OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ V C++

Filip Brázdil

1 ÚVOD

1.1 Historie, souvislosti

Programovací jazyk C++ je ze všech objektově orientovaných jazyků v praxi patrně nejrozšířenější. U nás jej možná o trošku předstihuje Turbo Pascal, ve světě je však jeho postavení podstatně pevnější. Je to dáno především návazností jazyka C++ na rozšířený a zavedený jazyk C. Zatímco jiné objektově orientované jazyky nacházejí své uplatnění ponejvíce na univerzitách a vůbec ve výzkumu, v programátorské praxi je třeba navazovat na existující projekty, využívat již napsané knihovny a mnohdy se nedostává času nebo peněz na kompletní přeškolení programátorů.

Návaznost na C dovolila programátorům pracovat se známým jazykem, seznamovat se s novými rysy objektově orientovaného programování a postupně je začít využívat. A i potom, když už psali nové programy objektově navržené, mohli využívat stávajících knihoven funkcí jazyka C. Z teoretického a didaktického hlediska tento způsob přístupu k C++ patrně není nejvhodnější, z hlediska produktivity práce, zvláště v přechodovém období, to pro mnoho lidí byl přístup jediný možný. Nyní počet programátorů v C++ již prudce roste, jazyk se začíná vyučovat na univerzitách (na některých už i v zahajovacích kursech), pořádají se semináře a konference, vycházejí časopisy, je k dispozici stále více překladačů a dalších programovacích nástrojů pro C++.

Kromě jazyka C, kde je návaznost zcela jasná, lze u C++ najít ještě jednu zřetelnou souvislost s dalším programovacím jazykem – Simulou 67. Jazyk Simula vznikl, jak již jeho jméno napovídá, v roce 1967 a byl určen především pro modelování a simulaci. Syntakticky vycházel z jazyka Algol 60 ale zavedl podstatná rozšíření, pro která bývá nyní považován za první objektově orientovaný jazyk na světě. Jako první zavedl pojem třídy (s klíčovým slovem class), jako uživatelsky definovaného typu, který měl definovány nejen datové složky, ale také algoritmy pro specifickou práci s nimi. Od každé třídy se daly odvozovat třídy další, které dědily její vlastnosti.

Jazyk C++ vznikl zhruba v letech 1985 až 1986 v Bellových laboratořích firmy AT&T a jeho autorem byl Bjärne Stroustrup. Od té doby pokračuje jeho stálý a poměrně rychlý

vývoj, který se dosud nezastavil. C++ je sice nadmnožinou jazyka C, ale není nějakou jednoduchou nadstavbou toho jazyka (takové nadstavby vznikly také, např. Objective C). Je to zcela nově navržený objektově orientovaný jazyk a návaznost právě na C je až druhotnou záležitostí. Stroustrup hledal programovací jazyk, který by použil jako základ syntaxe a C zvolil pro jeho rozšíření pod operačním systémem UNIX. Snaha o co největší kompatibilitu s existujícím jazykem C mu sice přinesla mnoho práce a komplikací, ale podle očekávání se odrazila v úspěšnosti jazyka. Zájem programátorů z praxe byl mnohonásobně vyšší, než by se dal očekávat u jazyka kompletně nového, s dosud nezavedenou syntaxí.

Kompatibilita C++ s jazykem C je sice vysoká, ale není úplná. Hlavním rozdílem je především to, že C++ má velmi silnou typovou kontrolu, zatímco C nemá téměř žádnou. Proto při překládání jazyka C překladačem pro C++ může vzniknout řada varovných nebo i chybových hlášení, lze je však odstranit vložením explicitního přetypování. Rozdíly jsou výraznější mezi C++ a staršími verzemi jazyka C (Kernighan & Ritchie), nežli oproti normované verzi ANSI C. Je to proto, že standard ANSI C vznikl už v době existence C++ a řadu jeho vlastností převzal.

Podobnosti C a C++ využívá velká část překladačů C++, zejména ve světě UNIXu, které jsou psány jako tzv. „cfront", to znamená překladač z C++ do C, přičemž výsledný program v C se překládá kompilátorem, který je nedílnou součástí každé implementace operačního systému UNIX. Jako cfront je realizován i překladač pocházející od AT&T, který se považuje pro C++ za standard. Pro C++ zatím neexistuje norma ANSI, normovací výbor (X3J16) dosud jedná. Nemá pozici právě lehkou, protože jazyk C++ se neustále vyvíjí a výbor je tak stavěn před požadavky zahrnovat do normy nové a nové rysy jazyka. AT&T cfront existuje již ve verzi 3.0 a řada dalších překladačů z něj přechází, popřípadě dále rozšiřuje. V poslední verzi jsou největší novinkou šablony funkcí a tříd (templates), pro verzi následující se uvažuje se zapracováním výjimek (exceptions).

1.2 Poznámka o terminologii

Budu používat terminologii, kterou jsme použili a částečně vytvořili při práci na českých překladech dokumentace firmy Borland. Vycházeli jsme částečně z existující praxe, částečně z existujících norem a částečně z vlastní fantazie a jazykového citu. Za výslednou podobu této terminologie jsou odpovědní Ing. Rudolf Pecinovský, CSc. a Ing. Filip Brázdil.

2 Rozšíření jazyka C++

2.1 Neobjektová rozšíření

2.1.1 Odkazy

Poměrně drobným, ale podstatným rozšířením jazyka C++ oproti C jsou odkazy. Odkaz je vlastně proměnná, která obsahuje adresu nějakého objektu (ne ve smyslu objektově orientovaného programování), která může být v některých případech použita místo objektu samotného. Mohu např napsat

```
int i;
int &j=i;                    // J je odkaz na i
j=0;                         // Přiřadí se proměnné i
```

Uvedené použití však není typické. Typické je použití odkazů ve formálních parametrech podprogramů. V jazyku C, když bylo potřeba ve funkci měnit hodnotu některého parametru, bylo nutno jako parametr předat ukazatel a ve funkci pracovat s ukazatelem. Použití odkazů práci s parametry zjednodušuje, odpadá jedna dereference. Tento způsob se hodí i při předávání rozsáhlejších objektů, které se v C vždy celé kopírovaly na zásobník. Bezpečnost předávaných parametrů proti změně ve funkci lze zajistit klíčovým slovem const.

2.1.2 Rozšiřování funkcí

V jazyku C++ je možná jedna věc, která při prvním setkání vzbuzuje u programátorů značné překvapení: několik funkcí může mít stejné jméno. Rozlišuje se mezi nimi podle počtu a typu parametrů. Této vlastnosti je vhodné využívat tehdy, když chceme různými funkcemi provádět podobnou operaci na různých typech parametrů. V klasické knihovně programovacích jazyků nacházíme např. dvojice sin() a sinl(), atan() a atanl() a další, které pro různý typ parametrů mají různá jména. V C++ mohou mít jméno stené a lišit se pouze typem nebo počtem parametrů.

Klasickým příkladem použití jsou třeba funkce max a min. V C se pro ně používají makra, ale zavedení opravdových funkcí umožňuje úplnou typovou kontrolu a bezpečnost.

Zvláštním případem tohoto jevu je předefinování standardních operátorů jazyka C jako je +, –, ale i třeba [], tak aby pracovaly s uživatelskými typy dat.

2.2 Objektově orientované rysy jazyka C++

2.2.1 Třídy

Chceme-li hovořit o objektově orientovaném programování v C++, první pojem, na který narazíme, je třída. Třída (klíčové slovo **class**) je zobecněním pojmu struktury z jazyka C. Hlavní rozdíl spočívá v tom, že ve třídě jazyka C++ se deklarují nejen datové složky struktury, ale i funkce (metody, member functions) pro práci s nimi. Je běžné, že v OO programu jsou všechny datové složky třídy zvenku nepřístupné a že se s nimi dá manipulovat pouze prostřednictvím metod. Cílem tohoto přístupu je umožnit pouze povolené manipulace s daty a zamezit vzniku chyb plynoucích ze změn implementace. Jako ukázku deklarace třídy si můžeme uvést deklaraci třídy String pro práci s textovými řetězci:

```
class String                    // Třída pro práci s řetězci
                                // Dvěma lomítky se v C++ označuje komentář
{
protected:
        unsigned delka;         // Délka řetězce
        char *retez;            // Vlastní data
public:
        String();               // Prázdný konstruktor
        String(const char *);   // Inicializující konstruktor
        String(const String &); // Kopírovací konstruktor
        ~String();              // Destruktor
        unsigned length() const { return delka;}
                                // Zjistí délku řetězce
        void up();              // Převede řetěz na velká písmena
        unsigned copy(char *cil) const { strcpy(cil, retez); return delka;}
                                // Kopíruje řetěz
        int operator[](const unsigned) const;
                                // N-tý znak z řetězu
};
```

V tomto příkladu je použito hned několik nových rysů jazyka C++ oproti C. Jak je budu postupně uvádět, budu se k příkladu vracet.

Prozatím si všimneme syntaxe deklarace: je skutečně odvozena od syntaxe struktury a mezi daty se volně vyskytují i deklarace funkcí. Jsou dvojího druhu: buďto pouze prototypy, jejichž těla budou deklarována někde později, nebo kompletní definice. Kompletní definice se používají pro velmi krátké funkce s jednoduchým, popřípadě žádným algoritmem (length je příkladem zcela typickým) a jsou automaticky překládány jako

fiktivní (inline) – ne tedy voláním funkce, ale přímým zápisem na místo použití; podobně jako při rozvoji makra. Funkce, které mají uvedeny pouze prototypy, je třeba později definovat. Aby se poznalo, že patří k této třídě, používá se následující syntaxe:

```
void String::up(void)          // Převod na velká písmena
      { char *ptr=retez;
      while (*ptr) { *ptr=toupper(*ptr);
                        ptr++;
                   }
}
```

Mezi funkcemi se také objevují dosud neznámá klíčová slova: **protected** a **public**. Patří k nim ještě **private** a společně určují možnosti přístupu k jednotlivým složkám třídy. Složky, deklarované v sekci **public** jsou volně dostupné zvenku pomocí známých operátorů pro struktury. Složky vypsané v sekci **protected** jsou přístupné pouze vlastním metodám třídy popřípadě třídy z ní odvozené. Bývá zvykem deklarovat všechny datové složky třídy jako **protected** (nebo **private**) a přístup k nim povolit pouze prostřednictvím funkcí definovaných jako **public**. V našem příkladu tento přístup reprezentuje chráněná datová složka delka, zpřístupněná pomocí funkce length(). Zajímá-li mne délka řetězce, nepoužiji Retez.delka, jako v C, ale Retez.length(). Toto samozřejmě nemohu použít na levé straně přiřazovacího příkazu, čímž zajišťuji zapouzdření dat. **protected** mohou být i funkce. Ty pak jsou volatelné pouze z jiných metod této třídy nebo třídy z ní odvozené.

Posledním klíčovým slovem pro řízení přístupu je **private**, které znamená nejpřísnější omezení: ke složce mají přístup pouze metody deklarované v této třídě, nikoli ve třídách odvozených.

V deklaracích funkcí je použito ještě klíčové slovo **const**. Je sice známé, ale v jiném kontextu. Zde je použito v deklaraci funkce za seznamem parametrů a znamená, že tato funkce nesmí měnit datové složky třídy. Překladač všechny takové pokusy zachytí a hlásí chybu.

Nesmírně důležitou vlastností tříd je to, že lze definovat nové třídy jako rozšíření některých už existujících. Taková nová odvozená třída má všechny složky i metody třídy původní a navíc to, co je obsaženo v její vlastní definici. Tomuto procesu se říká dědění a jedna třída může dědit i od několika rodičů současně (na rozdíl např. od Pascalu).

2.2.2 Konstruktory a destruktory

Mezi funkcemi deklarovanými v naší třídě String jsou i tři funkce, které vzbuzují pozornost tím, že mají stejné jméno, stejné navíc i se jménem třídy.

Možnost stejného jména u několika funkcí jsem již zmiňoval, ale proč mají tyto funkce stejné jméno jako celá třída a proč není uveden typ výsledku?

Tyto funkce jsou takzvané konstruktory třídy. Jsou to funkce, které se vyvolají při vytváření nového objektu dané třídy aby ho patřičným způsobem inicializovaly. Součástí konstruktoru může být alokování paměti, naplnění datových složek, testy zadaných inicializačních hodnot nebo cokoli jiného, co je pro danou třídu potřeba.

Pro každou třídu není nutné definovat konstruktor. Pokud ho programátor nedefinuje, vytvoří ho automaticky překladač. Nebo, jako v našem příkladu, je možné mít konstruktorů hned několik.

My máme konstruktory tři. První je prázdný konstruktor. Nemá žádné parametry a neumožňuje tedy inicializaci datových složek podle zadání. Jeho definice by mohla vypadat takto:

```
String::String(void)          // Prázdný konstruktor
{     delka=0;                // Prázdný řetěz
      retez=0;
}
```

Právě takový konstruktor by vytvořil i překladač sám, pokud bychom my žádný nedefinovali.

Druhý konstruktor v příkladu má parametr a umožňuje inicializaci klasickým znakovým řetězcem jazyka C. Součástí tohoto konstruktoru musí být alokace potřebné paměti pro řetěz.

Třetí konstruktor také umožňuje inicializaci a to jiným objektem téže třídy. Takový konstruktor se nazývá kopírovací, protože vytváří kopii existujícího objektu.

V našem příkladu je ještě jedna funkce, která nemá uveden typ výsledné hodnoty. Nemá však jméno stejné jako třída, ale jen podobné – uvozené vlnovkou. Tato funkce se nazývá destruktorem třídy a slouží ke korektnímu zrušení objektu. I destruktor může překladač vytvořit automaticky, ale tuto možnost nelze využít v případě, že objekt má alokovanou nějakou paměť, kterou je třeba uvolnit (stejně tak automaticky generovaný implicitní konstruktor nebude žádnou paměť alokovat). Na rozdíl od konstruktoru, destruktor je ve třídě jen jeden.

Konstruktory ani destruktory obvykle není třeba explicitně volat, o jejich správné použití se postará překladač.

2.2.3 Předefinování funkcí

Zmínil jsem se, že od každé třídy lze odvozovat třídy další. Odvozená třída pak dědí všechny složky i metody třídy rodičovské. V některých případech, když je odvozená třída specializovanější než rodičovská, je třeba určité metody nahradit, ne přejímat. Pro tyto případy je určen mechanismus překrývání funkcí. Nadefinuji-li v odvozené třídě funkci se stejným jménem, typem i parametry, jako má některá z metod třídy rodičovské, překryji tuto metodu a objekty odvozené třídy budou volat vždy nově definovanou funkci. Pokud je to třeba, lze metodu rodičovské třídy volat s použitím jména rodičovské třídy jako kvalifikátoru.

2.2.4 Virtuální funkce

Prosté překrývání funkcí má však jedno úskalí. Pracuji-li s objekty pomocí ukazatelů, používá překladač pro určení volané funkce typ ukazatele. Pokud bych například od třídy String odvodil třídu XString, v ní překryl metodu copy() a deklaroval Sptr jako ukazatel na String a Xptr jako ukazatel na XString, může nastat následující situace: Zavolám-li Sptr->copy(), vyvolá se metoda copy() třídy String. Zavolám-li Xptr->copy(), vyvolá se metoda copy() třídy XString. Ale: Ukazatel Sptr může ukazovat nejen na objekty třídy String, ale i na objekty tříd z ní odvozených, tedy i na objekty třídy XString. Ale i v tom případě se pro SPtr->copy() zavolá metoda copy() třídy String. A to jsem přece nechtěl.

V tomto případě nastává situace, kterou nelze řešit v době překladu, protože překladač nemůže vědět, na jaký objekt bude ukazatel ve skutečnosti ukazovat. Určení musí proběhnout až při výpočtu. C++ má pro tento účel vyhrazeno klíčové slovo **virtual**. Potřebuji-li jeden ukazatel použít pro objekty různých tříd (samozřejmě propojených dědičností) a volat vždy metodu patřičné třídy, označím metodu při deklaraci rodičovské třídy jako **virtual**. Volaná metoda se pak bude určovat až při běhu programu.

2.2.5 Vícenásobná dědičnost

Při dědění v C++ není situace tak nudná jako v normálním životě. Nová třída může mít rodiče jednoho, dva, nebo i více. Dědí ode všech.

S vícenásobnou dědičností je svázána řada jednodušších i složitějších pravidel, která určují přístupnost jednotlivých složek rodičovských tříd ve třídě odvozené, pořadí vyvolávání konstruktorů a destruktorů a další věci. Je řešena i situace, kdy odvozená třída dědí ze dvou tříd, které mají stejného předka. Složky tohoto společného předka se pak v odvozené třídě objeví dvakrát, pokud nebyl při odvozování označen klíčovým slovem **virtual**.

2.2.6 Operátory

Již jsem se zmínil, že rozšiřování funkcí se může týkat i standardních operátorů jazyka C++. A to nejenom operátorů +, –, * a /, ale i např. ++, &=, [], * (dereference ukazatele) nebo () (volání funkce).

To je velice příjemná vlastnost, která pomáhá zlepšit přehlednost programu. Je možné si nadefinovat obvyklé operátory pro libovolnou třídu. Pomocí znaménka * pak můžeme násobit komplexní čísla nebo matice, pomocí znaménka += napojovat řetězce atd.

V našem příkladu mám použit operátor indexování pro získání n–tého znaku z řetězu. Definice této funkce může vypadat například takto:

```
int operator[](const unsigned n)    // N–tý znak z řetězu
{
    return n<delka ? retez[n] : EOF;
}
```

2.2.7 Proudy

Zcela novým prvkem v jazyku C++ jsou proudy. Slouží pro obecný vstup a výstup. Data je možno posílat do proudu operátorem " a vyjímat operátorem „. Standardním zařízením jazyka C (stdin, stdout,...) zde odpovídají proudy cin, cout, cerr a clog. clog má podobnou funkci jako cerr, ale používá vyrovnávací paměť, takže odezva na výstupu nemusí být okamžitá.

Operace vstupu a výstupu s proudy jsou velice jednoduché, stačí napsat

```
int promenna;
cin >> promenna;
cout << „Načtená hodnota je: " << setw(6) << promenna << endl;
```

Na první pohled zde chybí formátovací řetězec, na jaký jsme zvyklí z funkcí scanf a printf. Operátory " a „ určují formátování podle typu vkládané proměnné. Operátorové funkce jsou rozšířené a pro každý typ proměnné se volá patřičná varianta.

Je fakt, že i pro jeden typ proměnné bývá třeba nastavovat formátovací možnosti. K tomu v C++ slouží tzv. manipulátory; v našem případě jsem použil setw(6), který nastaví šířku výstupního pole na šest znaků a endl, který zapíše konec řádku. Celý výstupní příkaz tedy odpovídá volání funkce printf(„Načtená hodnota je: %6d\n",promenna).

Výstupní řádek s operátory " je sice delší než s voláním funkce printf, ale jednak je přehlednější a potom má velikou výhodu v tom, že operátory " (a samozřejmě i „) může

sám uživatel libovolně rozšiřovat pro všechny svoje třídy. Funce printf a scanf byly naproti tomu omezeny pouze na standardně definované typy.

Na první pohled upoutá i zřetězení operátorů v příkazu. To je umožněno deklarací

```
istream &operator >> (istream &s, typ t);
ostream &operator << (ostream &s, typ t);
```

První operátor v řetězu přijímá jako parametr vstupní (výstupní) proud a po provedení operace tentýž proud vrací, takže se může stát vstupním parametrem pro operátor následující. Uvedený příklad odpovídá volání

```
operator<<(operator<<(operator<<(operator<<(cout, „Načtená hodnota je: "),
        setw(6)), promenna), endl);
```

2.2.8 Šablony

Pod slovem šablony (templates) se skrývá další významné rozšíření, které se v C++ objevilo až se standardem cfront 3.0. Jedná se o takzvané parametrizované typy.

Pro srozumitelnější vyjádření: pokud je třeba mít stejnou funkci pro operandy různých typů, je třeba ji definovat tolikrát, kolik různých typů může mít. V C navíc každá taková funkce musí mít jiné jméno. Podobnou situaci lze v jazyku C vyřešit makrem – např. makra min a max. Toto řešení je však použitelné pouze pro velice jednoduché funkce a navíc se zbavujeme možnosti typové kontroly parametrů. V C++ jsou proto definovány šablony. V programu napíšeme definici funkce bez uvedení typuparam etrů. Překladač potom sleduje všechna volání této funkce a vygeneruje kód funkce zvlášť pro každý použitý typ parametru. Pro zmiňovanou funkci max by vypadala šablona takto:

```
template <class t>
T max(T x, T y)
{ return (x > y) ? x : y ; }
```

Písmeno T zde zastupuje typ skutečných parametrů. Překladač vygeneruje tolik exemplářů funkce max, kolik je třeba. Podmínkou je, aby při volání byly oba parametry stejného typu a aby pro tento typ byl definován operátor >. Operátor > lze samozřejmě definovat pro libovolnou uživatelskou třídu libovolným způsobem – porovnání řetězců lexikograficky nebo podle délky, porovnání matic podle řádu nebo podle determinantu,...

Šablony lze v C++ použít nejen pro definici funkcí, ale i pro definici celých tříd, jejichž jedna nebo více složek je pak parametrického typu. Tato vlastnost přináší vynikající možnosti např. při definici skladových tříd. Je možno nadefinovat třídu implementující zásobník, frontu nebo množinu, aniž bychom předem věděli, jakého typu budou objekty ve třídě ukládané.

2.2.9 Výjimky

Chtěl bych se ještě zmínit o výjimkách. Ty sice dosud nebyly v C++ standardizovány (ani v ANSI výboru ani v AT&T cfront), ale s jejich uvedením se počítá.

Zpracování výjimek má přinést lepší možnosti ošetření chybových situací. Často se stává, že nějaká funkce může skončit se dvěma nebo více různými výsledky a nebo chybou. Možnost chyby přináší komplikace při vyhodnocování výsledku funkce (jeden příkaz if navíc, nelze použít přímo ve výrazu) a někdy i při volbě typu výsledku funkce (všecny funkce pracující se znaky jsou typu int, aby mohly rozlišit EOF od normálních znaků). Zdá se, že je vhodné najít způsob, jak v případě chyby opustit funkci jinou cestou než při normálním ukončení. Tomuto by měla napomoci nová klíčová slova **try, catch** and **throw**. Definice operátoru [] z naší třídy řetězců by pak mohla vypadat např. takto:

```
char operator[](const unsigned n)        // Funkce můžo vracet char
{
      if n> delka throw;                 // Ukončení při chybě
      return retez[n];                   // Normální ukončení
}
```

a její použití:

```
char c;
try c=str1[6];                           // Pokus o volání
      cout << „Šestý znak je " << c;     // Normální pokračování
catch cout << „Řetěz má méně než šest znaků";        // Chyba
```

3 Borland C++

3.1 Rozšíření jazyka

Překladače pro C++ od firmy Borland obsahují všechny rysy jazyka obsažené v cfront 3.0 a navíc některá rozšíření. Tato rozšíření jsou motivována z velké části faktem, že Borland C++ je určeno pro práci v operačním systému DOS na počítačích s procesory Intel 80x86. Proto je standardní knihovna rozšířena o celou řadu funkcí zprostředkujících služby DOSu a proto jsou zavedena některá nová klíčová slova. Nebudu zde uvádět rozšíření knihoven, ale zaměřím se na vlastní jazyk.

Vzhledem k segmentové organizaci paměti u procesorů Intel jsou v Borland C++ zavedeny paměťové modely, tak jak tomu bylo již u Turbo C. Použití toho kterého modelu pak určuje, jaká bude implicitní délka ukazatelů do kódu a do dat. Paměťových modelů je šest, **TINY, SMALL, MEDIUM, COMPACT, LARGE** a **HUGE**, ukazatele mohou být šestnáctibitové nebo dvaatřicetibitové. Modely **TINY, SMALL,** a **MEDIUM** mají

šestnáctibitové ukazatele do dat, modely TINY, SMALL a COMPACT šestnáctibitové ukazatele do programu. Model TINY je určen pro tvorbu souborů typu .COM a má program, data i zásobník v jediném segmentu, model HUGE naproti tomu umožňuje mít více než jeden segment statických dat. Implicitní velikost ukazatelů lze změnit při deklaraci s klíčovým slovem **near** nebo **far**.

Dalším přidaným klíčovým slovem je **pascal**, určené pro zavedení konvencí Turbo Pascalu. Identifikátory deklarované s tímto modifikátorem nerozlišují velká a malá písmena a funkce používají Pascalskou volací konvenci. Protějškem tohoto klíčového slova je **cdecl**, které zdůrazňuje, že na tento identifikátor se mají použít konvence jazyka C, resp. C++. Modifikátor **fastcall** je určen pro funkce, jimž se pro zvýšení rychlosti předávají parametry v registrech místo na zásobníku a modifikátor **interrupt** pro funkce určené k obsluze přerušení.

Ve vazbě na procesory Intel jsou v Borland C++ definovány pseudoproměnné _AL, _AH, _BL, _BH, _CL, _CH, _DL, _DH, _AX, _BX, _CX, _DX, _SI, _DI, _SP, _BP, _CS, _DS, _ES, _SS a _FLAGS, které přímo adresují jednotlivé registry procesoru.

Borland C++ dovoluje do programu vkládat jednotlivé příkazy nebo celé úseky v assembleru (uvozené klíčovým slovem **asm**), přičemž od verze 3.0 nepoužívá externí překladač assembleru.

Nejpodstatnější rozšíření jazyka, které už přesahuje hranice přizpůsobení DOSu jsou takzvané dynamicky rozlišované metody (dynamically dispatchable member functions), které se používají v knihovně Object windows. Toto rozšíření umožňuje přiřadit virtuálním metodám takzvaný přidělovací index, podle kterého mohou být volány. V Object Windows se toto využívá při volání metod jako odpovědi na zprávy Windows. Z důvodů kompatibility je však lepší omezit se na používání tohoto rysu pouze uvnitř ObjectWindows.

3.2 Integrované vývojové prostředí Borland C++

Integrované vývojové prostředí je od samého počátku silnou stránkou produktů firmy Borland. Ani v nových verzích nezůstává své pověsti nic dlužno.

S překladači Borland C++ se dodávají integrovaná prostředí dvě, jedno znakové, pro DOS i Windows, druhé grafické pro Windows. Základem obou prostředí je výkonný vícesouborový editor s mnoha okny. Okna se mohou navzájem překrývat a lze je přesouvat a měnit velikost myší. Editor používá konvence příkazů WordStar ke kterým časem přibývají rozšíření – použití nástěnky (clipboardu), označování bloků kombinací Shift–šipka. Chování editoru lze do značné míry modifikovat dodávaným překladačem maker. Novinkou je tzv. color syntax highlighting, které na obrazovce rozlišuje různými barvami klíčová slova, operátory, identifikátory, konstanty, komentáře a další prvky jazyka.

Jednotlivé funkce se ovládají klasicky přes menu nebo horké klávesy, v prostředí pro Windows také pomocí tlačítkového menu (speedbar). Vynikajícím prvkem zavedeným již u prvního Turbo C++ jsou tzv. Transfery. Ty dovolují z integrovaného prostředí spouštět externí programy a jejich výstup zobrazovat v dalším okně. Při instalaci se takto nastaví Turbo Debugger, Assembler a Profiler a vyhledávací program Grep.

Pro zpracování programů sestávajících z více zdrojových souborů je zde správce projektů. Ten sleduje závislosti jednotlivých souborů a při každém překladu zpracuje jen ty, které se od posledního překladu měnily. Je to integrovaná obdoba programu Make. Do projektu je rovněž možno zahrnout zdrojové soubory psané v jiném jazyce než v C++ a zpracovávat je překladačem zavedeným do některého z transferů. Správce souborů také udržuje informace o použitém paměťovém modelu, o zařazení informace pro ladění a o všech dalších nastaveních volitelných v menu Options.

Od verze 3.0 má překladač Borland C++ také výkonný optimalizátor, který zvyšuje využití registrů, vylučuje zbytečné proměnné, přesouvá invariantní kód mimo cykly, používá dvaatřicetibitové instrukce i registry atd.

3.3 Knihovny standardních objektů v Borland C++

3.3.1 Knihovna datových proudů

Knihovna datových proudů dodávaná k Borland C++ obsahuje dvacet tříd pro datové proudy. Protože některé z nich jsou pouze pomocné a jsou mezi nimi poměrně složité vztahy, nebudu popisovat detailně celou hierarchii, ale popíšu jen stručně ty nejdůležitější třídy.

Základem hierarchie je třída **ios**. Definuje základní složky děděné jejími následníky. Obsahuje stavové informace, formátovací prostředky, ukazatele na případné vyrovnávací paměti a další všeobecné složky.

Od třídy **ios** jsou mimo jiné odvozeny třídy **istream, ostream, fstream** a **strstream. istream** soustřeďuje vlastnosti potřebné pro vstup dat, **ostream** pro výstup dat, **fstream** obsahuje prostředky pro práci se soubory a **strstream** pro práci s řetězci. Od těchto tříd se pak dále odvozují **ifstream, iostream** a další kombinace. Na konci odvozování jsou třídy **fstream** a **strstream**, které shrnují všechny potřebné vlastnosti a funkce pro vstupní a výstupní operace se soubory resp. s řetězci. Jako specializovaná třída pro výstup na obrazovku je od **ostream** odvozen **constream**.

Každá třída pro datový proud obsahuje možnosti jako určitá skupina funkcí v C. Ve třídě **fstream** nalezneme metody open, close, read, write, get, put atd. Funkce printf a scanf jsou nahrazeny operátory " a „ a sadou manipulátorů. Operátory ve třídě **strstream** nahrazují funkce sprintf a sscanf. Ve třídě **constream** nalezneme metody clrscr a window

a manipulátory, jejichž jména nám také jsou známa: clreol, insline, lowvideo, setattr, setclr, setxy a další.

Knihovna datových proudů jako taková nepřináší téměř žádné přímé rozšíření oproti standardní knihovně vstupních a výstupních funkcí. Přináší však sjednocení a zapouzdření velkého počtu funkcí do několika málo tříd a především možnost rozšířit funkci těchto tříd na jakýkoli uživatelský typ nebo třídu.

3.3.2 Skladové třídy

Již od verze 2.0 je součástí Borland C++ rozsáhlá knihovna skladových tříd. Skladovými třídami se rozumí třídy, které implementují některé abstraktní datové typy, jako je např. fronta, zásobník, seznam a další. Knihovna skladových tříd tak nabízí implementaci těchto datových typů, přičemž konkrétní ukládaná data určí až uživatel. Ve verzi 3.0 a 3.1 jsou knihoven dvě kompletní sady, které obě poskytují stejné datové typy, ale liší se implementací. První sada knihoven je víceméně převzata z ve rze 2.0, druhá je implementována nově pomocí šablon.

U knihovny, která má implementovat abstraktní datový typ, přičemž není známo, jaká data (jakého typu) se do něj budou ukládat, je základním problémem právě specifikace ukládaných dat. Je to jako když stavíme pyramidu, ale chybí nám spodní vrstva. Vlastnosti nadstavby známe, ale nevíme na čem bude stát, jaká data se mají ve skladu ukládat. Z hlediska vyhrazení prostoru by se tento problém dal řešit poměrně snadno tak, že by se neukládala samotná data, ale ukazatel na ně. Bohužel tento přístup neřeší typovou kontrolu. Takový ukazatel by musel být typu void a muselo by používat přetypování. Jenže by se muselo používat i uvnitř funkcí knihovny, kterou píšeme a při překladu typ dat není znám. Takže se dostáváme zpět do výchozího bodu.

Dvě knihovny skladových tříd dodávané s Borland C++ se liší právě řešením tohoto problému. Principiálně jednodušší řešení používá šablon k definici abstraktního datového typu, přičemž ukládaný typ je právě parametrem šablony. Pro ty, kteří mají překladač s implementací šablon (BC++ 3.0 a vyšší) a kteří chtějí s použitím skladových tříd začít, je toto řešení jednoznačně vhodnější.

Druhé řešení, převzaté z verze 2.0, šablon využívat nemůže, protože ty v této verzi dosud implementovány nebyly a je poskytováno pro uživatele, kteří začali tuto starší knihovnu využívat. Jeho princip spočívá v tom, že všechny skladové třídy ukládají objekty třídy Object – nebo třídy od Object odvozené. Proto musí být typ jakéhokoli ukládaného objektu odvozen od Object. V praxi to není díky vícenásobné dědičnosti příliš náročný požadavek. Přesto tento přístup přináší obtíže při přetypování a oslabuje typovou kontrolu. Odvozování všech tříd od společného předka by také mohlo způsobit obtíže při použití další knihovny s podobnou organizací.

Od základní třídy Object jsou odvozeny i samotné skladové třídy a tak je možno do sebe vzájemně ukládat – vytvořit frontu množin atd...

3.4. Application Frameworks

Application Frameworks je společný název pro dvě objektové knihovny určené pro tvorbu uživatelského rozhraní. Pro textové aplikace pro DOS je to **Turbo Vision** a pro aplikace pro Windows je to **ObjectWindows**. Součástí dodávky Application Frameworks jsou i zdrojové programy obou těchto knihoven a také normální knihovny C++.

3.4.1 Turbo Vision

Turbo Vision je mohutná knihovna objektů pro vytváření interaktivního uživatelského rozhraní v textovém prostředí DOSu. Dává programátorovi k dispozici prostředky pro vytvoření obdobného uživatelského prostředí jako mají produkty firmy Borland včetně pohyblivých a překryvných oken s proměnlivou velikostí, dialogových rámečků, menu, stavového řádku i dialogových tlačítek. To vše samozřejmě s plnou podporou myši. Není již třeba psát program pro zobrazování oken a menu na obrazovce, stačí se zaměřit na to co se má v oknech objevit a jaké akce má dané menu spouštět.

TurboVision (TV) zavádí některé prvky programování řízeného událostmi. To znamená, že systémová část TV sama sleduje vstup z klávesnice, pohyby myši a stav jejích tlačítek a v případě významné události (řídicí kombinace, stisk tlačítka myši, „uchopení" rámečku okna ...) vyvolá funkci určenou k obsluze této události. Činnost funkce závisí už na programátorovi.

Všechny třídy TV plně podporují vstup a výstup pomocí datových proudů a sdružování objektů pomocí skladových tříd definovaných v Borland C++. Ukládání do proudů je nesmírně užitečná záležitost, v kterémkoli okamžiku můžete uložit stav aplikace (nebo její části – libovolného objektu) do souboru na disk jediným operátorem " a kdykoli později z disku opět obnovit.

3.4.1.1 Přehled základních tříd TVision

TApplication – Základní třída TV, sdružuje objekty tříd TDesktop (obsluha všeho na ploše obrazovky), TMenuBar (obsluha řádkového menu) a TStatusLine (obsluha stavového řádku). Metoda TApplication::Run() obsahuje vlastní tělo programu.

TDesktop – Třída pro vytvoření a obsluhu všech objektů v ploše obrazovky. Zajišťuje vytvoření, zobrazení a veškeré manipulace s okny a dalšími objekty na základě podnětů od uživatele (klávesnice, myš).

TWindow – Třída pro obsluhu klasických oken známých z produktů firmy Borland. Okno má hranici, záhlaví, lze je posouvat, zvětšovat, zmenšovat, otevírat a zavírat. Mezi

objekty uvnitř okna (tlačítka, dotazová pole) je možno přepínat klávesou Tab, Shift–Tab. Mezi jednotlivými okny se přepíná klávesami Alt–číslo_okna.

TMenuBar, TMenuBox – Třídy pro implementaci řádkových a roletových menu, vnořených do libovolné hloubky. Volbám v menu odpovídají patřičné funkce.

TStatusLine – Třída pro implementaci stavového a nápovědného řádku s podporou pro změny obsahu řádku v reakci na určené události.

TDialog – Třída pro obsluhu dialogových rámečků pro zadávání informací od uživatele. Může obsahovat řadu přepínačů, voleb a tlačítek.

TInputLine – Třída pro obsluhu dotazového řádku pro vkládání textové informace.

THistory – Třída pro obsluhu rámečku nabízejícího několik předcházejících voleb odpovídajícího dotazového řádku.

TListViewer – Třída pro zobrazování seznamů položek a výběr z nich.

TScroller, TScrollBar – Třídy pro podporu posouvání obsahu okna vůči jeho hranicím.

TButton – Třída pro obsluhu „tlačítka" – pole s textem, které uživatel může „stisknout" pomocí myši nebo klávesy Enter. TButton jako reakci na stisknutí vyvolá patřičnou funkci.

Tříd je samozřejmě mnohem více, uvedl jsem zde jen ty nejdůležitější.

3.4.2 ObjectWindows

Představují obdobu Turbo Vision pro prostředí Windows, velká část hlavních tříd má stejnou funkci a stejně se jmenuje. Z odlišnosti obou prostředí však nutně vyplývají i odlišnosti knihoven. Odlišnost ve vnějším provedení (text x grafika) je sice patrná na první pohled, z hlediska vnitřní struktury však není tak podstatná. Důležitější rozdíl je ten, že u Turbo Vision je celé řízení událostmi plně v režii programu samotného, ObjectWindows musí čekat na zprávy o událostech zpracovaných ve Windows. Odpadá tedy zpracování událostí jako takových, je však třeba zpracovávat mnohem více zpráv o událostech.

V této úloze se uplatňuje rozšíření syntaxe jazyka C++, které firma Borland zavedla. Virtuálním metodám ve třídách ObjectWindows je možno přiřadit číselný index a ObjectWindows pak zajistí, že pokud objekt přijme zprávu Windows s tímto indexem, vyvolá se automaticky takto určená virtuální metoda. Tak je možné definovat odezvy na zprávy Windows poměrně snadno a vyhnout se složitým strukturám přepínačů, které jsou k tomuto účelu potřeba při programování v C.

Literatura

[1] C++ Report ročník 1993, čísla 1 až 3, AT&T Bell labs, 1993

[2] Borland C++ 3.1, příručky, Borland International 1992

Autor: Ing. Filip Brázdil
AproSoft a.s.
Škroupovo nám. 9
130 00 Praha 3
tel.: (02) 627 21 35