

Při definici těchto operátorů indexování bychom měli myslet ještě na jednu věc: standardní operátor indexování v jazyku C++ vrací l-hodnotu, což znamená, že jej můžeme použít na levé straně přiřazovacího příkazu (viz ukázkový program C6 – 8 z této podkapitoly). Naše operátory by však tuto možnost poskytovat neměly. Operátor třídy *cTabulka* musí vytvořit nový objekt a nemůže proto vracet ani ukazatel, ani referenci,

a operátor třídy *cMesto* vrací kilometrové vzdálenosti mezi městy, u nichž nemáme zájem na tom, aby někdo jejich hodnoty v tabulce měnil (to bychom museli mít trochu jiné zadání).

Pascalisty poprosíme, aby se pokusili navrhnout ekvivalenty indexových operátorů jazyka C++. Ve třídě *cTabulka* to bude muset být procedura se dvěma parametry: prvním parametrem bude odkaz na objekt typu *cMesto*, jehož hodnota se nastaví v závislosti na hodnotě druhého parametru, kterým bude řetězec. Ve třídě *cMesto* to pak bude funkce, která očekává řetězec a vrací celé číslo, jež je vzdáleností mezi městem, odpovídajícím volající instanci, a městem, na něž odkazuje řetězec v parametru.

A nyní vám doporučujeme, abyste se pokusili předloženou úlohu alespoň v nástinu vyřešit. Pokud se vám bude zdát, že některé závěry z předchozích odstavců vedou k málo efektivnímu řešení, nebo že vedou k řešení, na které si netroufáte, neostýchejte se je změnit tak, aby vám vyhovovaly.

3.6 Operátor volání funkce ( )

Možnost přetěžování operátoru volání funkce je specialitou jazyka C++, ke které v Pascalu nemá smysl hledat nějaký ekvivalent. Tento operátor nám umožňuje používat instanci daného objektového typu jako proceduru nebo funkci.

Operátory volání funkce mají oproti ostatním operátorům jednu zvláštnost: nemají žádnou přesně definovanou aritu (počet operandů), takže si jejich homonyma můžeme zcela přizpůsobit svým potřebám.

V následující ukázce definujeme třídu symetrických matic *cSMat*. V ní přetížíme operátor funkčního volání s dvěma celočíselnými parametry. Tento operátor bude zastupovat maticový operátor indexování, který nám chyběl např. v příkladu na vyhledávání vzdálenosti měst.

```
/*  Příklad C6 - 9  */
/*****/ class cSMat /*****/
{
  public:
       cSMat( int n );
       ~cSMat()
       {delete [] P; }
       int& operator() ( int i, int j );
  //private:                      //Abychom nemuseli pro test
                                  //definovat zvláštní metody
       int N;
       int NN;
       int* P;
};
/********** class cSMat **********/

/*****/ cSMat::cSMat /*****/
( int n )
: N( n ),
```

```
NN( n*(n+1)/2 ),
P( new int [n] )
{
  int i, j;
  for( i=0; i < N; i++)
  for( j=0; j < N; j++ )
       (*this)(i,j) = i*10 + j;
}
/********** cSMat::operator() **********/

int& /*****/ cSMat::operator() /*****/
( int i, int j )
{
  assert( (i>=0) && (j>=0) && (i<N) && (j<N) );
  if( i < j )
       {int p=i; i=j; j=p; }
  return( this->P[(i * (i+1)) / 2 + j] );
}
/********** cSMat::operator() **********/
```

```
void /*****/ Test_71 /*****/
()
{
  cSMat M( 4 );
  int i;
  for( i=0; i < 4; i++ )
       M(i,i) += 100*M(i,i);
  cout << "\n\nVektor: ";
  for( i=0; i < M.NN; i++ )
       cout << M.P[i] << ", ";
}
/********** Test_71 **********/
```

3.7 Operátory přetypování

Přetypovávání bývá v praktických programech poměrně běžná operace. Velice často se totiž vyskytnou situace, v nichž bychom potřebovali, aby nějaký objekt vstupoval do operací, které jsou vyhrazeny pro objekty jiných typů.

V C++ je přetypování vždy spojeno s konverzí přetypovávané hodnoty. Z klasického jazyka C byla převzata syntax, podle níž se cílový typ zapisuje do závorek před přetypovávaný výraz, a navíc byla pro jednoslovně označené typy povolena i syntax známá z Turbo Pascalu. (Pozor na priority! V případě nutnosti je třeba použít závorek – např. *(char*)(s+3)*.) Syntaktická definice této operace je:

Přetypování:
(*Cílový_typ* **)** *Přetypovávaný_výraz*
Ident_cíl_typu **(** *Přetypovávaný_výraz* **)**

Roli identifikátoru typu mohou hrát i klíčová slova označující standardní typy, např. **long**, **double** apod. Při používání „pascalského“ způsobu zápisu musíme ale víceslovné

názvy (v tomto případě je slovem i hvězdička) cílových typů „zjednoslovnit“ použitím **typedef** – např.:

```
typedef char * uchar;
 //...
s2 = uchar(s1) + 5;
```

Jazyk C++ umožňuje definovat jako metodu objekového typu funkci, která bude provádět přetypování. Jméno této metody je tvořeno klíčovým slovem **operator**, za kterým uvedeme identifikátor cílového typu. Operátor přetypování nemá parametry a v jeho deklaraci neuvádíme typ vracené hodnoty, neboť ten je určen již jménem této funkce. Například

```
operator int();
```

je operátorová funkce, která bude přetypovávat na **int**.

V názvu operátoru přetypování můžeme použít hvězdičky nebo ampersandu – např.

```
operator void *();
```

je správně zapsaný prototyp operátoru přetypování na **void*.**

Podívejme se nyní na příklad:

```
/*  Příklad C6 – 10 | */
class cZlomek
{
  public:
       cZlomek( long C=0, long J=1 )
        {Cit=C; Jm=J; }
       operator double()          //Operátor přetypování na double
        {return (double(Cit) / Jm); }
       operator cStrT();          //Operátor přetypování na cStrT
       cZlomek operator*( cZlomek& Z )      //Operátor násobení
        {return cZlomek( Cit*Z.Cit, Jm*Z.Jm ); }
  private:
       long Cit, Jm;
};

/*****/ cZlomek::operator cStrT /*****/
()
{
  cStrT r = "";                  // 1 Pomocná proměnná
  char b[ 20 ];                  // 2 Paměť pro generaci řetězce
  ostrstream os( b, 20 );        // 3 Proud, který nám poskytne
  os << Cit << "/" << Jm << ends; // 4 formátovaný výstup
  r += b;                        // 5 Alokace vlastní paměti
  return r;                      // 6 Vrácení řetězce
}
/********** cZlomek::operator cStrT **********/

void /*****/ Test_1 /*****/ ()
{
  cZlomek Z2( 1, 2 );                                  // 1
  cZlomek Z3 = 3;                                      // 2
  double Pul = Z2;                                     // 3
```

```
   double Pi = cZlomek( 22, 7 );                          // 4
   cStrT sZ = cStrT( Z2 ) + " * " + Z3;                   // 5
   Pul = Pi * Z2 * Z3;                                    // 6
   cout << "\n\nPi * " << sZ << " = " << Pul;             // 7
   cout << "\nPi * (" << sZ << ") = " << Pi * (Z2 * Z3) << '\n';
 }
/********** Test_1 **********/
```

Předchozí příklad bude asi potřebovat několik vysvětlení.

Možná že při čtení tohoto příkladu některé z vás napadlo, proč v programu na řádku, označeném v komentáři číslem 1, definujeme pomocnou proměnnou *r*, když bychom se bez ní mohli klidně obejít. Příkaz na 5. řádku bychom přece mohli vynechat a příkaz z 6. řádku upravit na tvar

```
return b;
```

při němž by se vzhledem k typu výstupní hodnoty automaticky vyvolal konstruktor *cStrT::cStrT(const char*)*. To je sice pravda, ale protože jsme si zatím definovali třídu *cStrT* trochu nešikovně, neobdrželi bychom to, co bychom chtěli.

Pokud se podíváte zpět na definici tohoto konstruktoru, zjistíte, že jím vytvořený objekt používá původní řetězec. To je ale v našem případě naprosto špatné, protože tento řetězec je lokální automatickou proměnnou operátoru a po návratu z těla funkce zanikne. Vracený objekt by tedy odkazoval na již neexistující řetězec, a to by mohlo vést k fatálním chybám.

Abychom se této nepříjemné vlastnosti třídy *cStrT* vyhnuli, definovali jsme pomocnou proměnnou *r*, která na počátku ukazuje na prázdný řetězec. (Lepší by sice bylo upravit definici třídy, ale to si necháme na později.) V pátém řádku se pak k tomuto prázdnému řetězci přidá řetězec vytvořený ve znakovém poli *b*, což vede k automatické alokaci výsledného řetězce na haldě.

Zde jsme využili drobného opomenutí při konstrukci operátoru +=. V této definici jsme totiž sice vzali v úvahu možnost přičítání prázdného řetězce, ale zapomněli jsme na možnost přičítání k prázdnému řetězci.

Když se však na celou definici podíváte, bude vám zřejmé, že vzhledem k nedokonalé definici třídy *cStrT* bylo třeba ještě několika daších oklik. Ke všem se ještě časem vrátíme a pokusíme se z nich vyvodit i patřičná doporučení pro úpravu třídy *cStrT*.

Opusťme nyní třídu *cStrT* a vraťme se znovu k naší ukázce. V závěrečném testovacím podprogramu jsme v prvých dvou řádcích definovali dva zlomky – na tom nic pozoruhodného není. Pokud budete krokovat (F7) třetí řádek, můžete se přesvědčit, že se zde automaticky volá námi definovaný operátor *Zlomek::double()* stejně jako na řádku čtvrtém, kde jsme si však konvertovaný zlomek nejprve připravili explicitním voláním odpovídajícího konstruktoru.

Na pátém řádku nalezneme zajímavý součet tří objektů: prvý objekt je typu *cStrT*, druhý je typu **char*** a třetí typu *cZlomek*. Na první pohled by se možná mohlo zdát, že dané tři objekty sečíst nelze – jak nám vysvětlovali ve škole: „Není možné sčítat hrušky s jablky.“ Jak však za chvíli uvidíte, tentokrát to jde, a to proto, že jsme překladači ukázali cestu, jak z hrušek ta jablka udělat. Rozeberme si tento příkaz podrobněji:

Víme, že se operátory sčítání vyhodnocují zleva doprava. Nejprve se tedy provede levý součet a jeho výsledek pak bude levým operandem pravého součtu.

Prvým sčítancem našeho „dvojsoučtu“ je zlomek převedený explicitně na *cStrT*, druhým sčítancem je klasický textový řetězec. Překladač zjistí, že kdyby druhý argument převedl konstruktorem *cStrT(const char*)* na objekt typu *cStrT*, mohl by použít *cStrT::operator+(const cStrT)*. Tento operátor vrátí hodnotu typu *cStrT*, která se stává levým argumentem druhého součtu.

Při analýze druhého součtu překladač zjistí, že kdyby použil konverzní operátor *cZlomek::operator cStrT()*, mohl by použít stejný sčítací operátor jako v prvém součtu. Učiní tak, a výsledek je na světě.

Podívejme se nyní na šestý řádek. Situace na něm je velice podobná řádku předchozímu: zlomek *Z2*, který je pravým operandem levého součinu, se převede na **double** a tato dvě čísla se spolu vynásobí. Stejný postup se opakuje i při vyhodnocení pravého součinu.

Takto vypadá situace náramně jednoduše. Zkusme však pořadí činitelů otočit a vyhodnocovat součin

```
Z2 * Z3 * Pi
```

Pokud výraz na šestém řádku upravíte takto, oznámí vám překladač, že si s ním neví rady. Proč? Levý součin je pro něj triviální záležitost – operátor pro součin dvou zlomků je definován. Problém ovšem nastane ve chvíli, když překladač zjistí, že má vynásobit zlomek reálným číslem. (Než budete číst dále, zkuste se zamyslet, proč.) Tento součin je totiž možno vyhodnotit dvěma způsoby, a překladač se nedokáže sám rozhodnout, který z nich má použít. Při svém rozhodování vyžaduje vaši radu.

Prvou možností je převést levý operand (zlomek) na **double** a vynásobit dvě čísla. Pokud bychom chtěli, aby překladač vyhodnotil výraz tímto způsobem, musíme výraz přepsat do tvaru

```
double(Z2 * Z3) * Pi
```

Druhou možností je převést pravý operand na zlomek a vynásobit pak dva zlomky. Pokud bychom preferovali tento způsob řešení, museli bychom výraz přepsat do tvaru

```
Z2 * Z3 * cZlomek( Pi )
```

Jak si sami domyslíte, výsledky budou v obou případech odlišné.

Poslední věcí, u níž bychom se chtěli zastavit, je součin v 8. řádku. Všimněte si, jak uzávorkováním tohoto součinu ovlivníme výběr operátorů, které jsou pro jeho vyhodnocení použity. V našem příkladu sice dávají obě verze stejné výsledky, ale obecně by tomu tak být nemuselo.

Možná, že se nyní někteří z vás zděsili nad složitostí pravidel, kterými se překladač řídí. Pokud však budete jazyk používat delší dobu, zjistíte, že pravidla, kterými se překladač řídí, složitá vůbec nejsou – dokonce si troufáme tvrdit, že jsou ve většině případů dokonce průzračná. Programátor má však oproti počítači tu nevýhodu, že při tvorbě programu se mu neustále daří na některá z těchto průzračných pravidel zapomínat. Když pak po několikadenním ladění objeví příčinu chyby, ze které již několikrát obvinil ope-

rační systém, překladač a často i své spolupracovníky („Vy jste se mi v tom určitě hrabali!“), bývá mu často natolik stydno, že by šel domů nejraději kanálem.

Vraťme se však k vyhodnocování výrazů. Doporučujeme vám, abyste si připomněli zásady, které jsme se učili ve fyzice na základní škole, a u heterogenních výrazů si vždy udělali „rozměrovou zkoušku“. Tu uděláte tak, že si výraz pomocí závorek „rozbijete“ na jednotlivé operace a nad těmito operacemi provedete rozbor obdobný tomu, který jsme dělali v předchozích odstavcích.

Připomeňme si, že Pascal zná dva druhy přetypovávání: přetypování proměnných a přetypování výrazů, přičemž tyto dva druhy přetypování syntakticky nerozlišuje a snaží se vždy sám domyslet, který druh přetypování měl v danou chvíli programátor na mysli.

Při přetypování proměnných Pascal požaduje, aby přetypovávaná proměnná měla stejnou velikost (tj. aby zabírala stejný počet bajtů v paměti), jako instance typu, na který ji chceme přetypovat. Tímto přetypováním tedy pouze změním interpretaci dané oblasti paměti. Při přetypování výrazů překladač požaduje, aby jak zdrojový, tak cílový typ byly pořadové (ordinální). Přetypování pak spočívá v konverzi výsledků.

Použití operátorů přetypování je jednoduché – volají se stejně jako funkce:

Přetypování:

Ident_cíl_typu (Přetypovávaný_výraz)

I při definování homonym přetypovávacích operátorů narazíme na to, že se budou homonyma používat trochu jinak než jejich vzory, ale vzhledem k unaritě operátorů přetypování to nebude vadit.

Definici a příklad použití operátoru přetypování si ukážeme na témže příkladu se zlomky, který jsme použili v pasáži věnované jazyku C++.

```
(*  Příklad P6 – 4  *)
type
(*****) cZlomek (******) = object
  Cit, Jm : longint;
  constructor Init( C:integer; J:integer );           {Konstruktor }
  function Real:Real;                        {Přetypování na real }
  function Strg:String;                      {Přetypování na string}
  procedure Krat( Z1:cZlomek; Z2:cZlomek ); {Operátor násobení }
end;
(********** cZlomek **********)

constructor (*****) cZlomek.Init (******)
( C:integer; J:integer);
begin
  Cit := C;
  Jm := J;
end;
(********** cZlomek.Init **********)

function (*****) cZlomek.real (*****)
:real;
begin
  real := Cit / Jm;
```

```
end;
(********** cZlomek.real **********)

function (*****) cZlomek.Strg (*****)
:String;
var
   s1, s2 : String;
begin
   Str( Cit, s1 );
   Str( Jm, s2 );
   Strg := s1 + '/' + s2;
end;
(********** cZlomek.Strg **********)
procedure (*****) cZlomek.Krat (******)
( Z1:cZlomek; Z2:cZlomek );
begin
   Cit := Z1.Cit * Z2.Cit;
   Jm := Z1.Jm * Z2.Jm;
end;
(********** cZlomek.Krat **********)

procedure (*****) Test_1 (******)
 ;
var
   Z2, Z3, Zpom : cZlomek;
   Pi, Pul : real;
   sZ : String;
begin
   Z2.Init( 1, 2 );
   Z3.Init( 3, 1 );
   Pul := Z2.real;
   Zpom.Init( 22, 7 );
   Pi := Zpom.real;
   sZ := Z2.Strg + ' * ' + Z3.Strg;
   Pul := Pi * Z2.real * Z3.real;
   writeln( #13#10#13#10'Pi * ', sZ, ' =', Pul:9:6 );
   Zpom.Krat( Z2, Z3 );
   writeln( 'Pi * (', sZ, ') =', Pi*Zpom.real:9:6 );
end;
(********** Test_1 **********)
```

Programátoři v Pascalu mohli v tomto příkladu dávat mnohem méně pozor než programátoři v C++. Tento jazyk totiž nabízí velké možnosti, ale v mnoha situacích (viz naše ukázky) vyžaduje od programátora také mnohem větší pozornost a kázeň. S pomocí C++ je sice možno nadefinovat efektivnější a čitelnější programy, ale nabízí také mnohem více možností, jak vytvořit tyto definice nedokonale nebo dokonce špatně – viz naše definice třídy *cStrT*, ke které se budeme ještě vracet a společně ji zdokonalovat.

Vraťme se ale ještě k operátorům přetypování. Podobu jejich definic má programátor ve svých rukou, a proto je může naprogramovat tak, že se chování těchto operátorů nemusí přísně řídit pravidly platnými pro standardní přetypovací operátory.

Někdy je výhodné definovat homonyma, která opravdu převádějí hodnoty svých parametrů na hodnoty jiného typu (viz převod zlomku na jeho reálnou hodnotu a vlastně i jeho převod na textový řetězec v předchozích ukázkách), ale jindy je pro nás výhod-

nější, když hodnotou cílového typu pouze nějak charakterizujeme hodnotu typu zdrojového. V takovémto případě bychom si však měli rozmyslet, zda opravdu použijeme operátor přetypování (resp. v Pascalu metodu, která jej připomíná), nebo zda by nebylo lepší použít místo toho nějakou obyčejnou metodu.

Programátorům v C++ bychom chtěli ještě připomenout, že překladač musí mít stále jednoznačnou volbu. Jakmile si překladač může vybrat ze dvou možností, je zákonitě bezradný, protože se nedokáže rozhodnout, kterou z možných alternativ by programátor na daném místě rád viděl. (Ponechme stranou případ, kdy je skutečný význam napsané konstrukce pro programátora překvapením.) O tomto problému jsme již hovořili v souislosti s vyhodnocováním výrazu

```
Z2 * Z3 * Pi
```

Takovéto potíže způsobíme překladači i v případě, že si bude muset vybrat mezi konverzním konstruktorem a operátorem přetypování.

Představme si, že máme objekt *Hodnota*, který je typu *THodnoty*. Kromě toho máme v programu objektový typ *OTyp*. Podívejme se, jaké možnosti bude překladač zvažovat, objeví-li se v programu výraz

```
OTyp( Hodnota )
```

resp.

```
( OTyp ) Hodnota
```

Jestliže *THodnoty* není objektový typ, způsobí uvedený příkaz volání konverzního konstruktoru třídy **OTyp** s parametrem typu *THodnoty* (popřípadě s odkazem na tento typ). Pokud takový konstruktor neexistuje, nastane chyba.

Stejná situace nastane i tehdy, je-li *THodnoty* objektový typ, ale není v něm definován operátor přetypování na *OTyp*.

Je-li *THodnoty* objektový typ, ve kterém jsme definovali operátor přetypování na *OTyp*, a pokud přitom třída *OTyp* nemá konverzní konstruktor z typu *THodnoty*, bude volán operátor přetypování *THodnoty::operator OTyp()*.

Ve všech předchozích situacích byla volba jednoznačná. Problémy nastanou až ve chvíli, když je pro třídu *OTyp* definován konverzní konstruktor *OTyp::OTyp(Thodnoty)*, resp. *OTyp::OTyp(THodnoty&)* a když je ve třídě *THodnoty* zároveň definován operátor přetypování[10] *THodnoty::operator OTyp()*.

[10] Pozor: Překladače Turbo C++ 1.0 a Borland C++ 2.0 v této situaci reagují chybně – tváří se, že ve třídě *THodnoty* žádný operátor přetypování na *OTyp* není, a použijí konverzní konstruktor. Překladače verze 3.0 a vyšší však tuto nejednoznačnost odhalí a vydají patřičné chybové hlášení.

3.8 Vzdálenost dvou měst

V této podkapitole se vrátíme k úloze o vzdálenosti dvou měst, kterou jsme spolu začali rozebírat těsně před výkladem o operátoru volání funkce. Tam jsme navrhli řešení založené na třídách *cTabulka* a *cMísto*.

Možná, že vám připadá takovéto řešení zbytečně komplikované. Než se proto pustíme do jeho kódování, ukažme si nějaké opravdu jednoduché řešení naší úlohy, v němž se nebudeme snažit o možnost dalšího zobecnění a které bude opravdu řešit pouze naši úlohu (v zájmu maximální stručnosti nebudeme dokonce ani kontrolovat hodnoty vstupů):

```
/*  Příklad C6 - 11  */
const int MEST = 7;
char* Mesto[] =  {"Brno", "České_Budějovice", "Hradec_Králové",
                  "Ostrava", "Plzeň", "Praha", "Ústí_nad_Labem" };
int Dalka[] =    {186,
                  142, 217,
                  165, 346, 240,
                  296, 133, 206, 456,
                  202, 140, 112, 362, 94,
                  294, 232, 166, 454, 146, 92 };
void /*****/ Vzdalenost /*****/ ()
{
  while( 1 )
  {
       int i, j;
       cout << "\n\n\n";
       for( i=1; i <= MEST; i++ )
       cout << setw(3) << i << " - " << Mesto[i-1] << endl;
       cout << "\nZadej čísla měst, jejichž vzdálenost hledáš"
               " (0 = konec): ";
       cin >> i;
       if( !i ) break;
       cin >> j;
       if( i > j )
        {int k=i; i=j; j=k; }
       cout << "\nVzdálenost mezi městy " << Mesto[--i]
            << " a " << Mesto[--j] << " je ";
//Výstup vzdálenosti je v samostatném příkazu, abychom měli zaručeno,
// že se při výpočtu indexu použijí dekrementované proměnné
       cout << Dalka[i*(i-1)/2 + j] << " km\n";
  }
}
/********** Vzdalenost **********/
```

Abychom měli v obou jazycích stejnou výchozí pozici a aby pascalisté nemuseli luštit programy v C++, ukážeme si týž program také v Pascalu.

```
(*  Příklad P6 - 5  *)
const MEST = 7;
  DALEK = MEST*(MEST-1) div 2;
```

```
  nl = #10#13;
  Mesto : array[ 1..MEST ] of String[ 16 ] =
       ( 'Brno', 'České Budějovice', 'Hradec Králové',
         'Ostrava', 'Plzeň', 'Praha', 'Ústí_nad_Labem' );
  Dalka : array[ 1..DALEK ] of integer =
       ( 186, 142, 217,
         165, 346, 240, 296, 133, 206, 456,
         202, 140, 112, 362, 94, 294, 232, 166, 454, 146, 92 );

procedure (*****) Vzdalenost (*****)
;
var
  i, j, k : integer;
begin
  repeat
   writeln( nl, nl, nl );
   for i:=1 to MEST do
   writeln( i:3, ' - ', Mesto[i] );
   write( nl, 'Zadej čísla měst, jejichž vzdálenost hledáš ',
          '(0 = konec): ' );
   read( i );
   if( i <> 0 )then
   begin
        read( j );
        if( i > j )then
          begin
            k:=i; i:=j; j:=k;
          end;
      write( nl, 'Vzdálenost mezi městy ', Mesto[i],
             ' a ', Mesto[j], ' je ' );
      writeln( Dalka[(i-2)*(i-1) div 2 + j], ' km' );
   end;
  until( i = 0 );
end;
(********** Vzdalenost **********)
```

Takto bychom asi naši úlohu řešili v případě, že bychom potřebovali mít rychle k dispozici hotový program a nehodlali se pokoušet o jeho další zobecnění. Naše předchozí úvahy však vedly ke komplikovanějšímu řešení, neboť jsme si jím naopak chtěli otevřít bránu k řadě dalších rozšíření. Nyní se k němu vrátíme.

V předminulé podkapitole jsme si udělali základní rozvahu. Určili jsme si koncepční východiska a pokusili se o návrh základních datových struktur. Teď se pustíme do konkrétní realizace. Sami uvidíte, kolikrát budeme muset naše původní předběžné úvahy modifikovat. (Pro zkrácení navíc vynecháme z našich úvah metody, jejichž parametrem je řetězec s názvem města – ty si zkuste navrhnout sami a pak si je můžete porovnat s řešeními na doprovodné disketě.)

Nejprve se podívejme na deklaraci třídy *cTabulka*. Minule jsme si říkali, že by tato třída měla mít dva atributy: vektor názvů měst a vektor (pole) jejich vzdáleností. Zapomněli jsme však na jeden důležitý atribut, kterým je počet měst v tabulce. Pokud by měl program pracovat s jednou konkrétní množinou měst, mohli bychom v deklaraci třídy hned uvést i konkrétní rozměry, a počet měst v tabulce bychom si pamatovat nemuseli.

Avšak v takovém případě bychom ani nemuseli zavádět nějaké rafinované objektové typy, ale vystačili bychom s nějakým ekvivalentem úvodního prográmku. My však chceme položit dostatečně tvárné základy pro případná budoucí rozšíření, a proto musíme naše datové struktury konstruovat poněkud obecněji.

Pokud tedy budeme chtít definovat naši třídu obecněji, nesmíme ji vázat na žádný konkrétní počet měst, ale musíme v ní použít místo polí pouze ukazatele na tato pole, a pole alokovat v paměti až ve chvíli, kdy se dozvíme, jak budou velká. V případě názvů měst půjdeme dokonce ještě o jednu úroveň hlouběji. Chceme-li totiž uložit řetězce efektivně, nemůžeme je ukládat přímo do vektoru, protože pak bychom nemohli vyhradit pro každý řetězec pouze tolik paměti, kolik bude nezbytně potřebovat.

Zvolíme proto takové řešení, kdy zřídíme vektor, v němž budeme uchovávat pouze ukazatele na počátky jednotlivých řetězců, které budeme postupně zřizovat na haldě. Odpovídající atribut třídy *cTabulka* si pak definujeme jako ukazatel na toto pole ukazatelů na vlastní textové řetězce.

Tolik k atributům – podívejme se nyní na metody. Především budeme potřebovat konstruktor (jak jsme si řekli, bude jeho parametrem jméno souboru, z nějž celou tabulku načte) a homonymum operátoru výstupu, abychom dokázali naši tabulku také vytisknout.

Po ostatních metodách třídy *cTabulka* jsme nepožadovali nic více, než aby nám umožnily zjistit vzdálenost mezi dvěma městy. Pascalistům jsme se snažili namluvit, že pro přímé zjišťování vzdálenosti bude nejvhodnějším řešením definice funkční metody *Vzdalenost*, jejímiž parametry budou názvy měst, a jež bude vracet jejich vzdálenost. Pokud jste o tom přemýšleli, asi jste dospěli k závěru, že to nejlepší řešení není.

Při návrhu základní koncepce celého programu jsme si na začátku řekli, že uživateli ulehčíme ovládání tím, že jej nebudeme nutit zadávat celý název města, ale že programu bude stačit pouze několik počátečních písmen. Součástí definice by tedy měla být metoda (nazvěme si ji *Čti*), která čte vstup znak za znakem a okamžitě nabízí název města k odsouhlasení.

Pokud by metoda *Čti* vracela řetězec s plným názvem města, který bychom pak předávali funkci *Vzdálenost*, musela by funkce *Vzdálenost* znovu hledat v tabulce, o které město se jedná. Proto bude výhodnější, když *Čti* nebude vracet řetězec, ale index nalezeného města v tabulce (tj. objekt typu *cMísto*). Parametry metody *Vzdálenost* proto nebudou řetězce, ale celočíselné indexy. (Na takovéto věci obvykle přijdeme až ve chvíli, kdy začneme programovat.)

Obdobně se zachováme i v C++, kde budeme pro přímé zjišťování vzdáleností používat operátor funkčního volání se dvěma parametry typu **int**.

Abychom naše možnosti získání indexu do tabulky rozšířili, přetížíme ve třídě *cTabulka* indexový operátor (v Pascalu definujeme odpovídající metodu), který bude přijímat celočíselný argument a vracet objekt typu *cMísto*.

Nyní by se mohlo zdát, že již můžeme deklaraci třídy *cTabulka* zapsat. Tak tomu je, i není. Zapsat ji samozřejmě můžeme, ale odladit se nám ji nepodaří. Problém je v tom, že v ní používáme parametry typu *cMisto*, který jsme ještě nedeklarovali. V C++ sice můžeme tento problém vyřešit tak, že před definici třídy *cTabulka* umístíme předběžnou

deklaraci třídy *cMísto*, avšak Pascal se nám nijak „ukecat“ nepodaří. Abychom si tedy mohli předvést ukázky programů ve tvaru, který lze opsat a rovnou přeložit, musíme do nich včlenit i deklaraci třídy *cMisto*. Povězme si proto ještě o požadavcích na tuto třídu.

Minule jsme si o třídě *cMísto* řekli, že by měla mít dva atributy: adresu tabulky a index daného města v tabulce. Instance této třídy tak zprostředkují kompletní informace o daném městě. Mohli bychom je proto klidně považovat za takové zobecněné ukazatele.

Nyní bychom si měli ujasnit, jakými metodami třídu vybavíme do života. Určitě bude muset mít nějaký konstruktor, ale v průběhu programování jsme dospěli k závěru, že minule navrhované tři konstruktory jsou zbytečný přepych. Neparametrický konstruktor by nám sice dovolil definovat ekvivalent prázdného ukazatele, ale zároveň by nám definici třídy zkomplikoval – museli bychom testovat ve všech metodách inicializaci zpracovávané instance. Zůstaneme tedy u dvou konstruktorů dvouparametrických, přičemž naprogramování konstruktoru s řetězcovým parametrem si zkuste sami.

Problém s tím, že metody třídy *cTabulka* používají parametry (a vracejí hodnoty) typu *cMísto*, a naopak metody třídy *cMísto* používají parametry typu *cTabulka*, vyřešíme v C++ tak, že třídu *cMísto* deklarujeme nejprve pouze předběžně. Tím poskytneme překladači dostatek informací pro to, abychom se pak na ni mohli v definici třídy *cTabulka* odvolávat. Při následné definici třídy *cMísto* pak již překladač třídu *cTabulka* zná, a problémy proto odpadají.

```
/* Příklad C6 - 12 */
class cMisto; //Předběžná deklarace
/*****/ class cTabulka /*****/
{
  public:
       cTabulka( const char* Soubor );
       cMisto Cti();
       int operator() ( int Z, int Do ) const;
       cMisto operator[] ( int Misto ) const;
       friend ostream& operator<< ( ostream&, const cTabulka& );
  private:
       int Mest;                 //Počet měst v tabulce
       char** Jmeno;             //Ukazatel na vektor ukazatelů na jména
       int* Dalka;               //Ukazatel na vektor vzdáleností
       friend class cMisto;
};
/********** class cTabulka **********/

/*****/ class cMisto /*****/
{
  public:
       cMisto( const cTabulka* Tab, int Ind )
        : Tabulka( Tab ), Index( Ind ) {};
       int operator[] ( cMisto& Misto ) const;
  private:
       cTabulka const * Tabulka;
       int Index;
};
```

```
/********** class cMisto **********/
```

V této ukázce bychom vás chtěli ještě upozornit na skutečnost, že třída *cTabulka* deklaruje třídu *cMísto* jako přítele. Poznamenejme, že my jsme si nutnost tohoto přátelství uvědomili až ve chvíli, kdy jsme definovali operátor *cMisto::operator[](cMisto& Místo)*.

V Pascalu se z toho, že metody třídy *cTabulka* používají parametry typu *cMísto*, a naopak metody třídy *cMísto* používají parametry typu *cTabulka*, bez „ztráty květinky“ nevylžeme. Ať nadefinujeme kteroukoliv třídu jako první, vždy se objeví problém s tím, že používáme parametry třídy, která ještě není definovaná.

Máme dvě možnosti: buď definovat parametry oné nedefinované třídy jako netypové, nebo je nahradit ukazateli. (Jak víte, Pascal dovoluje definovat a používat i ukazatele na objekty dosud nedefinovaného typu.) V našem případě máme štěstí, neboť se nám budou ve třídě *cMísto* ukazatelové parametry docela hodit (v obecném případě to však často vede ke snížení přehlednosti programu).

```
(*  Příklad P6 - 6  *)
const
  MAX_PTR = MaxInt div sizeof(Pointer);
  MAX_INT = MaxInt div sizeof(integer);
type                              {Pomocné datové typy}
  uTabulka = ^cTabulka;
  uMisto = ^cMisto;
  uString = ^String;
  auString = array[ 0..MAX_PTR ] of uString;
  uauStr = ^auString;
  aInt = array[ 0..MAX_INT ] of Integer;
  uaInt = ^aInt;
  uChar = ^Char;

 (*****) cMisto = object (*****)
  Tabulka : uTabulka;
  IndexC : Integer;
  constructor InitInt( T:uTabulka; Misto:Integer );
  function Index( var M:cMisto ) : Integer;
end;
(********** cMisto = object **********)

 (*****) cTabulka = object (*****)
  Mest : Integer;          {Počet měst v tabulce }
  Jmeno : uauStr;          {Ukazatel na vektor ukazatelů na jména }
  Dalka : uaINt;           {Ukazatel na vektor vzdáleností }
  constructor Init( Soubor : String );
  procedure Cti( var Ret: cMisto );
  function Vzdalenost( m1, m2 : Integer ) : Integer;
  procedure IIndex( Misto:Integer; var Ret:cMisto );
  procedure Write;
  procedure WriteF( var F : text );
end;
(********** cTabulka = object **********)
```

Deklaraci tříd máme hotovu, nyní přejdeme k definicím jejich jednotlivých metod. Začneme přirozeně konstruktory.

Konstruktor objektů třídy *cMísto* je natolik jednoduchý, že jsme jej definovali přímo v deklaraci třídy. Trošku složitější to však bude s konstruktorem třídy *cTabulka*. Minule jsme si řekli, že v zájmu maximální tvárnosti našeho programu bude mít konstruktor třídy *cTabulka* jediný parametr, a tímto parametrem bude jméno souboru, v němž budou uložena data, na jejichž podkladě konstruktor danou instanci vytvoří.

Dejme tomu, že v aktuálním adresáři vytvoříme textový soubor s názvem KRAJE.TAB a do něj zapíšeme potřebné údaje v následujícím formátu:

```
(*  KRAJE.TAB  *)
7
Brno
186 České_Budějovice
142 217 Hradec_Králové
165 346 240 Ostrava
296 133 206 456 Plzeň
202 140 112 362 94 Praha
294 232 166 454 146 92 Ústí_nad_Labem
```

Nejprve je uveden počet měst zařazovaných do tabulky a za ním následují záznamy věnované jednotlivým městům. V každém záznamu jsou nejprve zapsány vzdálenosti daného města ke všem městům, které před ním v tabulce předcházejí, a poté název města. Aby bylo zpracování jednodušší, jsou mezery ve víceslovných názvech měst nahrazeny podtržítky.

Náš konstruktor tedy bude muset nejprve tento soubor otevřít, po zjištění počtu měst vyhradit patřičnou paměť a načíst patřičná data.

```
/*  Příklad C6 - 13  */
/*****/ cTabulka::cTabulka /*****/
( const char * Soubor )
{
  ifstream F( Soubor );          //Otevření inicializačního souboru
  char B[ 80 ];                  //Paměť pro předběžné načtení jména
  F >> Mest;                     //Načtení počtu měst
  Jmeno = new char* [ Mest ];    //Vektor ukazatelů na názvy měst
  Dalka = new int [ Mest*(Mest-1)/2 ];      //Vektor se vzdálenostmi
  int k = 0;                     //Index položky vzdáleností
  for( int i=0; i < Mest; i++ )  //Pro všechna města
  {
       for( int j=0; j < i; j++ ) //Načtení vzdálenosti k předchozím
  městům
       F >> Dalka[ k++ ];
       F >> B;                   //Předběžné načtení jména města
       int L = strlen( B );
       Jmeno[ i ] = new char[ L+1 ];        //Alokace paměti pro název
  města
       strcpy( Jmeno[ i ], B ); //Přenesení textu do vyhrazené paměti
  }//for
}
/********** cTabulka::cTabulka **********/
```

Pascalská definice je o trošku delší, protože standardní procedura *read* načítá řetězce i s úvodními bílými znaky, které proto musíme nejprve odstranit.

```
(*  Příklad P6 – 7  *)
constructor (*****) cTabulka.Init (*****)
( Soubor : String );
var
  F : Text;           {Inicializační soubor}
  B : String;         {Paměť pro předběžné načtení jména}
  i,j,k,L : Integer;

procedure (*****) skipws (***** lokální procedura *****)
;
{Procedura odstraní počáteční bíle znaky v přečteném jméně}
var
  i : Integer;
begin
  i := 1;
  while( B[i] <= ' ' )do         {Najdi první nebílý znak}
       inc(i);
  if( i>1 )then                  {Není-li prvním znakem řetězce}
       begin                     {setřes řetězec}
        move( B[i], B[1], Length(B)-i+1 );
       dec( Byte(B[0]), i-1 );   {Nastav novou délku řetězce}
  end;
end;
(********** skipws **********)

begin
  Assign( F, Soubor );           {Sdružení logického souboru s fyzickým}
  Reset( F );                    {Otevření logického souboru}
  ReadLn( F, Mest );             {Načtení počtu měst v tabulce}
  GetMem( Jmeno, SizeOf(Pointer)*Mest );
{Alokace vektoru ukazatelů na jména}
  GetMem( Dalka, SizeOf(Integer)*( Mest*(Mest-1) div 2 ) );
{Alokace vektoru pro uložení vzdáleností}
  k := 0;                        {Lineární index ukládané vzdálenosti}
  for i:=0 to Mest-1 do          {U každého místa se ...}
  begin
       for j:=0 to i-1 do        {...nejprve načtou vzdálenosti }
        begin                    {k předchozím místům}
         Read( F, Dalka^[ k ] );
         Inc(k);
        end;
       ReadLn( F, B );           {Načež se provizorně načte jeho název }
       skipws;             {Z názvu se odstraní úvodní bílé znaky }
       L := Length( B ) + 1;     {A tento upravený název se uloží }
       GetMem( Jmeno^[ i ], SizeOf(Char)*L );   {Alokuje se paměť }
       move( B, Jmeno^[ i ]^, L );          {do níž se název zkopíruje }
  end;
  Close( F );                    {Zavření inicializačního souboru}
end;
(********** cTabulka.Init **********)
```

V Pascalu musíme samostatně definovat i konstruktor třídy *cMísto*, nicméně jeho definice je naprosto triviální:

```
(*  Příklad P6 – 8  *)
constructor (*****) cMisto.InitInt (*****)
( T : uTabulka; Misto : Integer );
begin
  Tabulka := T;
  IndexC := Misto;
end;
```

Když máme objekty zkonstruované, měli bychom mít také možnost si je hned někam vytisknout. Dohodneme se, že při tisku tabulky dodržíme zhruba formát inicializačního souboru, s výjimkou úvodního údaje o počtu měst, který by v daném případě spíše rušil. Výstupní operace budeme definovat standardním způsobem.

V C++ použijeme homonymum operátoru výstupu:

```
/*  Příklad C6 – 14  */
ostream& /*****/ operator << /*****/
( ostream&o, const cTabulka& T )
//Zapíše tabulku do výstupního proudu ve formátu obdobném formátu
//obdobném formátu inicializačního souboru
{
  for( int i=0, k=0; i < T.Mest; i++ )
  {
   for( int j=0; j < i; j++ )               //Nejprve vzdálenosti
   o << setw( 4 ) << T.Dalka[ k++ ];        //k předchozím městům v tab.
  o << " +--" << T.Jmeno[ i ] << endl;      //Pak název města
  }
  return o;
}
/********** operator << **********/
```

V Pascalu použijeme naši oblíbenou dvojici metod *FWrite* a *Write*, kde první z nich zapisuje data do obecného souboru a druhá pak na standardní výstup.

```
(*  Příklad P6 – 9  *)
procedure (*****) cTabulka.WriteF (*****)
( var F : Text );
{Zapíše do souboru F tabulku v obdobném tvaru, v jakém byla zapsána
 v inicializačním souboru }
var
  i,j,k,L : Integer;
begin
  k := 0;                        {Index do tabulky vzdáleností}
  for i := 0 to Mest-1 do
  begin
       for j := 0 to i-1 do      {Nejprve vzdálenosti k předhozím }
       begin                     {místům v tabulce }
        System.Write( Dalka^[ k ]:4 );
```

```
        inc(k);
       end;
       WriteLn( ' +--', Jmeno^[ i ]^ );      {Pak název místa }
  end;
end;
(********** cTabulka.WriteF **********)

procedure (*****) cTabulka.Write (*****)
 ;
{Zapíše tabulku do standardního výstupního souboru}
begin
  WriteF( OutPut );
end;
(********** cTabulka.Write **********)
```

Nyní se dostáváme k vlastním operacím zjišťujícím vzdálenost. Nejprve si definujeme funkce pro přímé zjištění vzdálenosti dvou měst, zadaných jejich celočíselným indexem v tabulce.

V C++ bude, jak jsme si řekli, nejvhodnější přetížit operátor funkčního volání se dvěma celočíselnými parametry:

```
/*  Příklad C6 – 15  */
int /*****/ cTabulka::operator() /*****/
( int Z, int Do ) const
//Vrátí vzdálenost mezi dvěma místy o zadaných indexech
{
  if( Z == Do )
       return 0;
  if( Z < Do )
       {int i=Z; Z=Do; Do=i; }
  return Dalka[ Z*(Z-1)/2 + Do ];
}
/********** cTabulka::operator() **********/
```

V Pascalu jsme se rozhodli pro klasickou dvouparametrickou funkční metodu, kterou jsme nazvali *Vzdálenost*:

```
(*  Příklad P6 – 10  *)
function (*****) cTabulka.Vzdalenost (*****)
( m1, m2 : Integer ) : Integer;
{Vrátí vzdálenost mezi dvěma městy o zadaných indexech}
var
  i : Integer;
begin
  if( m1 = m2 )then
  begin
       Vzdalenost := 0;
       exit;
  end;
  if( m1 < m2 )then
  begin
       i := m1; m1 := m2; m2 := i;
  end;
```

```
  Vzdalenost := Dalka^[ m1*(m1-1) div 2 + m2 ];
end;
(********** cTabulka.Vzdalenost **********)
```

Na závěr této podkapitoly si ještě ukážeme definice obou indexových operátorů. Ve skutečnosti vlastně již nebudeme indexový operátor ve třídě *cTabulka* po provedených modifikacích potřebovat. Jeho definici tu však ponecháváme, abychom si připomněli možnost volání konstruktoru nového objektu v příkazu **return** a abychom pascalistům ukázali jednu z možností, jak se vyrovnat s tím, že Turbo Pascal neumí vracet hodnoty objektových typů.

```
/*  Příklad C6 - 16  */
inline cMisto /*****/ cTabulka::operator[] /*****/
( int i ) const
{
  return cMisto( this, i );
}
/********** cTabulka::operator() **********/
int /*****/ cMisto::operator[] /*****/
( cMisto& M ) const
//Vrátí vzdálenost z místa odkazovaného instací do místa argumentu
{
  if( Tabulka != M.Tabulka )
      abort();
  return (*Tabulka)( Index, M.Index );
}
/********** cMisto::operator() **********/
```

```
(*  Příklad P6 - 11  *)
procedure (*****) cTabulka.IIndex (*****)
( Misto:Integer; var Ret:cMisto );
begin
  Ret.InitInt( Addr(Self), Misto );
end;
(********** cTabulka.IIndex **********)

function (*****) cMisto.Index (*****)
( var M : cMisto ) : Integer;
{Vrátí vzdálenost z místa odkazovaného instancí do místa parametru}
begin
  if( Tabulka<>M.Tabulka ) then
      Halt;
  Index := Tabulka^.Vzdalenost( IndexC, M.IndexC );
end;
(********** cMisto.Index **********)
```

Podívejme se nyní na metodu *Čti*, která má za úkol interaktivně zjišťovat, které město má uživatel na mysli, a vlastní testovací program. My jsme použili následující řešení:

```
/*  Příklad C6 – 17  */
void /*****/ beep /*****/
()
//Pomocná procedura generující varovný akustický signál
{
  for( int i=0; i < 3; i++ )
  {
       sound( 2000 ); delay( 100 ); nosound(); delay( 20 );
  }
}
/********** beep **********/
cMisto /*****/ cTabulka::Cti /*****/
()
//Funkce pro interaktivní nalezení názvu města. Předpokládá, že názvy
//měst jsou v tabulce seřazeny podle abecedy.
{
  static const char ENTER=13;    //Kód znaku Enter
  int x = wherex();              //Výchozí pozice kurzoru
  int y = wherey();
  int i = 0;                     //Index nalezeného jména
  int j = 0;                     //Index testovaného písmene
  char c0[ 20 ];                 //Dosud odsouhlasená část jména
  while( 1 )
  {
       gotoxy( x, y );           //Vrať se na počátek názvu města
       cputs( Jmeno[i] );        //Vytiskni jméno - pro případ, že by
       clreol();            //předchozí bylo delší, vyčisti zbytek řádku
       gotoxy( x+j, y );         //Kurzor pod zadávaný znak
       char c = getch();
       if( c == ENTER )          //Jméno bylo nalezeno
       break;         //----------------------------->
       int i0 = i;               //Kam se vrátím, když nic nenajdu
       int Nalez = 0;       //Odpovídající jméno jsme ještě nenalezli
       while( (j==0) || !memcmp(Jmeno[i], c0, j-1) )
//Projdeme všechna jména se shodným, dosud odsouhlaseným počátkem
       {
        if( Jmeno[i][j] != c )//Aby nevznikaly problémy s diakritickými
       // znaménky, testujeme znak na nerovnost. Jinak by sem patřilo
       // efektivnější
       //if( Jmeno[i] [j] < c
         {//Příkaz musí být v bloku, aby si neuzurpoval následující else
          if( ++i >= Mest )
           break;     //------->
         }
         else
         {
          Nalez = 1;             //Našli jsme hledané
          break;                 //------->
         }
       }                         //while memcmp <--------//
       if( Nalez )
        c0[ j++ ] = c;           //Poslední odsouhlasený znak
       else
       {                         //Nenašli jsme takové
        i = i0;                  //Vrátíme se k výchozímu názvu
        beep();                  //Oznámíme neplatnou volbu
```

```
      }
  }             //while 1 <-----------------------------//
  gotoxy( x, y );               //Potřebujeme přesunout kurzor za název
  cputs( Jmeno[i] );            //- volím cestu nejmenšího odporu
  return cMisto( this, i );
}
/********** cTabulka::Cti **********/
```

Pascalská definice je opět o maličko složitější, protože její součástí je definice funkce, jejíž ekvivalent si mohou „pluskaři“ vyzvednout ze systémové knihovny. Až na tuto drobnou odchylku si však jsou obě definice velmi podobné:

```
(*  Příklad P6 – 12  *)
procedure (*****) cTabulka.Cti (*****)
( var Ret:cMisto );
{Funkce pro interaktivní nalezení názvu města. Předpokládá, že názvy
 měst jsou v tabulce seřazeny podle abecedy.}
procedure (*****) beep (*****)
;
{Pomocná procedura, generující varovný akustický signál}
var
  i : Integer;
begin
  for i:=0 to 2 do
  begin
       sound( 2000 ); delay( 100 ); nosound; delay( 20 );
  end;
end;
(********** beep **********)
var
  i,j,k,l : Integer;
  x,y : Integer;
  i0 : Integer;
  Nalez : Boolean;
  c : Char;
  c0 : String[ 40 ];            {Dosud odsouhlasená část jména}
  Jm : uString;
function (*****) ShodaPoc (*****)
: boolean;
{Pomocná procedura, porovnávající dvě oblasti paměti}
var
  i : Integer;
begin
  ShodaPoc := TRUE;
  for i:=1 to j do
       if( Jm^[i] <> c0[i] )then begin
        ShodaPoc := FALSE;
        exit;
       end
end;
(********** memcmp **********)
label
  L1, L2;
const
```

```
  ENTER = #13;                    {Kód znaku Enter}
begin
  x := wherex;                    {Výchozí pozice kurzoru}
  y := wherey;
  i := 0;                         {Index nalezeného jména}
  j := 0;                         {Index testovaného písmene}
  while( True )do begin
       GotoXY( x, y );            {Vrať se na počátek názvu města}
       Jm := Jmeno^[i];           {Pouze pro zvýšení efektivity}
       System.Write( Jm^ );       {Vytiskni jméno - pro případ, že by
                                   předchozí}
       ClrEol;                    {bylo delší, vyčisti zbytek řádku }
       GotoXY( x+j, y );          {Kurzor pod zadávaný znak}
       c := ReadKey;
       if( c=ENTER )then          {Jméno bylo nalezeno}
        goto L1;                  {----------------------------> }
       i0 := i;                   {Kam se vrátím, když nic nenajdu }
       Nalez := False;            {Odpovídající jméno jsme ještě
  nenalezli }
       while( ShodaPoc )do begin
{Projdeme všechna jména se shodným dosud odsouhlaseným počátkem.}
        if( Jm^[j+1] <> c )then begin       {Aby nevznikaly problémy
                 s diakritikou, testuji znak na nerovnost. Jinak by
                 sem patřilo efektivnější if( sp^[j+1]<c ) }
          inc(i);
          if( i >= Mest )then
            goto L2;              {----------> }
          Jm := Jmeno^[i];
        end
        else begin
          Nalez := True;          {Našli jsme hledané}
          goto L2;                {----------> }
        end;
       end;
       L2:;                       {while memcmp <---------/ }
       if( Nalez ) then
       begin
         inc(j);
         c0[ j ] := c;            {Poslední odsouhlasený znak}
       end
       else
       begin                      {Nenašli jsme takové}
        i := i0;                  {Vrátíme se k výchozímu názvu}
        Jm := Jmeno^[i];
        beep;                     {Oznámíme neplatnou volbu}
       end;
  end;
  L1:;                      {while True <---------------------------/ }
  GotoXY( x, y );           {Potřebujeme přesunout kurzor za název }
  System.Write( Jm^ );      {- volíme cestu nejmenšího odporu }
  Ret.InitInt( Addr(Self), i );
end;
(********** cTabulka.Cti **********)
```

Na závěr si tedy ukážeme testovací programek, který nám náš návrh prověří.

```
/*  Příklad C6 - 18  */
void /*****/ Test /*****/
()
{
  cTabulka T( "KRAJE.TAB" );
  crt << "\n\nKRAJE:\n" << T;    //Aby Cti mohla adresovat kurzor,
  do                             // musíme použít crt
  {
       crt << "\n\nVzdálenost z místa ";
       cMisto m1 = T.Cti();
       crt << " do místa ";
       cMisto m2 = T.Cti();
       crt << " je " << m1[ m2 ] << " km\n\n"
           << "Další dotaz (A/N): ";
  }while( toupper( getche() ) == 'A' );
 }
/********** Test **********/
```

```
(*  Příklad P6 - 13  *)
procedure (*****) Test (*****)
;
var
  T : cTabulka;
  m1,m2 : cMisto;
begin
  T.Init( 'KRAJE.TAB' );
  WriteLn;
  WriteLn( 'KRAJE:' );
  T.Write;
  WriteLn;
  repeat
       WriteLn;
       Write('Vzdálenost z místa ');
       T.Cti( m1 );
       Write(' do místa ');
       T.Cti( m2 );
       WriteLn( ' je ', m1.Index( m2 ), ' km');
       Write( 'Další dotaz (A/N): ');
  until( UpCase( ReadKey )<>'A' );
end;
(********** Test **********)
```

Čím tuto podkapitolu ukončit? Na předchozím příkladu jste mohli vidět několik věcí. Za prvé si myslíme, že dostatečně průkazně demonstroval, že před vlastním programováním je třeba si celý projekt vždy důkladně rozmyslet. Na počátku jsme se vám snažili nabídnout některé zásady budování naší třídy, nicméně hned na dalších stránkách se při hlubší analýze ukázalo, že mnohé z nich nepatřily k těm nejvhodnějším.

Počítejte s tím, že se vám málokdy podaří vyřešit problémy dané etapy tak, abyste je v některé z následujících etap nemuseli přepracovávat. Čím složitější projekt, tím je pravděpodobnost nezdaru větší, a tím jsou i všechny opravy dražší. Pokud opravujeme některé své vlastní předchozí chybné úvahy, jedná se o zcela jinou situaci, než když je

třeba modifikovat dispozice, podle nichž již pracuje na dílčích problémech několik skupin programátorů. Nechcete-li, aby vás vaše projekty přišly příliš draho, nepodceňujte etapy, které předcházejí vlastnímu kódování, tj. zápisu programu v daném programovacím jazyce.

Při porovnání našeho jednoduchého počátečního řešení s následujícím objektově orientovaným řešením asi mnohé z vás napadlo, že zavedením objektů se celý problém zbytečně zkomplikoval. To je však jen první dojem – a je mylný. Podívejme se na věc podrobněji.

Začněme u konstruktorů. Tím, že v původním nejjednoduším řešení nebyly žádné jejich ekvivalenty, nemohli jsme pracovat se žádnou jinou tabulkou než s tou, která byla natvrdo zakódována v programu. Pokud bychom chtěli původní program rozšířit o tuto možnost, museli bychom jej rozšířit i o kód realizující operace ekvivalentní vyvolání konstruktoru. Naprosto shodné je to i s tiskem tabulky.

Kód plusového homonyma operátoru funkčního volání a pascalské funkce *Vzdálenost* je v původním programu vpodstatě obsažen, i když ve zhuštěné podobě.

Homonyma operátorů indexace, resp. ekvivalentní pascalské metody, jsou v našem programu oproti předchozímu programu opravdu navíc. Otevírají nám však dveře pro další rozšíření, při němž můžeme hledat vzdálenosti mezi městy z různých tabulek.

Metoda *Čti*, kterou jsme definovali teď, v původním programu také chybí. S ní tam však chybí i elegance zadávání měst, jejichž vzájemnou vzdálenost se chceme dozvědět.

V souhrnu bychom tedy mohli konstatovat, že objektové řešení sice je delší, ale faktické zvětšení tak velké není, protože většina „nakynutého“ kódu buď přináší nové funkce nebo otevírá cestu k dalším rozšířením. Styl, s kterým byl navržen původní program, je vhodný pro jednoduchá okamžitá řešení. Styl, o nějž jsme se pokoušeli poté, je naopak vhodnější ve chvíli, kdy jste rozhodnuti program dále zdokonalovat a rozvíjet. Musíte si však dát pozor na to, aby genialita vaší koncepce nepřesahovala příliš genialitu programátorů, kteří ji budou realizovat, ale to je již jiná pohádka.

3.9 Operátor ->

Posledním z operátorů, o kterých si musíme povědět zvlášť, je operátor „->“. Vzhledem k tomu, jazyk Pascal žádný podobný operátor nenabízí, týká se celá tato podkapitola pouze C++.

Syntaktická pravidla pro jeho přetěžování jsou poněkud netypická, ale to je dáno i vlastnostmi standardní verze tohoto operátoru. Pro účely přetěžování se totiž operátor „->“ pokládá za unární. Připomeňme si, že jej smíme přetěžovat pouze jako nestatickou metodu objektového typu. Jeho jediným operandem je tedy instance, pro kterou jej zavoláme.

Při použití přetíženého operátoru „->“ musí stát vlevo od něj instance objektového typu a vpravo identifikátor složky objektového typu.

Je-li *a* instance třídy *A* a *b* její složka (atribut nebo metoda), znamená zápis

```
a -> b
```

totéž, jako

```
(a.operator->()) -> b
```

Jinými slovy: na hodnotu, kterou přetížený operátor „->“ vrátí, se znovu použije operátor ->.

Z toho plyne, že přetížený operátor „->“ musí vracet buď ukazatel na objekt (pak se použije standardní operátor „->“) nebo instanci nějakého jiného objektového typu, pro který jsme přetížili operátor „->“.

Použití operátoru „->“

Ukážeme si, jak lze využít přetíženého operátoru „->“ ke zpřehlednění zápisu programu.

Často se setkáváme s objektovými typy, do jejichž instancí neukládáme přímo užitečná data, ale pouze ukazatele na ně. Vezměme např. strukturu *udalost*, která obsahuje pořadové číslo, datum a popis nějaké události.

Dále definujeme třídu *Poznamky*, která bude obsahovat pole *událostí*. Toto pole bude mít na počátku velikost *KOLIK* prvků, a v případě potřeby se bude zvětšovat; o to se stará metoda *zvetsi*.

Třída *Poznamky* bude také obsahovat informace o počtu prvků v poli, index aktuálního prvku (toho, se kterým právě pracujeme) a index naposledy přidaného prvku.

O vkládání prvků se bude starat metoda *vloz_dalsi*, která vloží nový prvek na konec pole; pokud se nevejde, zvětší pole. Operátory „++“ a „--“ budou měnit hodnotu indexu aktuálního prvku (tedy určí jako aktuální předchozí nebo následující prvek v poli). Čtenář jistě dokáže sám navrhnout další metody pro práci s touto třídou.

```
/*  Příklad C6 – 19  */
#include <string.h>
// Když se alokace nepodaří
void chyba (){/* ...*/}

// Pole se alokuje po úsecích
// o délce KOLIK
const int KOLIK = 3;

// Struktura, obsahující užitečná data
struct udalost{
  unsigned cislo;
  int den, mesic, rok;
  char *popis;
// Konstruktor
  udalost(int c=0, int d=0, int m=0, int r=0, char* pop=0)
  :cislo(c), den(d), mesic(m), rok(r)
  {
       if(pop) {
       popis = new char[strlen(pop)];
       strcpy(popis, pop);
  }else
       popis = 0;
  }
};
```

```
// Třída, obsahující pole poznámek s daty
class Poznamky {
       udalost *pole_poznamek;
       int aktual;
       int posledni;
       int velikost;
  public:
       Poznamky();
       void zvetsi();
       void vloz_dalsi(int d, int m, int r, char* kom);
       void reset() {aktual = 0;}
       void operator++();
       void operator--();
       // ...a další metody
};

Poznamky::Poznamky()
:aktual(-1), posledni(-1), velikost(KOLIK)
{               // Zatím žádné údaje
  pole_poznamek = new udalost[KOLIK];
  if(!pole_poznamek) chyba();
  for(int i = 0; i < KOLIK; i++) pole_poznamek[i] = 0;
// Pole je prázdné
}

// Prodlouží pole o KOLIK položek:
// alokuje nové, staré do něj překopíruje
// a zruší
void
Poznamky::zvetsi(){
  udalost* pom_pole = pole_poznamek;
  pole_poznamek = new udalost[posledni+KOLIK+1];
  for(int i = 0; i <= posledni; i++)
  pole_poznamek [i] = pom_pole[i];
  delete [] pom_pole;
  velikost += KOLIK;
}

// Vloží do pole poznámek další
// položku; je-li potřeba, zavolá metodu
// zvětši
void
Poznamky::vloz_dalsi(int d, int m, int r, char * pop){
  if(posledni == velikost-1) zvetsi();
  posledni++;
  pole_poznamek[posledni].den = d;
  pole_poznamek[posledni].mesic = m;
  pole_poznamek[posledni].rok = r;
  pole_poznamek[posledni].cislo=posledni+1;
  pole_poznamek[posledni].popis=new char[strlen(pop)];
  if(!pole_poznamek[posledni].popis) chyba();
  strcpy(pole_poznamek[posledni].popis, pop);
  aktual=posledni;
}

// Změní index aktuálního prvku pole
void
```

```
Poznamky::operator++(){
   aktual ++;
   if(aktual > posledni) reset();
}

void
Poznamky::operator--(){
   aktual --;
   if(aktual < 0) aktual = posledni;
}
```

Jestliže nyní deklarujeme instanci *p* třídy *Poznámky*,

```
Poznamky p;
```

můžeme do ní postupně naskládat řadu poznámek k událostem:

```
p.vloz_dalsi(3,1,1991,"Padal sníh");
p.vloz_dalsi(15, 2, 1992, "Jirkovi přeskočilo");
p.vloz_dalsi(16, 2, 1992, "Už je zase normální");
p.vloz_dalsi(17, 2, 1992, "Teď zase pro změnu prší");
```

Jak ale zajistit přístup k uloženým datům? Je jasné, že nejčastěji budeme pracovat s „aktuálním“ prvkem, tedy s prvkem, určeným indexem *aktual*. V takovém případě můžeme použít služeb přetíženého operátoru „->“. Definujeme jej takto:

```
udalost *
Poznamky::operator->(){
   return pole_poznamek + aktual;
}
```

Pak můžeme napsat

```
int D = p->den;
```

a tento příkaz by znamenal totéž jako

```
int D = p.pole_poznamek[aktual].den;
```

kdyby nám něco podobného dovolila přístupová práva.

Pomocí takto přetíženého operátoru „->“ můžeme také měnit data, uložená v aktuálním prvku seznamu:

```
p->rok = 1887;
```

Skutečnost, že takto definovaný operátor „->“ umožňuje nejen číst uložená data, ale také je měnit, není vždy žádoucí; pokusme se to nějak napravit. Jak zařídit, abychom jej mohli používat pouze jednosměrně, tj. abychom si uložená data zabezpečili před nežádoucími změnami? Odpověď je jednoduchá: stačí operátoru „->“ předepsat, aby vracel ukazatel na konstantu:

```
const udalost *
Poznamky::operator->(){
   return pole_poznamek + aktual;
}
```

Pozor na nekonečnou rekurzi

Zopakujme si, jak je definováno použití přetíženého operátoru „->“: Příkaz

```
a -> b
```

se chápe jako

```
(a.operator->()) -> b
```

kde druhé použití „->“ může znamenat buď standardní operátor nebo opět přetížený operátor. Zde je ukryta rekurze. Pokud totiž vrátí přetížený operátor „->“ instanci nějakého objektového typu, bude se volat operátor ->“, přetížený pro vrácenou instanci, a překladač s ním bude zacházet podle stejných pravidel, jako s prvním operátorem „->“ atd.

Tato rekurze musí skončit tím, že některý z přetížených operátorů „->“ vrátí ukazatel na objekt a překladač použije standardní operátor „->“. Pokud se spleteme a vytvoříme nekonečnou rekurzi, mohou se některé překladače zhroutit.

Typický příklad, který se nám svého času podařil:

```
class A{
       int i;
  public:
       A(int j = 0);
       void set(int j);
       A operator->();            // !!!
};

A A::operator->(){
   return *this;
}
```

Operátor „->“, definovaný ve třídě *A*, vrací opět instanci třídy *A*! Jestliže tento operátor použijeme, např. v příkazu

```
A a;
a -> set(8);
```

bude zle. Na instanci, vrácenou operátorem „->“, se použije opět přetížený operátor, ten vrátí touž (nebo pozměněnou) instanci, na ni se opět použije přetížený operátor atd.

Výsledek závisí na překladači: Borlandské překladače ohlásí vyčerpání paměti (*out of memory*) a při překladu v prostředí ukáží jako chybný řádek místo, kde jsme operátor použili. Použijeme-li samostatný překladač, nedozvíme se, kde k chybě došlo, a hledání příčiny může trvat dost dlouho.

Podobně se zachová i Watcom C++ 10.5. Překladač Microsoft Visual C++ 1.5 rozpozná, že narazil na nekonečnou rekurzi v deklaraci operátoru, a ohlásí chybu v místě deklarace.

4. Dynamické datové typy

V této kapitole uděláme malou odbočku. Přestaneme se na chvíli věnovat výkladu vlastností probíraných jazyků a přesuneme se „o patro výše“. Budeme si povídat o některých datových strukturách, které ve svých programech používáme (přesněji mohli bychom používat).

Doposud jsme pracovali s objekty, jejichž velikost se v průběhu programu neměnila. Pokud někteří z vás namítnou, že u řetězců tomu tak nebylo, mají pravdu pouze napůl. Počet znaků daného textového řetězce se možná v průběhu programu měnil, ale velikost paměti vyhrazené pro tento řetězec zůstávala konstantní. (Za chvíli si o tom povíme podrobněji.)

V kapitole o práci s ukazateli jsme si mimo jiné říkali o dynamických proměnných, kterým se potřebná paměť přidělí až v průběhu programu. Ovšem i zde zůstávala jednou přidělená paměť vpodstatě konstantní. Pokud jsme potřebovali, aby proměnná pracovala s pamětí jiné velikosti, museli jsme nejprve původně přidělenou paměť vrátit a požádat o paměť novou.

Datové typy, jejichž objekty se chovají tak, jak jsme popisovali v předchozích dvou odstavcích, nazýváme **statické**. To proto, že velikost jejich objektů je v průběhu programu neměnná. V této kapitole si představíme některé **dynamické datové typy**, jejichž objekty mají tu vlastnost, že se jejich velikost v průběhu programu může dynamicky měnit.

Na počátku kapitoly jsme přiznali, že jisté vlastnosti dynamických datových typů mají i textové řetězce, protože jejich velikost se může v průběhu programu často měnit. Jejich implementace je však v obou jazycích statická: pro řetězec se vyhradí vektor znaků, do nějž se musí vejít – změny velikosti řetězce mohou probíhat pouze v rámci tohoto vektoru. Pokud bychom měli takovýchto řetězců více, mohli bychom velice rychle vyčerpat všechnu dostupnou paměť, a přitom bychom v ní měli ještě spoustu prázdného, avšak nevyužitelného místa.

Jedinou šancí, jak při větším počtu objektů proměnné velikosti vystačit s dostupnou pamětí, je dynamická reprezentace těchto objektů, při níž je každému objektu vždy přiděleno právě tolik paměťového prostoru, kolik v daném okamžiku potřebuje (přesněji řečeno požaduje), a veškerý zbylý paměťový prostor je k dispozici systému. Kdykoliv se v průběhu výpočtu nějaké místo uvolní, hned se vrátí systému, aby je mohl poskytnout objektu, jehož paměťové nároky mezitím vzrostly (takto jsme hospodařili s pamětí ve třídě *cStrT*).

V souvislosti s průběžným dynamickým přidělováním paměti vzniká jeden častý problém. Po chvíli přidělování a uvolňování paměti může nastat situace, kdy je systém požádán o blok paměti, který je větší, než kterýkoliv z bloků, jež má systém k dispozici. Přitom součet velikostí těchto malých bloků může být i několikanásobně větší, než je velikost bloku požadovaného po systému.

Systémy, které intenzivně pracují s dynamickými datovými typy, bývají pro tento případ často vybaveny speciálním programem, který se nazývá *garbage collector*, což bychom mohli doslovně přeložit jako sběrač smetí neboli **popelář**. Jeho úkolem je projít

haldu (paměť určenou pro dynamické přidělování dat), najít v ní všechny používané a nepoužívané bloky, všechny používané bloky „srazit“ k sobě a za ně (popřípadě před ně) umístit jeden velký blok volné paměti.

Na první pohled to vypadá snadně, ale tak jednoduché to zase není. Při práci s dynamickými datovými typy leží těžiště veškeré práce na ukazatelích. Jakmile tedy popelář nějaký blok přesune, musí najít všechny ukazatele, které na tento blok ukazují, a odpovídajícím způsobem změnit jejich hodnoty. Aby to vůbec dokázal, musí být součástí všech dynamických objektů i informace pro popeláře, které se musí při každé akci aktualizovat. Tím se ovšem efektivita celého programu výrazně snižuje, takže většina systémů správy dynamické paměti bývá koncipována tak, že práci popeláře nepodporuje.

Opusťme nyní povšechný úvod a podívejme se, jaké dynamické datové typy se v programech používají nejčastěji.

Seznam

Seznam (*list*) je asi nejpoužívanějším dynamickým datovým typem. Je tvořen lineárně uspořádanou množinou prvků. To znamená, že ke každému prvku s výjimkou posledního existuje následník a ke každému prvku s výjimkou prvního existuje předchůdce. Někdy se používají i kruhové seznamy, v nichž následníkem posledního prvku je prvek první a naopak předchůdcem prvního prvku je prvek poslední.

Seznamy se implementují jako zřetězené záznamy, kdy jednou ze složek záznamu je ukazatel na *následující* prvek. Pokud mohou být hodnoty jednotlivých prvků seznamu hodně různorodé, pak je nejvýhodnější, aby měl tento záznam pouze dvě složky: ukazatel na další prvek a ukazatel na hodnotu daného prvku. Touto hodnotou pak již může být prakticky cokoliv – např. další seznam.

Takovýmto seznamem lze poměrně snadno procházet, avšak pouze jedním směrem. Pokud naše aplikace vyžaduje, abychom k danému prvku uměli rychle najít nejen jeho následníka, ale také jeho předchůdce, implementuje se seznam tak, že součástí záznamu o daném prvku je navíc i adresa jeho předchůdce. Hovoříme pak o **dvojitě zřetězeném seznamu** nebo o **dvousměrném seznamu** (*double linked list*).

První prvek seznamu bývá často označován **hlava** (*head*), a seznam, který vznikne z původního seznamu oddělením hlavy, bývá označován jako **ocas** (*tail*).

Seznamy hrají klíčovou úlohu v oblasti umělé inteligence – jejich důležitost lze poznat již z jména klíčového programovacího jazyka umělé inteligence: název LISP je totiž akronym z anglického *list processing* neboli zpracování seznamů. Dokonce jsou konstruovány počítače, jejichž hardware je navržen tak, aby na něm byla práce se seznamy co neefektivnější (počítače řady PC mezi ně nepatří).

Možná jste slyšeli o programovacím jazyku Logo, který se někde používá při výuce matematicky a fyziky na některých základních a středních školách. V něm je seznam jediným strukturovaným datovým typem – dokonce i textové řetězce jsou tam často reprezentovány jako seznamy znaků.

Seznamy však hrají velice důležitou roli i v klasickém programování, kde mimo jiné slouží k implementaci ostatních dynamických datových struktur. Mohli bychom dokon-

ce říci, že dynamické datové struktury se implementují buď staticky přes vektory (např. nám známé textové řetězce) nebo dynamicky přes seznamy.

Zásobník

Zásobník (*stack*) je po seznamu druhým nejpoužívanějším dynamickým datovým typem. Je klíčovou datovou strukturou mnoha aplikací. Realizuje registr typu LIFO (*last in, first out* – poslední dovnitř, první ven). U zásobníku jsou, stejně jako u fronty, o které budeme mluvit dále, základními operacemi testování prázdnosti zásobníku, přidání prvku do zásobníku a jeho vyjmutí za zásobníku.

Zásobník funguje obdobně jako hromádka papírků s poznámkami: papírek, který jsme na ni odložili naposledy, z ní zpět odebereme jako první, protože leží na vrchu. Zásobníky se používají při vyhodnocování matematických výrazů při překladech programů nebo při převádění původně rekurzivních algoritmů na zpracování pomocí cyklů.

Se zásobníkovou strukturou jsme se setkali při ladění našich programů, když jsme si přes příkaz CTRL-F3 vyvolávali okno zobrazující hierarchii volání procedur. Datovou strukturou řídící volání procedur a návraty zpět do volajícího programu je právě zásobník. Podprogram, který přišel poslední, odchází první, protože nejprve se musíme vrátit z podprogramu volaného, a teprve pak můžeme ukončit podprogram volající.

Fronta

Fronta (*queue*) je datová struktura, která ve své základní podobě realizuje registr typu FIFO (*first in, first out* – první dovnitř, první ven, česky „kdo dřív přijde, ten dřív mele“). Základními operacemi s frontou jsou test prázdnosti fronty, přidání dalšího prvku do fronty a vyjmutí prvku, který je na řadě. Objekty, které jsou ve frontě, se chovají tak, jak jsme ve frontách zvyklí: operace vyjmutí prvku, který je na řadě, předá ten z prvků čekajících ve frontě, který do ní byl zařazen jako první.

V programech se často používají některé speciální druhy front, které se chovají trochu jinak. Mezi nejpoužívanější patří **dvojitá fronta**, *deque*[11], která umožňuje přidávat a odebírat prvky z obou konců fronty (fronta ze života: na začátek přicházejí prominenti a konec opouští ti, kteří již nemohou déle čekat) a **fronta s předbíháním** (*priority queue*), v níž se nový prvek zařazuje do fronty podle své důležitosti, takže na řadu přijdou důležitější prvky vždy před prvky méně důležitými.

Strom

Strom (*tree*) bychom mohli považovat za zobecněný seznam, v němž může mít daný prvek i více než jednoho následníka (někdy se naopak seznamům říká degenerované stromy). Prvek, který nemá žádného předchůdce (tj. není ničím následníkem) se nazývá kořen stromu, prvky, které nemají žádné další následníky, se nazývají listy stromu.

Stromy se mimo jiné používají v programech, které hrají hry – často můžete slyšet o tom, že program prohledává strom řešení. Současná situace je vrcholem stromu, z nějž vychází tolik větví (má tolik následníků), kolik je možných (přesněji kolik je uvažova-

[11] Název *deque* vznikl jako zkratka z *double queue*. Výslovnost je „dek“.

ných) tahů. Z každého takto vzniklého vrcholu pak vychází tolik větví, kolik je možných protitahů atd. – dokud stačí paměť a čas.

Množina

Co je to množina (*set*) víme z matematiky. Mezi základní množinové operace patří přidání prvku do množiny (každý prvek může být v množině přítomen pouze jednou, takže nové přidání stejného prvku neudělá nic), vyjmutí prvku z množiny a test na přítomnost prvku v množině. Kromě toho bývají implementovány i operace s celými množinami: sjednocení, průnik a rozdíl množin.

V Pascalu jsou množiny (i když staticky implementované) mezi datovými typy podporovanými přímo v definici jazyka. Do výkladu jsme je prozatím nezařadili, nechali jsme si je až do doby, kdy budete znát základy objektově orientovaného programování, abychom mohli definici množinových datových typů v Pascalu a C++ co nejvíce sjednotit.

Množiny jsou užitečným datovým typem, který však řada programátorů v zájmu zvýšení efektivity nahrazuje celočíselnými datovými typy, a operace přidání a vyjmutí prvku nahrazuje operacemi nahazování a shazování odpovídajících bitů (proto také nejsou součástí základní definice jazyka C ani C++). Domnívám se však, že operace s množinami jsou v Turbo Pascalu implementovány dostatečně efektivně a jejich používání většinou programy zpřehledňuje bez ztráty efektivity.

Kupa

Kupa (v borlandském manuálu *bag* – batoh) je datový typ podobný množině. Liší se však od ní tím, že se v ní může vyskytovat daný prvek i vícekrát. Zmiňujeme se zde o ní proto, že je implementována v borlandské knihovně kontejnerů (*container class library*).

Slovník

Slovník (*dictionary*) je množina položek, které mají dvě části: jednu, která musí být v rámci slovníku jedinečná, protože je daná položka podle této části do slovníku zařazována a v něm vyhledávána, a druhou, která může obsahovat další informace o položce. Jedná se vlastně o množinu rozšířenou o vyhledávání položky podle hodnoty její části – klíče. I o slovníku se zde zmiňujeme hlavně proto, že je implementován v borlandské knihovně skladových tříd.

Existuje ještě řada dalších tříd dynamických datových typů, ale ty jsou většinou jednoúčelové, šité na míru konkrétní aplikaci.

4.1 Seznam

Seznam patří mezi datové typy, kterým říkáme **kontejnery** – setkáte se i s překladem **sklady** (*container classes*), protože jejich hlavním účelem je „skladovat“ nějakou mno-

žinu položek „pod jednou střechou“. Přitom jednotlivé položky můžeme do kontejneru dodávat a zase je z něj odebírat. (Poznamenejme, že mezi sklady patří např. také pole.)

Pokud budeme chtít implementovat nějaký sklad (např. pole nebo seznam), musíme nejprve definovat typ položek, které budeme do tohoto skladu ukládat – na to jsme ale zvyklí již od polí. V případě seznamů však musíme navíc definovat také typ prvků seznamu, tj. těch struktur, které obsahují uloženou informaci spolu s odkazem na další a/nebo na předchozí prvek. Pro prvky seznamu budeme po vzoru jazyka LISP používat název **atomy** (popřípadě **položky**) a pro ukládaná data **hodnoty**.

Před definicí datového typu atomu si musíme rozmyslet, zda atomy mají obsahovat celou ukládanou informaci nebo pouze ukazatel na ni. Kromě toho se také musíme rozhodnout, zda budeme používat jednosměrně zřetězený seznam, jehož atomy sice zaberou méně místa v paměti, avšak některé operace trvají podstatně déle, nebo zda bude pro naše účely vhodnější obousměrně zřetězený seznam, kde však za rychlost zaplatíme zvýšenou spotřebou paměti.

V následujícím výkladu budeme souběžně ukazovat možné implementace základních operací pro oba typy seznamů, abyste mohli porovnat jejich komplikovanost a odhadnout jejich rychlost. Abychom si navíc následující výklad maximálně zjednodušili, mohli se soustředit na vlastní seznamy a nerozptylovali se operacemi nad uloženými hodnotami, budeme v našich příkladech pracovat se seznamem celých čísel.

Abychom však mohli naše procedury použít později i pro jiné typy hodnot, přejmenujeme si v této kapitole celočíselný typ na *tHodn*. Když pak budeme chtít později použít naše procedury pro jiný datový typ, stačí na *tHodn* přejmenovat nový typ hodnot ukládaných do seznamu. (Časem se naučíme elegantnější způsob.)

Pro naše příklady si definujeme následující typy atomů:

```
/*  Příklad C7 – 1  */
typedef int tHodn;          //Typ hodnoty jednotlivých atomů seznamu

/*****/ class cSAtom /*****/
{                           //Atom jednosměrně zřetězeného seznamu
  public:
       tHodn Hodnota;       //Hodnota atomu
       cSAtom* Dalsi;       //Ukazatel na následníka
};
/********** class cSAtom **********/

/*****/ class cDAtom /*****/
{                           //Atom obousměrně zřetězeného seznamu
  public:
       tHodn Hodnota;       //Hodnota atomu
       cDAtom* Dalsi;       //Ukazatel na následníka
       cDAtom* Predch;      //Ukazatel na předchůdce
};
/********** class cDAtom **********/
```

V obou definovaných třídách jsou všechny jejich složky veřejné. To sice není nejlepší řešení, ale ušetří nám práci v dalším výkladu. Časem si ukážeme elegantnější a bezpečnější řešení tohoto problému.

Jak víme, Pascal neumí řídit přístup k jednotlivým složkám tříd – umí pouze lokalizovat v rámci modulu ty z nich, které uvedeme v sekci označené klíčovým slovem **private**. To však má za následek vymizení daných složek z informací poskytovaných o dané struktuře debuggerem. Nejvýhodnější reakcí na tuto skutečnost bude asi to, že klíčové slovo **private** uzavřeme do podmíněně překládané sekce tak, že po dobu ladění daného modulu překládáno nebude, a použijeme jej až ve chvíli, kdy bude daný modul odladěn a my se budeme chtít pojistit proti tomu, abychom na inkriminované složky v jiných modulech omylem „nesáhli“.

```
(*  Příklad P7 – 1  *)
const
  nl = #10#13;              {Přechod na novou řádku}
type
  tHodn = INTEGER;          {Typ hodnoty jednotlivých atomů seznamu}
  pSAtom = ^cSAtom;
  pDAtom = ^cDAtom;
(*****) cSAtom = object (** Atom jednosměrně zřetězeného seznamu **)
{$ifndef LADIM}
  private
{$endif}
       Hodnota : tHodn;     {Hodnota atomu }
       Dalsi : pSAtom;      {Ukazatel na následníka}
end;
(********** cSAtom=object **********)

(*****) cDAtom = object (*** Atom obousměrně zřetězeného seznamu ***)
{$ifndef LADIM}
  private
{$endif}
       Hodnota : tHodn;     {Hodnota atomu }
       Dalsi : pDAtom;      {Ukazatel na následníka}
       Predch : pDAtom;     {Ukazatel na předchůdce}
end;
(********** cDAtom=object **********)
```

V dalších příkladech už budeme šetřit místem a klíčové slovo **private** s příslušnými „dolarovými poznámkami“ (direktivami v komentáři) vynecháme. V případě potřeby si je určitě dokážete doplnit sami.

Struktura atributů vlastního seznamu je na první pohled jednoduchá. Většinou po ní nepožadujeme nic víc, než aby nám ukázala na první a u dvojitě zřetězeného seznamu někdy také na poslední prvek. Pro jednoduchý seznam by tedy mohla dokonce stačit i obyčejná ukazatelová proměnná, která by ukazovala na první prvek. Ta by nám však nedovolila sdružit se svojí hodnotou žádné metody, a proto budeme seznam definovat jako objektový datový typ.

Než však přistoupíme k vlastní definici, zamysleme se nejprve nad tím, jaké operace bychom měli nad seznamem definovat.

Pro seznamy si definujeme pouze neparametrický konstruktor, který nemusí dělat nic jiného, než inicializovat ukazatele na počátek a konec seznamu nulovou hodnotou reprezentující prázdný ukazatel.

Destruktor seznamů je trochu náročnější. Pokud bychom ponechali destrukci seznamu na implicitním destruktoru, odstranil by z paměti pouze instanci typu *cSList*, resp. *cDList*, ale všechny atomy, tvořící vlastní seznam, by nadále zůstaly (a překážely) na haldě. To znamená, že náš destruktor musí při destrukci seznamu zdestruovat také všechny jeho atomy.

Způsob konstrukce a zejména pak destrukce seznamu hrozí možnou kolizí při přiřazování seznamů a při jejich konstrukci kopírovacím konstruktorem. Protože totiž instance seznamových typů obsahují pouze ukazatele na počátek příp. konec seznamu, je zřejmé, že pokud použijeme implicitní přiřazovací operátor nebo implicitní kopírovací konstruktor, překladač vytvoří pouze kopie těchto ukazatelů. Jestliže pak jeden ze seznamů destruujeme, zničíme automaticky i druhý, a ve chvíli, kdy bychom začali destruovat druhý seznam, „sestřelili“ bychom s vysokou pravděpodobností systém.

Možnosti řešení jsou dvě: buďto definujeme kopírovací konstruktor a přiřazovací operátor (v Pascalu ekvivalentní proceduru) tak, že vytvoří kopii původního seznamu, nebo jejich použití prostě zakážeme. Které řešení je lepší, to záleží na konkrétní plánované aplikaci. My se pro tuto chvíli přikloníme k druhému řešení, protože je jednodušší a protože předpokládáme, že v případě potřeby si dokážete správné verze kopírovacího konstruktoru a přiřazovacího operátoru definovat sami.

Zákaz použití je sice jednodušší, ale v Pascalu ne zcela realizovatelný. S kopírovacím konstruktorem je to snadné: když jej nedefinujeme, nemůžeme jej používat. Horší je to s přiřazovacím operátorem, protože jeho použití není možno překladači zakázat. Programátor se tedy musí ohlídat sám.

V C++ je řešení poměrně prosté. Deklarujeme obě metody v sekci **private**. To stačí, definovat je již nemusíme. Pokud by překladač potřeboval použít kopírovací konstruktor nebo přiřazovací operátor mimo metody naší třídy a spřátelené funkce, ohlásí, že jsou pro něj nepřístupné. Pokud by je potřeboval použít v metodách třídy či jejich přátel, použije je jako kteroukoliv externí funkci, ale sestavovací program (linker) nám pak oznámí, že dané funkce nikde nenašel. Účelu tedy bylo dosaženo.

Podívejme se nyní na další metody, které by bylo užitečné v seznamu definovat. Především bychom měli umět otestovat prázdnost seznamu. Dále bychom neměli zapomenout na metody, které budou prvky do seznamu přidávat, a metody, které je z něj budou odebírat.

Abychom si ozřejmili některé rozdíly mezi jednoduše a dvojitě zřetězeným seznamem, zařadíme metody, které budou přidávat (odebírat) prvek jak na začátek seznamu, tak na jeho konec. Výsledná podoba definic obou seznamů by tedy mohla být např. následující:

```
/*  Příklad C7 – 2  */
/*****/ class cSList /*****/
{                          //Seznam jednosměrně zřetězených atomů
  public:
       cSList():Prvni( 0 ) {}
```

```
      ~cSList();
      cSList& PridejPoc( tHodn );
      cSList& PridejKon( tHodn );
      tHodn UberPoc();
      tHodn UberKon();
      int Prazdny() {return( Prvni == 0); }
      int operator!() {return( Prvni == 0); }
      cSList& operator+=( tHodn h ) {return PridejKon( h ); }
      friend ostream& operator << ( ostream&, const cSList& );
  private:
      cSAtom* Prvni;
//Následující deklarace pouze brání překladači použít dané metody v
  programu
      cSList( const cSList& );
      cSList& operator= ( const cSList& );
};
/********** class cSList **********/

/*****/ class cDList /*****/
{                          //Seznam obousměrně zřetězených atomů
  public:
      cDList():Prvni(0), Posl(0) {}
      ~cDList();
      cDList& PridejPoc( tHodn );
      cDList& PridejKon( tHodn );
      tHodn UberPoc();
      tHodn UberKon();
      int Prazdny() {return( Prvni == 0); }
      int operator!() {return( Prvni == 0); }
      cDList& operator+=( tHodn h ) {return PridejKon( h ); }
      friend ostream& operator << ( ostream&, const cDList& );
  private:
      cDAtom* Prvni;
      cDAtom* Posl;
//Následující deklarace pouze brání překladači použít dané metody v
  programu
      cDList( const cDList& );
      cDList& operator=( const cDList& );
};
/********** class cDList **********/
```

Jako námět pro zpřehlednění programů pracujících se seznamy nabízíme přetížený operátor logické negace, který bude testovat prázdnost seznamu, a metodu, přidávající prvek na konec seznamu, jako přetížený operátor +=.

```
(*  Příklad P7 - 2  *)
type
(*****) cSList = object (** Seznam jednosměrně zřetězených atomů **)
  Prvni : pSAtom;
  constructor Init;
  destructor Done;
  procedure PridejPoc( h:tHodn );
  procedure PridejKon( h:tHodn );
  function UberPoc:tHodn;
```

```
  function UberKon:tHodn;
  function Prazdny:boolean;
  procedure writef( var F:Text );
  procedure write;
end;
(********** class cSList **********)

(*****) cDList = object (*** Seznam obousměrně zřetězených atomů ***)
  Prvni : pDAtom;
  Posl : pDAtom;
  constructor Init;
  destructor Done;
  procedure PridejPoc( h:tHodn );
  procedure PridejKon( h:tHodn );
  function UberPoc:tHodn;
  function UberKon:tHodn;
  function Prazdny:boolean;
  procedure writef( var F:Text );
  procedure write;
end;
(********** class cDList **********)
```

Podívejme se nejprve na konstruktory a destruktory. Konstruktory jsou triviální, a v pascalských programech, kde je není možno definovat v rámci deklarace třídy, je pro úsporu místa ani neuvádíme – jistě si je dokážete definovat sami. S destruktory je to složitější. Protože však je destruktor pro jednosměrný i obousměrný seznam vpodstatě totožný, uvádíme pouze první z nich; na doprovodné disketě najdete všechny definice.

```
/* Příklad C7 - 3  */
/*****/ cSList::~cSList /*****/
()
//Destruktor jednosměrně zřetězeného seznamu
  { cSAtom* a;  //Pomocná proměnná
  while( (a = Prvni) != 0 )      //Dokud seznam není prázdný
  {
       Prvni = a->Dalsi;         //První ukazuje až na druhý atom
       delete a;                 //a původní první atom tak můžeme
                                 //smazat
  }
}/********** cSList::~cSList **********/
```

```
(*  Příklad P7 - 3  *)
destructor (*****) cSList.Done (*****)
;
{Destruktor jednosměrně zřetězeného seznamu}
var
  a : pSAtom;              {Pomocná proměnná}
begin
  a := Prvni;
  while( a <> NIL )do      {Dokud seznam není prázdný}
  begin
       Prvni := a^.Dalsi;  {První ukazuje až na druhý atom}
       dispose( a );       {a první atom tak můžeme smazat}
       a := Prvni;
```

```
  end;
end;
(********** cSList.Done **********)
```

Než se podíváme na „obyčejné“ metody, nabídneme vám nejprve testovací prográmek, kterým si budete moci ověřit správnost toho, co jsme naprogramovali. Uvádíme jej v předstihu, abyste si mohli každou metodu vyzkoušet, hned jak si ji naprogramujete, a nemuseli jste čekat na závěr výkladu. Pokud uzavřete části s dosud nedefinovanými metodami do komentářů či do podmíněně překládaných sekcí, můžete si každou vykládanou partii vyzkoušet hned.

```
/*  Příklad C7 – 4  */
void /*****/ Test /*****/
()
{
  static const Esc = 27 - '0';
  cSList S;
  cDList D;
  int i = 1;
  crt << "\n\nTest práce se seznamy\n";
  do
  {
       crt << "\n\n"
           << "Jednoduchý: " << S
           << "Dvojitý: " << D
           << "\n1 - Přidat na počátek"
              "\n2 - Přidat na konec"
              "\n3 - Ubrat z počátku"
              "\n4 - Ubrat z konce"
              "\n\nEsc Konec testu: ";
       switch( getche() - '0' )
       {
           case 1:
              S.PridejPoc( i );
              D.PridejPoc( i*100 );
              i++;
              break;
           case 2:
              S.PridejKon( i );
              D.PridejKon( i*100 );
              i++;
              break;
           case 3:
              S.UberPoc();
              D.UberPoc();
              break;
           case 4:
              S.UberKon();
              D.UberKon();
              break;
           case Esc:
              i = -1;
              break;
```

```
           default:
              crt << "\nŠpatné zadání, opakuj\n";
        }
   }while( i > 0 );
}/********** Test **********/
```

```
(*  Příklad P7 – 4  *)
procedure (*****) Test (*****)
;
const
  Esc = #27;
var
  S : cSList;
  D : cDList;
  i : integer;
  c : char;
begin
  S.Init;
  D.Init;
  i := 1;
  write( nl, nl, 'Test práce se seznamy', nl );
  repeat
       write( nl, nl );
       write( 'Jednoduchý: ' ); S.write;
       write( 'Dvojitý: ' ); D.write;
       write(   nl,
           '1 - Přidat na počátek', nl,
           '2 - Přidat na konec', nl,
           '3 - Ubrat z počátku', nl,
           '4 - Ubrat z konce', nl, nl,
           'Esc Konec testu: ' );
       c := ReadKey;
       writeln( c );
       case c of
           '1': begin
              S.PridejPoc( i );
              D.PridejPoc( i*100 );
              Inc( i );
              end;
           '2': begin
              S.PridejKon( i );
              D.PridejKon( i*100 );
              Inc( i );
              end;
           '3': begin
              S.UberPoc;
              D.UberPoc;
              end;
           '4': begin
              S.UberKon;
              D.UberKon;
              end;
           Esc: i := -1;
           else writeln( nl, 'Špatné zadání, opakuj' );
       end;
```

```
  until( i <= 0 );
end;
(********** Test **********)
```

Abychom mohli doopravdy začít testovat, musíme si ještě vytvořit prostředky, které nám zobrazí výslednou podobu seznamu v nějakém rozumném tvaru. Podoby těchto operátorů (metod) jsou opět pro obě třídy téměř shodné, a proto uvádíme pro změnu pouze druhou z nich. Vzhledem k tomu, že pro odlišení jsou v předchozím testu hodnoty uložené v obousměrném seznamu 100krát větší, doporučujeme vám, abyste ve verzi pro jednosměrný seznam zapsali za oddělovací čárku o dvě mezery více (v řádku, označeném komentářem „Odděl jej“).

```
/*  Příklad C7 – 5  */
ostream& /*****/ operator << /*****/
( ostream& o, const cDList& L )
//Operátor výstupu obousměrně zřetězeného seznamu
{
  cDAtom* a = L.Prvni;      //Ukazatel aktuálního atomu
  o << "( ";                //Úvodní otevírací závorka
  while( a != 0 )           //Cyklus přes všechny atomy
  {
       o << a->Hodnota;     //Vytiskni hodnotu atomu
       a = a->Dalsi;        //Nastav ukazatel na následníka
       if( a )              //Pokud nějaký následník existuje
        o << ", ";          //Odděl jej
  }
  o << " )\n";              //Závěrečná zavírací závorka
  return o;
}/********** operator << **********/
```

```
(*  Příklad P7 – 5  *)
procedure (*****) cSList.writef (*****)
( var F : Text );
{Výstup hodnot jednosměrně zřetězeného seznamu do souboru F}
var
  a : pSAtom;
begin
  a := Prvni;                              {Ukazatel aktuálního atomu }
  System.write( F, '( ' );                 {Úvodní otevírací závorka }
  while( a <> NIL )do                      {Cyklus přes všechny atomy }
  begin
       System.write( F, a^.Hodnota );      {Vytiskni hodnotu atomu }
       a := a^.Dalsi;            {Nastav ukazatel na následníka }
       if( a <> NIL )then        {Pokud nějaký následník existuje}
        System.write( ', ' );              {Odděl jej }
  end;
  writeln( ' )' );                         {Závěrečná zavírací závorka
  }
end;
(********** cSList.writef **********)

procedure (*****) cSList.write (*****)
;
```

```
{Výstup hodnot jednosměrně zřetězeného seznamu na standardní výstup}
begin
  writef( output );
end;
(********** cSList.write **********)
```

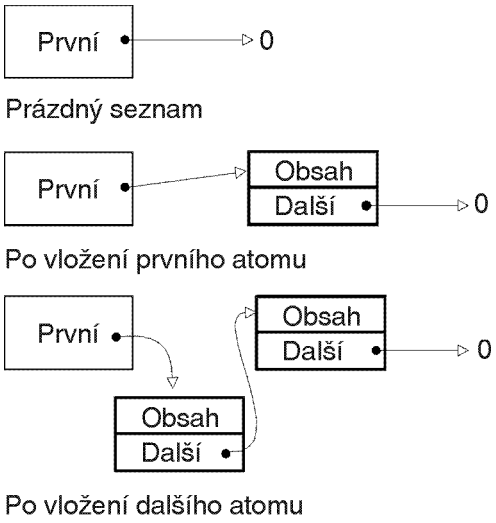
Podívejme se nyní na „obyčejné“ metody. Test prázdnosti seznamu je jednoduchý (v pascalských příkladech jej pro úsporu místa ani uvádíme, v C++ jsme ten jeden řádek obětovali). Seznam je prázdný, pokud je prázdný ukazatel na jeho počátek. Trošku obtížnější by to mohlo být se zbylými operacemi. Abychom vám je trochu přiblížili, pokusíme se potřebná přesměrování ukazatelů nakreslit.

Začneme přidáním hodnoty na počátek seznamu. To je poměrně jednoduché. Víme, že hodnotu musíme do seznamu přidat tak, že vytvoříme atom, do nějž hodnotu uložíme, a který bude navíc obsahovat i odkazy na svého předchůdce a popřípadě i následníka (viz obr. na následující straně).

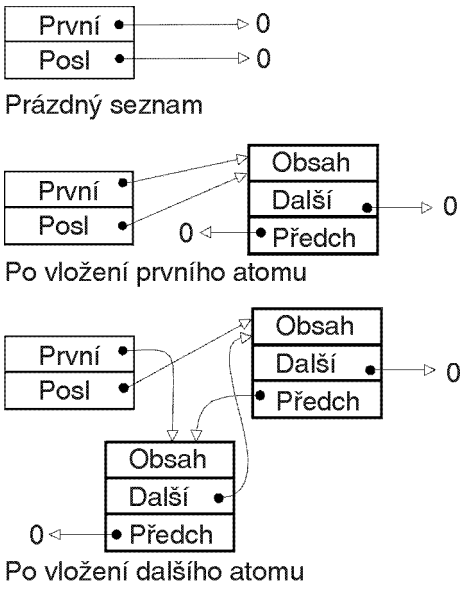
Přidáváme-li atom na počátek seznamu, stane se přidávaný atom novým prvním atomem (hlavou seznamu) a původní hlava bude jeho následovníkem. U obousměrně zřetězeného seznamu však nesmíme zapomenout na to, že pokud přidáváme první atom, bude tento atom nejen prvním, ale také posledním (protože jediným) prvkem seznamu, a že tedy musíme odpovídajícím způsobem modifikovat i hodnotu ukazatele na poslední prvek seznamu.

```
/*  Příklad C7 - 6  */
cSList& /*****/ cSList::PridejPoc /*****/
( tHodn h )
//Přidá hodnotu na počátek jednosměrně zřetězeného seznamu
{
  cSAtom* a = new cSAtom;  //Vytvoř atom pro přidávanou hodnotu
  a->Hodnota = h;          //Ulož do něj přidávanou hodnotu
  a->Dalsi = Prvni;        //Bývalý první bude následníkem
  Prvni = a;               //nového prvního
  return *this;
}/********** cSList::PridejPoc **********/

cDList& /*****/ cDList::PridejPoc /*****/
( tHodn h )
//Přidá hodnotu na počátek obousměrně zřetězeného seznamu
{
  cDAtom* a = new cDAtom;  //Vytvoř atom pro přidávanou hodnotou
  a->Hodnota = h;          //Ulož do něj přidávanou hodnotu
  a->Dalsi = Prvni;        //Bývalý první bude jeho následníkem
  a->Predch = 0;           //Jako první však nebude mít předchůdce,
  Prvni->Predch = a;       //ale bude předchůdcem bývalého prvního
```



Přidání atomu na počátek jednosměrně zřetězeného seznamu



Přidání atomu na počátek obousměrně zřetězeného seznamu

```
  Prvni = a;                //a stane se novým prvním
  if( !Posl )               //Pokud byl seznam dosud prázdný
       Posl = a;            //stane se přidaný atom i posledním
return *this;
}/********** cDList::PridejPoc **********/
```

```
(*  Příklad P7 – 6  *)
procedure (*****) cSList.PridejPoc (*****)
( h : tHodn );
{Přidá hodnotu na počátek jednosměrně zřetězeného seznamu}
var
  a : pSAtom;
begin
  new( a );                 {Vytvoř atom pro přidávanou hodnotu}
  a^.Hodnota := h;          {Ulož do něj přidávanou hodnotu }
  a^.Dalsi := Prvni;        {Bývalý první bude následníkem }
  Prvni := a;               {nového prvního }
end;
(********** cSList.PridejPoc **********)

procedure (*****) cDList.PridejPoc (*****)
( h :tHodn );
{Přidá hodnotu na počátek obousměrně zřetězeného seznamu}
var
  a : pDAtom;
begin
  new( a );                 {Vytvoř atom pro přidávanou hodnotu }
  a^.Hodnota := h;          {Ulož do něj přidávanou hodnotu }
  if( Posl = NIL )then      {Pokud byl seznam dosud prázdný }
  Posl := a;                {stane se nový atom zároveň posledním}
  a^.Dalsi := Prvni;        {Bývalý první bude jeho následníkem }
  a^.Predch := NIL;         {Jako první však nebude mít předchůdce,}
  Prvni^.Predch := a;       {ale bude předchůdcem bývalého prvního }
  Prvni := a;               {a stane se novým prvním }
end;
(********** cDList.PridejPoc **********)
```

Nyní se pokusíme odebrat atom z počátku seznamu. Musíme si dát především pozor na to, abychom odebírali atomy pouze ze seznamu, který nějaké atomy vůbec obsahuje. Pokud je seznam prázdný, musí program vyvolat chybovou situaci. V našem příkladu pouze napíše hlášení na obrazovku a vrátí nulu. V reálných programech však musí být toto ošetření trochu jiné – k tomu se v budoucnu ještě vrátíme.

Vlastní odebrání atomu opět není příliš složité. Nesmíme však atom zrušit dříve, než si někde zapamatujeme adresu jeho následovníka, který se po zrušení daného atomu stane prvním atomem seznamu. V obousměrně zřetězených seznamech musíme navíc myslet i na to, že pokud je rušený atom jediným atomem v seznamu, je nejen atomem prvním, ale zároveň také posledním, a nesmíme proto zapomenout „vyprázdnit“ také ukazatel na poslední prvek.

Samostatné obrázky pro odebrání atomu jsme nekreslili, protože vše potřebné je možno vyčíst z obrázků pro přidání atomu.

```
/*  Příklad C7 – 7  */
tHodn /*****/ cSList::UberPoc /*****/
()
//Vrátí hodnotu odebranou z počátku jednosměrně zřetězeného seznamu
{
  if( !Prvni )             //Pokud je seznam prázdný, není co odebrat
  {
       cerr << "\a\n\nSeznam je prázdný\n\n\a";
       return( 0 );
  }
  cSAtom* a = Prvni; //Adresu bývalého prvního ulož do pomocné proměnné
  Prvni = a->Dalsi;        //Nastav nový první
  tHodn h = a->Hodnota;    //Zapamatuj si hodnotu původního prvního
  delete a;                //Zruš původní první
  return h;                //Vrať zapamatovanou hodnotu
}/********** cSList::UberPoc **********/

tHodn /*****/ cDList::UberPoc /*****/
()
//Vrátí hodnotu odebranou z počátku obousměrně zřetězeného seznamu
{
  if( !Prvni )        //Pokud je seznam prázdný, není co odebrat
  {
       cerr << "\a\n\nSeznam je prázdný\n\n\a";
       return( 0 );
  }
  cDAtom* a = Prvni; //Adresu býval0ho prvního ulož do pomocné proměnné
  Prvni = a->Dalsi;        //Nastav nového prvního
  if( Prvni )
       Prvni->Predch = 0;  //Nebude mít žádného předchůdce
  else                     //Byl jediným atomem v seznamu
  Posl = 0;                //=> byl i poslední a seznam je nyní prázdný
  tHodn h = a->Hodnota;    //Zapamatuj si hodnotu původního prvního
  delete a;                //a zruš jej
  return h;                //Vrať zapamatovanou hodnotu
}/********** cDList::UberPoc_ **********/
```

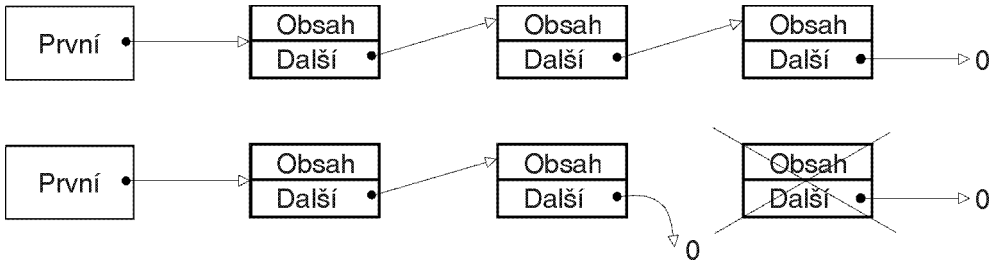
```
(*  Příklad P7 – 7  *)
function (*****) cSList.UberPoc (*****)
: tHodn;
{Vrátí hodnotu odebranou z počátku jednosměrně zřetězeného seznamu}
var
  a : pSAtom;              {Adresa rušeného atomu}
begin
  if( Prazdny )then        {Pokud je seznam prázdný, není co odebrat }
  begin
       System.write( nl, nl, 'Seznam je prázdný', nl, nl );
       UberPoc := 0;
       exit;
  end;
  a := Prvni;   {Adresu bývalého prvního ulož do pomocné proměnné}
  Prvni := a^.Dalsi;       {Nastav nový první }
  UberPoc := a^.Hodnota;   {Budeme vracet hodnotu původního prvního }
  dispose( a );            {Zruš původní první }
end;
(********** cSList.UberPoc **********)
```

```
function (*****) cDList.UberPoc (*****)
: tHodn;
{Vrátí hodnotu odebranou z počátku obousměrně zřetězeného seznamu}
var
   a : pDAtom;
begin
   if( Prazdny )then        {Pokud je seznam prázdný, není co odebrat}
   begin
        System.write( nl, nl, 'Seznam je prázdný', nl, nl );
        UberPoc := 0;
        exit;
   end;
   a := Prvni; {Adresu bývalého prvního ulož do pomocné proměnné}
   Prvni := a^.Dalsi;       {Nastav nového prvního }
   if( Prvni <> NIL )then   {Pokud v seznamu zůstal ještě nějaký atom }
        Prvni^.Predch := NIL {První nemá žádného předchůdce }
   else
        Posl := NIL;        {Rušený atom byl poslední }
   UberPoc := a^.Hodnota;   {Budeme vracet hodnotu původního prvního }
   dispose( a );            {Zruš původní první }
end;
(********** cDList.UberPoc **********)
```

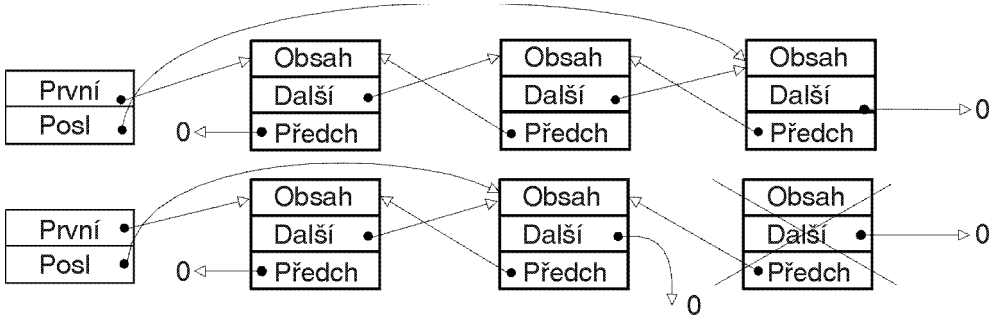
Přidání atomu na konec seznamu je u jednosměrně zřetězeného seznamu poměrně nepříjemná záležitost, protože musíme nejprve projít celým seznamem a najít jeho poslední atom.

U obousměrně zřetězených seznamů nemusíme procházet celý seznam, protože součástí datové struktury je ukazatel na poslední atom a ten zase obsahuje ukazatel na předposlední atom. Stejně jako u přidávání atomu na počátek seznamu však nesmíme zapomenout, že přidáváme-li atom do prázdného seznamu, přidáváme s ním nejen poslední, ale zároveň i první atom, a musíme tedy patřičně modifikovat i druhý z koncových ukazatelů.

Operace potřebné pro přidání atomu na konec seznamu si můžete odvodit z následujících obrázků.



Odebrání atomu z konce jednosměrně zřetězeného seznamu



Odebrání atomu z konce obousměrně zřetězeného seznamu

```
/*  Příklad C7 - 8  */
cSList& /*****/ cSList::PridejKon /*****/
( tHodn h )
//Přidá hodnotu na konec jednosměrně zřetězeného seznamu
{
  cSAtom *a = Prvni;  //Ukazatel na aktuální atom
  cSAtom *p;          //Předchůdce aktuálního atomu
  while( a != 0 )     //Najdeme konec seznamu
  {
       p = a;
       a = a->Dalsi;  //Dokud má atom následníka, není poslední
  }                   //Po skončení cyklu je a == 0
  a = new cSAtom;     //Nový atom
  if( Prvni )         //Seznam není prázdný
       p->Dalsi = a;  //Nový bude následníkem doposud posledního
  else                //Seznam je dosud prázdný
       Prvni = a;     //Nový atom bude zárovň i prvním atomem seznamu
  a->Dalsi = 0;       //Nebude již mít žádného následníka
  a->Hodnota = h;     //Přiřadíme mu požadovanou hodnotu
  return *this;
}/********** cSList::PridejKon **********/

cDList& /*****/ cDList::PridejKon /*****/
( tHodn h )
//Přidá hodnotu na konec obousměrně zřetězeného seznamu
{
  cDAtom* a = new cDAtom;  //Nový atom
  a->Dalsi = 0;            //bude poslední => nebude mít následníka
  a->Predch = Posl;        //Jeho předchůdcem bude bývalý poslední
  if( Posl )               //Seznam není prázdný
       Posl->Dalsi = a;    //Nový bude předchůdcem bývalého posl.
  else                     //Seznam je prázdný
       Prvni = a;          //Nový atom je zároveň prvým atomem sezn.
  a->Hodnota = h;          //Přiřadíme mu požadovanou hodnotu
  Posl = a;                //a nastavíme jej jako nový poslední
  return *this;
```

```
(*  Příklad P7 - 8  *)
procedure (*****) cSList.PridejKon (*****)
( h : tHodn );
{Přidá hodnotu na konec jednosměrně zřetězeného seznamu}
var
   a : pSAtom;              {Ukazatel na aktuální atom }
   p : pSAtom;              {Předchůdce aktuálního atomu}
begin
   a := Prvni;
   while( a <> NIL )do      {Najdeme konec seznamu }
   begin
        p := a;
        a := a^.Dalsi;      {Dokud má atom následníka, není poslední}
   end; {a = 0 }
   new( a );                {Nový atom }
   if( Prvni <> NIL )then   {Pokud seznam nebyl prázdný }
        p^.Dalsi := a       {Bude nový atom následníkem dosud posledního
   }
   else
        Prvni := a;         {Nový atom bude jediným atomem seznamu }
   a^.Dalsi := NIL;         {Poslední atom nemá žádného následníka }
   a^.Hodnota := h;         {Přiřadíme mu požadovanou hodnotu }
end;
(********** cSList.PridejKon **********)

procedure (*****) cDList.PridejKon (*****)
( h : tHodn );
{Přidá hodnotu na konec obousměrně zřetězeného seznamu}
var
   a : pDAtom;
begin
   new( a );     {Nový atom }
   a^.Dalsi := NIL;         {bude poslední :=> nebude mít následníka}
   a^.Predch := Posl;       {Jeho předchůdcem bude bývalý poslední }
   if( Posl <> NIL )then    {Pokud seznam nebyl prázdný }
        Posl^.Dalsi := a    {Bude nový atom předchůdcem dosud posl. }
   else
        Prvni := a;         {Nový atom je jediným atomem seznamu }
   a^.Hodnota := h;         {Přiřadíme mu požadovanou hodnotu }
   Posl := a;               {a nastavíme jej jako nový poslední }
end;
(********** cDList.PridejKon **********)
```

Poslední ze základních operací, které jsme si pro seznam deklarovali, je odebrání atomu z konce seznamu (viz obr. 3 a 4). U jednosměrně zřetězených seznamů je tato operace (stejně jako operace přechozí) spojena s nalezením posledního a předposledního atomu v seznamu. Pak teprve můžeme poslední atom zrušit a původní předposlední atom modifikovat na nový poslední atom. U obousměrně zřetězených seznamů musíme opět ošetřit případ, kdy odebíráme poslední atom seznamu, a seznam tudíž bude prázdný.

```
/*  Příklad C7 - 9  */
tHodn /*****/ cSList::UberKon /*****/
()
//Vrátí hodnotu odebranou z konce jednosměrně zřetězeného seznamu
```

```
{
  if( !Prvni )             //Je-li seznam prázdný - není co odebrat
  {
       cerr << "\a\n\nSeznam je prázdný\n\n\a";
       return( 0 );
  }
  cSAtom *p;               //Předchůdce aktuálního atomu
  cSAtom *a = Prvni;       //Ukazatel na aktuální atom
  while( a->Dalsi != 0 )   //Najdi konec seznamu
  {                   //Poslední atom je ten, který nemá následníka
       p = a;              //p = předchůdce nového aktuálního atomu
       a = a->Dalsi;       //Novým akt. atomem je následník stávajícího
  }
  if( a == Prvni )         //Byl jediným atomem seznamu
       Prvni = 0;          //Seznam nyní bude prázdný
  else                     //Nebyl jediným - jeho předchůdce bude novým
       p->Dalsi = 0;       // posledním => nebude již mít následníka
  tHodn h = a->Hodnota;    //Zapamatuj si hodnotu původního posl.atomu
  delete a;                //Zruš jej
  return h;                //a zapamatovanou hodnotu vrať
}/********** cSList::UberKon **********/

tHodn /*****/ cDList::UberKon /*****/
()
//Vrátí hodnotu odebranou z konce obousměrně zřetězeného seznamu
{
  if( !Prvni )             //Je-li seznam prázdný - není co odebrat
  {
       cerr << "\a\n\nSeznam je prázdný\n\n\a";
       return( 0 );
  }
  cDAtom* a = Posl;        //Ukazatel na rušený atom
  Posl = a->Predch;        //Novým posledním bude jeho předchůdce
  if( Posl )               //Rušený atom nebyl jediným atomem seznamu
       Posl->Dalsi = 0;    //Nový poslední nebude mít následníka
  else                     //Byl jediným
       Prvni = 0;          //Seznam bude nyní prázdný
  tHodn h = a->Hodnota;    //Zapamatuj si hodnotu původního posl.atomu
  delete a;                //Zruš jej
  return h;                //a zapamatovanou hodnotu vrať
}/********** cDList::UberKon **********/
```

```
(*  Příklad P7 – 9  *)
function (*****) cSList.UberKon (*****)
: tHodn;
{Vrátí hodnotu odebranou z konce jednosměrně zřetězeného seznamu}
var
  a : pSAtom;                  {Ukazatel na aktuální atom }
  p : pSAtom;                  {Předchůdce aktuálního atomu}
begin
  if( Prvni = NIL )then    {Je-li seznam prázdný - není co odebrat }
  begin
       System.write( nl, nl, 'Seznam je prázdný', nl, nl );
       UberKon := 0;
       exit;
```

```
  end;
  a := Prvni;                    {Ukazatel na aktuální atom }
  while( a^.Dalsi <> NIL )do     {Najdi konec seznamu }
  begin               {Poslední atom je ten, který nemá následníka}
       p := a;   {p := předchůdce nového aktuálního atomu}
       a := a^.Dalsi;       {Novým akt. atomem je následník stávající}
  end;
  if( a = Prvni )then       {Rušený atom byl jediným atomem seznamu }
       Prvni := NIL
  else                      {Nebyl - jeho předchůdce bude novým }
       p^.Dalsi := NIL;     {posledním :=> nebude již mít následníka}
  UberKon := a^.Hodnota;    {Vrátíme hodnotu původního posled. atomu}
  dispose( a );             {Zruš jej }
end;
(********** cSList.UberKon **********)

function (*****) cDList.UberKon (*****)
: tHodn;
{Vrátí hodnotu odebranou z konce obousměrně zřetězeného seznamu}
var
  a : pDAtom;               {Ukazatel na rušený atom }
begin
  if( Prvni = NIL )then     {Je-li seznam prázdný, není co odebrat}
  begin
       System.write( nl, nl, 'Seznam je prázdný', nl, nl );
       UberKon := 0;
       exit;
  end;
  a := Posl;                {Rušíme poslední atom }
  Posl := a^.Predch;        {Novým posledním bude jeho předchůdce }
  if( Posl <> NIL )then     {Seznam nezůstal prázdný }
       Posl^.Dalsi := NIL   {Nový poslední nebude mít následníka }
  else
       Prvni := NIL;        {Seznam je nyní prázdný }
  UberKon := a^.Hodnota;    {Vrátíme hodnotu původního posledního atomu}
  dispose( a );             {Zruš jej }
end;
(********** cDList.UberKon **********)
```

4.2 Ještě o seznamech

V předchozí podkapitole jsme si ukázali objektovou implementaci seznamů. V této podobě najdete seznamy ve většině učebnic pokročilejšího programování. Pokud se však podíváte na implementaci seznamů v borlandské knihovně kontejnerových tříd (*container class library*) v C++, zjistíte, že ve verzích 3.0 a 3.1 tam jsou seznamy implementovány trochu jinak. (Turbo C++ 1.0 a Borland C++ 2.0 seznamy implementovaly přibližně tak, jak jsme si ukázali, Borland C++ 4.0 a pozdější je implementují pomocí šablon – ty budeme probírat v dalším dílu.) Pokud byste se chtěli začíst do zdrojových textů těchto knihoven, povíme si předem o nejdůležitějších odchylkách.

Vynecháme-li odchylky způsobené využitím dědičnosti, kterou budeme probírat až později, pak u seznamů v nových verzích knihoven především objevíte, že jednotlivé

atomy neobsahují přímo hodnoty, ale pouze ukazatele na ně. O této možnosti jsme však již hovořili, a proto by vás neměla nijak nepřekvapit. Co by vás však překvapit mohlo, to je ukazatel na konec seznamu, který najdete i mezi atributy jednosměrně zřetězeného seznamu.

V borlandských knihovnách není prvek odebírán ze seznamu podle toho, kde leží, ale podle toho, jakou má hodnotu. V obousměrně zřetězeném seznamu dokonce najdete dvě verze těchto metod: jedna hledá odebíranou hodnotu od počátku seznamu a druhá ji hledá od jeho konce. Odebírání s vyhledáváním jsme však v předchozí podkapitole nezavedli proto, že jde vlastně o dvě operace: vyhledání hodnoty a poté její odebrání. Domníváme se, že by to výklad zbytečně zatemňovalo; nyní by však již pro vás neměl být problém si takovéto metody v případě potřeby naprogramovat.

Možná vás překvapí, že v borlandských knihovnách najdete mezi atributy seznamu kromě ukazatelů na začátek a konec seznamu i jeho první a poslední atom. Tyto „atributové“ atomy se však nepoužívají pro uchování hodnot, resp. ukazatelů na ně, ale slouží pouze jako zarážky, označující okraje skutečného seznamu. Zavedení zarážkových krajních atomů mezi atributy totiž zjednodušuje některé metody, a hlavně citelně zjednoduší programování některých operací.

Netroufáme si jednoznačně tvrdit, které řešení je lepší. Klasické řešení (tj. řešení, které jsme implementovali my a které bylo použito ve starších verzích překladačů) je většinou (i když ne vždy) úspornější na paměť a při jednodušších požadavcích na schopnosti seznamů je celkově efektivnější. Řešení z knihoven Borland C++ 3.0 a 3.1 je zase v některých případech rychlejší a snáze se programuje (je tedy menší pravděpodobnost chyby), protože při něm odpadá většina výjimečných situací, které je nutno programovat zvlášť (jde např. o vložení atomu do prázdného obousměrně zřetězeného seznamu). Kromě toho při „novém“ řešení odpadnou i některé problémy, na něž narazíme v současném výkladu. Mohli bychom je tedy označit za výhodnější v případě, kdy naše požadavky na schopnosti seznamů a s nimi spolupracujících tříd budou komplexnější.

My prozatím zůstaneme u naší minulé implementace. K k implementaci, použité v nových verzích borlandské knihovny kontejnerových tříd (tj. k implementaci s krajními nárazníkovými atomy jako atributy seznamu) se vrátíme, až si vyložíme principy dědičnosti a polymorfismu a až si budeme na příkladu borlandské knihovny kontejnerových tříd ukazovat možnou aplikaci těchto principů. (Při té příležitosti si ukážeme, jak je možno obdobnou knihovnu definovat v Pascalu.)

Nyní si dovolíme definici seznamu trochu zesložitit.

Doposud jsme v zájmu maximálního zjednodušení definovali atomy tak, že obsahovaly přímo uloženou hodnotu – v našem případě celé číslo. V praxi se však často používá druhá možnost, tj. atomy, které obsahují pouze ukazatele na objekty. (Stará verze knihovny kontejnerových tříd jazyka C++ nabízela dokonce pouze „ukazatelovou implementaci“; nová verze nabízí pro jistotu obě varianty.) Použití ukazatelů má několik výhod. Jednou z nich je, že objekty, ukládané do seznamu, bývají dost často rozsáhlejší datové struktury, a jejich kopírování by bylo značně neefektivní (může jít i o struktury, které kopírovat nelze – např. soubory). S dalšími výhodami se seznámíme, až budeme probírat dědičnost.

Poznámka:
Zavedením ukazatelů se nám terminologie maličko zkomplikuje. Při „ukazatelové implementaci" totiž mohou být do seznamu uloženy téměř libovolné objekty, přičemž u mnohých z nich nelze dost dobře hovořit o jejich hodnotě (zkuste definovat hodnotu souboru nebo seznamu). V dalším textu proto nebudeme hovořit o hodnotách, ale o objektech zařazovaných do seznamu (a obecně do kontejneru). Položkou seznamu (a obecně kontejneru) pak budeme nazývat uložený objekt, resp. ukazatel na něj (podle implementace). Přitom budeme abstrahovat od toho, zda daný typ kontejneru potřebuje pro zařazení položky nějakého prostředníka (v případě seznamů atom) nebo žádného prostředníka nepotřebuje (klasická pole).

„Ukazatelové řešení" nám kromě flexibility v typech uložených dat umožňuje i to, aby v seznamu (a obecně v libovolném kontejneru) byly uloženy objekty (přesněji ukazatele na objekty) různých typů, a abychom navíc mohli daný objekt bez zbytečného plýtvání pamětí uložit zprostředkovaně (tj. přes ukazatel) do několika seznamů, resp. kontejnerů. Představte si např., že potřebujeme pro každý z nabízených výrobků vytvořit seznamy osob, které si jej zamluvily. Je přece zbytečné ukládat desetkrát adresu osoby, která si zamluvila 10 výrobků.

Pokud připustíme, že odkaz na nějaký objekt může být uložen v několika kontejnerech, musíme si rozvážit, co se má provést s daným objektem ve chvíli, kdy odkaz na něj v kontejneru rušíme. Máme zrušit i daný objekt?

Poznámka:
V borlandské knihovně kontejnerových tříd je tento rys označován jako vlastnictví. Pokud daný kontejner (v našem případě seznam) objekty do něj uložené vlastní, automaticky je při své destrukci ruší. Pokud kontejner uložené objekty nevlastní, ponechává jejich zrušení jiným částem programu.

Při tvorbě knihovny není možné dopředu určit, které z obou řešení je lepší. Nejjednodušší je doplnit metodu, odstraňující danou položku (přesněji ukazatel na uložený objekt) z kontejneru, o parametr, kterým určíme, zda se má s tímto ukazatelem odstranit i objekt, na kterou onen ukazatel ukazuje.

V C++ však před námi ihned vyvstane problém, jak sdělit destruktoru, zda dotyčný kontejner své prvky vlastní či nikoliv. Jak víte, destruktory v C++ žádné parametry mít nesmějí (aby mohly být volány automaticky). Informace o vlastnictví objektů kontejnerem se jim tedy předává tak, že se k atributům seznamu přidá ještě příznak, který určuje, zda má destruktor při rušení ukazatelů na objekty zrušit i objekty, na něž tyto ukazatele ukazují, či zda tyto objekty „zanechá svému osudu".

Tento atribut deklarujeme jako nestatický. Nelze totiž očekávat, že by všechny seznamy své objekty vlastnily, nebo naopak, že by žádný seznam své objekty nevlastnil.

Zavedení atributu s informací o vlastnictví hodnot může být výhodné i v Pascalu. Tam sice destruktorům předávat parametry můžeme (destruktory se musí v Pascalu volat explicitně a není tedy problém jim nějaký ten parametr předat), avšak řešení s dodatečným atributem je přece jenom o něco bezpečnější, tj. méně náchylné k chybám programátora.

Na rozdílnost reakcí v závislosti na vlastnění či nevlastnění objektu seznamem musíme myslet i ve vztahu k návratové hodnotě metody, která ruší položku seznamu. Doposud totiž obě metody odebírající položku ze seznamu vracely její hodnotu. V nové definici budeme muset zařídit, aby metoda vrátila hodnotu rušené položky (tj. ukazatel na objekt) pouze v případě, že odkazovaný objekt nebude rušen, a v opačném v případě aby vrátila prázdný ukazatel.

Vraťme se ale od obecných úvah o vlastnictví objektů kontejnery zpět k seznamům, a to speciálně k seznamům jednosměrně zřetězeným. Jak jste si mohli všimnout, operace s prvky na konci seznamu (jak přidání, tak i rušení) je značně pomalá. Proto je dobré si vždy rozmyslet, zda tyto operace budeme opravdu potřebovat a zda by nebylo výhodnější je vypustit.

Pokud bude výhodné implementovat přidávání položky na konec jednosměrně zřetězeného seznamu (např. pokud budeme prostřednictvím jednosměrně zřetězeného seznamu implementovat frontu), doporučujeme upravit definici seznamu tak, aby obsahoval i ukazatel na poslední atom seznamu, jako jsme to definovali u seznamů zřetězených obousměrně.

Tato úprava zefektivní operaci přidání položky na konec seznamu, avšak nijak nezvýší efektivitu odebrání položky z konce seznamu. Pokud budete odebírat položky z konce seznamu pouze výjimečně, dá se tato nízká efektivita většinou přetrpět. V opačném případě byste si měli rozmyslet, zda by pro vaše účely nebyl výhodnější seznam zřetězený obousměrně.

Předpokládáme, že všechny výše uvedené modifikace, tj. nahrazení hodnoty v atomu ukazatelem na ni, přidání atributu popisujícího vlastnictví hodnot uložených do seznamu a přidání ukazatele na poslední atom jednosměrně zřetězeného seznamu, jsou natolik jednoduché, že si je dokážete připojit k implementaci do seznamů probíraných v předchozím oddílu. (Ostatně najdete je na doprovodné disketě.)

Zde si ukážeme alespoň novou podobu deklarací tříd. Nelekněte se však neznámých přátel, ke kterým se budou třídy v těchto deklaracích hlásit. Se všemi se postupně seznámíte v následující kapitole, věnované iterátorům.

```
/*  Příklad C7 – 10  */
typedef int tObj;           //Typ objektů ukládaných do seznamu
typedef tObj *pObj;          //Ukazatel na objekty ukládané do seznamu

//V následujícím výčtovém typu nabízíme dvě sady identifikátorů
//výčtových konstant. Vyberte si podle toho, které hledisko
//je vám bližší.
enum eMoje {CIZI, MOJE, NASTAVENE, eMoje,
            NEMAZ=0, MAZ };

/*****/ class cSAtom /*****/
{                           //Atom jednosměrně zřetězeného seznamu
  private:
       tObj *Objekt;        //Objekt
       cSAtom *Dalsi;       //Ukazatel na následníka
       friend class cSList;
       friend class cSLIter;
       friend ostream& operator << ( ostream&, const cSList& );
```

```
};
/********** class cSAtom **********/

/*****/ class cDAtom /*****/
{          //Atom obousměrně zřetězeného seznamu
  private:
       tObj *Objekt;             //Objekt atomu
       cDAtom *Dalsi;            //Ukazatel na následníka
       cDAtom *Predch;           //Ukazatel na předchůdce
       friend class cDList;
       friend class cDLIter;
       friend ostream& operator << ( ostream&, const cDList& );
       friend ostream& operator << ( ostream&, const cDLIter& );
};
/********** class cDAtom **********/

/*****/ class cSList /*****/
{                                 //Seznam jednosměrně zřetězených atomů
  public:
       cSList( int Vlastn=CIZI ) : Prvni(0), Posl(0), Vlastnik(Vlastn)
  {}
       ~cSList();
       void PridejPoc( tObj* );
       void PridejKon( tObj* );
       tObj* UberPoc( int Zrusit=NASTAVENE );
       tObj* UberKon( int Zrusit=NASTAVENE );
       int Prazdny() {return( Prvni == 0); }
       int operator!() {return( Prvni == 0); }
       friend ostream& operator << ( ostream&, const cSList& );
  private:
       cSAtom* Prvni;
       cSAtom* Posl;
       int Vlastnik;
//Následující deklarace pouze brání překladači použít
//dané metody v programu
       cSList( const cSList& );
       cSList& operator= ( const cSList& );
       friend class cSLIter;
};
/********** class cSList **********/

/*****/ class cDList /*****/
{          //Seznam obousměrně zřetězených atomů
  public:
       cDList( int Vlastn = 0 ):Prvni(0), Posl(0), Vlastnik(Vlastn) {}
       ~cDList();
       void PridejPoc( tObj* );
       void PridejKon( tObj* );
       tObj* UberPoc( int Zrusit=NASTAVENE );
       tObj* UberKon( int Zrusit=NASTAVENE );
       int Prazdny() {return( Prvni == 0); }
       int operator!() {return( Prvni == 0); }
       friend ostream& operator << ( ostream&, const cDList& );
  private:
       cDAtom* Prvni;
       cDAtom* Posl;
```

```
      int Vlastnik;
//Následující deklarace pouze brání překladači použít
//dané metody v programu
      cDList( const cDList& );
      cDList& operator=( const cDList& );
      friend class cDLIter;
      friend ostream& operator << (ostream&, const cDLIter& );
};
/********** class cDList **********/
```

```
(*  Příklad P7 – 10  *)
const
  nl = #10#13;              {Přechod na novou řádku}
type
  tData = integer;          {Typ hodnoty jednotlivých atomů seznamu}
  pData = ^tData;
  eVlastn = ( CIZI, MOJE, NASTAVENE );

  pSAtom = ^cSAtom;
(*****) cSAtom = object (** Atom jednosměrně zřetězeného seznamu **)
      Data : pData;         {Ukazatel na ukládaný objekt }
      Dalsi : pSAtom;       {Ukazatel na následníka atomu}
  end;
(********** cSAtom=object **********)

  pDAtom = ^cDAtom;
(*****) cDAtom = object (*** Atom obousměrně zřetězeného seznamu ***)
      Data : pData;         {Ukazatel na ukládaný objet }
      Dalsi : pDAtom;       {Ukazatel na následníka atomu}
      Predch : pDAtom;      {Ukazatel na předchůdce atomu}
  end;
(********** cDAtom=object **********)

 (*****) cSList = object (** Seznam jednosměrně zřetězených atomů **)
      Prvni : pSAtom;
      Posl : pSAtom;
      Vlastnik : eVlastn;
      constructor Init;
      constructor InitV( Vl : eVlastn );
      destructor Done;
      procedure PridejPoc( d:pData );
      procedure PridejKon( d:pData );
      function UberPoc : pData;
      function UberKon : pData;
      function UberPocV( Vl : eVlastn ) : pData;
      function UberKonV( Vl : eVlastn ) : pData;
      function Prazdny : boolean;
      procedure writef( var F:Text );
      procedure write;
  end;
(********** class cSList **********)

 (*****) cDList = object (*** Seznam obousměrně zřetězených atomů ***)
      Prvni : pDAtom;
      Posl : pDAtom;
      Vlastnik : eVlastn;
      constructor Init;
```

```
      constructor InitV( Vl : eVlastn );
      destructor Done;
      procedure PridejPoc( d:pData );
      procedure PridejKon( d:pData );
      function UberPoc : pData;
      function UberKon : pData;
      function UberPocV( Vl : eVlastn ) : pData;
      function UberKonV( Vl : eVlastn ) : pData;
      function Prazdny : boolean;
      procedure writef( var F:Text );
      procedure write;
  end;
(********** class cDList **********)
```

Zkuste si do deklarace seznamu přidat také deklarace některých dalších metod. Máme na mysli např. metody, které bychom mohli nazvat *Počátek* a *Konec* a které vracejí odkaz na první a poslední položku seznamu, a metodu *Vyprázdni*, která vyčistí seznam (tj. odstraní z něj veškeré atomy), aniž by jej destruovala. Navíc můžete přidat i metody ubírající ze seznamu prvek s danou hodnotou. Domníváme se, že všechny tyto metody jsou natolik jednoduché, že není potřeba, abychom zde uváděli jejich možné implementace.

Než přejdeme k další kapitole, chtěli bychom vás ještě upozornit na jednu změnu, kterou musíte udělat v testovacím programu. Dosud jsme používali atomy, jež přímo obsahovaly hodnoty. Nebyl proto problém tyto hodnoty v průběhu cyklu generovat a do vytvářených atomů je ukládat. Nyní atomy obsahují pouze ukazatele na hodnoty. Abychom mohli na hodnotu ukázat, musí se jim nejprve někde v paměti vyhradit prostor. V testovacím programu proto nesmíme zapomenout vyhradit nejprve potřebnou paměť pro každou hodnotu, kterou chceme uložit do seznamu, a pak do seznamu vložit ukazatel na tuto paměť. Patřičná část testovacího programu proto v C++ dostane tvar:

```
// ...
case 1:
  S.PridejPoc( new int(i) );
  D.PridejPoc( new int(i*100) );
  i++;
  break;
 // ...
```

a v Pascalu tvar:

```
{...}
1:begin
      new( pi );
      pi^ := i;
      S.PridejPoc( pi );
      new( pi );
      pi^ := i*100;
      D.PridejPoc( pi );
      Inc( i );
  end;
{...}
```

5. Iterátory

V minulé kapitole jsme se již několikrát zmínili, že pro práci se seznamy se často používají tzv. **iterátory**. Mohli bychom je prozatím definovat jako **zobecněné parametry cyklů**. Podívejme se na problém opět z obecnějšího hlediska všech kontejnerů.

Víme, že datové struktury, kterým říkáme kontejnery, nám slouží k tomu, abychom mohli sdružovat skupiny objektů a pracovat s těmito skupinami jako s jediným celkem (objektem). V mnoha případech však s celým kontejnerem nepracujeme a sdružení objektů do jednoho kontejneru nám slouží především tomu, abychom měli všechny objekty snadno dostupné pro operace, které chceme provádět nad jednotlivými objekty.

Typickým příkladem takovéto operace může být např. výpočet aritmetického průměru. Jistě budete souhlasit, že je mnohem snazší spočítat aritmetický průměr čísel, která máme uložena ve vektoru, než počítat aritmetický průměr hodnot roztroušených po jednotlivých proměnných.

Pokud jsme neznali jiné kontejnery než pole, bylo vše jasné. Jakmile jsme potřebovali provést nějakou akci se všemi nějakým způsobem určenými hodnotami pole, naprogramovali jsme cyklus s parametrem, kterým jsme postupně indexovali jednotlivé položky, s nimiž jsme žádané operace prováděli. Jistě by nebylo k zahození, kdybychom měli obdobnou možnost i pro ostatní typy kontejnerů: seznamy, fronty, množiny atd. Kdybychom i pro ně mohli definovat nějaký parametr cyklu, který bychom vhodným způsobem měnili, a získávali tak přístup k jednotlivým hodnotám v kontejneru uloženým.

V klasických seznamech mohl být takovýmto parametrem ukazatel na aktuální atom – nazvěme si tento ukazatel pracovně **kurzor;**. Za pomoci tohoto kurzoru bychom procházeli seznamem a on by nám zprostředkovával přístup k jednotlivým atomům, přičemž přechod na další atom bychom v C++ realizovali výrazem

```
Kurzor = Kurzor -> Dalsi
```

a v Pascalu příkazem

```
Kurzor := Kurzor^.Dalsi
```

Pokud by bylo třeba procházet seznamem odzadu, bylo v obousměrně zřetězených seznamech možno obdobně přesměrovat kurzor na předchůdce označeného atomu. (V jednosměrně zřetězených seznamech by tato operace byla, jak sami jistě odhadnete, značně neefektivní.)

Uvedený postup má ale jednu nevýhodu. Předpokládá totiž, že na všech místech programu, kde budete chtít takovýto kurzor použít, musíte zveřejnit vnitřní konstrukci atomu a zpřístupnit ji tak, aby bylo možno získat adresu jeho následníka či předchůdce. To však není nejlepší postup, neboť se tím zbytečně zvyšuje riziko nekorektního zásahu do dané datové struktury. (Správně by vlastně okolní program neměl o existenci atomů vůbec vědět.)

Z toho totiž plyne, že každý zásah do definice dané datové struktury – v našem případě seznamu – znamená, že musíme projít celý program, vyhledat všechna místa, kde

se s danou datovou strukturou pracuje, a v případě potřeby program upravit, aby respektoval novou podobu datové struktury.

Předchozí nevýhody částečně odstraňuje použití procedury, např.

```
Dalsi( Kurzor )
```

Všimněte si, že ač se pohybujeme po seznamu, samotný seznam mezi parametry této „přesunovací“ procedury být nemusí. Tuto proceduru bychom v případě seznamů mohli v C++ definovat jako spřátelenou s atomy, které by tak mohly své atributy zbytku programu znepřístupnit. V Pascalu bychom však museli za získanou bezpečnost zaplatit ztrátou efektivity, neboť v něm se nemůžeme vyhnout volání procedury a s ním spojené režii. (V C++ lze volání procedury obejít definicí vložené funkce.)

Poznámka:
Pascal sice také nabízí možnost definice vložených funkcí, avšak tyto funkce je třeba programovat ve strojovém kódu, což je horor nejtěžšího kalibru. Programování v asembleru je vedle programování ve strojovém kódu procházka rajskou zahradou.

I modifikace kurzoru prostřednictvím volání procedury (byť vložené) však zůstává stále duchem v klasickém, tj. objektově neorientovaném programování. Jak jsme si řekli na počátku výkladu, přenáší objektově orientované programování značnou část práce a zodpovědnosti směrem k datům. Objektově orientovaným řešením v našem případě bude definice speciálního „kurzorového“ datového typu, jehož instance se budou umět přesouvat po jednotlivých hodnotách uložených v daném kontejneru. Takto definovaný kurzor se nazývá **iterátor** a používá se především jako parametr cyklů, v nichž se postupně zpracovávají jednotlivé položky uložené v kontejneru.

Zamysleme se nyní nad tím, jak bychom takový iterátor co nejlépe definovali. Aby mohl iterátor procházet kontejnerem, musí něco vědět o jeho uspořádání. Příslušný kontejner tedy musí deklarovat „svůj“ iterátor jako svého přítele. (V modifikovaných definicích atomů a seznamů na konci minulé kapitoly jsme se již s těmito deklaracemi přátel setkali.)

V našem případě, tj. v případě seznamu, musí mít iterátor navíc možnost přístupu k jednotlivým atributům atomu. Pokud tedy budou atributy atomu soukromé (v naší upravené poslední definici to tak již je), musí být iterátor deklarován i mezi přáteli atomu. Takto můžeme ošetřit návaznost na nejbližší okolí.

Podívejme se nyní, jaké služby by nám měl iterátor poskytovat. Především se musí umět inicializovat, tj. musí se umět nastavit na nějakou počáteční hodnotu – nejlépe na počátek (nebo třeba také na konec) seznamu. Napadají nás tři možné způsoby inicializace:

- ✧ konstruktorem při definici daného iterátoru,
- ✧ zvláštní metodou,
- ✧ přiřazením hodnoty jiného iterátoru.

Vzhledem k tomu, že kopírovací konstruktor a přiřazovací operátor nám vyhovují ve svých implicitních verzích, stačí definovat konstruktor a inicializační metodu (nazveme

ji *Reset*), jejichž parametrem bude odkaz na seznam, na jehož počátek se daný iterátor nastaví.

Iterátor nám musí umět předat odkazovanou hodnotu. Přesněji řečeno: musí nám umět předat odkaz či ukazatel na tuto hodnotu, abychom ji mohli také změnit. Pro tento účel je možno použít klasickou metodu; nám se ale v C++ zalíbila možnost přetížit operátor volání funkce bez parametrů. V Pascalu nazveme tuto metodu *Hodnota*.

Iterátor musí nabízet operátor inkrementace nebo nějaký jeho ekvivalent. Tento operátor by měl vracet odkaz či ukazatel na uloženou hodnotu a jako vedlejší efekt by měl posunout iterátor na další hodnotu. Často (např. u obousměrně zřetězených seznamů) by se nám hodila i doplňková metoda, při níž se iterátor vrátí na hodnotu předchozí.

Otázkou zůstává, zda se tyto „přesunové“ operátory mají chovat „prefixově“ či „postfixově“, či zda máme naprogramovat obě varianty. Výhodnost jednotlivých řešení již záleží na konkrétní aplikaci. My vám zde ukážeme obě možnosti.

Poznámka:
Při deklaraci operátorů inkrementace a dekrementace jsme si řekli, že by bylo rozumné zavést i v Pascalu nějakou konvenci, kterou bychom v identifikátoru charakterizovali „prefixovost“ či „postfixovost“ definovaného operátoru. Začali jsme používat způsob značení, při němž na ten kraj identifikátoru dané funkce, na jehož straně by u ekvivalentního operátoru v C++ stála inkrementovaná či dekrementovaná proměnná, přidáme znak, který tuto proměnnou zastupuje (většinou x, ale může být i jiný). Operátory s „prefixovým“ chováním, tj. operátory, které nejprve provedou operaci a vrací hodnotu objektu po provedení operace, označíme podle této konvence INCx a DECx, operátory s „postfixovým“ chováním, tj. operátory, které si nejprve připraví hodnotu, kterou budou vracet, a teprve pak provedou žádanou operaci, označíme xINC a xDEC.

Vedle možnosti přesunu iterátoru na další či předchozí položku seznamu bychom také potřebovali umět nějakým způsobem zjistit, zda jsme již vyčerpali všechny hodnoty uložené v kontejneru, takže můžeme cyklus ukončit.

V C++ toho můžeme dosáhnout dvěma způsoby. Pokud bude „přesunovací“ operátor vracet ukazatel na uloženou hodnotu, může v případě vyčerpání kontejneru vrátit prázdný ukazatel. Pokud bude vracet referenci na uloženou hodnotu, musíme definovat ještě speciální „nesmyslnou“ hodnotu, na níž předá odkaz v případě, že vyčerpal kontejner. To se ovšem nemusí vždy hodit, protože hodnoty ukládané do kontejneru mohou být paměťově náročné.

Oba dva typy vracených hodnot (tj. ukazatele a reference) mají své výhody a nevýhody. Pro nás je ovšem rozhodnutí jednoduché. Protože bychom rádi výklad obou jazyků maximálně sjednotili a protože pascalské funkce umějí vracet pouze ukazatele, zvolíme v obou jazycích prvou z uvedených alternativ. Přesunovací operátor tedy bude vracet ukazatel.

Nezávisle na typu hodnoty vracené inkrementačním operátorem bývá někdy výhodné mít k dispozici i metodu, která nám podá informaci o stavu vyčerpanosti kontejneru. Borlandské knihovny skladových tříd pro tento účel definují operátor přetypování na **int**, jenž v případě vyčerpanosti seznamu vrátí nulovou hodnotu a v opačném případě

hodnotu nenulovou. Domníváme se však, že byste si tuto metodu dokázali bez obtíží naprogramovat sami, a proto v naší definici chybí. Místo ní používáme zjišťování odkazu na aktuální hodnotu.

S přesuny a zjišťováním vyčerpanosti seznamu souvisí ještě jedna otázka: Budeme potřebovat znát polohu iterátoru vůči seznamu i poté, kdy seznam vyčerpáme? Není tak zcela nesmyslné hovořit o tom, že iterátor ukazuje na fiktivní hodnotu, která leží za poslední nebo před první hodnotou seznamu. V takovém případě ale musíme umět rozlišit neinicializovaný iterátor od iterátoru ukazujícího mimo seznam (tj. od iterátoru ukazujícího na onu výše zmíněnou fiktivní hodnotu) nebo zakázat možnost definice neinicializovaného iterátoru.

Pokud chceme povolit obě „přespočetné“ polohy iterátoru, musíme navíc umět rozlišit, zda iterátor právě ukazuje před seznam nebo za něj. Jinými slovy: nestačí nám pro to jedna hodnota (např. nulový, prázdný ukazatel).

Pokud použijeme implementaci se zarážkovými atomy jako atributy seznamu, můžeme obě krajní polohy rozlišit poměrně snadno, neboť v tomto případě určují krajní polohy právě ukazatele na ony zarážky a neinicializovaný iterátor může symbolizovat např. prázdný ukazatel.

Pokud budeme chtít zůstat u naší jednodušší implementace bez zarážek, musíme se smířit s tím, že budeme moci používat pouze jednu krajní polohu, a z toho ovšem zákonitě plyne, že plnohodnotný bude také pouze jeden směr pohybu iterátoru po seznamu. Onu jedinou „přespočetnou“ hodnotu pak můžeme reprezentovat prázdným ukazatelem.

V některých aplikacích bychom potřebovali mít možnost přidat a/nebo odebrat hodnotu v místě, na něž iterátor ukazuje. Každý z předchozích dvou požadavků má však malý háček.

Při vkládání hodnoty si musíme ujasnit, zda budeme chtít danou hodnotu (v případě seznamů nový atom, jehož některá složka bude na tuto hodnotu odkazovat) přidat před hodnotu (atom), na kterou ukazuje iterátor, nebo za ni. Při tomto rozhodování však navíc musíme vzít v úvahu i to, zda budeme chtít mít možnost vkládat nové hodnoty do seznamu i v případě, kdy z něj již iterátor „vyběhne“. Pak bychom totiž museli zajistit, aby bylo možno vkládat pouze **před** iterátor vyběhnuvší za seznam, nebo pouze **za** iterátor vyběhnuvší před seznam.

Při odebírání hodnoty (a s ní i atomu, který na ni odkazuje) ze seznamu je situace na první pohled jednodušší, protože můžeme opravdu odebrat právě tu hodnotu (ten atom), na níž iterátor ukazuje (samozřejmě s výjimkou fiktivní hodnoty za koncem, resp. před počátkem seznamu.) V případě seznamů to však můžeme provést „snadno“ pouze u seznamů zřetězených obousměrně. Existuje i řešení, kterým se realizace dané operace výrazně zjednoduší i pro jednosměrně zřetězené seznamy – ale to si spolu s některými dalšími problémy jednosměrně zřetězených seznamů a některými specialitami iterátoru jednoduše zřetězeného seznamu necháme na později.

Poslední metodou, kterou bychom měli pro iterátory definovat, je nějaká forma výstupu. Při ladění programu totiž často potřebujeme vědět, kam náš iterátor v daném okamžiku právě ukazuje. Jednou z možností (i když ne vždy nejlepší) je vytisknout se-

znam, po němž iterátor přebíhá, a v tomto seznamu nějakým výrazným způsobem označit hodnotu, na kterou iterátor právě ukazuje.

Předchozí úvahy vedou k následující definici iterátorů pro dvojitě zřetězený seznam:

```
/*  Příklad C8 – 1  */
/*****/ class cDLIter /*****/
{
  public:
       cDLIter();
       cDLIter( cDList& L );
//Kopírovací konstruktor nám vyhovuje v implicitní verzi.
//Stejně tak destruktor a operátor přiřazení
       void Reset( cDList& L );
       pData operator () ();
       pData operator ++ ();
       pData operator ++ (int);
       void operator -- ();
       int operator == ( cDLIter i );
       int operator != ( cDLIter i );
       void PredVloz ( pData );
       void Zrus ( int Vlastn=CIZI );
       friend ostream& operator << ( ostream&, const cDLIter& );
  private:
       cDList *Seznam;
       cDAtom *Kurzor;
};
/********** class cDLIter **********/
```

```
(*  Příklad P8 – 1  *)
type pDList = ^cDList;
type (*****) cDLIter (*****)     = object
  Seznam : pDList;
  Kurzor : pDAtom;
  constructor Init;
  constructor InitList( var L : cDList );
  procedure Reset( var L : cDList );
  function Hodnota : pData;
  function INCc : pData;   {"Prefixová" verze }
  function iINC : pData;   {"Postfixová" verze}
  procedure DECc;     {"Prefixová" verze }
  function Rovno( i : cDLIter ) : Boolean;
  function Ruzne( i : cDLIter ) : Boolean;
  procedure PredVloz( d : pData );
  procedure Zrus( Vlastn : eVlastn );
  procedure WriteF( var F:Text );
  procedure Write;
end;
(********** class cDLIter **********)
```

A nyní vám opět jednou navrhneme, abyste se pokusili naprogramovat těla deklarovaných metod sami, a pak je prověřili v příkladu, který bude dostatečně jednoduchý, ale nebude zase zcela triviální. Zkuste např. definovat proceduru, která setřídí seznam. Můžete použít metodu, která bude vycházet z třídění polí pomocí přímého výběru. Abyste

se měli ve svých úvahách o co opřít, ukažme si, jak bychom tuto metodu definovali pro pole celých čísel:

```
/*  Příklad C8 — 2  */
void /*****/ Setrid /*****/
( int A[], int N )
{
  for( int i=0; i < N-1; i++ )   //Cyklus přes všechny prvky
  {                              // s výjimkou posledního
       int min = A[i];           //Předpokládáme, že aktuální prvek má
       int imin = i;             //ve zbytku pole nejmenší hodnotu
       for( int j=i+1; j < N; j++ )         //Ověříme předpoklad
        if( A[j] < min )         //j-tý prvek je zatím nejmenší
       min = A[ (imin = j) ];    //Zapamatujeme si index i hodnotu
       if( i != imin )           //Pokud náš původní předpoklad
       {                         //nevyšel,
        A[imin] = A[i];          //vyměníme nejmenší prvek s i-tým
        A[i] = min;
       }                         //Prvních i prvků je nyní
  }    //for i                   //setříděno
}/********** Setrid **********/
```

```
(*  Příklad P8 — 2  *)
type ARRAY100INT = array[ 0..100 ] of Integer;
procedure (*****) SetridPole (*****)
( var A : ARRAY100INT; N : Integer );
var i,j,min,imin : Integer;
begin
  for i := 0 to N-2 do           {Cyklus přes všechny prvky }
  begin                          {s výjimkou posledního }
       min := A[i];              {Předpokládáme, že aktuální prvek má }
       imin := i;                {ve zbytku pole nejmenší hodnotu }
       for j := i+1 to N-1 do    {Ověříme předpoklad }
       if( A[j] < min )then      {j-tý prvek je zatím nejmenší }
       begin
         imin := j;
         min := A[ imin ];       {Zapamatujeme si index i hodnotu }
       end;
       if( i<>imin )then         {Pokud náš původní předpoklad }
       begin                     {nevyšel }
         A[imin] := A[i];        {Vyměníme nejmenší prvek s i-tým }
         A[i] := min;
       end;                      {Prvních i prvků je nyní }
  end;     {for i}               {setříděno }
end;
(********** Setrid **********)
```

V předchozím příkladu se po nalezení nejmenšího prvku v dosud nesetříděné části pole prohodí hodnota tohoto prvku s hodnotou prvního prvku nesetříděné části. Nejlepším řešením pro seznamy bude asi procedura, která prohodí ukazatele na hodnoty v obou atomech. Takovouto metodu jistě dokážete snadno naprogramovat; zkuste ale z cvičných důvodů zaměnit tyto hodnoty pomocí operací vložení hodnoty do seznamu a jejího vy-

jmutí ven. (Alespoň se přesvědčíte o správnosti definic těchto metod.) Možnou podobu těchto metod najdete dále.

K testování svých programů můžete použít následující proceduru:

```
/*  Příklad C8 – 3  */
void /*****/ TestIter /*****/
()
{
   static int P[] = {5, 3, 2, 1, 4 };
   static int N = 5;
   int i;
   int a[10];
   cout << "\n\nPole před setříděním: ";
   for( i=0; i < N; i++ )
   cout << " " << (a[i] = P[i]);
   Setrid( a, N );
   cout << "\nPole po setřídění: ";
   for( i=0; i < N; i++ )
   cout << " " << a[i];
   cDList S;
   for( i=0; i < N; i++ )
   S.PridejKon( &P[i] );
   cout << "\n\n\nSeznam před tříděním: " << S << "\n";
   Setrid( S );
   cout << "\nSeznam po setřídění: " << S;
}/********** TestIter **********/
```

```
(*  Příklad P8 – 3  *)
const P :
   array[ 0..4 ] of Integer = ( 5, 3, 2, 1, 4 );
   N : Integer = 5;
var
   i : Integer;
   a : ARRAY100INT;
   S : cDList;

procedure (*****) TestIter (*****)
;
begin
   Writeln;
   Writeln;
   Write('Pole před setříděním: ');
   for i := 0 to N-1 do
   begin
        a[i] := P[i];
        Write( ' ', a[i]);
   end;
   Writeln;
   SetridPole( a, N );
   Write('Pole po setřídění: ');
   for i:=0 to N-1 do
        Write( ' ', a[i] );
   Writeln;
   S.Init;
```

```
  for i := 0 to N-1 do
      S.PridejKon( addr(P[i]) );
  Writeln;
  Writeln;
  Write('Seznam před tříděním: '); S.Write; Writeln;
  SetridSeznam( S );
  Writeln;
  Write('Seznam po setřídění: '); S.Write; Writeln;
end;
(********** TestIter **********)
```

Nyní si ukážeme, jak by mohly vypadat definice třídění seznamů:

```
/*  Příklad C8 — 4  */
void /*****/ Setrid /*****/
( cDList& S )
{
  for( cDLIter i(S); i(); i++ )  //Cyklus přes všechny prvky
  {                              // s výjimkou posledního
      tData* min = i();          //Předpokádáme, že aktuální prvek má
      cDLIter imin = i;          //ve zbytku pole nejmenší hodnotu
      for( cDLIter j=i; ++j; )   //Ověříme předpoklad
        if( *j() < *min )        //j-tý prvek je zatím nejmenší
          min = (imin=j)();      //Zapamatujeme si index i hodnotu
      if( i != imin )            //Pokud náš původní předpoklad nevyšel
      {//Prohod( i, imin );      //Vyměníme nejmenší prvek s i-tým
       i.PredVloz( imin() ); cout << "i+ " << i;
       imin.PredVloz( i() ); cout << "imin+ " << imin;
       j = i;
       --i;
       j.Zrus(); cout << "i- " << i;
       imin.Zrus(); cout << "----- " << S << "\n";
       x();
      }
  }   //for i                    //Prvních i prvků je nyní setříděno
}/********** Setrid **********/
```

```
(*  Příklad P8 — 4  *)
procedure (*****) SetridSeznam (*****)
( var S : cDList );
var
  i,j,imin : cDLIter;
  min : pData;
begin
  i.InitList( S );
  j.Init;
  imin.Init;
  while( i.Hodnota<>nil ) do
  begin
      min := i.Hodnota;          {Předpokládáme, že aktuální prvek má }
      imin := i;                 {ve zbytku pole nejmenší hodnotu }
      j := i;
      j.INCc;
      while( j.Hodnota<>nil ) do
```

```
      begin
        if( j.Hodnota^ < min^ )then {j-tý prvek je zatím nejmenší }
        begin
         imin := j;             {Zapamatujeme si index i hodnotu }
         min := imin.Hodnota;
        end;
        j.INCc;
      end;

      if( i.Ruzne(imin) )then {Pokud náš původní předpoklad nevyšel }
      begin
        i.PredVloz( imin.Hodnota ); Write('i+ '); i.Write;
        imin.PredVloz( i.Hodnota ); Write('imin+ '); imin.Write;
        j := i;
        i.DECc;
        j.Zrus(CIZI); Write('i- '); i.Write;
        imin.Zrus(CIZI); Write('----- '); S.Write; Writeln;
      end;
      i.iINC;
  end;
end;
(********** SetridSeznam **********)
```

Jak jste si jistě všimli, mezi byl atributy iterátoru také odkaz na seznam, po jehož položkách se iterátor pohybuje. Tento atribut potřebujeme, abychom mohli iterátor vybavit metodami pro přidání a odebrání prvku, přesněji abychom v těchto metodách mohli v případě, kdy bychom přidávali nebo odebírali první či poslední položku v seznamu, modifikovat také vnitřní proměnné seznamu, které ukazují na první a poslední atom. Prázdnost ukazatele na seznam, po němž se bude iterátor pohybovat, nám může zároveň signalizovat neinicializovanost iterátoru.

Kdybychom v seznamech používali zarážek na obou koncích, nebyl by tento dodatečný atribut iterátoru nutný. K tomu se ale vrátíme později.

Možná, že vás napadá, proč jsme hned neimplementovali seznam se zarážkami jako atributy. Odpověď je jednoduchá: Kdybychom si nejprve neprošli různá úskalí klasické koncepce, mohla by se vám zdát implementace se zarážkami zbytečně složitá, stejně jako se kdysi zdála nám. Kromě toho v mnoha aplikacích naše klasická jednoduchá implementace bohatě stačí, a často je i efektivnější. Znovu opakujeme, že to, která z výše zmíněných dvou koncepcí je lepší, záleží na konkrétní aplikaci.

Nyní se konečně dostaneme k implementaci metod iterátoru[12].

```
/*  Příklad C8 – 5  */
ostream& /*****/ operator << /*****/
( ostream& o, const cDLIter& i )
//Pomocný operátor výstupu pro ladění programů s iterátory
{
  cDAtom* a = i.Seznam->Prvni;   //Ukazatel na aktuální atom
  o << "( ";                    //Úvodní otevírací závorka
  while( a != 0 )               //Cyklus přes všechny atomy
```

[12] O makru *assert* viz kapitola 7.

```
  {
       if( a == i.Kurzor )       //Atom, na který iterátor ukazuje,
       o << "-->";               // označ šipkou
       o << *a->Data;            //Vytiskni hodnotu atomu
       a = a->Dalsi;             //Nastav ukazatel na následníka
       if( a )                   //Pokud nějaký následník existuje,
       o << ", ";                //odděl jej
  }
  o << " )\n";                   //Závěrečná zavírací závorka
  return o;
}/********** operator << **********/

inline cDLIter::cDLIter()
: Seznam(0), Kurzor(0) {}
inline cDLIter::cDLIter( cDList& L )
  {Reset( L ); }
inline void cDLIter::Reset( cDList& L )
  {Seznam = &L; Kurzor = L.Prvni; }
inline int cDLIter::operator == ( cDLIter i )
  {return (Seznam == i.Seznam) && (Kurzor == i.Kurzor); }
inline int cDLIter::operator != ( cDLIter i )
  {return !(*this == i); }

pData /*****/ cDLIter::operator () /*****/
()
//Operátor volání funkce slouží k předání hodnoty označené iterátorem
{
  assert( Seznam != 0 );    //Iterátor musí být již inicializován
  if( Kurzor != 0 )         //Iterátor někam ukazuje
       return Kurzor->Data;
  else
       return 0;            //Seznam je vyčerpán
}/********** cDLIter::operator ++ **********/

tData* /*****/ cDLIter::operator ++ /*****/
()
//Prefixová verze operátoru inkrementace
{
  assert( (Seznam != 0) ); //Iterátor musí být již inicializován
  if( (Kurzor != 0) &&     //Pokud ještě nebyl seznam vyčerpán
       ((Kurzor = Kurzor->Dalsi) != 0) )
       return Kurzor->Data;
  else
       return 0;            //Seznam je vyčerpán
}/********** cDLIter::operator ++ **********/

tData* /*****/ cDLIter::operator ++ /*****/
( int )
//Postfixová verze operátoru inkrementace
{
  assert( (Seznam != 0) ); //Iterátor musí být již inicializován
  if( Kurzor != 0 )
  {
       tData* pom = Kurzor->Data;
       Kurzor = Kurzor->Dalsi;
       return pom;
  }
```

```
  else
       return 0;
}/********** cDLIter::operator ++ **********/

void /*****/ cDLIter::operator -- /*****/
()
//Operátor dekrementace - slouží pouze jako pomocný
{
  assert( (Seznam != 0) && (Kurzor != Seznam->Prvni) );
//Iterátor musí být již inicializován a nesmí ukazovat na počátek
  if( Kurzor != 0 )
       Kurzor = Kurzor->Predch;
  else
       Kurzor = Seznam->Posl;
}/********** cDLIter::operator -- **********/

void /*****/ cDLIter::PredVloz /*****/
( tData *d )
//Zařadí do seznamu objekt d před položku, na níž ukazuje iterátor
{
  assert( (Seznam != 0) ); //Iterátor musí být inicializován
  cDAtom *a = new cDAtom;
  a ->Data = d;
  a ->Dalsi = Kurzor;
  if( Kurzor == 0 )        //Iterátor ukazuje za konec seznamu
  {
       a ->Predch = Seznam->Posl; //Přidání na konec seznamu
       Seznam->Posl->Dalsi = a;
       Seznam->Posl = a;
  }
  else
  {
       a ->Predch = Kurzor->Predch;
       Kurzor ->Predch = a;
       if( Kurzor == Seznam->Prvni )
         Seznam->Prvni = a;
       else
         a->Predch->Dalsi = a;
  }
  if( Kurzor == Seznam->Prvni )
       Seznam->Prvni = a;
}/********** cDLIter::PredVloz **********/

void /*****/ cDLIter::Zrus /*****/
( int Vlastn )
//Vyjme ze seznamu hodnotu, na kterou ukazuje iterátor, a pokud je
//parametr Vlastn nenulový, smaže ji. Po odebrání hodnoty ukazuje na
//následující
{
  assert((Seznam != 0) && (Kurzor != 0));
//Iterátor musí být již inicializován a musí někam ukazovat
  if( Kurzor == Seznam->Prvni )
       Seznam->Prvni = Kurzor->Dalsi;
  else
       Kurzor->Predch->Dalsi = Kurzor->Dalsi;
  if( Kurzor == Seznam->Posl )
       Seznam->Posl = Kurzor->Predch;
```

```
  else
       Kurzor->Dalsi->Predch = Kurzor->Predch;
  if( Vlastn == NASTAVENE )
       Vlastn = Seznam.Vlastnik;
  if( Vlastn == MOJE )
       delete Kurzor->Data;
  delete Kurzor;
}/********** cDLIter::Zrus **********/
```

```
(*  Příklad P8 — 5  *)
procedure (*****) cDLIter.WriteF (*****)
( var F : text );
{ Pomocný operátor výstupu pro ladění programů s iterátory }
var a : pDAtom;
begin
  a := Seznam^.Prvni;            {Ukazatel na aktuální atom }
  system.Write( F, '(' );        {Úvodní otevírací závorka }
  while( a<>nil ) do             {Cyklus přes všechny atomy }
  begin
       if( a=Kurzor )then        {Atom, na který iterátor ukazuje, }
         system.Write( F, '-->' );          {označ šipkou }
       system.Write(F, (a^.Data)^ );        {Vytiskni hodnotu atomu }
       a := a^.Dalsi;            {Nastav ukazatel na následníka }
       if( a<>nil )then          {Pokud nějaký následník existuje, }
         system.Write( F, ', ' );           {odděl jej }
  end;
  system.Writeln( F, ' )' );     {Závěrečná zavírací závorka }
end;
(********** cDIter.WriteF **********)

procedure (*****) cDLIter.Write (*****)
;
begin
  WriteF( OutPut );
end;
(********** cDIter.Write **********)

constructor (*****) cDLIter.Init (*****)
;
begin
  Seznam := nil;
  Kurzor := nil;
end;

constructor (*****) cDLIter.InitList (*****)
( var L : cDList );
begin
  Reset( L );
end;

procedure (*****) cDLIter.Reset (*****)
( var L : cDList );
begin
  Seznam := addr(L);
  Kurzor := L.Prvni;
end;
```

```
function (*****) cDLIter.Rovno (*****)
( i : cDLIter ) : Boolean;
begin
  Rovno := (Seznam=i.Seznam) and (Kurzor=i.Kurzor);
end;

function (*****) cDLIter.Ruzne (*****)
( i : cDLIter ) : Boolean;
begin
  Ruzne := not Rovno(i);
end;

function (*****) cDLIter.Hodnota (*****)
: pData;
{ Metoda slouží k předání hodnoty označené iterátorem }
begin
  assert( Seznam<>nil );         {Iterátor musí být již inicializován }
  if( Kurzor<>nil )then          {Iterátor někam ukazuje }
       Hodnota := Kurzor^.Data
  else
       Hodnota := nil;           {Seznam je vyčerpán }
end;
(********** cDLIter::operator ++ **********)

function (*****) cDLIter.INCc (*****)
: pData;
{ Prefixová verze operátoru inkrementace }
begin
  assert( Seznam<>nil );         {Iterátor musí být již inicializován }
  if( (Kurzor<>nil) )then
  begin
       Kurzor := Kurzor^.Dalsi;
       if( Kurzor<>nil )then     {Pokud ještě nebyl seznam vyčerpán }
        begin
         INCc := Kurzor^.Data;
         exit;
       end;
  end;
  INCc := nil;                   {Seznam je vyčerpán }
end;
(********** cDLIter.INCc **********)

function (*****) cDLIter.iINC (*****)
: pData;
{ Postfixová verze operátoru inkrementace }
var pom : pData;
begin
  assert( Seznam<>nil );         {Iterátor musí být již inicializován }
  if( Kurzor<>nil )then
  begin
       pom := Kurzor^.Data;
       Kurzor := Kurzor^.Dalsi;
       iINC := pom;
       exit;
  end;
  iINC := nil;
end;
```

```
(********** cDLIter.iINC **********)

procedure (*****) cDLIter.DECc (*****)
;
{ Operátor dekrementace - slouží pouze jako pomocný }
begin
  assert( (Seznam<>nil) and (Kurzor<>Seznam^.Prvni) );
{Iterátor musí být již inicializován a nesmí ukazovat na počátek }
  if( Kurzor<>nil )then
       Kurzor := Kurzor^.Predch
  else
       Kurzor := Seznam^.Posl;
end;
(********** cDLIter.DECc **********)

procedure (*****) cDLIter.PredVloz (*****)
( d : pData );
var a : pDAtom;
{ Zařadí do seznamu objekt d před položku, na níž ukazuje iterátor }
begin
  assert( Seznam<>nil );         {Iterátor musí být inicializován }
  new(a);
  a^.Data := d;
  a^.Dalsi := Kurzor;
  if( Kurzor=nil )then           {Iterátor ukazuje za konec seznamu }
  begin
       a^ .Predch := Seznam^.Posl;          {Přidej na konec seznamu }
       Seznam^.Posl^.Dalsi := a;
       Seznam^.Posl := a;
  end
  else
  begin
       a^.Predch := Kurzor^.Predch;
       Kurzor^.Predch := a;
       if( Kurzor = Seznam^.Prvni )then
         Seznam^.Prvni := a
       else
         a^.Predch^.Dalsi := a;
  end;
  if( Kurzor = Seznam^.Prvni )then
       Seznam^.Prvni := a;
end;
(********** cDLIter::PredVloz **********)

procedure (*****) cDLIter.Zrus (*****)
( Vlastn : eVlastn );
{ Vyjme ze seznamu hodnotu, na kterou ukazuje iterátor, a pokud je
  parametr smazat nenulový, smaže ji. Po odebrání hodnoty ukazuje
  na následující.
}
begin
  assert( (Seznam<>nil) and (Kurzor<>nil) );
{Iterátor musí být již inicializován a musí někam ukazovat }
  if( Kurzor = Seznam^.Prvni )then
       Seznam^.Prvni := Kurzor^.Dalsi
  else
```

```
      Kurzor^.Predch^.Dalsi := Kurzor^.Dalsi;
  if( Kurzor = Seznam^.Posl )then
      Seznam^.Posl := Kurzor^.Predch
  else
      Kurzor^.Dalsi^.Predch := Kurzor^.Predch;
  if( Vlastn = NASTAVENE )then
      Vlastn := Seznam^.Vlastnik;
  if( Vlastn = MOJE )then
      dispose( Kurzor^.Data );
  dispose( Kurzor );
end;
(********** cDLIter.Zrus **********)
```

V našem povídání jsme se zatím zabývali iterátory pro obousměrně zřetězené seznamy, které jsou sice náročnější na paměť, ale zato nám umožňují realizovat řadu operací bez ztráty efektivity. Nyní se pokusme vysledovat odlišnosti, s nimiž se setkáme v případě iterátorů pro jednosměrně zřetězené seznamy.

První zvláštností, na kterou bychom vás chtěli upozornit, je přidávání položky do seznamu. U jednosměrně zřetězeného seznamu je totiž naše rozhodování o trochu komplikovanější, neboť vložení nové položky (nového atomu) před položku (atom), na níž ukazuje iterátor, je časově poměrně náročná operace. Musíme totiž projít seznam a najít předchůdce daného atomu, abychom mohli modifikovat jeho atribut ukazující na následníka. Proto je pravděpodobné, že dáme přednost vkládání nového atomu za atom označený iterátorem.

Pokud však budeme chtít vkládat novou položku až za položku označenou iterátorem, nesmíme považovat hodnotu iterátoru „vyběhnuvšího“ ze seznamu za plnohodnotnou, protože vkládání nové položky za fiktivní položku za koncem seznamu je nutně nekorektní operace. Buďto bychom tedy museli vkládat položku před položku označenou iterátorem nebo se musíme smířit s neplnoprávností vyběhlého iterátoru. Ať se rozhodneme jakkoliv, vždycky nás čekají nějaké nepříjemnosti. (Za chviličku si ukážeme, jak z toho ven.)

Do problémů s efektivitou se dostaneme i u odebírání položek ze seznamu. U jednosměrně zřetězených seznamů nás totiž při odebírání položky, na níž ukazuje iterátor, opět čeká procházení seznamu a hledání předchůdce odebírané položky. Proto se pro jednosměrně zřetězené seznamy tato operace někdy v zájmu zvýšení efektivity definuje jako operace odebrání položky za položkou označenou iterátorem. V tu chvíli ovšem zákonitě vyvstává otázka, jak odebrat první prvek.

Na první pohled se před námi rýsují dvě možnosti: buď se smíříme se sníženou efektivitou a budeme, byť neefektivně, odebírat přímo označenou položku, nebo budeme chtít využít vyšší efektivnosti odebírání následného prvku a seznam doplníme nějakým hluchým prvním prvkem, za kterým bude teprve následovat skutečný první prvek seznamu.

Možná, že vás napadla ještě třetí možnost, a to definovat iterátor jednosměrně zřetězeného seznamu tak, že si bude kromě adresy atomu, na který ukazuje, pamatovat jen tak pro sebe i adresu jeho předchůdce. (Nazvěme si tento atribut *Predch*.) Pak by bylo

možno bezproblémově jak mazat položku označenou iterátorem, tak vkládat před tuto položku položky další.

Pokud bychom implementovali iterátory s atributem *Predch*, ukazujícím na atom předcházející atomu odkazovanému atributem *Kurzor*, mohli bychom bez problému pracovat i s iterátorem ukazujícím na onu fiktivní položku za poslední položkou seznamu, protože tuto hodnotu iterátoru bychom reprezentovali prázdným ukazatelem v atributu *Kurzor* a ukazatelem na poslední atom seznamu v atributu *Predch*. Sami si asi jistě domyslíte, že při nastavení iterátoru na počátek seznamu by to bylo právě obráceně: ukazatel na předchůdce aktuálního atomu (atribut *Predch*) by byl prázdný a ukazatel na aktuální atom (atribut *Kurzor*) by ukazoval na první atom seznamu. (Protože jednosměrně zřetězeným seznamem probíháme pouze dopředu, nemusíme o iterátoru ukazujícím před seznam vůbec uvažovat.)

Tato implementace, tj. implementace s dodatečným atributem, je za určitých podmínek výhodná. Většina problémů se sníženou efektivitou na první pohled odpadne. Objeví se však některé problémy nové.

Hlavním problémem, který před námi při naší rozšířené implementaci iterátoru vyvstane, je problém synchronizace. Pokud totiž ukazují dva iterátory na tutéž položku a jeden z nich před tuto položku vloží položku další, přestane být hodnota atributu *Predch* u druhého iterátoru platná, a bude ji potřeba na počátku metod, které ji využívají, **synchronizovat** se skutečností.

Na první pohled by se mohlo zdát, že můžeme daný iterátor synchronizovat tak, že by se atribut *Predch* pouze přesunul přes vložené položky. Toho můžeme dosáhnout tak, že do metod, vyžadujících synchronizovaný iterátor, vložíme v C++ příkaz

```
if( Kurzor != Seznam->Prvni )
  while( Predch->Dalsi != Kurzor )
       Predch = Predch->Dalsi;
```

V Pascalu bude tento příkaz mít podobu

```
if( Kurzor <> Seznam^.Prvni )then
  while( Predch^.Dalsi <> Kurzor )do
       Predch := Predch^.Dalsi;
```

To by však nesměla existovat možnost, že bychom předchozí položku ze seznamu odebrali. Pak by totiž mohl program při plnění předchozího příkazu vzít místo platné adresy nějaké „smetí“ a bloudit pak bezcílně pamětí, dokud by nám při čekání na jeho výsledky nedošla trpělivost.

Pokud tedy z povahy aplikace musíme připustit, že smažeme položku před iterátorem, musíme hodnotu atributu *Predch* synchronizovat tak, že pokorně začneme u počátku seznamu a hledáme atom, na nějž ukazuje atribut *Kurzor* – stejně, jako bychom jej hledali v případě, kdybychom si v iterátoru žádnou adresu předchozího atomu nepamatovali.

To však ještě nestačí. Pokud budeme ochotni podstoupit dohledávací anabázi pouze tehdy, pokud to bude opravdu nutné (v opačném případě nemělo smysl zavádět další atribut, protože bychom stejně pokaždé procházeli celý seznam od počátku), musíme umět rozeznat situace, kdy to nutné je.

Jednou z možností je test z výše uvedeného příkazu, při němž zjišťujeme, zda následníkem atomu, na nějž ukazuje atribut *Predch*, je opravdu atom, na nějž odkazuje atribut *Kurzor*. Tento test by však mohl být splněn i v případě, kdy bychom atom odkazovaný atributem *Predch* odebrali ze seznamu a okupovanou paměť vrátili zpět do volně paměti. Atribut *Predch* by totiž na uvolněnou paměť i nadále ukazoval, a dokud by program tuto paměť nepřidělil dalšímu objektu a dokud by ji tento nový objekt nemodifikoval, dál bychom v tomto uvolněném objektu nacházeli adresu bývalého následníka.

Aby tomu tak nebylo, musíme při rušení atomu nejprve „vyprázdnit“ nebo jinak znehodnotit jeho ukazatel na následníka. Navíc musíme zaručit, že žádný objekt, kterému bude v budoucnu tato paměť přidělena, do ní neumístí hodnotu, která by falešně vyhověla našemu testu. Tento požadavek vypadá na první pohled nesplnitelně, ale kupodivu existuje řada situací, ve kterých je přirozeně splněn. Typickým příkladem je např. naše úloha s tříděním, protože při ní se po celou dobu života iterátoru na haldě nealokují jiné objekty, než atomy seznamu. Uvolněný atom tedy v tomto případě nemůže být přepsán jiným objektem, než nově vzniklým atomem. Pokud tedy bude v takovémto případě náš test splněn, nemůže to znamenat nic jiného než to, že nově alokovaný atom je předchůdcem atomu, na který ukazuje atribut *Kurzor*.

Sami asi cítíte, že se touto argumentací dostáváme na tenký led. Bezpečnější by bylo se na žádné „zaručené“ vedlejší podmínky nespoléhat. Nikdy totiž nevíme, kdy je při následných modifikacích programů porušíme, aniž bychom si při tom uvědomili, že tím nepřímo zavádíme chybu někam úplně jinam, a to navíc chybu velice záludnou a těžko odhalitelnou.

Pokud jsou pro nás obousměrně zřetězené seznamy příliš paměťově náročné a pokud bychom chtěli využít zvýšené efektivity diskutovaného řešení, měli bychom na krizové situace myslet přímo v programu a zabezpečit potřebnou synchronizaci explicitně. To znamená, že bychom po každém přidání nebo odebrání položky měli prověřit korektnost atributů všech iterátorů, které mohly odkazovat na odebíranou položku nebo na položku před ní či za ní.

Explicitní synchronizace se nám vyplatí pouze tehdy, pokud je počet iterátorů, jejichž konzistentnost by mohla být danou operací narušena, dostatečně malý — přesněji řečeno: pokud je mnohem menší než očekávaná délka seznamu. V opačném případě bychom totiž mohli synchronizací iterátorů ztratit daleko více, než kolik bychom touto „efektivnější“ implementací iterátoru mohli získat.

Synchronizaci můžeme realizovat dvěma způsoby. Buď pro ni definujeme samostatnou metodu, nebo rozšíříme metody pro vkládání a odebírání položky ze seznamu o další parametry, v nichž budeme předávat odkazy na iterátory, které potřebujeme synchronizovat.

Oba způsoby mají své výhody a nevýhody. Nevýhodou prvního způsobu, samostatné metody, je to, že musíme pro každou jednotlivou synchronizaci znovu volat proceduru, a kromě vlastní synchronizace tak snižovat efektivitu programu ještě o režii spojenou s voláním procedury. Pokud se však pro definici synchronizační funkce přesto rozhodneme, definujeme ji asi následovně:

```
/*  Příklad C8 — 6  */
```

```
inline void /*****/ cSLIter::Sync /*****/
( const cSLIter& I )
//Synchronizace iterátoru po přidání nebo odebrání položky ze seznamu.
//Iterátor I ukazuje na následníka přidané či rušené položky.
{
  if( Kurzor == I.Kurzor )
       Predch = I.Predch;
//Pokud bychom chtěli ošetřit i možnost odebrání položky odkazované
//synchronizovaným iterátorem, rozšířili bychom definici o příkazy:
  else if( Predch == I.Predch )
       Kurzor = I.Kurzor;
//V tom případě ovšem pomalu přestává být výhodné mít tuto metodu
//definovanou jako vloženou funkci, protože už je moc dlouhá.
}/********** cSLIter::SyncI **********/
```

```
(*  Příklad P8 — 6  *)
procedure (*****) cSLIter.Sync (******)
( const I : cSLIter );
{ Synchronizace iterátoru po přidání nebo odebrání položky ze seznamu.
  Iterátor I ukazuje na následníka přidané či rušené položky. }
begin
  if( Kurzor = I.Kurzor )then
       Predch := I.Predch
{Pokud bychom chtěli ošetřit i možnost odebrání položky odkazované
 synchronizovaným iterátorem, rozšířili bychom definici o příkazy: }
  else if( Predch = I.Predch )then
       Kurzor := I.Kurzor;
end;
(********** cSLIter.SyncD **********)
```

Jak jste si jistě všimli, při synchronizaci po odebrání položky ze seznamu jsme v odpovídajících metodách naznačili i možnost „vzpamatování“ iterátoru „z úmrtí v rodině“, tj. z odebrání položky, na kterou iterátor odkazuje. Explicitní synchronizace nám tak dovoluje toto „úmrtí v rodině“ připustit, tj. připustit použití iterátoru, kterému jsme před chvíli odkazovanou položku vyjmuli ze seznamu. Myslíme si však, že je lepší se této možnosti ve svých programech raději vyhýbat.

Poznámka:
Pokud u obousměrně zřetězených seznamů připustíme použití iterátoru, jehož položka byla odebrána ze seznamu, musíme také zavést odpovídající synchronizační funkci.

Pojďme ale dále. Pokud dáme přednost druhému způsobu synchronizace, tj. dodatečným parametrům metod pro přidání a odebrání položky ze seznamu, nejenže odpadne režie s voláním procedury, ale navíc máme rovnou k dispozici všechny potřebné informace, které jsme při prvním způsobu museli synchronizační proceduře předávat jako parametry.

I druhý způsob má ovšem své nevýhody. Jeho hlavní nevýhodou je především to, že obecně budeme při každém přidání, resp. odebrání položky synchronizovat jiný počet iterátorů. S tím se sice v C++ můžeme snadno vypořádat pomocí definice metody s pro-

měnným počtem parametrů, Pascal to ale neumožňuje, a proto budeme muset vymyslet způsob, jak to obejít.

Bez nepříjemných důsledků však nakonec nezůstaneme ani v C++, protože tam jsme u metody pro odebírání prvku využívali implicitní hodnoty parametru specifikujícího, zda se má po odebrání položky ze seznamu zrušit i uložený objekt či nikoliv. Této výhody bychom se museli buď vzdát, nebo definovat nějakou další (nejspíše vloženou) metodu, která tento parametr vypustí. (Možnost synchronizace po odebrání odkazované položky již neuvádíme – za prvé vám ji nedoporučujeme a za druhé si ji v případě potřeby jistě dokážete doplnit sami.)

Pokud se tedy rozhodneme jít touto cestou, definujeme novou podobu metody pro odebrání položky ze seznamu asi takto:

```
/*  Příklad C8 — 7  */
/*****
  Deklarováno:
  void Zrus( int Vlastn=NASTAVENE, cSLIter *I=0, ...);
*****/
void /*****/ cSLIter::Zrus /*****/
( int Vlastn, cSLIter *I, ...)
//Vyjme ze seznamu položku, na níž ukazuje iterátor, a pokud je "jeho",
//smaže ji. Po odebrání hodnoty ukazuje na následující položku.
//Před ukončením činnosti synchronizuje všechny iterátory předané
//jako parametry. Seznam parametrů je ukončen prázdným ukazatelem.
{
  assert((Seznam != 0) && (Kurzor != 0));
//Iterátor musí být již inicializován a musí někam ukazovat
  if( Kurzor == Seznam->Prvni )
  {
       Seznam->Prvni = Kurzor->Dalsi;
       Predch = 0;
  }
  if( Kurzor == Seznam->Posl )
  Seznam->Posl = Predch;
  if( Vlastn == NASTAVENE )
       Vlastn = Seznam->Vlastnik;
  if( Vlastn == MOJE )
       delete Kurzor->Data;
  cSAtom *pom = Kurzor;
  Kurzor = Kurzor->Dalsi;
  Predch->Dalsi = Kurzor;
  delete pom;
//Následující použití proměnné pI není zcela korektní, protože
//nepoužívá postup zpracování proměnného počtu parametrů doporučený
//normou ANSI, ale využívá některá specifika implementace překladačů
//firmy Borland. Později si řekneme, proč to děláme právě takto.
  for( cSLIter **pI = &I;
  *pI != 0;
  pI++ )
  {
       if( (*pI)->Kurzor == Kurzor )
       (*pI)->Predch = Predch;
  }
```

```
}/********** cSLIter::Zrus **********/

inline void /*****/ cSLIter::Zrus /*****/
( cSLIter& I )
//Zruší položku ze seznamu a zabezpečí synchronizaci iterátoru I.
//Další osud položky závisí na nastavení vlastnických práv seznamu.
{
  Zrus( NASTAVENE, &I, 0 );
}/********** cSLIter::Zrus **********/
```

V Pascalu zvolíme kompromis, při kterém definujeme metodu se třemi synchronizovanými iterátory jako parametry. V případě, že by bylo potřeba synchronizovat více iterátorů, použijeme pro jejich synchronizaci samostatnou metodou.

```
(*  Příklad P8 — 7  *)
procedure (*****) cSLIter.Zrus (*****)
( Vlastn : eVlastn );
{ Vyjme ze seznamu položku, na níž ukazuje iterátor, a pokud je "jeho",
  smaže ji. Po odebrání hodnoty ukazuje na následující }
var
  pom : pSAtom;
begin
  assert( (Seznam<>nil) and (Kurzor<>nil) );
{Iterátor musí být již inicializován a musí někam ukazovat }
  if( Kurzor = Seznam^.Prvni )then
  begin
       Seznam^.Prvni := Kurzor^.Dalsi;
       Predch := NIL;
  end;
  if( Kurzor = Seznam^.Posl )then
       Seznam^.Posl := Predch;
  if( Vlastn = NASTAVENE )then
       Vlastn := Seznam^.Vlastnik;
  if( Vlastn = MOJE )then
       dispose( Kurzor^.Data );
  pom := Kurzor;
  Kurzor := Kurzor^.Dalsi;
  Predch^.Dalsi := Kurzor;
  dispose( pom );
end;
(********** cSLIter.Zrus **********)

procedure (*****) cSLIter.ZrusS (*****)
( Vlastn : eVlastn; i1, i2, i3 : pSLIter );
{ Vyjme ze seznamu položku, na níž ukazuje iterátor, a pokud je "jeho",
  smaže ji. Zároveň synchronizuje iterátory, na něž ukazují parametry,
  přičemž prázdnost jednoho ukazatele znamená i prázdnost ukazatelů
  následujících, a tím i nepotřebnost jejich synchronizace. Po odebrání
  položky ukazuje na položku následující za položkou právě odebranou }
begin
  Zrus( Vlastn );
{ Následující část kódu je trochu těžkopádná, chceme se ale vyhnout
  "nečistým" praktikám, vycházejícím ze znalosti mechanismu překladu.
  Použití vektoru nebo seznamu by komplikovalo program, ve kterém
  hodláme tuto metodu použít.
```

```
}
  if( i1 <> NIL )then
  begin
       if( Kurzor = i1^.Kurzor )then
         Predch := i1^.Predch;
       if( i2 <> NIL )then
       begin
         if( Kurzor = i2^.Kurzor )then
           Predch := i2^.Predch;
         if( i3 <> NIL )then
         begin
           if( Kurzor = i3^.Kurzor )then
             Predch := i3^.Predch;
         end; {3}
       end; {2}
  end; {1}
end;
(********** cSLIter.ZrusS **********)
```

Jak vidíte, zpracování většího počtu parametrů není zrovna nejelegantnější. Časem si ukážeme metody, jak takovéto zpracování zelegantnit a zefektivnit – budeme však muset použít některé „nečisté“ praktiky.

Před definicí metody pro přidání položky do seznamu bychom se měli zamyslet nad otázkou hodnoty iterátoru po provedené operaci. U iterátorů obousměrně zřetězených seznamů jsme to měli jednoduché – tam jsme nechali iterátor nadále ukazovat na původní položku, protože nebyl problém iterátor v případě potřeby o jednu položku v seznamu vrátit.

Návrat o položku zpět však je, jak víme, pro iterátory jednosměrně zřetězených seznamů neefektivní operace, a to i při naší rozšířené implementaci s atributem ukazujícím na předchozí položku. Pokud se mu budeme chtít vyhnout, napadne nás asi modifikace metody pro vložení položky, jejímž vedlejším efektem by bylo přesunutí iterátoru na vkládanou položku. To je samozřejmě možné, ale toto řešení většinou znepřehlední programy, které pak budou nutně vypadat jinak, než tytéž programy využívající seznamů zřetězených obousměrně – a to není dobré.

Druhou možností je přidání další metody (tj. metody pro vložení položky, která by přesunula iterátor na vkládanou položku), avšak ta se nám také nelíbí, protože nevýhody první možnosti nijak výrazně neodstraňuje.

Třetí možností pak je modifikovat stávající metodu přidáním dalšího parametru, v němž by metoda vracela iterátor ukazující na vloženou položku. Tato možnost sice také vede k modifikacím programů, nicméně tyto modifikace jsou daleko drobnější a nejsou tak náchylné k chybám. Proto se přimlouváme právě za ni.

Po zakomponování všech diskutovaných modifikací by tedy mohla definice metody pro vložení nové položky do seznamu vypadat např. následovně:

```
/*  Příklad C8 – 8  */
void /*****/ cSLIter::PredVloz /*****/
( tData *d, cSLIter *Puv, cSLIter *I, ...)
//Zařadí do seznamu objekt d před položku, na níž ukazuje iterátor
```

```
//Pokud je ukazatel Puv neprázdný, nastaví iterátor, na nějž tento
//ukazatel ukazuje, na právě vloženou položku.
//Počínaje parametrem I pak synchronizuje iterátory určené ukazateli
//tak dlouho, dokud nenarazí na prázdný ukazatel.
{
  assert( (Seznam != 0) );       //Iterátor musí být inicializován
  cSAtom *a = new cSAtom;        //Adresa přidávaného atomu - položky
  a->Data = d;
  a->Dalsi = Kurzor;             //Současná akt. položka bude následník
  if( Kurzor == 0 )              //Ukazuje iterátor za konec seznamu?
  eznam->Posl = a;               //Nastav nový konec seznamu
  if( Kurzor == Seznam->Prvni )  //Vkládáme před počátek seznamu?
       Seznam->Prvni = a;        //ANO - nastav nový počátek seznamu
  else                      //NE - vkládáme až někam za počátek seznamu
       Predch->Dalsi = a;        // Navaž předchozí atom
  if( Puv != 0 )                 //Potřebujeme si poznamenat pozici?
  {
       Puv->Kurzor = a;          //ANO - Iterátor Puv bude ukazovat na
       Puv->Predch = Predch;     // právě vloženou položku
       Puv->Seznam = Seznam;
  }
  Predch = a;                    //Vložená položka bude novou předchozí
  for( cSLIter **pI = &I;        //Synchronizace vyjmenovaných iterátorů
       *pI != 0;
       pI++ )
  {
  if( (*pI)->Kurzor == Kurzor )  //Pokud ukazovali na položku, před
       (*pI)->Predch = Predch;   //kterou jsme vkládali, aktualizuje
  }        //se jím předchůdce odkazované položky
}********** cSLIter::PredVloz **********/
```

```
(*  Příklad P8 — 8  *)
procedure (*****) cSLIter.PredVlozS (*****)
( d : pData; Puv : pSLIter; i1, i2, i3 : pSLIter );
{ Zařadí do seznamu objekt d před položku, na níž ukazuje iterátor.
  Pokud je ukazatel Puv neprázdný, nastaví iterátor, na nějž tento
  ukazatel ukazuje, na právě vloženou položku. Počínaje parametrem i1
  pak synchronizuje iterátory určené ukazateli tak dlouho, dokud
  nenarazí na prázdný ukazatel nebo dokud nevyčerpá všechny tři. }
var
  a : pSAtom;
begin
  assert( Seznam<>nil );   {Iterátor musí být inicializován }
  new(a);  {a = Adresa přidávaného atomu - položky }
  a^.Data := d;
  a^.Dalsi := Kurzor;      {Současná aktuální položka bude následníkem}
  if( Kurzor=nil )then     {Ukazuje iterátor za konec seznamu? }
       Seznam^.Posl := a;  {Nastav nový konec seznamu }
  if( Kurzor = Seznam^.Prvni )   {Vkládáme před počátek seznamu? }
  then
       Seznam^.Prvni := a  {ANO - nastav nový počátek seznamu }
  else                     {NE - vkládáme až někam za počátek seznamu}
       Predch^.Dalsi := a; {Navaž předchozí atom }
  if( Puv <> NIL )then     {Potřebujeme si poznamenat pozici? }
  begin
```

```
      Puv^.Kurzor := a;    {ANO - Iterátor Puv bude ukazovat na }
      Puv^.Predch := Predch;    {právě vloženou položku }
      Puv^.Seznam := Seznam;
  end;
  Predch := a;             {Vložená položka bude novou předchozí }
  if( i1 <> NIL )then      {Synchronizace vyjmenovaných iterátorů }
  begin
      if( Kurzor = i1^.Kurzor )then        {Pokud ukazovali na položku,
                                            před }
          Predch := i1^.Predch;            {kterou jsme vkládali,
                                            aktualizuje }
      if( i2 <> NIL )then       {se jim předchůdce odkazované položky}
      begin
          if( Kurzor = i2^.Kurzor )then
                Predch := i2^.Predch;
          if( i3 <> NIL )then
          begin
                if( Kurzor = i3^.Kurzor )then
                  Predch := i3^.Predch;
          end; {3}
      end; {2}
  end; {1}
end;
(********** cSLIter::PredVloz **********)
```

Nová podoba definice datového typu iterátorů jednosměrně zřetězeného seznamu, která již zahrnuje vše, o čem jsme hovořili, tedy vypadá následovně:

```
/*  Příklad C8 – 9  */
/*****/ class cSLIter /*****/
{
  public:
      cSLIter();
      cSLIter( cSList& L );
//Kopírovací konstruktor nám vyhovuje v implicitní verzi.
//Stejně tak destruktor a operátor přiřazení
      void Reset( cSList& L );
      pData operator () ();
      pData operator ++ ();
      pData operator ++ (int);
      void operator -- ();
      int operator == ( cSLIter i );
      int operator != ( cSLIter i );
      void PredVloz ( pData d, cSLIter *Puv=0, cSLIter *I=0, ...);
      void Zrus ( cSLIter &I );
      void Zrus ( int Vlastn=NASTAVENE, cSLIter *I=0, ...);
      void SyncD ( const cSLIter& D );
      void SyncI ( const cSLIter& I );
      friend ostream& operator << (ostream&, const cSLIter& );
  private:
      cSList *Seznam;
      cSAtom *Kurzor;
      cSAtom *Predch;
};
```

```
/********** class cSLIter **********/

(*  Příklad P8 – 9  *)
type
pSList = ^cSList;
pSLIter = ^cSLIter;
(*****) cSLIter (*****) = object
       Seznam : pSList;
       Kurzor : pSAtom;
       Predch : pSAtom;
       constructor Init;
       constructor InitList( var L : cSList );
       procedure Reset( var L : cSList );
       function Hodnota : pData;
       function incX : pData;              {"Prefixová" verze }
       function Xinc : pData;              {"Postfixová" verze}
       function Rovno( i : cSLIter ) : Boolean;
       function Ruzne( i : cSLIter ) : Boolean;
       procedure PredVloz ( d : pData );
       procedure PredVlozS( d : pData;
       Puv: pSLIter; i1, i2, i3 : pSLIter );
       procedure Zrus ( Vlastn : eVlastn );
       procedure ZrusS( Vlastn : eVlastn; i1, i2, i3 : pSLIter );
       procedure Sync (var I : cSLIter );
       procedure WriteF( var F:Text );
       procedure Write;
  end;
(********** class cSLIter **********)
```

Definice ostatních metod již neuvádíme, neboť si je jistě dokážete bez problémů napsat sami.

6. Deklarace typů uvnitř třídy

Nyní na chvilku přerušíme naše povídání o kontejnerech a iterátorech a zastavíme se u jedné z vlastností novějších verzí jazyka C++, která umožňuje vztáhnout výhody zapouzdření i na definice datových typů.

Možnost deklarovat nové datové typy uvnitř třídy zavádí (přesněji dotahuje do konce) jazyk C++ až od verze 2.1, která je implementována borlandskými překladači od verze 3.0 výše (opravdu funguje až v překladačích verze 4.x). Starší verze borlandských překladačů implementovaly verzi jazyka 2.0, ve které jsme tuto možnost využívat nemohli. Přesněji: mohli jsme definovat nové datové typy uvnitř definic objektových typů, ale vlastnosti takto definovaných datových typů se nijak nelišily od vlastností datových typů definovaných mimo třídu.

To znamená, že v C++ můžeme v definici třídy definovat nejen atributy a metody, ale i nové vnořené (a tedy lokální) datové typy. Vnořovat přitom můžete jak typy neobjektové, tak typy objektové. S vnořením neobjektových datových typů se můžeme setkat např. v definici datových proudů, která je součástí standardní knihovny; s vnořováním objektových datových typů se pak můžete setkat např. v knihovně kontejnerových tříd (*container class library*), o které jsme se již zmiňovali v souvislosti se seznamy.

Vnoření datových typů používáme především proto, abychom předem zamezili případné možné záměně s dalšími identifikátory, resp. abychom neblokovali některé identifikátory pro další použití. V takovém případě definujeme vnořovaný datový typ jako veřejně přístupný. Pokud chceme definovat datový typ určený pouze pro danou třídu a její přátele, definujeme jej jako soukromý.

Pokud se budeme chtít na vnořený datový typ později odvolávat, musíme tak učinit s plnou kvalifikací, což znamená, že před vlastní jméno použitého datového typu musíme napsat identifikátor třídy, do které je vnořen, a oddělit jej čtyřtečkou. U vnořených výčtových typů pak musíme při jejich používání kvalifikovat i všechny jejich výčtové konstanty.

Příklady vnořených výčtových typů najdeme např. v definici třídy *ios* v souboru IOSTREAM.H. V následujícím příkladu si ukážeme tu část deklarace datového proudu *ios*, která obsahuje definice výše zmiňovaných výčtových datových typů.

```
/*  Příklad C9 - 1  */
class ios {
  public:
//Stavové bity proudů
       enum io_state {
           goodbit = 0x00,  //Vše je OK
           eofbit = 0x01,   //Byl dosažen konec souboru
           failbit = 0x02   //Poslední operace byla neúspěšná
           badbit = 0x04,   //Pokus o nepovolenou operaci
           hardfail = 0x80  //Blíže nespecifikovaná chyba
       };
//Podmínky nastavované při otevírání proudů
       enum open_mode {
```

```
            in = 0x01,       //Otevřít pro čtení
            out = 0x02,      //Otevřít pro zápis
            ate = 0x04,      //Po otevření najeď za konec souboru
            app = 0x08,      //Přidávej pouze na konec souboru
            trunc = 0x10,    //Existuje-li soubor, vyčisti jej
            nocreate = 0x20, //Nesmí se zřídit nový soubor
            noreplace= 0x40, //Nesmí se přepsat existující soubor
            binary = 0x80    //Otevři soubor v binárním modu
        };
//Referenční bod pro pořadí hledané položky
        enum seek_dir {
            beg=0,           //Počátek souboru - begin
            cur=1,           //Aktuální pozice - current
            end=2            //Konec souboru - end
        };
//Anonymní výčtový typ s formátovacími příznaky
        enum {
            skipws = 0x0001,          //Přeskakuj na vstupu bílé znaky
            left = 0x0002,            //Zarovnávej výstup vlevo
            right = 0x0004,           //Zarovnávej výstup vpravo
            internal = 0x0008,        //Zarovnávej oboustranně
            dec = 0x0010,             //Desítková soustava
            oct = 0x0020,             //Osmičková soustava
            hex = 0x0040,             //Šestnácktová soustava
            showbase = 0x0080,        //Označ soustavu vystupuj. čísel
            showpoint = 0x0100,       //Zobrazuj desetinnou tečku
            uppercase = 0x0200,       //Šetnáctková čísla velkými písmeny
            showpos = 0x0400,         //Zobraz + u kladných čísel
            scientific= 0x0800,       //Semilogaritmický tvar
            fixed = 0x1000,           //Běžný tvar reálných čísel
            unitbuf = 0x2000,         //Po zápisu spláchni všechny proudy
            stdio = 0x4000            //Po zápisu spláchni stdout a
                                      //stderr
        };
//
//Zbytek definice proudu s deklaracemi atributů a metod
//
}; //class ios
```

Poslední z vnořených výčtových datových typů nemá dokonce ani jméno, takže nemůžeme definovat žádné jeho instance. Slouží pouze k pojmenování některých hodnot, tj. k přidělení identifikátorů jednotlivým formátovacím příznakům. Jestliže budeme chtít v programu takto pojmenované příznaky používat, musíme před deklarovaným názvem příznaku specifikovat třídu, v jejíž definici se deklarace tohoto názvu nachází – v našem případě třídu *ios*.

Podívejme se na příkladek:

```
/*  Příklad C9 – 2  */
void /*****/ TiskPolozky /*****/
( const ostream& o, const char* Popis, int Hodnota )
{
  const SireTextu = 50;
  const SireCisla = 10;
```

```
  long f = o.flags();
  char c = o.fill( '.' );
// Použití formátovacích příznaků
  o.setf( ios::left, ios::adjustfield );
  o << setw( SireTextu ) << Popis;
  o.fill( '0' );
  o.setf( ios::right, ios::adjustfield );
  o << setw( SireCisla ) << Hodnota << endl;
  o.flags( f );
  o.fill ( c );
}/***** TiskPolozky *****/
```

Stejně jako klasické datové typy můžeme uvnitř tříd definovat i vnořené objektové typy. Vnořením definice objektového typu do deklarace jiného objektového typu však ovlivňujeme pouze oblast viditelnosti jeho identifikátoru a přístupová práva k typu jako celku. V žádném případě tím neměníme přístupová práva k jeho složkám.

To znamená, že přístupová práva složek obklopující třídy ke složkám třídy vložené a naopak přístupová práva složek vložené třídy ke složkám třídy ji obklopující, nejsou vložením definice nijak ovlivněna – jsou dána jednoznačně specifikátory **public**, **protected** a **private**.

Pokud tedy chceme v metodách obklopující třídy pracovat se soukromými nebo chráněnými složkami vnořené třídy, musí vnořená třída definovat obklopující třídu mezi svými přáteli a naopak, chceme-li ve vnořené třídě pracovat se soukromými nebo chráněnými složkami třídy, která ji obklopuje, musí být vnořená třída deklarována jako přítel třídy obklopující.

Jediný rozdíl mezi používáním instancí vnořeného veřejně přístupného objektového typu v obklopující třídě a mimo ni je v tom, že v obklopující třídě můžeme používat samotné jméno vnořeného typu, kdežto v jiných místech programu musíme používat jméno kvalifikované názvem obklopující třídy.

Jestliže tedy ve třídě *A* definujeme typ *T*, můžeme se na něj v deklaraci třídy *A* (a v tělech jejích metod) odvolávat pomocí identifikátoru *T*. Kdekoli jinde musíme psát *A::T*.

Můžeme si to vyzkoušet na příkladu seznamů. Definujme typ atomů jako vnořený typ v datovém typu odpovídajícího seznamu. (Vzhledem k tomu, že oblasti jejich viditelnosti jsou různé, mohou mít oba „atomové datové typy“ stejný identifikátor *cAtom*.) V obou seznamech a jejich metodách pak budeme moci používat identifikátory „atomových typů“ bez kvalifikace, kdežto v obou iterátorech musíme daný typ kvalifikovat datovým typem, v jehož definici je definice příslušného „atomového typu“ vnořena.

```
/*  Příklad C9 – 3  */
typedef int tData;          //Typ objeků ukládaných do seznamu
typedef tData *pData;       //Ukazatel na objekty ukládané do seznamu
enum eVlastn {CIZI =0, MOJE,     NASTAVENE, _eVlastn,
           NEMAZ=0, MAZ };

/*****/ class cSList /*****/
{ //Seznam jednosměrně zřetězených atomů
  public:
```

```
//===== Zde jsou vynechány deklarace metod =====
       friend ostream& operator << ( ostream&, const cSList& );
  private:
  /*****/ class cAtom /*****/
  {                        //Atom jednosměrně zřetězeného seznamu
       private:
           tData *Data;     //Ukazatel na objekt uložený do seznamu
           cSAtom *Dalsi;   //Ukazatel na následníka
           friend class cSList;
           friend class cSLIter;
           friend ostream& operator << ( ostream&, const cSList& );
           friend ostream& operator << ( ostream&, const cSLIter& );
  };/********** class cSAtom **********/

  cAtom* Prvni;
  cAtom* Posl;
  int Vlastnik;
  cSList( const cSList& );
  cSList& operator= ( const cSList& );
  friend class cSLIter;
  friend ostream& operator << (ostream&, const cSLIter& );
};
/********** class cSList **********/

/*****/ class cSLIter /*****/
{
  public:
//===== Zde jsou vynechány deklarace metod =====
       friend ostream& operator << (ostream&, const cSLIter& );
  private:
       cSList *Seznam;
       cSList::Atom *Kurzor;
       cSList::Atom *Predch;
};
/********** class cSLIter **********/

/*****/ class cDList /*****/
{ //Seznam obousměrně zřetězených atomů
  public:
//== Zde jsou vynechány deklarace metod ==
       friend ostream& operator << ( ostream&, const cDList& );
  private:
  /*****/ class cAtom /*****/
  {    //Atom obousměrně zřetězeného seznamu
       private:
       tData *Data;         //Ukazatel na objekt uložený do seznamu
           cAtom *Dalsi;    //Ukazatel na následníka
           cAtom *Predch;   //Ukazatel na předchůdce
           friend class cDList;
           friend class cDLIter;
           friend ostream& operator << ( ostream&, const cDList& );
           friend ostream& operator << ( ostream&, const cDLIter& );
  };/***** class cDAtom *****/

  cDAtom* Prvni;
  cDAtom* Posl;
  int Vlastnik;
```

```
  friend class cDLIter;
  friend ostream& operator << (ostream&, const cDLIter& );
};
/********** class cDList **********/

/*****/ class cDLIter /*****/
{
  public:
//===== Zde jsou vynechány deklarace metod =====
       friend ostream& operator << ( ostream&, const cDLIter& );
  private:
       cDList *Seznam;
       cDList::Atom *Kurzor;
};
/********** class cDLIter **********/
```

6.1 Přístupová práva ke vnořeným typům

Norma jazyka C++ říká, že specifikace přístupových práv **public**, **protected** a **private** se vztahuje na vnořené typy stejně jako na ostatní složky tříd. (Přístupová práva se prostě týkají jmen, deklarovaných v jednotlivých sekcích, bez ohledu na to, co ta jména popisují.)

To ale neznamená, že to budou všechny překladače nutně respektovat. Vzhledem k tomu, že starší verze C++ přístupová práva na datové typy nevztahovaly, snaží se např. BC++ 3.0 a 3.1 o jakýsi kompromis a zacházejí s vnořenými objektovými typy jako s veřejně přístupnými, i když je deklarujeme v sekci **private**. Borlandské překladače plně uplatňují přístupová práva ke vnořeným typům až počínaje verzí 4.0.

Na závěr této kapitolky jsme pro vás připravili krátký nefunkční prográmek, na kterém si můžete vyzkoušet, jak váš překladač zachází s přístupovými právy ke vnořeným datovým typům – nakolik je ve shodě nebo v rozporu s normou.

```
/*  Příklad C9 – 4  */
class /***/ A /***/         //Obklopující třída
{
  public:
       void af();           //Metoda
       typedef int AI;      //Lokální veřejný typ
  private:
       AI ai;          //Lokální datový typ není třeba kvalifikovat
       class /***/ B /***/  //Vnořená třída - měla by se chovat
       {   //jako soukromá
       public:
         typedef int BI;    //Lokální veřejný typedef
         BI bf();           //Metoda
       private:
         typedef int JJ;    //Soukromý lokální typ
         BI bi;
         static BI sbi;
       //static JJ sbj;     //Není možné, protože statické atributy je
           //nutno definovat mimo třídu a tam je soukromý
```

```
          //vnořený datový typ nedostupný
      friend void fb( B&b );
      };/***** B *****/
      B ab;
      B::BI abi;          //Typ je lokální ve třídě B
      typedef int II;     //Soukromý lokální typ
      friend void fa( AI i );
};/***** A *****/

A::B::BI A::B::sb;    //Definice statického atributu vnořené třídy B
//A::B::JJ A::B::sbj;       //Nelze, protože zde je typ A::B::JJ
  nedostupný
A ea;
A ::AI eai;
A ::B eb;
A ::B::BI ebi;
//A::II eii;          //Není možno použít, protože typ není dostupný

#pragma warn -par     //Direktivy potlačující varovná hlášení
#pragma warn -aus     //o nepoužití deklarovaných proměnných
#pragma warn -use     //a parametrů

void fa( A::AI i )    //Funkce fa je přítelem třídy A, a má proto
{                     //přístup k jejím soukromým složkám včetně
  A::II ii=0;         //vnořených typů - musí je však kvalifikovat.
//A::B::JJ j;         //Není ale přítelem třídy B, a nemůže proto
}                     //používat její soukromé složky a typy.

void fb( A::B& b )    //Funkce fb je přítelem třídy B, a může proto
{                     //používat všechny její složky a typy včetně
  A::B::JJ j=3;       //soukromých. Vnořené datové typy však musí
  b.bi += je-li;      //odpovídajícím způsobem kvalifikovat.
}

void A::af()          //Metoda af může používat všechny složky
{                     //třídy A, ale pouze veřejné složky třídy B -
  AI ai;              //třída B totiž ani třídu A ani metodu af
  B::BI bi;           //nevyhlásila za svého přítele. Metoda tedy může
  II aii;             //použít pouze veřejné vnořené datové typy
//B::JJ bjj;          //třídy B, avšak může je kvalifikovat
  B b;                //zkráceným způsobem.
  fb( b );
}

A::B::BI A::B::bf()   //Pro metodu bf platí ve vztahu ke třídě A
{                     //totéž, co platilo pro metodu af ve vztahu
  return( bi );       //ke třídě B.
}

void main()
{
  A a;                //Ve funkci main jsou dostupné pouze veřejné
  A::AI ai;           //složky a veřejné vnořené datové typy obou
  A::B b;             //tříd, protože tyto třídy nevyhlásily funkci
  A::B::BI bi;        //main za svého přítele.
//A::II ii            //Navíc je při použití dostupných vnořených
//A::B::JJ jj         //datových typů nutná jejich plná kvalifikace.
  fa( ai );
```

```
    fb( b );
    a.af();
    b.bf();
}
```

7. Správa paměti: operátory new a delete

Při řešení praktických úloh se může stát, že nám koncepce operátoru **new**, kterou nabízí překladač, z nějakých důvodů nevyhovuje.

Typickým příkladem byly např. problémy, na něž jsme naráželi při přetěžování operátorů, které nemohly vracet reference na objekty. Tenkrát jsme si vysvětlovali, proč není vhodné, aby některé operátory vracely reference na objekty. Hlavním důvodem bylo to, že objekt, alokovaný uvnitř operátoru a předaný ven referencí, není dost dobře možno z haldy odstranit. Jako typický příklad jsme si uváděli operátor "+", který vytváří nový objekt, jenž je součtem obou jeho operandů.

Při té příležitosti jsme si také řekli, že si v podobných případech můžeme pomoci využitím metody, kterou jsme kdysi objevili v jednom překladači zdrojových programů z Turbo Pascalu 4.0 do jazyka C a kterou nazveme **bazén** (v literatuře se také setkáte s označením **aréna**). Tato metoda je založena na skutečnosti, že pomocné objekty, které jsou rozšířenými operátory vytvářeny, bývají většinou pouze dočasné, že jich je v daný okamžik potřeba vždy jen poměrně malý počet, a že doba, po kterou jsou potřeba, bývá krátká.

Princip této metody je nesmírně jednoduchý a je možno jej aplikovat na všechny datové typy – objektové i neobjektové: V paměti (tedy nejspíše na haldě) vyhradíme nějaký prostor, kterému budeme říkat bazén. V tomto bazénu budeme postupně alokovat všechny dočasné proměnné. Když při alokaci dojdeme až na konec bazénu, začneme další proměnné alokovat opět od počátku. Při tom budeme předpokládat, že všechny proměnné, jež předtím okupovaly prostor, který se právě chystáme znovu přidělit, již skončily svůj aktivní život a v paměti více méně jen zaclánějí, takže je s klidem přepíšeme.

Jak jste si jistě všimli, tato metoda tedy alokuje neustále nové a nové proměnné, a žádné z nich z paměti neuvolňuje. Nepoužívá tedy operátor **delete** (C++), resp. proceduru *dispose* (Pascal), a nemusíme proto přemýšlet nad tím, kdy a jak tento operátor, resp. tuto proceduru zavolat.

Tato koncepce však přináší také některá omezení. Jedním z nich je požadavek, aby destruktor nebylo třeba volat. Pokud má být výše uvedená alokační strategie bezpečná, nesmí mít datové typy, jejichž instance v bazénu alokujeme, nějaký netriviální destruktor, protože při práci s bazénem se žádný destruktor nevolá. Existují však možnosti, jak tento problém vyřešit. K tomu se za chvíli ještě vrátíme.

Nyní, když již víme, že by se nám nějaká speciální podoba operátoru **new** resp,. procedury *New* (a popřípadě i jeho protějšku – operátoru **delete** resp. procedury *dispose*) mohla hodit, ukažme si, jaké možnosti nám v tomto směru dávají probírané jazyky.

Jazyk C++ nabízí pro alokaci proměnných operátor **new** a pro jejich dealokaci operátor **delete**. Pro oba operátory můžete definovat homonyma. Tyto operátory a jejich homonyma však mají některé specifické vlastnosti, které je třeba mít na paměti.

Oba operátory – jak **new** tak **delete –** lze definovat jako řadové funkce nebo jako metody objektových typů. Pokud je ale definujeme jako řadové funkce, **nemusí mít – na rozdíl od ostatních operátorů – žádný parametr objektového typu**.

Jazyk C++ nám navíc dovoluje nejen definovat homonyma, ale dokonce i předefinovat standardní verze operátorů **new** a **delete**. Tyto operátory jsou totiž definovány jako knihovní funkce, které můžeme nahradit svými vlastními verzemi.

Poznámka:
ANSI C++ rozlišuje operátor **new** *pro alokaci jednoduché proměnné a operátor* **new[]**, *který se používá pro alokaci polí. Například:*

```
// ANSI C++
class T * uT = new T; // použije se new
int *i = new int[1000]; // použije se new[]
```

Podobně se rozlišují operátory **delete** *a* **delete[]**. *Zde se poměrně výrazně liší i syntaxe – ale to již známe, o tom jsme si povídali dříve.*

```
// ANSI C++
delete uT; // volá se delete
delete [] i; // volá se delete[]
```

Poznamenejme, že některé překladače (mj. také BC++) tolerují při uvolňování polí neobjektových typů zápis

```
delete i; // lze v BC++
```

Pro dynamicky alokovaná pole objektových typů je však zápis

```
delete [] i;
```

nezbytný, protože jinak se nezavolají destruktory pro všechny prvky pole.

ANSI C++ umožňuje také definovat vlastní verze těchto „polních" operátorů. Setkáme se s tím v borlandských překladačích počínaje verzí 4.0 a povíme si o nich později v této kapitole.

Při definicích homonym nebo dokonce nových verzí těchto operátorů však musíte dodržet několik pravidel.

Operátor new

Všechny definice operátoru **new** a jeho homonym musí:

1. vracet hodnotu typu **void***, ve které předají volajícímu programu adresu alokované paměti;
2. mít první parametr typu *size_t*[13] – překladač v něm předává velikost paměti, kterou je třeba alokovat.

[13] Připomeňme si, že *size_t* je typ, deklarovaný v hlavičkovém souboru *alloc.h* jako
`typedef unsigned size_t;`

Operátor delete

Všechny definice operátoru **delete** a jeho homonym musí:

1. vracet **void** (jinými slovy nevracet nic) – jedná se tedy o procedury;
2. mít první parametr typu **void*** (volající program v něm předává adresu rušeného prvku);
3. pro případ, že by alokovaná velikost paměti neodpovídala velikosti odpovídající proměnné, smí mít i druhý parametr typu *size_t*.

Jiné požadavky na globální operátory a jejich homonyma kladeny nejsou. Vraťme se ale k operátoru **new**. Při jeho volání se první parametr neuvádí. Překladač jej automaticky skrytě dodá sám, obdobně, jako např. předává metodám parametr **this**. V tomto parametru se předává operátoru velikost paměti, kterou má operátor **new** vyhradit.

Pokud voláme homonymum, které má ještě nějaké další parametry, zapíšeme jejich hodnoty do kulatých závorek za klíčové slovo **new**, jako kdyby to byla funkce. S touto koncepcí nám sice autoři jazyka C++ operátor **new** trochu „popascalili“ (v Pascalu se při volání procedur a funkcí, které nemají žádné parametry, nepíší kulaté závorky), ale vzhledem k tomu, že ve většině případů budete používat **new** bez parametrů, tak se při této koncepci zápis většiny programů nepatrně zjednoduší.

Stejnou „pascalskou“ syntax použili autoři jazyka i pro označení datového typu alokované proměnné, které se uvádí za klíčovým slovem **new** a případným seznamem parametrů. Pokud se spokojíme s bezparametrickým konstruktorem, stačí zde uvést pouze identifikátor datového typu (a tím pádem u instancí objektových typů i identifikátor konstruktoru). Pokud budeme chtít pro vytvoření instance objektového typu použít konstruktor s parametry nebo pokud budeme chtít neobjektové proměnné přiřadit počáteční hodnotu, zapíšeme seznam parametrů (nebo přiřazovanou hodnotu) do závorek za jméno typu, podobně jako při volání funkce. (Při inicializaci instancí, vytvořených pomocí operátoru **new**, nelze použít zápis s přiřazením, na který jsme zvyklí z „obyčejných“ deklarací, protože by mohl být nejednoznačný.)

Pokud budeme chtít definovat homonymum operátoru **new** „šité na míru“ nějaké třídě, tj. pokud je budeme chtít definovat jako metodu dané třídy, musíme respektovat některá další omezení.

Především musíme mít na paměti, že operátor **new** je statickou metodou se všemi z toho vyplývajícími vlastnostmi (nemá např. skrytý parametr **this**). Překladač to však ví a netrvá na tom, abychom při definici jeho homonyma klíčové slovo **static** explicitně uváděli. Nezávisle na tom, zda je uvedeme či neuvedeme, překladač toto homonymum vždy definuje jako statickou metodu.

Druhou věcí, na kterou musíte při definici homonyma tohoto operátoru myslet je, že pokud homonymum definujeme jako metodu nějaké třídy, automaticky tím pro danou třídu zakryjeme všechny globální verze tohoto operátoru[14]. Pokud bychom chtěli někte-

[14] Proč deklarací **new** jako metody zakryjeme všechna globální **new**? Vysvětlení je jednoduché, ale často se na něj zapomíná: Třída (tj. deklarace třídy a těla všech jejích metod) představuje samostatný obor viditelnosti. Jestliže tedy ve třídě deklarujeme jakoukoli

rou z globálních verzí použít, musíme ji buď explicitně kvalifikovat operátorem **::** (čtyřtečka), nebo si můžeme definovat ekvivalent potřebné verze globálního **new** jako další metodu své třídy, která bude mít jako jediný příkaz volání odpovídající globální verze. (Takovéto homonymum je možno s výhodou definovat jako vloženou funkci.)

Třetí zvláštností, na kterou bychom neměli při definici homonym operátoru **new** pro danou třídu zapomenout, je, že při alokaci vektoru (pole) se homonymum definované pro danou třídu nepoužívá. Ve starších verzích C++ použije překladač globální verzi operátoru, v ANSI C++ použije operátor **new[ ]**[15].

Abychom pořád jen neteoretizovali, ukážeme si malý demonstrační příklad, v němž se pokusíme předvést nejdůležitější vlastnosti operátoru **new**.

Nejprve si uvedeme použité direktivy preprocesoru.

```
/*  Příklad C10 – 1  */
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
#pragma warn -par     //Direktivy potlačující varovná hlášení
#pragma warn -aus     //o nepoužití deklarovaných proměnných
#pragma warn -use     //a parametrů
```

V následující ukázce si definujeme třídu *A*, která bude obsahovat dva nestatické atributy: pořadí vzniku dané instance mezi instancemi třídy (to abychom ve výpisu poznali, o kterou instanci se jedná) a krátký text pro ještě snazší identifikaci. Kromě toho bude obsahovat i statický atribut obsahující počet doposud vzniklých instancí (vzniklých, nikoliv však nutně existujících).

Kromě konverzního konstruktoru, předávajícího instanci identifikační řetězec, a bezparametrického konstruktoru, potřebného pro definici neinicializovaného vektoru instancí dané třídy, bude naše třída obsahovat pouze dvě další metody, kterými budou homonyma operátoru **new**. První z nich (ta s druhým parametrem typu **char**) bude implementovat bazén, druhou z nich definujeme jen proto, abychom uvnitř třídy zpřístupnili globální verzi standardního **new**.

Pro metodu, realizující bazén, jsme přidali do třídy *A* další dva statické atributy: prvním je pole instancí typu A[16] a druhým je index ukazující, kam bude do bazénu umístě-

---

funkci s názvem **operator new**, zastíníme tím automaticky všechny deklarace stejnojmenných funkcí v nadřízeném oboru viditelnosti - tj. na úrovni souboru.

[15] Poznamenejme, že standardní verze globálního operátoru **new[ ]** volá globální operátor **new** (tedy funkci **operator new(***size_t***)**).

[16] Všimněte si, že jsme v deklaraci třídy zapsali pouze "otevřenou" deklaraci. Velikost pole jsme zadali až v následující definiční deklaraci

```
A A::bazen[ A::MAX_NBA ];
```

na příští instance. Oba tyto atributy by sice mohly být lokální v operátorové funkci, ale umístili jsme je mezi atributy třídy proto, abychom si ukázali, že statické atributy nohou být instancemi "své" třídy. Zdůrazňujeme však, že to platí **pouze pro statické atributy**! "Obyčejný", tedy nestatický atribut naopak instancí své třídy být nesmí.

V definici třídy jsme deklarovali nepojmenovaný veřejně přístupný výčtový typ. Ten vlastně pouze zavádí dvě konstanty: *MAX_NBA* udává maximální počet prvků současně umístitelných do bazénu třídy *A* (tj. velikost tohoto bazénu) a *MAX_CHR* udává velikost vektoru znaků, který je nestatickým atributem třídy *A*.

Obě tyto konstanty bychom samozřejmě mohli deklarovat pomocí direktiv **#define**, protože však jde o záležitosti, která se týkají opravdu jen a pouze třídy *A*, je vhodné umístit je dovnitř.

Mohlo by se zdát, že bychom k tomuto účelu mohli použít konstantních atributů třídy *A*; jenže takovéto konstanty nejsou dostatečně konstantní na to, aby mohly sloužit k definici pole, neboť jejich hodnota se nastavuje až v konstruktoru, tedy v době běhu programu. Velikost našeho pole musí znát ale již překladač.

```
/*  Příklad C10 – 2  */
class /*****/ A /*****/
{
  public:
       enum {    //Výčtový typ nahrazuje konstanty
           MAX_NBA = 5,     //Počet prvků umístitelných v bazénu třídy A
           MAX_CHR = 10,    //Velikost znakových polí ve třídě A
       };
       A( char *Text );     //Kopírovací konstruktor
       A();                 //Bezparametrický konstruktor
       void* operator new( size_t Velikost, char poc );
       void* operator new( size_t Velikost );
       friend ostream& operator << ( ostream&, A& );
  private:
       int ai;              //Pořadí konstrukce dané instance
       char as[ MAX_CHR ];  //Pole znaků s přechovávaným textem
       static int an;       //Celkový počet vygenerovaných instancí
       static A bazen[];    //Bazén pro alokaci dočasných proměných
       static int bf;       //Index prostoru pro další prvek
};
/********** A **********/
int A::an = 0;              //Nebyla zatím zkonstruována žádná instance
int A::bf = 0;              //První proměnná se umístí na počátek bazénu
A A::bazen[ A::MAX_NBA ];   //Bazén pro dočasné proměnné
```

V následující části programu nabízíme definice konstruktorů třídy *A*. Jejich hlavním úkolem je především podání zprávy o tom, že byly zavolány, abychom měli přehled o dění v programu a nemuseli jej krokovat.

Pokud bychom uvedli velikost pole už v definici třídy, ohlásil by kterýkoli z borlandských překladačů až po verzi 4.52 včetně chybu - použití nedefinované struktury. Jde o nedopatření v borlandských překladačích.

Bezparametrický konstruktor má navíc za úkol oddělit na obrazovce výstup tohoto běhu programu od výstupů předcházejících.

Jedna z věcí, které nám na obdobných zkušebních (téměř bychom mohli říci výzkumných) programech vadí, je to, že po spuštění programu lze někdy obtížně poznat, kde končil výstup předchozí seance a kde začíná výstup programu právě běžícího.

Poznámka:
Tisk umístěný na počátek procedury main nám nepomůže, protože před spuštěním procedury main se musí nejprve zkonstruovat všechny globální proměnné, a tím i vyvolat konstruktory, které pro nás občas vypustí nějaké ty kontrolní tisky. Univerzálním řešením je definice jednoúčelového objektového typu, který nemá žádný atribut a jehož jedinou metodou je bezparametrický konstruktor. Tento konstruktor nemá žádný jiný úkol, než nějakým způsobem oddělit jednotlivé výstupy. Pokud jako první globální instanci definujeme instanci právě tohoto datového typu, dosáhneme kýženého oddělení výstupů.

Možná vás na první pohled napadá, že bychom mohli definovat vhodnou proceduru, která bude mít takové oddělení výstupů – nebo jinou akci, kterou chceme provést na samém počátku programu. Této proceduře bychom pomocí direktivy **#pragma startup** *přiřadili vhodnou prioritu.*

Bohužel, to nefunguje. Pokud totiž budeme chtít pracovat s výstupy, směřujícími do datových proudů jazyka C++, musíme programu poskytnout možnost tyto datové proudy nejprve zkonstruovat, a teprve pak se můžeme pokoušet nějakým způsobem vizuálně oddělit minulý a současný výstup. Toho můžeme dosáhnout nejjednodušeji právě tak, že na vhodné místo programu umístíme definici naší globální proměnné. Přitom jediným účelem této definice bude vyvolat konstruktor, který provede žádanou akci – v našem případě vizuální oddělení minulého a stávajícího výstupu.

V našem příkladu jsme zvolili jednodušší variantu. Protože je program napsán tak, že prvními konstruovanými instancemi objektového typu budou složky vektoru A::bazen*, umístili jsme oddělující kód do jejich konstruktoru.*

```
/*  Příklad C10 – 3  */
/*****/ A::A /*****/
()
//Bezparametrický konstruktor třídy A
{
  if( an == 0 ) //Oddělení výstupu na obrazovce od předchozího
  cout << "\n\n\n\n\n==========================================";
  cout << "\nA::A–Prvek: " << (ai=an++)
       << " - nový text: " << "Prazdny"
       << " - pův.text: " << as;
  strncpy( as, "Prázdný" , MAX_CHR );
}/********** A::A **********/

/*****/ A::A /*****/
( char *Text )
//Kopírovací konstruktor třídy A
{
  cout << "\nA::A(Txt) Prvek: " << (ai=an++)
       << " - nový text: " << Text
       << " - pův.text: " << as;
```

```
  strncpy( as, Text, MAX_CHR );
}/********** A::A **********/

ostream& /*****/ operator << /*****/
( ostream& o, A& a )
{
  return o
       << "\nInstance: " << a.ai
       << ". - Text = \"" << a.as
       << "\", (Adr-Buf) = " << (&a-a.bazen)
       << ", an = " << a.an;
}/********** operator << **********/
```

Poslední funkcí z předchozí ukázky byl přetížený operátoru výstupu. V něm jsme ve srovnání s předchozími definicemi provedli jednu maličkou inovaci: využili jsme toho, že operace výstupu je výraz, který vrací referenci na výstupní proud, a vložili jsme ji jako vracenou hodnotu za klíčové slovo **return**.

Třetí vystupující hodnotou (popisné textové řetězce nepočítám) je rozdíl dvou ukazatelů. Jistě si vzpomenete, že rozdíl ukazatelů vrací počet instancí mezi těmito dvěma ukazateli. Protože oba ukazatele jsou typu *A**, zjistíme tak přímo pořadí instance v bazénu.

Tato informace pro nás bude užitečná i v případě, kdy budou instance alokovány někde jinde v paměti (např. v jiném bazénu). Stačí pouze, když si zjistíme, jaký je index prvé instance v této nové oblasti, a pak již pokračujeme v průběžném indexování.

Podívejme se nyní na definice lokálních homonym operátoru **new**. Připomínáme znovu, že jakmile definujeme jedno homonymum, automaticky se tak znepřístupní všechny globální verze tohoto operátoru.

Jestliže tedy definujeme v naší třídě lokální homonymum operátoru **new** a přitom chceme mít nadále k dispozici původní verzi operátoru **new**, musíme ji explicitně kvalifikovali operátorem **::** (čtyřtečka) nebo si musíme definovat lokální verzi, která nám zpřístupní verzi globální. Domníváme se ale, že operátor čtyřtečka není zase tak nesympatický, takže se k podobným trikům budete uchylovat spíše výjimečně.

Již v pasáži o základních charakteristikách operátoru **new** jsme si řekli, že k této redefinici můžete s výhodou využít vložené funkce. Pokud však tuto redefinici „zpestříte“ dalšími dodatky jako např. my, vložená funkce se vám již nevyplatí.

```
/*  Příklad C10 - 4  */
void* /*****/ A::operator new /*****/
( size_t Velikost, char poc )
{
  cout << "\n=== A::new( size_t " << Velikost
       << ", char " << poc << " )"
       << " - bf = " << bf;
  void *ret = &bazen[ bf ];
  if( ++bf >= MAX_NBA )
  bf = 0;
  memset( ret, poc, Velikost );
  return ret;
}/********** A::operator new **********/
```

```
void* /*****/ A::operator new /*****/
( size_t Velikost )
{
  cout << "\n=== A::new( size_t " << Velikost << " )";
  return new char [ Velikost ];
}/********** A::operator new **********/
```

V souvislosti s bazénovou verzí lokálního homonyma operátoru **new** bychom chtěli ještě upozornit na jednu věc: jistě jste si všimli, že v těle přetíženého operátoru jsme údaj o velikosti alokované oblasti použili pouze pro její inicializaci zadaným znakem. Mlčky jsme tedy předpokládali, že alokovaná velikost je stejná jako velikost instancí typu *A*. To však nemusí vždy platit. Podrobněji se k tomu vrátíme, až si budeme vyprávět o dědičnosti.

Další část našeho „výzkumného“ programu obsahuje definice globálního operátoru **new**. Pro globální verze operátoru **new** naštěstí neplatí to, že pokud definujete rozšiřující homonymum, tj. homonymum s dodatečnými parametry, tak že by se původní verze operátoru stala nedostupnou. Budou to prostě dvě operátorové funkce se stejným jménem a s různým počtem parametrů v jednom oboru viditelnosti vedle sebe, takže budou obě dostupné. Novou verzi standardního operátoru jsme definoval pouze proto, abychom mohli celý postup alokace a konstrukce snáze sledovat, a to jak pomocí kontrolních výpisů, tak i pomocí detailního krokování.

V ukázce nalezneme homonymum, které se stará o bazénovou alokaci pro jakýkoliv typ proměnných. Bazén a důležité ukazatele definujeme jako jeho lokální statické proměnné. Toto řešení je výhodnější než řešení ukázané v lokálních verzích operátorů, protože poskytuje větší bezpečnost před nežádoucími zásahy „nepovolaných“ částí programu.

Věc, která by nás mohla – a měla – na následujících definicích opravdu zarazit, je test globální proměnné *MAIN* a následné větvení akcí. Toto větvení jsme do programu zařadili proto, že operátor **new** je několikrát volán z incializační části programu, která se vykoná ještě před vstupem do procedury *main* (musí se např. alokovat systémové datové proudy, do kterých pak budeme zapisovat). Než jsme si to uvědomili a program patřičně modifikovali, počítač nám několikrát dokonale „zkameněl“.

Program jsme proto po prohlédnutí upravili tak, že do vstupu do procedury *main* se používá vpodstatě původní postup (jak je to přesně si vysvětlíme později) a bazénová verze alokace se spustí až po vstupu do této procedury.

Abyste se mohli přesvědčit o intenzitě volání operátoru **new** před vstupem do funkce *main*, doplnili jsme jeho definici o dvě lokální statické proměnné, které monitorují počet volání a celkový počet alokovaných bajtů.

Než se na tento program podíváte, rádi bychom vás upozornili, že jde o **dokonalý příklad toho, jak se v C++ programovat nemá**. Zde jsme se k němu uchýlili především proto, abychom si mohli snadno ukázat některé vlastnosti homonym operátoru **new**.

```
/*  Příklad C10 - 5  */
const MAX_BAZ = 70;              //Velikost globálního bazénu
```

```
int MAIN = 0;                    //Indikace vstupu do procedury main

void* /*****/ operator new /*****/
( size_t Velikost, char poc )
{
  static unsigned Volani = 0;    //Počítá celkový počet volání a
  static unsigned Bytu = 0;      //celkový počet alokovaných bajtů
  Volani++;
  Bytu += Velikost;
  if( MAIN )
  {//Větev operátoru vykonávaná po vstupu do funkce main
       static char bazen[ MAX_BAZ ];        //Lokální statická proměnná
       static char *ab = bazen;  //Ukazatel na počátek
                                 //připravené oblasti
       static const char *uk = &bazen[ MAX_BAZ ];      //Konec bazénu
       cout<< "\n=== ::new( size_t " << Velikost
           << ", char " << poc << " )"
           << " - index = " << (ab - bazen);
       void *ret = ab;           //Předpokládaná adresa alokace
       if( (ab += Velikost) >= uk )         //Vejde se do bazénu?
       {                         //Nevejde => budeme ji alokovat od
        ret = ab = bazen;        // počátku bazénu
        ab += Velikost;          //Proměnnou ab přesuneme za ni
        if( ab >= uk )           //Pokud se stále nevejde
          abort();               // předčasně ukončíme program
       }
       return memset( ret, poc, Velikost );
  }
  else          //Tato větev operátoru se vykonává v inicializační
  {             //části programu před vstupem do funkce main.
       return malloc( Velikost ); //Standardní alokace na haldě
  }
}/********** operator new **********/

void* /*****/ operator new /*****/
( size_t Velikost )
//Tento operátor jsme definovali jenom proto, abychom přesně věděli,
//kdy jej překladač použije
{
  if( MAIN )
       cout << "\n=== ::new( size_t " << Velikost << " )" << endl;
  return new( 'N' ) char[ Velikost ];
}/********** operator new **********/
```

Nyní si zkuste následující testovací prográmek. Zkuste si sami zakrývat jednotlivé definice homonym operátoru **new** a dívejte se, jak na to bude překladač reagovat.

```
/*  Příklad C10 - 6  */
A *pa0 = new( '0' ) A( "Nultý" );          //Lokální s inicializací
A *pah = ::new( 'H' ) A( "Halda" );        //Globální s inicializací -
       // volá se před main => alokuje se na haldě pomocí malloc

void /*****/ Test /*****/ ()
{
  MAIN = 1;
  cout << *pa0
       << *pah << '\n';    // cout << '\n; je totéž jako cout << endl;
```

```
  A *pa1, *pa2;
  pa1 = new( '1' ) A( 'Prvni' );            //Lokální s inicializací
  cout << *pa1 << '\n';
  pa2 = new A( "Druhy" );                   //Lokální bez inicializace
  cout << *pa2 << '\n';
  A *pa3 =::new( '3' ) A( "Třetí" );        //Globální s inicializací
  cout << *pa3 << '\n';
  A *pa4 =::new A( "Čtvrtý" );              //Globální bez inicializace
  cout << *pa4 << '\n';
       A *pv = new( 'X' ) A [5];            //Globální s inicializací
  for( int i=0; i < 5; i++ )                //aplikovaný hromadně
       cout << pv[ i ];
  cout << '\n';
}/********** Test **********/
```

Co je špatně?

Nyní se vrátíme k definici globálního operátoru **new** a rozebereme si, co je na něm tak odstrašujícího, že by se nám o něm mohlo v noci zdát.

První věc, která se nám na něm nelíbí, je větvení algoritmu podle hodnoty jakési globální proměnné. Už to může být zdrojem mnoha problémů při pozdějších úpravách programu, neboť závislost operátoru **new** na této proměnné není (při použití) nijak zřejmá.

Snadno se může stát, že někde dále budeme v programu potřebovat tu verzi operátoru, která alokuje paměť v haldě. Budeme tedy přepínat funkci operátoru **new** tím, že budeme měnit hodnoty nějaké na pohled naprosto nesouvisející proměnné? Podívejte se, jak by to vypadalo:

```
MAIN = 1;  // Alokace v bazénu
A* a1 = new A("Další");
// ...
MAIN = 0;  // Alokace v haldě
A* a1 = new A("Nějaký jiný");
```

Jistě s námi souhlasíte, že to není ani přehledné, ani bezpečné. Snadno se může stát, že při nějaké úpravě použijeme proměnnou *MAIN* k jiným účelům – a co se bude dít, to můžeme jen hádat.

Proto je lepší použít dvě různá homonyma operátoru **new** a rozlišit je podle typu parametrů. Například takto:

```
/*  Příklad C10 – 7  */
static unsigned Volani = 0;      //Počítá celkový počet volání a
void* /*****/ operator new /*****/
( size_t Velikost )
// Ukazuje počet volání
{
  Volani++;
  void *p = malloc(Velikost);
  if(p) memset(p, 'N', Velikost);
return p;
}/********** operator new **********/

void* /*****/ operator new /*****/
( size_t Velikost, char poc )
```

```
{
  static unsigned Bytu = 0;      //Celkový počet alokovaných bajtů
  Volani++;
  Bytu += Velikost;
  static char bazen[ MAX_BAZ ];  //Lokální statická proměnná
  static char *ab = bazen;       //Ukazatel na počátek připrav. oblasti
  static const char *uk = &bazen[ MAX_BAZ ];          //Konec bazénu
  cout << "\n=== ::new( size_t " << Velikost
       << ", char " << poc << " )"
       << " - index = " << (ab - bazen);
  void *ret = ab;                //Předpokládaná adresa alokace
  if( (ab += Velikost) >= uk )   //Vejde se do bazénu?
  {        //Nevejde => budeme ji alokovat od
      ret = ab = bazen;          // počátku bazénu
      ab += Velikost;            //Proměnnou ab přesuneme za ni
      if( ab >= uk )             //Pokud se stále nevejde
        abort();                 // předčasně ukončíme program
  }
  return memset( ret, poc, Velikost );
}/********** operator new **********/
```

Nyní jsou volání na první pohled rozlišitelná,

```
A* a1 = new A('Další'); // V bazénu
A* a1 = new('v') A('Nějaký jiný'); // V haldě
```

a nemůže se stát, že náhodným přepsáním nějaké globální proměnné změníme chování celého programu. To ale není vše. **Ani po této úpravě není nový globální operátor new nejlepší, neboť jeho chování se v několika ohledech podstatně odlišuje od chování standardního new**.

V programování se – stejně jako třeba při jízdě po silnici – vyplatí dodržovat konvence, tj. chovat se tak, jak to ostatní očekávají. Jinak riskujeme, že zmateme ostatní, kteří budou někdy náš program upravovat, nebo dokonce i sami sebe, až zapomeneme, jakou že podivnou fintu jsme kde vymysleli.

Poslední globální verze operátoru **new** se odchyluje od „normálu“ ve dvou ohledech:

1. Standardní operátor **new** alokuje vždy alespoň 1 bajt. Protože *malloc(0)* vrátí 0, vrací náš operátor **new** 0 v případě, že je *Velikost* == 0, což je nepřípustné.
2. Standard jazyka C++ předepisuje, že pomocí standardní funkce *set_new_handler* můžeme určit funkci, kterou operátor **new** zavolá, pokud se alokace nepodaří[17]. Také

[17] Připomeňme si, že funkce *set_new_handler* je deklarována ve standardním hlavičkovém souboru *new.h* takto:

```
typedef void NHF(void);
NHF *set_new_handler(NHF*);
```

Parametrem funkce *set_new_handler* je ukazatel na nový „handler“, funkci typu **void** bez parametrů, která se postará o ošetření nedostatku paměti. Ukazatel na tento „handler“ se uloží do globální proměnné *_new_handler*,

```
extern NHF* _new_handler;
```

to náš operátor neumí, takže pokud bychom zapomněli, že v programu používáme své vlastní **new**, mohli bychom se divit.

Poznamenejme, že pokud definujeme homonyma, která se liší počtem parametrů, nemusíme na tato pravidla brát ohled: používáme-li nestandardní operátor **new**, očekáváme od něj také nestandardní chování, a to, že používáme nestandardní operátor **new**, je vidět na první pohled.

Vezmeme-li v úvahu všechny výhrady, které jsme si řekli, dospějeme k následujícímu tvaru globálního jednoparametrického **new**:

```
/*  Příklad C10 – 8  */
void* /*****/ operator new /*****/
( size_t Velikost )
{
  Volani ++;
  void * p;
  Velikost = Velikost ? Velikost : 1;       // Vždy alespoň 1 bajt
  while ( (p = malloc(Velikost)) == 0
            && _new_handler != 0)           // Když se nepovedla alokace
       _new_handler();                      // a je nastaven handler,
                                            // zavolej ho
  if(p) memset(p, 'N', Velikost);
return p;
}
```

7.1 Operátory pro alokaci polí

Již jsme si řekli, že ANSI C++ umožňuje také přetěžovat operátory pro alokaci a dealokaci polí **new[ ]** a **delete[]**. Platí pro ně naprosto stejná pravidla jako pro „obyčejné“ operátory **new** a **delete**, takže si rovnou ukážeme jednoduchý příklad.

```
/*  Příklad C10 – 9  */
#include <new.h>
#include <iostream.h>
class /*****/ B /*****/
{
       int b;
  public:
       B(int i=0): b(i){cout << "konstruktor B" << endl;}
       ~B(){ b = -1; cout << "destruktor B" << endl; }
       void* operator new(size_t, void* p); // Obyčejné new
```

Funkce *set_new_handler* vrací ukazatel na předchozí „handler“.
Upozornění pro čtenáře, kteří používají překladače Borland C++ 4.0 a pozdější: Operátor **new**, implementovaný v těchto překladačích, při neúspěšné alokaci standardně vyvolává výjimku typu *xalloc*. Protože výjimkami se budeme zabývat až v příštím dílu, doporučujeme přinutit jej k obvyklému chování příkazem

```
set_new_handler(0);
```

```
      void* operator new[] (size_t, void* p);         // New pro pole
      void operator delete(void* p);       // Obyčejné delete
      void operator delete[](void* p);     // Delete pro pole
};
/********** B **********/

void* /*****/ B::operator new /*****/
(size_t, void* p)
{
  cout << "B::operator new pro jednoduchou promennou" << endl;
  return p;
}/********** B::operator new **********/

void* /*****/ B::operator new[] /*****/
(size_t, void* p)
{
  cout << "B::operator new pro pole" << endl;
  return p;
}/********** B::operator new **********/

void /*****/ B::operator delete /*****/
(void*)
{
  cout << "delete pro jednoduchou promennou" << endl;
}/********** B::operator delete **********/

void /*****/ B::operator delete[] /*****/
(void*)
{
  cout << "delete pro pole" << endl;
}/********** B::operator delete **********/
```

Ve třídě *B* jsme definovali jako metody jak „obyčejné“ operátory **new** a **delete**, tak i jejich „polní“ verze **new[ ]** a **delete[]**. Operátory **new** a **new[]** prostě jen vypíší zprávu o svém zavolání a vrátí ukazatel, který dostanou jako druhý parametr. K čemu jsou tedy vlastně dobré? Umožňují volat konstruktor pro objekt, který již existuje!

Podobně operátory **delete** a **delete[]** nedělají na první pohled nic jiného, než že vypíší zprávu. Ve skutečnosti slouží k volání destruktoru.

Podívejme se na příklad použití:

```
/*  Příklad C10 - 10  */
void /*****/ Test_nd /*****/
()
{
  char bazen[100];    // Zase nějaký bazén...
  B* b1 = new(bazen) B(22);                              // 1
  B* b2 = new(bazen + sizeof(B)) B[10];                  // 2
  cout << "adresa, zadana operatoru new[] je "
       << (void*)(bazen + sizeof(B)) << endl
       << "adresa, vracena timto operatorem je "
       << b2 << endl;
  delete b1;                                             // 3
  delete [] b2;                                          // 4
}/********** Test_nd **********/
```

V řádku, označeném `// 1`, alokujeme na počátek bazénu jednu instanci třídy *B*. Přitom se použije „obyčejný“ operátor *B::***new**. V řádku `// 3` pak tuto instanci zrušíme – zavoláme na ni prostřednictvím operátoru *B::***delete** destruktor.

Podobně v řádku `// 2` alokujeme počínaje druhým prvkem pole 10 prvků typu *B*. Překladač použije operátor *B::***new[]** a ten desetkrát zavolá konstruktor. V řádku // 4 pak toto pole zrušíme – operátor **delete[]** se postará o zavolání destruktoru pro všechny prvky. O tom nás – vedle krokování v Turbo Debuggeru – přesvědčí zprávy, které tyto konstruktory vypisují.

Všimněte si ale jedné zrady: operátor **new[]** vrátí jinou adresu, než mu předáme jako parametr! Tento operátor si totiž před alokované pole poznamená do dvou bajtů jeho velikost. To musíme brát při práci s bazénem vždy v úvahu.

Jak víme, Pascal nám pro alokaci dynamických proměnných nabízí proceduru *new* a pro jejich dealokaci proceduru *dispose*. Kromě toho máme možnost vyhradit pro proměnnou i paměť jiné velikosti, než jaká odpovídá jejímu typu (toho se využívá zejména u polí). K tomuto účelu nám slouží procedura *GetMem* a k uvolnění takto alokované paměti pak procedura *FreeMem*.

Situaci nám usnadní skutečnost, že Pascal nepovažuje (na rozdíl od C++) *new* za operátor, ale za proceduru, které předáváme ukazatel na alokovanou proměnnou jako parametr předávaný referencí (odkazem). Tím nám totiž Pascal umožňuje definovat nějakou jinou proceduru, která by řešila problém alokace způsobem, jenž by nám více vyhovoval, a která by přitom používala téměř stejnou syntaxi jako standardní procedura *new* podporovaná překladačem.

Aby to však zase nebylo tak jednoduché, *new* přece jenom není běžná procedura, ale procedura, pro kterou autoři překladače definovali rozšířenou syntaxi. Ta povoluje, aby druhým parametrem procedury byl zápis volání konstruktoru, který se na alokovanou proměnnou hned aplikuje. Naštěstí se však nejedná o rozšíření, které by nebylo možno bez větších obtíží obejít.

S využitím standardních prostředků (tj. standardně definovaných procedur a funkcí, při jejichž definici se nebudeme prohřešovat proti pravidlům jazyka) můžeme nestandardní alokaci instancí objektových typů řešit tak, že napřed zavoláme naši vlastní alokační proceduru, a poté na alokovanou proměnnou aplikujeme konstruktor. Pokud bychom místo procedury *new* použili proceduru *GetMem*, museli bychom postupovat naprosto stejně.

V podstatě totožné problémy budeme řešit i v případě uvolňovací procedury *dispose*, ve které pro změnu můžeme jako druhý parametr zadat volání destruktoru. V tomto případě však budeme logicky muset pořadí otočit: nejprve explicitně zavoláme destruktor, a poté naši modifikovanou uvolňovací proceduru. Stejně musíme postupovat i v případě, kdy místo procedury *dispose* potřebujeme použít proceduru *FreeMem*.

Pokusme se nyní vše ukázat na testovacím programu. Půjde o obdobu programu, jaký jsme si uvedli dříve v této kapitole, když jsme hovořili o možnostech vlastních definic alokačních procedur v jazyku C++. Nejprve si definujme potřebnou datovou strukturu a předem deklarujme globální funkce, které budeme chtít použít.

Pascalská definice bude o něco jednodušší než definice z jazyka C++, protože Pascal nám neumožňuje definovat statické atributy ani statické metody, a my musíme všechny tyto deklarace odstěhovat do volného prostoru mimo třídu. Ve třídě tedy zůstanou dva atributy: pořadí vzniklé instance a přiřazený text. Z metod tu pak zůstane pouze jeden konstruktor – u polí musíme stejně konstruovat jednotlivé složky my, takže nutnost definice bezparametrického konstruktoru odpadá. Přibude procedura pro tisk instance dané třídy, kterou je v Pascalu vhodné definovat jako metodu.

Atributy bychom měli správně definovat jako soukromé, ale pak by nám debugger odmítal cokoliv sdělit o jejich hodnotách. Proto porušíme zásady objektově orientovaného programování a definujeme je jako veřejné. Abychom však měli představu o správné podobě věcí, ponechali jsme direktivy ovládající přístup ke složkám v komentářích. Poznamenejme, že direktiva **public** je povolená pouze v Turbo Pascalu verze 7.0.

Na závěr jsme pak definovali proměnnou *an* počítající instance a deklarovali tři funkce pro bazénovou alokaci. Nesnažili jsme se však již definovat funkce s různým počtem parametrů, protože v Pascalu se stejně jedná stále o totéž. Pouze jsme chtěli připomenout, že v důsledku toho, že Turbo Pascal nezná statické funkce, musíme jak proceduru specializovanou na alokaci instancí daného objektového datového typu, tak i relativně obecnou globální alokační proceduru definovat jako globální procedury.

Alokační procedury jsou ve skutečnosti pouze dvě. Proceduru *BazNew7* deklarujeme pouze proto, abychom si na ní pak mohli ukázat, jak při řešení tohoto problému využít některých vlastností Turbo Pascalu verze 7.0.

```
(*  Příklad P10 – 1  *)
const
  MAX_NBA = 5;        {Počet prvků umístitelných v bazénu třídy TA }
  MAX_CHR = 10;       {Velikost znakových polí ve třídě TA }
  nl = #10#13;

type
  TpA = ^TA;
(*****) TA=object (*****)
  { private{}
       ai : integer;              {Pořadí konstrukce dané instance }
       as : string[ MAX_CHR ];    {Pole znaků s přechovávaným textem }
  {public{}
       constructor Init1( Text : String );  {Kopírovací konstruktor }
       procedure write;
  end; (********** TA **********)

const
  an : integer = 0;              {Celkový počet vygenerovaných instancí}

  procedure newA( var p:TpA; poc:char ); forward;
  procedure BazNew( var p:Pointer; Velikost:word; poc:char ); forward;
  procedure BazNew7( var p:Pointer; Velikost:word; poc:char ); forward;
```

Následující definice konstruktoru a tisk instance třídy *TA* mají za cíl pouze informovat o průběžném stavu věcí.

Pokud čtete obě jazykové verze, víte, že v C++ jsme do konstruktoru umísťovali kód pro oddělení výstupu aktuálního běhu programu od výstupů předchozích. V Pascalu však stačí, když takovýto kód umístíme na počátek inicializační části, protože tento tisk nebude předběhnut žádným automaticky vyvolaným programem.

Vlastní bazén a jeho pomocné proměnné zatím nedefinujeme, protože není vhodné zbytečně zveřejňovat věci, které zveřejněny být nemusí. Pokud nepotřebujeme, aby byl bazén sdílen několika procedurami nebo funkcemi, je vhodné jej definovat jako lokální v alokační proceduře.

```
(*  Příklad P10 – 2  *)
constructor (*****) TA.Init1 (*****)
( Text : string );
{Kopírovací kosntruktor třídy TA}
begin
  System.write( nl, 'TA.Init-Prvek: ', an,
       ' - nový text: ', Text, ' - pův.text: ', as );
  as := Text;
  ai := an;
  Inc( an );
end;
(********** TA.Init1 **********)

procedure (*****) TA.write (*****)
;
begin
  System.write( nl, 'Instance: ', ai,
       ". - Text = "', as, ', an = ', an );
end;
(********** TA.write **********)
```

V následujícím prográmku najdete specializovanou verzi bazénové alokační procedury a spolu s ní procedurku zaplňující definovanou oblast zadaným znakem; ve standardní knihovně nenabízí Pascal totiž funkci, která by byla obdobou céčkovské *memset*. Zaplňovací proceduru nedefinujeme jako lokální, aby ji mohla sdílet i obecná bazénová alokační procedura – tu si definujeme za chvilku.

```
(*  Příklad P10 – 3  *)
procedure (*****) StrFill (*****)
( p:Pointer; Velikost:word; c:char );
(* === verze pro Pascal 6.0 === *)
type
  tc = array[0..10000] of char;
  tpc= ^tc;
var
  pc : tpc;
  i : integer;
begin
  pc := p;
  for i:=0 to Velikost-1 do
  pc^[i] := c;
end;
(********** StrFill **********)
```

```
procedure (*****) newA (*****)
( var p:TpA; poc:char );
var
  bazen : array[0..MAX_NBA]of TA;          {Bazén pro alokaci dočasných
                                            proměnných}
const
  bf : integer             {Index prostoru pro další prvek - }
      = 0;                 {první proměnná se umístí na počátek bazénu}
begin
  write( nl, '=== newA1( char = ', poc, ' )',
       ' - bf = ', bf );
  p := @bazen[ bf ];
  Inc( bf );
  if( bf > MAX_NBA )then
       bf := 0;
  StrFill( p, sizeof(TA), poc );
end;
(********** newA **********)
```

Porovnáme-li následující globální bazénovou alokační proceduru s jejím „pluskovým“ ekvivalentem, zjistíme, že je opět o něco jednodušší. Za prvé se nesnaží (a ani nemůže) o náhradu systémového *new*, takže se nepotřebujeme starat, zda její chování odpovídá konvencím a zda ji nějaká část programu nepoužije „za našimi zády“ předčasně. Z toho důvodu jsme do ní nezařadili ani globální proměnné, které by počítaly jednotlivá volání procedury a celkový počet alokovaných bajtů.

```
 (*  Příklad P10 - 4  *)
const MAX_BAZ = 100;  {Velikost globálního bazénu }

procedure (*****) BazNew (*****)
( var p:Pointer; Velikost:word; poc:char );
const
  bazen : array[0..MAX_BAZ] of char        {Lokální statická proměnná }
      = '';
  ab : word = 0;           {Ukazatel na počátek připravené oblasti}
begin
  write( nl, '=== BazNew( size_t = ', Velikost,
       ', char ', poc, ' ) - index = ', ab );
  p := @bazen[ ab ];       {Předpokládaná adresa alokace }
  Inc( ab, Velikost );
  if( ab > MAX_BAZ )then   {Vejde se do bazénu? }
  begin                    {Nevejde => budeme ji alokovat od }
       p := @bazen[0];     {počátku bazénu }
       ab := Velikost;     {Proměnnou ab přesuneme za ni }
       if( ab >= MAX_BAZ )then begin        {Pokud se stále nevejde,
  předčasně}
         writeln( nl, nl, 'CHYBA' );        {ukončíme program}
         halt;
       end;
  end;
  StrFill( p, Velikost, poc );
end;
(********** BazNew **********)
```

Následující testovací prográmek se snaží ukázat, jak je možno speciální alokační proceduru používat. Nelekněte se toho, že je v něm použita i procedura *BazNew7*, kterou jsme doposud nedefinovali. Povíme si o ní v následujícím dílu, takže pokud chcete prográmek testovat hned, uzavřete prozatím její volání mezi komentářové závorky nebo je vyčleňte pomocí direktiv pro podmíněný překlad.

```
(*  Příklad P10 – 5  *)
var
  pah : TpA;
  pa0 : TpA;

procedure (*****) Test (*****)
;
type
  TvA = array[ 0..4 ] of TA;
  TpvA = ^TvA;
var
  pa1, pa2, pa3, pa4 : TpA;
  pv : TpvA;
  i : integer;
begin
  if( an = 0 )then    {Oddělení výstupu na obrazovce od předchozího}
  write( nl,nl,nl,nl,'=========================================' );
  newA( pa0, '0' );   {Globální proměnná se specializovaným new }
  pa0^.Init1( 'Nultý' );
  pa0^.write; writeln;
  newA( pa1, '1' );   {Lokální proměnná se specializovaným new }
  pa1^.Init1( 'Prvni' );
  pa1^.write; writeln;
  BazNew( Pointer(pa2), sizeof(TA), '2' );{Lokální proměnná s globálním
  bazénovým new}
  pa2^.Init1( 'Druhý' );
  pa2^.write; writeln;
  BazNew7(Pointer(pa3), sizeof(TA), '3' );{Lokální proměnná s globálním
  bazénovým new}
  pa3^.Init1( 'Třetí' );
  pa3^.write; writeln;
  new( pa4, Init1( 'Čtvrtý' ) ); {Lokální standardně alokovaná }
  pa4^.write; writeln;           {a inicializovaná proměnná }
  new( pah, Init1( 'Halda' ) );  {Globální standardně alokovaná }
  pah^.write; writeln;           {a inicializovaná proměnná }
  BazNew( Pointer(pv), 5*sizeof(TA), 'X' );
  for i:=0 to 4 do begin
       pv^[i].Init1( 'Vektor' );
       pv^[i].write; writeln;
  end;
  writeln;
end;
(********** Test **********)
```

8. Dodatek

V tomto dodatku najdete především stručný přehled rozšíření, se kterými se můžeme setkat v Turbo Pascalu 7.0 a v Delphi. Kromě toho jsme sem zahrnuli i několik užitečných programátorských triků.

8.1 Nejdůležitější novinky Turbo Pascalu 7.0

Většina výkladu o Pascalu v této knize je založena na Turbo Pascalu verze 6.0. Proto se nyní podívejme, jaká hlavní rozšíření najdeme v poslední dosovské verzi 7.0. O rozšířeních, které přinesl další borlandský překladač Pascalu – Delphi – si stručně povíme v následující podkapitole.

Direktiva public

Prvním rozšířením, s nímž jste se již v našich ukázkách setkali, je zavedení klíčového slova **public**. Toto klíčové slovo uvozuje v definici objektového typu sekci deklarací veřejně přístupných složek. Sekce **private** a public přitom můžeme libovolně prokládat.

Nepříjemné ovšem je, že o atributech v sekcích **private** nevědí nejen ostatní moduly, ale bohužel ani debugger, takže se domníváme, že při ladění stejně nebudou programátoři sekci **private** používat. Jinými slovy, alespoň při ladění budou nechávat všechny složky veřejné tak, jak to kdysi zavedla verze 5.5.

Konstantní parametry

Druhým rozšířením je zavedení konstantních parametrů podprogramů. Konstantní parametry se chovají jako parametry předávané hodnotou, přičemž překladač zaručuje, že jejich hodnotu programátor v těle procedury omylem nezmění.

Otevřená pole

Dalším příjemným rozšířením je zavedení otevřených polí jako parametrů. Možná si vzpomínáte, že když jsme začali pracovat s poli, stěžovali jsme si na pascalskou sterilní typovou kontrolu, která neumožňovala definovat čistým způsobem proceduru nebo funkci, jejímž parametrem by bylo pole o předem neznámém počtu prvků. Obcházeli jsme to tak, že jsme pro daný parametr vypnuli typovou kontrolu a definovali jej jako netypový (předávali jsme jej odkazem bez udání typu). Uvnitř procedury jsme pak definovali nějaké dostatečně velké pole, které mělo prvky stejného typu, a toto pole jsme direktivou **absolute** uložili v paměti na místo, odkud jsme převzali onen netypový parametr – mohli bychom říci, že jsme tímto polem onen netypový parametr překryli.

Nová verze Pascalu nám již umožňuje definovat parametry podprogramu jako otevřená pole, tj. jako pole, jejichž přesný rozměr definovaný podprogram ještě nezná. Ta-

kovýto parametr deklarujeme podobně jako „obyčejné" pole, pouze neuvedeme specifikaci indexu, např.:

```
function Suma( const Pole: array of real );
```

Otevřená pole můžete předávat všemi třemi způsoby, tj. hodnotou, odkazem (referencí) i jako konstanty (viz předchozí deklarace).

Otevřená pole pak můžete v programu používat stejně jako pole, jejichž index se pohybuje od 0 do *N*-1, kde *N* je počet prvků pole (tedy jako pole v Céčku). Oproti klasickému jazyku C autoři Pascalu svá pole ještě vylepšili: podprogram je schopen skutečné rozměry daného pole zjistit, a to dokonce hned dvěma způsoby: buď pomocí knihovní funkce *Sizeof*, která vrátí velikost odpovídajícího skutečného parametru v bajtech, nebo pomocí knihovní funkce *High*, která vrátí index posledního prvku.

Abychom si o této nové možnosti Pascalu udělali názornou představu, připravili jsme kratičkou ukázku se dvěma podprogramy.

Prvním z nich je procedura *Seq*, jíž se odkazem předává pole celých čísel. Jejím úkolem je inicializovat toto pole čísly počínaje hodnotou jejího druhého parametru.

Druhým podprogramem je pak funkce *Prumer*, jejímž prvním parametrem bude pole *Mereni*, které bude předáváno jako konstantní parametr. Překladač tedy zaručí, že se jeho hodnoty v těle funkce Prumer nezmění. Funkce spočte aritmetický průměr prvých *N* prvků pole *Mereni*, přičemž N bude jejím druhým parametrem. Tato funkce tedy respektuje to, že na konci pole mohou být i položky, jejichž hodnoty se do aritmetického průměru započítávat nebudou.

```
(*    Příklad P11 – 1    *)
procedure (*****) Seq (*****)
( var Pole : array of integer; Start:integer );
var
  i:integer;
begin
  for i:=Start to High( Pole ) do
      Pole[ i ] := i;
end;
(********** Seq **********)
function (*****) Prumer (*****)
( const Mereni : array of real; N:integer )
: real;
var
  i:integer;
  s:real;
begin
  s := Mereni[ 0 ];
  for i:=1 to N do
  s := s + Mereni[ i ];
  Prumer := s / N;
end;
(********** Prumer **********)
```

Znakové řetězce končící nulou

Další novinkou sedmé verze pascalského překladače je nový datový typ, který je ekvivalentem klasických řetězců jazyka C a který je v manuálu označován **jako nulou končící řetězec** (*null terminated string*). Tento typ budeme v dalším textu pro úsporu místa označovat NKŘ.

Když kdysi – před téměř patnácti lety – přišel Turbo Pascal s datovým typem řetězců (*string*) a začlenil řetězcové operace do syntaxe jazyka, sklidil za to velký potlesk. Postavil tím totiž práci s řetězci v Pascalu na obdobnou úroveň, s jakou se do té doby z rozšířených jazyků pyšnil pouze Basic. S příchodem "Woken" (rychle se vžívající český název pro nechutně neskloňovatelná MS Windows) se však situace trochu změnila. Asi víte, že Wokna jsou naprogramována v jazyku C (v C nebo C++ je ostatně naprogramována naprostá většina velkých programových balíků), a proto jsou i veškeré řetězcové operace postaveny na koncepci řetězců používané v tomto jazyce – tj. na řetězcích, které nejsou uvozeny ničím, co by dávalo předem tušit jejich délku, a místo toho jsou ukončeny prázdným znakem, tj. znakem s kódem 0.

Aby nedocházelo k potížím při komunikaci s "wokenními" obslužnými procedurami, bylo nutno do Pascalu začlenit céčkovsky koncipované textové řetězce – NKŘ. Operace s nimi však tentokrát nejsou zakomponovány do syntaxe jazyka, ale pracuje se s nimi (podobně jako v jazycích C a C++) prostřednictvím řady procedur a funkcí, které jsou všechny součástí jednotky (unit) *strings*[18].

Spolu s NKŘ převzali tvůrci Turbo Pascalu 7.0 z Céčka i základy adresové aritmetiky a práce s poli znaků.

Než se pustíme do podrobnějšího výkladu, ukážeme si ještě možné podoby procedur *StrFill7* a *BazNew*, ve kterých jsme využili některých možností nabízených sedmou verzí překladače.

```
(*   Příklad P11 – 2   *)
procedure (*****) StrFill7 (*****)
( p:Pointer; Velikost:word; c:char );
var
  pc : PChar;
  kc : pChar;
begin
  pc := p;
  kc := pc + Velikost;
  while( pc < kc )do
  begin
       pc^ := c;
       Inc( pc );
  end;
(**)
end;
(********** StrFill7 **********)
```

[18] O významu a použití jednotlivých podprogramů z této jednotky se zde nebudeme rozepisovat, neboť všechny potřebné informace najdete v manuálech nebo v nápovědě k překladači.

```
procedure (*****) BazNew7 (*****)
( var p:Pointer; Velikost:word; poc:char );
const
  bazen : array[0..MAX_BAZ] of char        {Lokální statická proměnná }
          = '';
  ab : PChar = @bazen;           {Ukazatel na počátek připrav.oblasti}
  uk : PChar = @bazen[MAX_BAZ];  {Ukazatel na konec bazénu }
begin
  write( nl, '777 BazNew7( size_t = ', Velikost,
       ', char ', poc, ' ) - index = ', (ab - @bazen) );
  p := ab;                       {Předpokládaná adresa alokace }
  Inc( ab, Velikost );
  if( ab > uk )then              {Vejde se do bazénu? }
  begin                          {Nevejde => budeme ji alokovat od }
      p := @bazen[ 0 ];          {počátku bazénu }
      ab := @bazen[ Velikost ]; {Proměnnou ab přesuneme za ni }
      if( ab >= uk )then begin  {Pokud se stále nevejde, předčasně}
        writeln( nl, nl, 'CHYBA' );        {ukončíme program}
        halt;
      end;
  end;
  StrFill7( p, Velikost, poc );
end;
(********** BazNew7 **********)
```

Především si řekneme, že pro práci s NKŘ musíme nastavit přepínač pro použití rozšířené syntaxe: v dialogovém okně [*Options* | *Compiler*] v bloku *Syntax options* zaškrtneme *Extended syntax* nebo použijeme „dolarovou poznámku" {$X}.

Deklarace

NKŘ nemá – na rozdíl od tradičního typu *string* – žádné podstatné omezení délky. Je totiž omezena pouze velikostí segmentu paměti, tedy délkou 65535 znaků; pro velkou většinu použití by to mělo stačit. Skládá se z posloupnosti libovolného počtu nenulových znaků, ukončené nulovým znakem #0. V Turbo Pascalu jej deklarujeme jako pole znaků, indexované od 0:

```
type
  Tretez = arrray[0..MAXCHAR] of char;
```

Při práci s NKŘ lze s výhodou používat ukazatele typu PChar. To jsou vlastně ukazatele na char, pro které je zavedena rozšířená kompatibilita vzhledem k přiřazení a prom které jsou definovány některé aritmetické operace.

Proměnné typu *PChar* můžeme přiřadit hodnotu typu *PChar*, *string* nebo NKŘ, např. takto:

```
var
  up, uq, ur: PChar;
  nkr: Tretez;
begin
 up := nkr;                    {přiřazujeme NKŘ}
 uq := 'Skoč do zdi, postavo!' {přiřazujeme řetězcový literál}
 ur := up;                     {přiřazujeme proměnné typu PChar}
end;
```

První z těchto příkazů způsobí, že se do proměnné *up* uloží adresa počátku pole *nkr*. Druhý příkaz přiřadí proměnné *uq* adresu místa v paměti, kde bude uložen citovaný povzbudivý výrok, a to jako řetězec, zakončený nulou. Stejného výsledku bychom dosáhli, kdybychom napsali

```
var
  up: PChar
const
  pom: array[0..22] of char = 'Skoč do zdi, postavo!'#0;
begin
 up := @pom;
end;
```

Všimněte si, že jsme v inicializaci konstanty *pom* museli ke znakovému řetězci připojit i nulový znak, #0.

Řetězcové literály můžeme použít i při inicializaci typových konstant typu *PChar*:

```
const
 tyden: arrayof PChar = (
       'pondělí', 'úterý', 'středa', 'čtvrtek',
       'pátek', 'sobota', 'neděle');
```

Indexování ukazatelů

Ukazatele typu *PChar* můžeme idexovat, jako kdyby to byla pole. Vezměme např. proměnnou *up*, které jsme před chvilkou přiřadili jistý výkřik. Je-li *c* proměnná typu **char**, přiřadíme jí příkazem

```
c := up[3];
```

hodnotu znaku ‘č’, který je v čtvrtý v řetězci, na který *up* ukazuje (a protože jde o pole indexované od nuly, má index 3).

Adresová aritmetika

Vezměme dva ukazatele *P* a *Q* typu *PChar* a číslo *I* typu *word*. Pak můžeme při rozšířené syntaxi provést následující aritmetické operace:

1. K ukazateli můžeme přičíst hodnotu typu *word* (celé číslo bez znaménka):
   `P + I` nebo `I + P`
   Výsledkem je ukazatel, který ukazuje o *I* bajtů dále než *P*. Jinak řečeno, celé číslo *I* se přičte k ofsetové části ukazatele *P*.
2. Od ukazatele můžeme odečíst hodnotu typu *word*:
   `P - I`
   Výsledkem je ukazatel, který ukazuje na adresu o *I* bajtů nižší než *P*. Jinak řečeno, celé číslo *I* se odečte od ofsetové části ukazatele P.
3. Můžeme od sebe odečíst dva ukazatele,
   `P - Q`

a dostaneme počet bajtů, o který leží P v paměti dále než Q. V tomto případě se odečtou ofsetové části obou ukazatelů a výsledkem hodnota typu word. Takovéto odečítání má ale smysl pouze v případě, že P i Q ukazují do stejného NKŘ.

NKŘ a standardní procedury

Je-li povolena rozšířená syntax, lze na NKŘ použít standardní procedury *Read*, *Readln* a *Str*. Standardní procedury *Write*, *Writeln*, *Val*, *Assign* a *Rename* lze použít jak na NKŘ tak i na ukazatele typu *PChar*.

8.2 Delphi a Object Pascal

Delphi je profesionální vývojový prostředek, který firma Borland uvedla na trh v roce 1995. Je určen ke tvorbě aplikací pro Windows (verze 2.0, uvedená počátkem roku 1996, je určena pro Windows 95) a je založen na Object Pascalu (OP) – tedy na objektově orientovaném dialektu jazyka Pascal, který navazuje na předchozí implementace (Turbo Pascal, Borland Pascal).

Popis Delphi, jeho jazyka, prostředí a nástrojů, které programátorovi nabízí, by vydal sám o sobě na několik svazků. To si bohužel vzhledem k omezenému rozsahu této knihy nemůžeme dovolit, a proto jsme zařadili pouze informativní přehled novinek, se kterými se v nové variantě borlandského Pascalu setkáme.

V tomto dílu se podíváme pouze na neobjektové vlastnosti jazyka a na zapouzdření objektových typů.

Neobjektové novinky Object Pascalu

Mnohé z novinek, se kterými se v nejnovější variantě borlandského Pascalu setkáváme, mají za cíl usnadnit spolupráci s programy, napsanými v jiných programovacích jazycích, zejména v C a C++.

Booleovské a celočíselné typy

Vedle tradičního typu *boolean* se v OP setkáme s typy *ByteBool*, *WordBool* a *LongBool*, které zabírají 1, 2, resp. 4 bajty.

Pokud jde o celá čísla, rozlišuje OP tzv. *generické* a *fundamentální* typy.

Rozsah a formát fundamentálních typů nezávisí na procesoru ani na operačním systému, pro který je program určen, a měl by tedy být stejný ve všech implementacích OP (tj. jak ve verzi pro Windows 3.1 tak i ve verzi pro Windows 95).

Fundamentální celočíselné typy jsou *Shortint* a *Byte* (jeden bajt se znaménkem a bez něj), Smallint a Word (dvoubajtová čísla se znaménkem a bez něj) a Longint (čtyřbajtová čísla se znaménkem). Jejich rozsahy uvádí následující tabulka:

typ	rozsah
Shortint	-128 .. 127
Smallint	-32768 .. 32767
LongInt	-2147483648 .. 2147483647
Byte	0 .. 256
Word	0 .. 65535

Generické typy jsou *Integer* a *Cardinal*. Typ *Integer* představuje v 16bitovém prostředí dvoubajtové, ve 32bitovém prostředí čtyřbajtové celé číslo se znaménkem. Typ *Cardinal* představuje v 16bitovém prostředí dvoubajtové, ve 32bitovém prostředí čtyřbajtové celé číslo bez znaménka. Jejich rozsahy najdete v následující tabulce:

typ	rozsah v 16bitovém prostředí	rozsah ve 32bitovém prostředí
Integer	-32768 .. 32767	-2147483678 .. 2147483647
Cardinal	0 .. 65535	0 .. 4294967295

Procedurální typy (ukazatele na procedury a funkce)

Již předchozí verze borlandského Pascalu znaly procedurální a funkcionální typy (tj. ukazatele na procedury nebo funkce). OP v Delphi umožňuje používat i ukazatele na metody objektových typů. Deklarují se podobně jako „obyčejné" procedurální typy, pouze obsahují navíc frázi **of object**; např.:

```
type
TobjMet = procedure(i: integer) of object;
```

Proměnná tohoto typu se skládá ze dvou ukazatelů: z ukazatele na metodu a z reference na objekt, ke kterému patří.

Je-li *Met* metoda třídy *Tobj*, přiřadíme její adresu proměnné *t* typu *TobjMet* příkazem

```
t := @Tobj.Met;
```

Procedury a funkce

Tradičně se procedury a funkce v Pascalu řídí pascalskou volací konvencí. Jde o způsob překladu volání těchto podprogramů, který znamená: parametry se ukládají na zásobník v pořadí, v němž jsou zapsány v příkazu, kterým ji voláme. Při návratu z procedury či funkce se o úklid na zásobníku stará zavolaný podprogram.

OP v Delphi umožňuje předepsat u některých procedur a funkcí volací konvenci jazyka C (parametry se na zásobník ukládají v pořadí obráceném, než v jakém jsou zapsány, a o úklid zásobníku se po návratu stará volající podprogram). K tomu slouží direktiva *cdecl*, která se zapisuje za hlavičku – podobně jako např. direktivy *near* nebo *far*.

Příklad:

```
function f(int n): integer; cdecl;
begin
 {...}
```

```
end;
```

Pascalská volací konvence ovšem vede zpravidla k efektivněji přeloženému programu. Proto se céčkovská volací konvence bude hodit především při volání funkcí z knihoven, napsaných v jazyku C.

Vracená hodnota

Jednou z příjemných novinek je, že funkce mohou vracet hodnotu téměř libovolného typu. Jedinou výjimku tvoří „staré“ objektové typy, deklarované pomocí klíčového slova **object**, a soubory.

V definiční deklaraci funkce můžeme vypočtený výsledek přiřadit identifikátoru funkce, podobně jako v předchozích verzích. Kromě toho můžeme vypočtenou hodnotu přiřadit lokální proměnné *Result;*.

Přiřazení proměnné *Result* znamená totéž jako přiřazení identifikátoru funkce; ovšem proměnnou *Result* můžeme používat také ve výrazech. Příklad uvidíme v následujícím odstavci.

Otevřená pole

O otevřených polích jsme již hovořili v souvislosti s novinkami Borland Pascalu 7.0. V Delphi najdeme další rozšíření, tzv. konstruktor otevřeného pole.

Při volání podprogramu, který má jako parametr otevřené pole, můžeme jako skutečný parametr uvést výčet prvků tohoto pole v hranatých závorkách. Podívejme se např. na funkci *Sum*, která vypočte součet prvků pole libovolné délky:

```
function Sum(var A: array of integer): integer;
var
i: word;
begin
  Result := 0;
  for i := 0 to High(A) do Result := Result + A[i];
end;
```

Při volání můžeme použít konstruktor otevřeného pole;

```
j := Sum([1, i, 5*j+2, k+l]);
```

Třídy

V OP zůstaly zachovány objektové typy tak, jak jsme se s nimi seznámili v předchozích kapitolách. Vedle toho ale přichází OP s novým objektovým typem – třídou (*class*).

Deklarace třídy je na první pohled podobná deklaraci tradičního typu **object**; na první pohled se liší jen klíčovým slovem **class**, které nahradilo tradiční a poněkud nešťastné klíčové slovo **object**.

Instance tříd

Instance tříd jsou vždy dynamické, tj. vytvářejí se až za běhu programu. Deklarujeme-li *x* proměnnou třídy *T*, deklarujeme vlastně referenci na objekt; např.:

```
type T = class
i: integer;
{...}
end;
var x: T;
```

Proměnná *x* je reference na objekt typu *T*. To znamená, že obsahuje ukazatel, který se ale automaticky dereferencuje. Píšeme tedy

```
x.i := 11;
```

nikoli

```
x^.i := 11;
```

Proměnné typu třída můžeme také přiřadit hodnotu **nil**. Několik proměnných typu třída může obsahovat referenci (odkaz) na týž objekt.

Složky tříd

Třída je opět strukturovaný typ, který může obsahovat atributy (v manuálech označované jako *field* - tedy pole), metody (*method*) a vlastnosti (*property*).

Atributy a metody již známe, zastavme se tedy krátce u vlastností. Vlastnost, property, je pojmenovaná složka objektů dané třídy spolu s akcemi, sdruženými s přístupem k ní (tj. s čtením a změnami jejích hodnot). Podobně jako atributy slouží i vlastnosti k ukládání hodnot, které nějak charakterizují určitou instanci dané třídy, na rozdíl od nich poskytují ale větší možnost řídit přístup k nim.

Deklarace vlastnosti začíná klíčovým slovem property, za kterým následuje identifikátor vlastnosti. Pak mohou následovat specifikace metod, které se mají používat ke čtení hodnot vlastnosti, k ukládání hodnot do ní a předpis implicitní hodnoty.

Deklarace vlastnosti může mít např. tvar

```
type
  TBod = class
  private
       x, y: integer;
       HodnotaBarvy: integer;
       procedure NastavBarvu(Hodn: integer);
  public
       property Barva: integer read HodnotaBarvy write NastavBarvu;
{...a další ...}
end;
```

Použijeme-li vlastnost ve výrazu, vezme („přečte“) se její hodnota prostřednictvím metody nebo pole, uvedené za specifikátorem read. Je-li Bod instance třídy TBod, znamená příkaz

```
i := Bod.Barva;
```

totéž jako

```
i := Bod.HodnotaBarvy;
```

a příkaz

```
Bod.Barva := BILA;
```

neznamená nic jiného než

```
Bod.NastavBarvu(BILA);
```

Metody nebo atributy, které se při práci s vlastností budou používat, musíme deklarovat předem.

Přístupová práva

OP nabízí dva nové specifikátory přístupových práv. Vedle možností **public** a **private**, které byly již v Borland Pascalu 7.0, tu najdeme ještě možnosti **published**, **protected** a **automated**.

Možnosti **protected** a **automated** se týkají dědičnosti, proto se jim zatím vyhneme. Možnost **published** znamená stejná přístupová práva jako možnost **public**, navíc se pro „publikované“ složky generují informace, potřebné pro určování jejich typu za běhu programu. Používá se zejména pro vlastnosti (property) u tzv. komponent, předdefinovaných objektů, které lze používat ke „stavebnicovému“ (vizuálnímu) návrhu aplikací. S vygenerovanými informacemi o typech totiž dokáže prostředí Delphi pracovat i v době návrhu aplikace a díky tomu můžeme měnit vlastnosti komponent při programování pomocí vestavěných nástrojů.

Předběžné deklarace

Občas se stane, že chceme ve třídě *A* použít odkaz na třídu *B*, a naopak ve třídě *B* potřebujeme odkaz na třídu *A*. V takovém případě použijeme předběžnou deklaraci jedné z tříd.

```
type
  A = class; {předběžná deklarace}
  B = class
       alfa: A; {reference na třídu A}
       {...}
  end;
  B = class
       beta: B;
  {...}
  end;
```

Dále uvidíme, že deklarace atributu *alfa* vytvoří pouze referenci, tedy vlastně ukazatel, takže překladači nebude vadit, že neví, jak třída *B* vlastně vypadá.

Reference na instance, reference na třídy

Je-li *A* třída, pak deklarací

```
var x: A;
```

vytvoříme referenci na objekt třídy *A*. Hodnotou proměnné *x* tedy bude adresa konkrétní instance třídy *A*.

OP ale umožňuje také pracovat s referencemi na třídu jako celek. Referenci na třídu deklarujeme pomocí fráze **class of**, za kterou následuje identifikátor třídy. Podívejme se na příklad:

```
{deklarace třídy}
type
  TBod = class
       {...}
  end;
{reference na třídu}
  RefBod = class of TBod;
```

Proměnné typu *RefBod* přiřadíme hodnotu příkazem

```
var rb: RefBod;
{...}
rb := TBod;
```

Všimněte si, že identifikátor třídy představuje hodnotu typu „reference na danou třídu“. Proměnné tohoto druhu mají význam především ve spojitosti s dědičností a s metodami tříd.

Metody tříd

OP také (konečně) zavedl metody tříd - tedy metody, které pracují s třídou jako celkem, nikoli s jednotlivou instancí. (Můžeme to chápat také tak, že „obyčejné“ metody, tj. metody instancí, pracují s referencí na instanci, zatímco metody tříd pracují s referencí na třídu jako celek).

Hlavička metody třídy začíná klíčovým slovem **class**; jinak se neliší od hlavičky metody instancí. Deklarujeme např. *VypisInfo* ve třídě *TBod* jako metodu třídy:

```
type
  TBod = class
  public
       class
procedure VypisInfo;
  {...}
  end;
class procedure TBod.VypisInfo;
begin
  {...}
end;
```

V metodách třídy se nemůžeme odvolávat na atributy, neboť metody třídy nepracují se žádnou konkrétní instancí. Ze stejného důvodu v nich nemůžeme také volat metody instancí. Můžeme v nich ovšem volat jiné „třídní“ metody dané třídy. Při volání metod třídy můžeme ke kvalifikaci použít referenci na třídu (tj. identifikátor třídy nebo proměnnou typu reference na třídu, které jsme přiřadili hodnotu); můžeme také použít referenci na instanci, podobně jako při volání „obyčejných“ metod.

Self

Podobně jako v metodách „starých“ objektových typů, deklarovaných pomocí klíčového slova **object**, je i v metodách objektových typů, deklarovaných pomocí klíčového slova **class**, k disposici implicitní parametr *Self*.

V metodách instancí obsahuje *Self* odkaz na instanci, pro kterou danou metodu voláme. V metodách tříd obsahuje odkaz (referenci) na třídu, pro kterou danou metodu voláme.

8.3 Makro assert

Toto makro je definováno v hlavičkovém souboru *ASSERT.H* (ten musíte samozřejmě před použitím makra vložit příkazem **#include**) a jeho úkolem je otestovat podmínku, kterou mu předáváte jako parametr. V případě, že tato podmínka není splněna, přeruší běh programu a do standardního chybového výstupu zapíše zprávu o chybě. Tato zpráva bude obsahovat jméno souboru a číslo řádku, na němž se makro nachází, a testovanou podmínku.

Pokud svůj program odladíte a budete chtít tato testování z programu v zájmu zvýšení efektivity odstranit, stačí pouze na počátku každého modulu definovat preprocesorovou konstantu *NDEBUG*:

```
#define NDEBUG
```

Pascalské knihovny podobné makro nenabízejí. Není ovšem problém definovat si funkci, která je zastoupí.

8.4 Použití prázdného ukazatele

Na závěr si ukážeme jednu pomůcku, která může usnadnit céčkařům život.

Jednou z poměrně častých chyb je, že použijeme prázdný ukazatel (obsahující 0), jako kdyby obsahoval skutečně plnohodnotnou adresu. Program s takovouto chybou může dlouhou dobu běžet bez problémů, až najednou nečekaně „spadne".

Borlandské překladače nám nabízejí pomůcku, jak tuto chybu v některých paměťových modelech odhalit. Ve všech paměťových modelech kromě drobného totiž umísťují na počátek datové oblasti 4 volné bajty, za kterými následuje copyright, např.

```
Borland C++ - Copyright 1991 Borland Intl.
```

Novější překladače, tj. Borland C++ 4.x, zde ukládají 4 volné bajty, za kterými následuje

```
NULL CHECK
```

a další volné bajty.

Pokud náhodou použijete prázdný blízký ukazatel na data (v malém a středním modelu jsou implicitní) a zapíšete něco na adresu, na kterou „ukazuje", přepíšete nejspíš právě tuto oblast, která se však naštěstí na nic důležitého nepoužívá.

Ukončovací kód, používaný v borlandských překladačích, zavolá po skončení hlavní části programu (tj. toho, co jsme napsali my) ještě před předáním řízení operačnímu systému epilog programu, který zkontroluje obsah výše zmíněného počátku datové oblasti. Pokud zde najde něco jiného, než úvodní čtyři nulové bajty následované odpovídajícím textem (přesněji něco s jiným kontrolním součtem), napíše na obrazovku zprávu

```
Null pointer assignement
```

a my hned víme, že program není zcela v pořádku. Problémem ovšem zůstává, jak avizovanou chybu v programu najít.

Možností, jak si hledání usnadnit, je několik: Poměrně nenáročná je definice funkce, která bude tisknout obsah počátku datové paměti – nejprve čtyři datové bajty, pak textový řetězec. Volání této funkce umístíme na klíčová místa programu tak, abychom kontrolou jejího výstupu dokázali nejprve zhruba a pak stále jemněji lokalizovat místo, kde k chybě dochází.

Pokud nechceme narušovat výstupy svého programu na obrazovku, můžeme si pro tuto funkci definovat nějaký její vlastní výstupní proud, do nějž bude funkce zapisovat výsledky svých testů. V naší definici bychom tedy např. přidali mezi deklarace statických objektů deklaraci

```
static ofstream cerr( "Null_Poi.Ass" );
```

Tato lokální deklarace by pak překryla deklaraci stejnojmenného globálního proudu a místo systémového chybového výstupu by výstup naší funkce směřoval do definovaného souboru - v našem případě do souboru NULL_POI.ASS. Pokud bychom chtěli používat standardní chybový proud, stačí odsunout uvedenou deklaraci do komentáře. Výsledná definice by pak mohla vypadat např. následovně:

```
/*   Příklad C11 — 1   */
static void x( char * Text )
{
  //static ofstream cerr( "NulPoAss" );
  static int * L = 0;
  static int * H = (int *) 2;
  static char * B = (char*) 4;
  cerr << "\n" << Text << ": "
       << *H << ", " << *L << " - " << B << endl;
}
```

Výhodou řešení pomocí funkce je to, že takto můžeme funkci programu kontrolovat i při běžném provozu s tím, že uživatele, který náš program testuje nebo již dokonce používá, požádáme, aby v případě havárie zahrnul soubor NULL_POI.ASS mezi odkládané soubory, ze kterých se pak budeme snažit vyčíst co bylo příčinou chyby.

Pokud chceme tuto chybu hledat při běžném ladění pomocí debuggeru, nemusíme se obtěžovat s definicí funkce, ale můžeme kontrolovat obsah počátku datové paměti přímo ve sledovacím okně. Pro ten případ doporučujeme umístit do sledovacího okna následující dvě položky:

```
*(void far * *) 0, p
(char *) 4
```

První položka se poptává na hodnotu prvých čtyř bajtů; dokud je vše v pořádku, hlásí systém hodnotu NULL. Druhá položka se ptá na následující řetězec. Jeho správná hodnota závisí na typu překladače, avšak neobávám se, že byste narušení tohoto řetězce nerozpoznali.

Při ladění byste mohli využít ještě třetí možnost, a to použít samostatný debugger, který umí zastavit program ve chvíli, kdy se změní obsah zadané oblasti paměti. Výhodou tohoto řešení je, že chybné místo najdete hned poté, jakmile navodíte situaci, při které k oné chybě dochází. Nevýhodou je, že pokud nejste náležitě hardwarově vybaveni, chod programu se velice citelně zpomalí. Pokud však používáte počítač s mikroprocesorem 386 (stačí SX) nebo vyšším, můžete použít debugger TD386, který dokáže ohlídat označenou oblast paměti, aniž by se běh programu nějak viditelně zpomalil. (Tato poznámka je dnes už téměř zbytečná; počítače s procesory 80286 a staršími se už téměř nevyskytují.)

Klíčovým problémem celého ladění však nebývá většinou ani tak lokalizace chyb, o kterých již víme, ale navození situací, v nichž se tyto chyby projevují. V tom vám však již neporadíme.

Rejstřík

I

K

L

M

N

O

P

Q

R

Ř

S

Rudolf Pecinovský, Miroslav Virius

Objektové programování I

Uèebnice s pøíklady v Turbo Pascalu a Borland C++

Odpovìdný redaktor Michal Dvoøák
Návrh a grafická úprava obálky Adéla Bìlovská

Poèet stran 232

Vydala Grada Publishing, spol. s r.o.
Na Poøíèí 17, Praha 1

1996
Vydání 1.

Vytiskly Tiskárny Havlíèkùv Brod, a.s.
Husova ulice 1881, Havlíèkùv Brod