1. Programování pro DOS a některé zastaralé součásti C++

Tento text obsahuje výklad a příklady, které jsem vyřadil ze druhého vydání skripta Programování v C++. Jde především o programování pro operační systém DOS.

1.1. Příkazy (kapitola 6)

Příkaz ASM

Tento příkaz se v době programování pro DOS používal daleko více než dnes. Proto i příklad, který byl uveden v prvním vydání skripta, obsahoval konstrukci typickou pro programování pro DOS.

Příklad
Napíšeme funkci, která zjistí, zda je k počítači připojena myš, a pokud ano, kolik má tlačítek. Využijeme překladač Borland C++ 3.1.
K tomu potřebujeme vědět, že ovládání myši je na PC připojeno obvykle na přerušení 51. Chceme-li je použít, musíme do registru AX uložit hodnotu 0 — tím zadáváme, že chceme použít funkci č. 0 tohoto přerušení pro práci s myší. Vynulujeme také registr BX a pak vyvoláme přerušení 51, tj. v šestnáctkové soustavě `33h` (to je zápis hexadecimální konstanty v Turbo Asembleru, *nikoli* v Borland C++).
Po návratu z tohoto přerušení najdeme v registru BX počet tlačítek myši (nebo 0, není-li k počítači myš připojena). Protože naše funkce bude typu `int` a takové funkce vracejí hodnotu v registru AX, přesuneme obsah BX do AX.

```
int IndikaceMysi()
{
  asm {
    xor ax,ax;        // nulování registrů AX a BX
    mov bx,0;         // dvěma různými způsoby
    int 33h;          // volání rutiny pro obsluhu přerušení
    mov ax,bx         // přesun výsledku do registru AX
  }
}
```

Poznamenejme, že překladač se bude zlobit, že tato funkce nevrací hodnotu, i když je typu `int` — neobsahuje totiž příkaz `return`. Jde však pouze o varování, nikoli o chybu, a funkce ve skutečnosti *vrací* výsledek — ovšem my jsme to naprogramovali v asembleru, a proto to překladač neodhalí.

1.2. Ukazatele na PC

Pro ukazatele na PC platí skoro vše, co jsme si řekli v kapitole 8 skriprta. Bohužel, něco je občas trochu jinak.

Problém spočívá v poněkud netradičním zacházení s pamětí a s adresami, se kterým se setkáváme u procesorů firmy Intel. V důsledku toho se na PC obvykle rozlišují tři druhy ukazatelů: blízké (*near*), vzdálené (*far*) a normalizované (*huge*). Pro první dva druhy lze používat adresovou aritmetiku pouze s jistou dávkou opatrnosti, pro třetí (normalizované ukazatele) je sice adresová aritmetika plně k disposici, ale práce s nimi je pro změnu pomalejší.

Trocha historie

Abychom pochopili, o co vlastně jde, musíme se nejprve vrátit trochu do historie. Předchůdcem mikroprocesoru Intel 8086, který se stal základem prvních osobních počítačů, byl čip Intel 8080. Tento procesor byl osmibitový. Uměl ovšem zkombinovat dva bajty a vytvořit

z nich 16bitovou adresu. Adresový prostor počítačů založených na těchto procesorech byl proto maximálně 65 536 bajtů (64 KB).

Jejich následník, procesor Intel 8086, byl již šestnáctibitový (do jednoho registru se vešly 2 B). Mohlo by se tedy zdát logické, že bude umět zkombinovat 4 bajty a tak vytvořit 32bitovou adresu; výsledkem by byl adresový prostor 4 GB, tedy něco přes 4 miliardy bajtů. To se ovšem v oné době zdálo nesmyslně mnoho, a tak se u firmy Intel rozhodli pro jinou možnost, která údajně vedla k jednodušší architektuře procesoru (a také snad k částečné zpětné kompatibilitě s procesorem 8080).

Adresování na 8086 a v reálném režimu pozdějších procesorů

Adresa na těchto procesorech je tvořena nikoli 32, ale pouhými 20 bity; to znamená, že adresový prostor má velikost 1 048 576 B (1 MB). Operační paměť je rozdělena na úseky o délce 16 B, které se nazývají odstavce (*paragrafy*). Souvislá část paměti, která začíná na hranici paragrafu a jejíž délka nepřesahuje 64 KB, se nazývá *segment*.

Adresu určitého bajtu na PC s tímto procesorem vyjádříme tak, že udáme nejprve pořadové číslo odstavce, na jehož hranici začíná segment, ve kterém tento bajt leží, a pak relativní adresu bajtu v daném segmentu (tzv. *ofset*). Pro adresu se používá zápis

segement : ofset

Obyčejně hovoříme o *segmentové* a *ofsetové*[1] části adresy.

Procesor si z této dvojice vypočte „skutečnou“ adresu (tedy pořadové číslo bajtu v paměti) podle vztahu

*segment*16 + ofset*

Je jasné, že jeden určitý bajt lze popsat mnoha různými adresami. Např. adresu čtyřicátého bajtu v paměti můžeme napsat 2:8 nebo 1:24 nebo 0:40. Někdy ale potřebujeme, aby byl zápis jednoznačný; proto se zavádí tzv. *normalizovaná adresa*, ve které je vždy

```
0 <= ofset <= 15.
```

Normalizovaná adresa 40. bajtu v paměti je 2:8.

Stejným způsobem zachází s pamětí i procesor 80186 (vždy) a procesory 80286 a pozdější, pokud běží v tzv. reálném režimu (např. pod operačním systémem DOS).

Adresy a registry

Každý program se skládá z několika oblastí; v jedné je kód (programové instrukce), v další data, v další zásobník, ve kterém se ukládají např. lokální automatické proměnné (proměnné, které deklarujeme v tělech funkcí a které existují, jenom pokud se provádí tělo funkce, a pak zase zaniknou). Tyto oblasti jsou obvykle tvořeny samostatnými segmenty a označují se jako kódový segment, datový segment a zásobníkový segment.

Procesor si za běhu programu uchovává segmentové části adres v různých pomocných registrech. Např. segmentovou část adres z datového segmentu má v registru DS, segmentovou část adres z kódového segmentu má v registru CS, segmentovou část adres ze zásobníkového segmentu má v registru SS. Jestliže tedy potřebujeme něco udělat s proměnnou *A*, která leží v datovém segmentu, stačí, když zadáme ofsetovou část její adresy — segmentovou si procesor doplní sám (Zadáváme tedy relativní adresu vzhledem k počátku datového segmentu.). To je docela efektivní.

Na druhé straně v rozsáhlejších programech nemusíme vystačit s jedním segmentem pro kód, s jedním segmentem pro data a s jedním pro zásobník. Představte si, že náš program má několik datových segmentů a my potřebujeme proměnnou, která leží v jiném segmentu, než

[1] O*ffset* znamená anglicky *posunutí*.

který právě určuje registr DS. V takovém případě musíme uvést obě části adresy, segmentovou i ofsetovou. Práce s takovouto adresou ovšem procesoru potrvá déle.

S podobnými problémy se můžeme setkat i v případě ukazatelů na kód (na funkce). Pokud voláme funkci, která leží v aktuálním kódovém segmentu (v tom, jehož segmentovou adresu máme v registru CS), stačí použít blízký ukazatel, tj. stačí uvést pouze ofsetovou část adresy. (Tomu se říká „blízké volání funkce“.) Jestliže ale potřebujeme zavolat funkci, která je v jiném kódovém segmentu, než na který právě ukazuje registr CS, musíme použít vzdálenou adresu. (Tomu se říká „vzdálené volání“.)

Také při návratu z funkce se používají podle okolností blízké nebo vzdálené ukazatele.

Blízké, vzdálené a normalizované ukazatele

Každý programovací jazyk implementovaný na PC, který umožňuje pracovat s ukazateli, se musí nějak vyrovnat se segmentovou architekturou paměti (stále hovoříme o programování pro reálný režim, ve kterém se všechny procesory chovají jako Intel 8086). Poměrně jednoduché řešení zvolil Turbo Pascal: Všechny ukazatele na data jsou čtyřbajtové, obsahují tedy jak segmentovou tak i ofsetovou část, a programátor s tím nemůže nic dělat.

Překladače C++ (a také překladače jazyka C) nabízejí programátorům obvykle možnost volit, jaké ukazatele chce používat. Programátor může jednak určit tzv. paměťový model, a tím předepsat implicitní tvar ukazatelů, a za druhé může v deklaraci explicitně určit druh jednotlivých ukazatelů.

Pokud píšeme malý program, ve kterém se všechna data vejdou do jednoho segmentu, stačí nám pracovat pouze s ofsetovou částí ukazatelů a nechat počítač, aby si automaticky doplnil segmentovou část z registru DS (resp. SS u lokálních automatických proměnných). Použijeme tedy tzv. *blízké ukazatele* (anglicky se označují jako *near pointers*). Deklarujeme je pomocí nestandardního modifikátoru[2] `__near`, který zapisujeme za označení typu a před hvězdičku:

```
int __near *uni;
```

Proměnná `uni` je blízký ukazatel na `int`. Má velikost 2 B a bude obsahovat pouze ofsetovou část adresy.

Jestliže ale píšeme rozsáhlejší program, ve kterém se všechny proměnné nevejdou do jednoho datového segmentu, budeme potřebovat celou adresu (jak segment tak i ofset). použijeme tedy tzv. vzdálené ukazatele. Ty se deklarují pomocí klíčového slova `__far`, např.

```
int __far *ufi;
```

Proměnná `ufi` je vzdálený ukazatel na `int`. Má velikost 4 B a bude obsahovat jak segmentovou tak i ofsetovou část adresy.

Pokud potřebujeme vzdálený ukazatel s normalizovanou adresou, použijeme tzv. normalizovaný ukazatel, deklarovaný pomocí klíčového slova `__huge`, např.

```
int __huge *uhi;
```

Normalizované ukazatele mají pochopitelně také velikost 4 B.

Pro blízké ukazatele lze používat adresovou aritmetiku; musíme si ale dát pozor, abychom nepřekročili hranici segmentu. Jestliže např. ukazatel `uni` obsahuje ofset 65534, bude po provedení operace

```
uni++;
```

obsahovat 0 (`uni` je ukazatel na `int`, takže se přičte 2, a výsledek — 65536 — přesahuje rozsah šestnáctibitových celých čísel bez znaménka; vezme se tedy výsledek modulo 65536, což je 0).

[2] Ve starších implementacích se tato klíčová slova psala bez úvodních podtržítek, tj. `near`, `far` a `huge`.

Pro vzdálené ukazatele nemusí správně fungovat porovnání pomocí operátorů >, >= atd. (Funguje, pokud mají oba ukazatele stejnou segmentovou část.) Podobně nemusí fungovat odečítání ukazatelů. Přičítání čísel k ukazatelům funguje, pouze pokud nepřekročíme hranici segmentu — podobně jako u blízkých ukazatelů.

Pro normalizované ukazatele můžeme adresovou aritmetiku používat bez obav. Už jsme si ale řekli, že práce s nimi je pomalejší, neboť program je musí neustále udržovat v normalizovaném tvaru.

Kde vzít segment?

Používáme-li blízké ukazatele, doplňuje si překladač segmentovou část adresy sám z některého ze segmentových registrů. Přitom platí:

- jde-li o adresu proměnné uložené v datovém segmentu, použije obsah registru DS;
- jde-li o adresu lokální automatické proměnné, použije obsah registru SS;
- jde-li o adresu programového kódu (např. funkce), použije obsah registru CS.

Některé implementace jazyka C++ umožňují přímo v deklaraci blízkého ukazatele předepsat, se kterým segmentovým registrem má být sdružen. Např. v Borland C++ lze použít modifikátory `__ss`, `__cs`, `__ds` a `__es`, v Microsoft C++ klíčové slovo `__based`.

Vedle toho můžeme deklarovat ukazatele, které budou obsahovat pouze segmentovou část adresy. K tomu nám v Borland C++ i v Microsoft C++ poslouží modifikátor __seg. Pro tyto „segmentové“ ukazatele nelze používat obvyklou adresovou aritmetiku, můžeme je ale sčítat s blízkými ukazateli (a vzniknou vzdálené ukazatele).

MK_FP a jiná makra

V hlavičkovém souboru `dos.h` (Borland C++, Visual C++ 1.5) resp. `i86.h` (Watcom C++) najdeme makro `MK_FP(seg, ofs)`, které umožňuje sestavit vzdálený ukazatel ze dvou čísel typu `unsigned`. Chceme-li naopak rozložit vzdálený ukazatel na segmentovou a ofsetovou část, použijeme makra `FP_SEG(uk)`, resp. `FP_OFF(uk)`.

Příklad

Pracuje-li grafický adaptér v textovém režimu, ukládají se data zobrazovaná na monitoru v paměti počínaje adresou `0xB800:0x0`, a to tak, že v sudých bajtech jsou vždy zobrazované znaky a v lichých „atributy“ — tj. příznaky vyjadřující barvu textu a pozadí. Pokud bychom chtěli pracovat s obrazovkovou pamětí přímo, nikoli prostřednictvím knihovních funkcí, mohli bychom napsat např. následující úsek programu:

```
#include <dos.h>
typedef unsigned radek[80];
radek (__far * Obr) = (radek __far*)MK_FP(0xB800, 0);
for(int i = 0; i < 10; i++) Obr[0][i] = 'a'+i+256*7;
```

Zde nejprve deklarujeme `radek` jako jméno pro typ „pole 80 čísel typu `unsigned`“. (Číslo typu `unsigned` zabírá v 16bitovém prostředí 2 B, takže stačí uložit do nižšího bajtu znak a do vyššího jeho atributy. Atribut 7 odpovídá šedé barvě.)

Dále definujeme vzdálený ukazatel `Obr` na typ `radek` a přiřadíme mu adresu začátku obrazové paměti. Makro `MK_FP` vrací ukazatel typu `void *`, proto ho musíme přetypovat.

Paměťové modely v reálném režimu

V předchozím oddílu jsme si řekli, že na PC máme k disposici 3 druhy ukazatelů. Programátor se o ně ovšem obvykle nemusí starat: pokud nepředepíše pomocí modifikátorů `__near`, `__far` resp. `__huge` něco jiného, budou všechny ukazatele stejného druhu, který překladač určí podle paměťového modelu použitého při překladu.

Překladače C/C++ pro reálný režim obvykle rozlišují 6 paměťových modelů. Jednotlivé modely se liší nejen tím, který druh ukazatelů je implicitní, ale i celkovým uspořádáním paměti a maximální možnou velikostí programu.

Ukážeme si obvyklý význam těchto modelů (detaily se mohou v různých implementacích lišit). V závorkách uvádíme anglické názvy.

Paměťový model pro překládaný program určujeme buď nastavením některého z přepínačů prostředí nebo některou z voleb v příkazové řádce překladače — podrobnější informace je třeba hledat v dokumentaci.

Drobný model (tiny)

V tomto modelu jsou všechny ukazatele — na data i na kód (funkce) — implicitně blízké. Data i kód jsou ve stejném segmentu, takže celý program nesmí být větší než 64 KB. Program nemá vlastní zásobník. Tento model se hodí pro velmi malé programy.

Malý model (small)

Také v tomto modelu jsou všechny ukazatele implicitně blízké, data a kód jsou však v různých segmentech. To znamená, že přeložený kód smí zabírat maximálně 64 KB a stejně velká mohou být i globální data (celý program tedy může zabírat až 128 KB). Malý model se hodí pro středně velké programy.

Střední model (medium)

V tomto modelu se implicitně používají vzdálené ukazatele na kód a blízké ukazatele na data. Program může mít i více kódových segmentů, ale jen jeden datový segment. Jinak řečeno: kód může zabírat až 1 MB, zatímco data jsou omezena na 64 KB. Střední model se hodí pro velké programy s malým objemem dat.

Kompaktní model (compact)

Kompaktní model je tak trochu opakem středního modelu: pro data se implicitně používají vzdálené ukazatele, pro kód blízké. To znamená, že kód může zabírat maximálně 64 KB, zatímco data až 1 MB. Kompaktní model se používá pro nepříliš velké programy s rozsáhlými daty.

Velký model (large)

V tomto modelu se používají vzdálené ukazatele jak pro kód tak pro data. Program může mít více kódových i datových segmentů; velikost kódu i dat je omezena hodnotou 1 MB. Používá se pro velké programy s velkým rozsahem dat.

Rozsáhlý model (huge)

Také v tomto modelu se používají implicitně vzdálené ukazatele jak pro kód tak pro data. Od velkého modelu se liší mj. implicitním způsobem překladu funkcí — ale o tom budeme hovořit později.

Poznámka

Všimněte si, že normalizované ukazatele se implicitně nepoužívají v žádném z paměťových modelů. Ty musíme vždy explicitně deklarovat.

Reference

V oddílu věnovaném referencím jsme si řekli, že to jsou vlastně ukazatele, které se automaticky dereferencují. To znamená, že i pro reference budeme občas potřebovat modifikátory `__near` a `__far` (`__huge` nebude mít nejspíš smysl, neboť pro reference nelze používat adresovou aritmetiku).

Funkce pro práci s pamětí na PC

Operátor `new` a funkce `malloc()` a `calloc()` vracejí v „malých“ modelech (drobný, malý a střední) blízké ukazatele a ve zbývajících třech vzdálené ukazatele. Připomeňme si, že funkce `malloc()`, `calloc()` a `realloc()` mají prototypy

```
void* malloc(size_t s);
void* calloc(size_t n, size_t s);
void* realloc(void* p, size_t s);
```

„Velikost“ vraceného ukazatele se bude řídit použitým paměťovým modelem. Navíc je typ `size_t` v „malých“ paměťových modelech definován jako `unsigned`, zatímco ve „velkých“ jako `unsigned long`. To znamená, že v „malých“ modelech můžeme pomocí těchto funkcí alokovat bloky paměti o velikosti nejvýše 64 KB, zatímco ve velkých modelech až do 4 MB (což v DOSu nemá význam).

Pokud chceme v malém nebo středním modelu alokovat větší úsek paměti, můžeme použít funkce `farmalloc()`, `farcalloc()` a `farrealoc()`, které mohou mít parametry typu `unsigned long`. Paměť alokovanou pomocí těchto funkcí uvolníme pomocí funkce `void farfree(void *)`.

Funkce

```
size_t coreleft();
unsigned long farcoreleft();
```

umožňují zjistit, kolik zbývá volné paměti (jsou k disposici jen v některých implementacích).

Jak je to v chráněném režimu

Dvaatřicetibitové procesory

80386 byl první dvaatřicetibitový procesor pro PC. Dnes se s ním už nesetkáme; převážná většina osobních počítačů je osazena procesory Intel Pentium a lepšími; z programátorského hlediska — přesněji z hlediska práce s pamětí — se však neliší. Pro tyto procesory je ale reálný režim v podstatě nepřirozený. Opravdu všech možností, které poskytují, lze totiž využít pouze v chráněném režimu.

Chráněný režim

Také v chráněném režimu se adresa skládá ze dvou částí; první z nich se nazývá selektor, druhá — stejně jako v reálném režimu — ofset.

Paměť je opět rozdělena na segmenty, které ovšem tentokrát mohou být velké až 4 GB (16 MB na 80286). Začátky všech segmentů — spolu s dalšími informacemi — má procesor uloženy v tzv. tabulce deskriptorů.

Tabulka deskriptorů je vlastně obyčejné pole, uložené někde v paměti, a selektor je index, který určuje jednotlivý prvek (deskriptor) v tomto poli.

Adresy se tedy v chráněném režimu zadávají nepřímo. Procesor dostane selektor a ofset; na základě selektoru si najde v tabulce deskriptorů adresu začátku segmentu, k té přičte ofset, a tak získá skutečnou adresu požadovaného bajtu. Protože je v chráněném režimu na procesorech 80386 a vyšších ofset dvaatřicetibitový, mohou být segmenty tak velké (obr. 8.4).

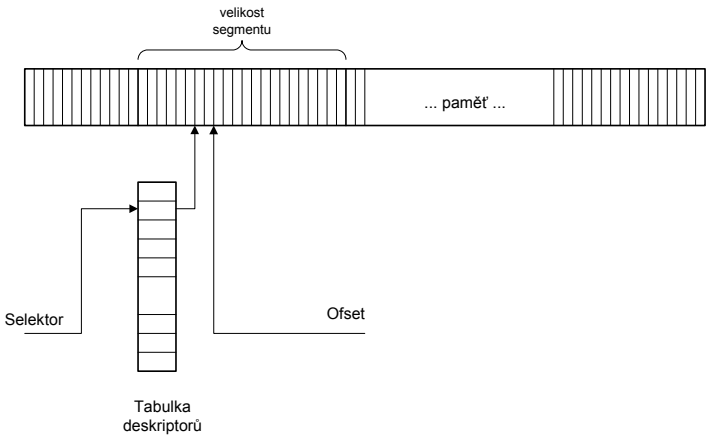
Deskriptor obsahuje kromě skutečné adresy začátku segmentu ještě další informace:

- délku segmentu,
- příznak určující, zda jde o kódový nebo datový segment,
- příznak stupně ochrany.

Procesor tak může kontrolovat, zda nepřekračujeme hranice segmentu. Kromě toho kontroluje, zda se nesnažíme zapisovat data do kódového segmentu nebo použít data v datovém segmentu jako instrukce a zda máme vůbec právo do tohoto segmentu přistupovat. Pokud

některý ze zákazů porušíme, způsobíme chybu, označovanou jako „obecné porušení ochrany“ (*general protection fault*) a náš program skončí.

Procesor tedy — alespoň v jisté míře — chrání data i kód před neoprávněnými zásahy. Od toho pochází také název „chráněný režim“ (*protected mode*). Poznamenejme, že samozřejmě existují cesty, jak tuto ochranu obejít a jak napsat např. program, který modifikuje sám sebe.



Obr. 1.1 Přístup do paměti v chráněném režimu procesorů 80286 a novějších

Ukazatele v chráněném režimu

Z toho, co jsme si o dosud řekli, plyne, že i v chráněném režimu se ukazatel skládá ze dvou částí, z nichž jednu by si mohl procesor doplnit automaticky. Měli bychom tedy mít opět možnost používat jak blízké, tak i vzdálené ukazatele.

Překladače však zpravidla vycházejí z představy, že segment o maximální možné velikosti 4 GB je dostatečný pro jakékoli použití, a proto nenabízejí jinou možnost, než používat blízké ukazatele. Patrně nejrozšířenější dvaatřicetibitové překladače, Borland C++ 4.0 a pozdější a Microsoft Visual C++ 2.0 a pozdější, nabízejí pouze tzv. plochý (flat) model paměti. Je to dvaatřicetibitová obdoba drobného modelu: celý program je v jednom segmentu.

Na druhé straně např. dvaatřicetibitový překladač Watcom C++ umožňuje rozdělit program do několika segmentů a používat i vzdálené ukazatele. Zmíněný překladač nabízí také několik paměťových modelů pro dvaatřicetibitové aplikace. Využívají se zejména při psaní aplikací pro dosovské extendery.

1.3. Modifikátor volatile

Tento modifikátor se často používá v nízkoúrovňových konstrukcích, které závisejí na platformě, pro niž je program určen. Zde si ukážeme použití ve funkci pro obsluhu přerušení v operačním systému DOS.

Následující příklad je třeba překládat některým ze starších překladačů, které měly DOS jako cílovou platformu — může to být např. Borland C++ 3.1 nebo starší, Microsoft Visual C++ 1.5 nebo starší nebo Watcom 10.5 nebo starší.

Poznamenejme, že v těchto překladačích měly i hlavičkové souboru jazyka C++ příponu `.h`, proto zde uvidíte mj. `<iostream.h>`, nikoli `<iostream>`. Pomocné funkce, jako `getvect()` a další nástroje pro programování pro DOS, najdeme v hlavičkovém souboru `<dos.h>`.

Příklad

Ukážeme si použití nestálé proměnné (`volatile`) na jednoduchém příkladu funkce, která způsobí, že program pro DOS bude čekat stanovenou dobu. Užijeme k tomu přerušení 08 (časovače), které je vyvoláváno cca osmnáctkrát za sekundu (po 55 ms).

O funkcích pro obsluhu přerušení (modifikátor `interrupt`) budeme mluvit v příštím oddílu, zatím stačí vědět, že DOS tuto funkci zavolá vždy, když dojde k odpovídajícímu přerušení. V následujícím programu je to funkce `Timer()`, kterou bude volat každých 55 ms a která bude měnit hodnotu proměnné `cas`.

```
// Přeloženo v Borland C++ 3.1
#include <dos.h>        // Starý překladač, proto je
#include <iostream.h>   // <iostream.h>, nikoli <iostream.h>

volatile long cas;             // Proměnná pro počítání času
void interrupt (far *Fun)(...); // Ukazatel na funkci pro
                                // obsluhu přerušení

void interrupt Timer(...)       // Obsluha přerušení 08
{
  Fun();                   // Nejprve zavoláme původní obsluhu
  cas++;
  outportb(32,32);      // Oživíme řadič přerušení
}

void Wait(long Sec)     // Čekání
{
  Sec *= 18;               // Přes kolik přerušení je třeba čekat
  cas = 0;
  while(cas < Sec);     // Test hodnoty proměnné cas
}

int main()
{
  // Zapamatujeme si původní funkci pro obsluhu přerušení
  Fun = getvect(8);
  setvect(8,Timer);     // Nastavíme novou
  cout << "Počkáme 2 vteřiny." << endl;
  Wait(2);                 // Čekáme
  cout << "Už jsme se dočkali.";
  setvect(8,Fun); // Obnovíme původní funkci pro obsluhu přerušení
  return 0;
}
```

Proměnná `cas` se mění asynchronně, a proto jsme ji deklarovali jako nestálou (`volatile`). Funkce `Wait()` nejprve proměnnou `cas` vynuluje a pak ji v cyklu porovnává se zadanou hodnotou.
V hlavním programu získáme pomocí funkce `getvect()` adresu systémové funkce pro obsluhu tohoto přerušení 08 a uložíme ji v globální proměnné `Fun` typu vzdálený ukazatel na funkci typu `void interrupt`. Pak pomocí funkce `setvect()` předepíšeme, že se k ošetření přerušení 08 má používat funkce `Timer()`. Před skončením programu vrátíme původní funkci pro obsluhu přerušení.
Ve funkci `Timer()` nejprve zavoláme původní funkci pro obsluhu přerušení 08 (to je „programátorská slušnost“, neboť tato funkce může mít na starosti řadu věcí, které by jinak nefungovaly). Pak změníme hodnotu proměnné `cas`.
Poznamenejme, že po každém hardwarovém přerušení je třeba „oživit“ řadič přerušení, aby byl schopen přijmout další signál přerušení. (Tím se vlastně sděluje, že přerušení bylo ošetřeno.) To uděláme tak, že na port 32 zapíšeme hodnotu 32. K tomu použijeme funkce `outportb()`. Všechny tyto funkce mají prototypy v hlavičkovém souboru `<dos.h>`.

1.4. Tvar deklarátoru

Nechť platí deklarace

`T` $\mathcal{D}$`;` (1)

kde `T` je jméno typu (např. `unsigned long`). [Uvádíme jen body, které jsou ve skriptu změněny.]

b) Je-li $\mathcal{D}$ deklarátor tvaru `*E`, bude `E` identifikátor proměnné typu ukazatel na typ `T`.

V překladačích pro reálný režim na PC lze před takovýto deklarátor zapsat některý z modifikátorů `__near`, `__far`, `__huge` nebo `__seg` (v Borland C++ také `__cs`, `__ds`, `__ss` nebo `__es`). Pak jde o explicitní deklaraci blízkého, vzdáleného, normalizovaného nebo segmentového ukazatele nebo blízkého ukazatele sdruženého s jedním z uvedených registrů. Pokud takovýto modifikátor neuvedeme, použije se ukazatel blízký nebo vzdálený podle paměťového modelu.

c) Je-li $\mathcal{D}$ tvaru `&E`, bude `E` identifikátor reference na typ `T`. Připomeňme si, že C++ nedovoluje reference na typ `void`. V překladačích pro reálný režim na PC lze před takovýto deklarátor zapsat modifikátor `__near` nebo `__far`. Pak jde o explicitní deklaraci blízké nebo vzdálené reference.

e) Je-li $\mathcal{D}$ deklarátor tvaru `F()` nebo `F(`*specifikace_formálních_parametrů*`)`, je `F` identifikátor funkce typu `T` (vracející hodnotu typu `T`). Následuje-li za deklarátorem tělo funkce, jde o definici, jinak jde o prototyp funkce (deklaraci, která pouze informuje překladač o existenci funkce).

Informativní deklarace tvaru `F()` znamená v C++ totéž co `F(void)`, tedy funkci bez parametrů. V jazyce C znamená tato deklarace funkci, o jejíchž parametrech neuvádíme žádné informace.

V prototypu můžeme ve *specifikaci_formálních_parametrů* uvést jména těchto parametrů; nemají však žádný význam, pouze usnadňují čtení programu.

V překladačích pro reálný režim na PC můžeme před identifikátor `F` uvést některý z modifikátorů `__near`, `__far` nebo `__huge`; dostaneme tak explicitní deklaraci blízké, vzdálené nebo robustní funkce (kap. 12.5). Pokud žádný modifikátor neuvedeme, bude funkce blízká nebo vzdálená podle použitého paměťového modelu.

V překladačích pro PC zde také můžeme uvést některý z modifikátorů označujících volací konvenci (`__cdecl`, `__pascal`, `__fastcall`, `__stdcall` aj.) Pokud jej neuvedeme, použije překladač implicitní volací konvenci.

1.5. Funkce a paměťové modely

Blízké a vzdálené funkce

V kapitole o ukazatelích jsme si řekli, že programech pro 16bitová prostředí můžeme používat blízké nebo vzdálené ukazatele. To se týká i funkcí, neboť volání funkce v sobě obsahuje skok na adresu začátku procedury a návrat z funkce znamená skok na adresu za místem, dokud jsme ji zavolali.

Proto překladače C a C++ pro 16bitová prostředí rozlišují také tzv. *blízké* (*near*) a *vzdálené* (*far*) funkce. Při volání blízké funkce se použijí blízké ukazatele, při volání vzdálené funkce se použijí vzdálené ukazatele.

Implicitně jsou všechny funkce v drobném, malém a kompaktním modelu blízké; ve středním a velkém modelu jsou implicitně vzdálené.

Chceme-li ve svém programu použít funkci, která neodpovídá implicitním požadavkům paměťového modelu, musíme ji explicitně deklarovat. K tomu použijeme modifikátory `__near` (pro blízké funkce) nebo `__far` (pro vzdálené funkce). Modifikátor je součástí deklarátoru funkce.

Příklady

```
int __far F();
int __far *f1();
```

```
int * __far f2();
int __near * far f3();
```

`F()` je vzdálená funkce typu `int`, `f1()` je funkce implicitního typu, která vrací vzdálený ukazatel na `int`, `f2()` je vzdálená funkce, která vrací ukazatel na `int` implicitní velikosti, a `f3()` je vzdálená funkce, která vrací blízký ukazatel na `int`.

Robustní funkce

V rozsáhlém modelu jsou všechny funkce implicitně robustní (*huge*). Při jejich volání se používají vzdálené ukazatele, navíc se při vstupu do takovéto funkce aktualizuje obsah registru DS. Pokud bychom chtěli explicitně deklarovat robustní funkci, musíme použít modifikátor `__huge`.

1.6. Funkce pro obsluhu přerušení

Funkce pro obsluhu přerušení jsou specialitou programů pro operační systém DOS. Abychom pochopili oč jde, musíme si nejprve povědět několik slov o přerušeních jako takových.

Přerušení

Určité události způsobí, že procesor přeruší svoji činnost a řeší vzniklou situaci, „ošetří přerušení“. Poté se může řízení vrátit zpět.

Přerušení mohou být způsobena hardwarovými příčinami nebo vyvolána programově; na PC jsou označena čísly 0 – 255. Hardwarové příčiny mohou být např.

dělení nulou	00,
přetečení	04,
signál časovače	08,
vstup z klávesnice	09

atd. Softwarové přerušení lze na PC vyvolat instrukcí asembleru

`INT` *`n`*

kde *n* je číslo přerušení. V Borland C++ lze také užít standardní funkci

`void geninterrupt(int` *`cislo_preruseni`*`),`

která má stejný (význam.

Mezi nejčastěji užívaná softwarová přerušení patří volání služeb DOSu (přerušení 33, tj. 0x21).

Procesory třídy 80x86 rezervují prvních 1024 bytů operační paměti pro 256 *vzdálených* ukazatelů na procedury pro obsluhu přerušení (tyto ukazatele se nazývají „vektory přerušení“). Ne všechny vektory přerušení jsou obsazeny, tj. pro některá přerušení nejsou obslužné rutiny definovány. Ty může obsadit uživatel vlastními funkcemi. Kromě toho lze předefinovat standardní procedury pro obsluhu přerušení.

Po příchodu přerušení procesor dokončí instrukci, kterou právě vykonává, uloží na zásobník obsah registrů FLAGS, CS a IP (registr příznaků a adresa instrukce, na kterou se po ošetření přerušení vrátí), zakáže další přerušení a provede vzdálený skok (skok s užitím čtyřbajtové adresy) na místo určené vektorem přerušení. Po ošetření přerušení obnoví pomocí uložené hodnoty obsah registru FLAGS a bude pokračovat tam, kde skončil při příchodu přerušení.

Po hardwarových přerušeních zůstane možnost dalších přerušení blokována. Oživení řadiče přerušení dosáhneme tím, že na port 32 zapíšeme číslo 32. Pro práci s porty na PC lze užít funkcí `inportb()` a `outportb()`.

Funkce pro obsluhu přerušení

Překladače C a C++ pro DOS umožňují, abychom si definovali své vlastní funkce pro obsluhu přerušení. Každá taková funkce musí být typu `void interrupt`. Její parametry mohou být registry procesoru 80x86, a to v pořadí

`bp, di, si, ds, es, dx, cx, bx, ax, ip, cs, flags.`

Deklarujeme je jako parametry typu `unsigned`. Za registrem `flags` mohou následovat další parametry.

Funkce pro obsluhu přerušení vždy nejprve uloží na zásobník obsah registrů AX, BX, CX, DX, ES, DS, SI, DI a BP. (Registry CS, IP a FLAGS uložil již procesor před jejím voláním.) Parametry funkce pro obsluhu přerušení pak zpřístupňují v těle funkce právě tyto uložené hodnoty.

Formální parametry `bp, ..., cs, flags`, odkazují právě na tyto *uložené* obsahy registrů. (Proto je rozumné použít uvedené identifikátory a dodržet jejich pořadí.)

Před návratem z funkce se obsahy registrů automaticky obnoví. Pokud změníme ve funkci pro obsluhu přerušení hodnotu některého z parametrů `bp, ..., flags`, použije se změněná hodnota.

Pokud ve funkci pro obsluhu přerušení chceme užívat aritmetické operace s reálnými čísly, musíme nejprve uložit stav koprocesoru 80x87 a před návratem jej obnovit.

Použití

K instalaci funkce pro obsluhu přerušení slouží funkce `setvect()`. Chceme-li získat ukazatel na aktuálně instalovanou funkci pro obsluhu určitého přerušení, použijeme funkci `getvect()`. S příkladem jsme se setkali v této kapitole oddílu 3 věnovaném cv-modifikátorům.

Poznámky

- Překlad funkce pro obsluhu přerušení končí instrukcí `IRET` (návrat z přerušení). Tato instrukce způsobí skok na vzdálenou adresu uloženou na vrcholu zásobníku a přitom ze zásobníku vyjme návratovou adresu a obsah registru FLAGS. Při překladu ostatních funkcí se na PC užívá instrukce `RET`, která nezahrnuje manipulaci s registrem příznaků.
- Funkce pro obsluhu přerušení můžeme volat i jako obyčejné funkce.

1.7. Objektové datové proudy v C++ – stará implementace

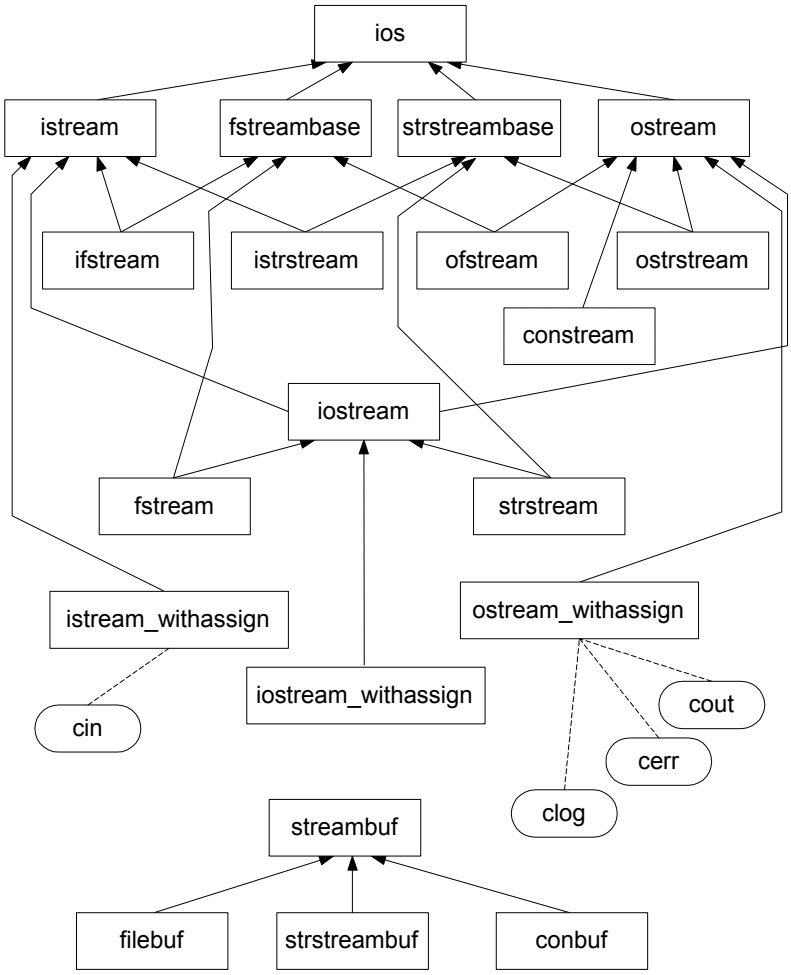
V tomto oddílu se seznámíme s klasickou implementací objektových datových proudů v podobě, v jaké se používala před přijetím standardu ISO.

Objektová koncepce knihovny datových proudů a využití přetížených operátorů přináší řadu výhod, z nichž asi nejdůležitější je možnost snadno definovat vlastní verzi vstupního resp. výstupního operátoru >> resp. << pro své vlastní datové typy, možnost definovat vlastní manipulátory a dokonce i možnost definovat nové datové proudy.

Třídy, které tvoří proudy

Datové proudy jazyka C++ jsou založeny na dvou hierarchiích objektových typů (obr. 1). Jedna z nich je odvozena od třídy `streambuf` a obsahuje objekty, které se starají o fyzickou realizaci vstupních a výstupních operací.

Potomci třídy `streambuf` obsahují metody specifické pro proudy orientované na soubory,do řetězců a příp. na konzolu.



Obr. 1.1 Hierarchie tříd tvořících datové proudy v klasické implementaci

Třída ios

Tato třída definuje vlastnosti, které jsou společné všem datovým proudům. Je virtuálním předkem dalších tříd a je definována v `iostream.h`.

Obsahuje ukazatel `bp` na sdružený objekt typu `streambuf`. Tento atribut využívají především metody potomků. Dále zde najdeme atribut `state` typu `int`, který obsahuje příznaky možných chybových stavů proudu. Tyto příznaky popisuje veřejně přístupný výčtový typ `ios::io_state`.

```
enum io_iostate {
  goodbit  = 0,         // Vše je OK
  eofbit   = 1,         // Nalezen konec souboru
  failbit  = 2,         // Poslední operace se nezdařila
  badbit   = 4,         // Pokus o neplatnou operaci
  hardbit  = 8          // Hardwarová chyba
};
```

Tabulka 1.1 Stavové příznaky proudů

Atribut `x_flags` typu `long` obsahuje formátovací příznaky. Ty jsou popsány pomocí nepojmenovaného veřejně přístupného výčtového typu:

```
enum {
  skipws     =     0x1, // Při vstupu budou přeskoč bílé znaky
  left       =     0x2, // Zarovnej výstup vlevo
  right      =     0x4, // Zarovnej výstup vpravo
  internal   =     0x8, // První znak výstupního pole je znaménko
                        // nebo označení číselné soustavy (např. 0x)
  dec        =    0x10, // Desítková soustava
  oct        =    0x20, // Osmičková soustava
  hex        =    0x40, // Šestnáctková soustava
  showbase   =    0x80, // Při výstupu indikuj číselnou soustavu
  showpoint  =   0x100, // Zobraz se desetinnou tečku
  uppercase  =   0x200, // Při výstupu v šestnáctkové soustavě
                        // použij velká písmena
  showpos    =   0x400, // Kladná čísla vystupují se znaménkem "+"
  scientific =   0x800, // Výstup reálných čísel v semilogaritmickém
                        // tvaru
  fixed      =  0x1000, // Výstup reálných čísel ve tvaru
                        // s pevnou řádovou čárkou
  unitbuf    =  0x2000, // Spláchni proud po výstupu
  stdio      =  0x4000, // Spláchni proudy sdružené se stdout
                        // a stderr po výstupu
};
```

Tabulka 1.2 Formátovací příznaky proudů

Atributy `x_precision`, `x_width` a `x_fill` jsou typu `int` a obsahují přesnost, šířku výstupního pole a vyplňovací znak. Implicitní hodnota prvních dvou je 0, u posledního je to mezera.

S těmito atributy můžeme pracovat buď pomocí metod nebo pomocí manipulátorů. Podívejme se nyní na některé z metod (manipulátorům věnujeme samostatný oddíl). Většina metod třídy `ios` nastavuje nebo vrací hodnoty atributů a tak zjišťuje stav proudu nebo určuje formátování vstupu a výstupu.

Hodnoty jednotlivých stavových bitů v atributu `state` lze zjišťovat pomocí veřejně přístupných metod `bad()`, `eof()`, `fail()` a `good()`, které vracejí výsledek typu `int`. Voláním metody `ios::clear()()` můžeme nastavit nebo vynulovat příznaky chyb (kromě příznaku `hardfail`).

Metoda `int ios::rdstate()` vrací atribut `state`, tj. slovo obsahující všechny chybové příznaky daného proudu.

Následující metody existují ve dvou variantách. Metoda bez parametrů pouze zjistí aktuální hodnotu určitého příznaku, metoda s jedním parametrem nastaví novou hodnotu a vrátí hodnotu původní. Jde o tyto metody:

```
long ios::flags();        // vrátí hodnotu formátovacích příznaků,
long ios::flags(long P);    // uložených v x_flags;
int ios::precision();
int ios::precision(int P);  // Vracejí přesnost
int ios::width();         // Vracejí nastavenou šířku vstupního
int ios::width(int s);      // nebo výstupního pole
char ios::fill();           // Vrátí nastavený vyplňovací znak
char ios::fill(char Vypln);
```

Metoda

```
long ios::setf(long P)
```

vrátí předchozí nastavení formátovacích příznaků a nastaví novou hodnotu, danou parametrem `P`. Můžeme také použít přetíženou verzi metody `setf()` se dvěma parametry, ve které první parametr obsahuje hodnotu nastavovaného příznaku a druhý parametr říká, které pole chceme nastavit. Možné hodnoty druhého parametru jsou definovány jako konstanty ve třídě `ios` a jejich identifikátory ukazuje tabulka 1.3.

```
  adjustfield  // určuje, že nám jde o zarovnání
  basefield    // určuje, že nám jde o základ číselné soustavy
  floatfield   // určuje, že nám jde o formát reálných čísel
```

Tabulka 1.3 Možné hodnoty druhého parametru metody setf()

Např. příkaz

```
cout.setf(ios::hex, ios::basefield);
```

říká, že chceme změnit tu část nastavení, které určuje základ číselné soustavy pro vstup nebo výstup celých čísel. Příkaz

```
cout.setf(ios::scientific, ios::floatfield);
```

způsobí výstup reálných čísel v semilogaritmickém tvaru.

Chceme-li zjistit stav proudu, můžeme použít přetížené operátory `!` a `void*()`. Operátor `!` vrací 1, jestliže se poslední vstupní nebo výstupní operace s proudem nepodařila, tj. je-li nastaven některý z příznaků `eofbit`, `failbit` nebo `badbit`. Operátor přetypování na `void*` vrací nulu, pokud se poslední operace nepodařila, a ukazatel na proud, jestliže proběhla v pořádku.

Ve třídě `ios` je dále definován výčtový typ `ios::openmode`. Hodnoty tohoto výčtového typu lze skládat pomocí operátoru `|` jako bitové příznaky.

```
enum open_mode {
  in        =   0x1,     // Otevře soubor pro čtení
  out       =   0x2,     // Otevře soubor pro zápis
  ate       =   0x4,     // Po otevření najde konec souboru;
                         // to však lze změnit pomocí
                         // metody ostream::seekp()
  app       =   0x8,     // Po otevření najde konec souboru;
                         // veškerý výstup půjde na konec souboru
                         // a nelze to změnit
  trunc     =  0x10,     // Pokud soubor existuje, smaže se
  nocreate  =  0x20,     // Pokud soubor neexistuje, nastane chyba
  noreplace =  0x40,     // Pokud soubor existuje, nastane chyba
  binary    =  0x80,     // Soubor se otevře jako binární
                         // (implicitně se otevírá v textovém režimu)
};
```

Tabulka 1.4 Režimy otevření souboru

Od třídy `ios` jsou odvozeny třídy `istream` a `ostream`, které představují základ vstupních a výstupních proudů, třída `fstreambase`, která je základem proudů, orientovaných na soubory, a třída `strstreambase`, která je základem paměťových proudů. Také tyto třídy se v programech zpravidla přímo nepoužívají. Od nich jsou pak odvozeny třídy `fstream`, `strstream`, `istream_withassign`, `ostream_withassign` a některé další, které již opravdu slouží ke vstupním a výstupním operacím.

Pokud výslovně neuvedeme něco jiného, jsou následující třídy deklarovány v hlavičkovém souboru `iostream.h`.

Třída istream

Tato třída je základem vstupních proudů. V ní je pro účely formátovaného vstupu přetížen operátor >> (*extraktor*). Jako příklad si ukážeme deklaraci tohoto operátoru pro typ `long`:

```
istream & istream::operator>> (long& _l);
```

Všimněte si, že tento operátor vrací odkaz na datový proud, pro který jej zavoláme. To znamená, že např. výraz

```
cin >> i;
```

představuje odkaz na proud `cin` ve stavu po přečtení proměnné `i`. Díky tomu můžeme přetížené operátory zřetězovat. Jestliže se při čtení do proměnné `i` nějakým způsobem změnil stav proudu `cin`, bude následující operace probíhat již se změněným proudem.

Ve třídě `istream` jsou mimo jiné definovány metody

```
long istream::tellg();
istream& istream::seekg(long);
istream& istream::seekg(long, seek_dir);
```

První z nich umožňuje zjistit aktuální pozici v souboru, druhá a třetí umožňuje tuto pozici změnit. Metodě `seekg(long)` zadáváme jako jediný parametr hodnotu vrácenou metodou `tellg()`, metodě `seekg(long, seekdir)` zadáme jako druhý parametr jednu z hodnot `beg`, `cur` nebo `end`. První parametr určuje velikost posunu v bajtech, druhý říká, odkud to máme počítat — zda od počátku, od aktuální pozice nebo od konce.

Třída ostream

Tato třída je pro změnu základem výstupních proudů. Je v ní (jako metoda) přetížen operátor << (*insertor*), který slouží k formátovanému výstupu. Jako příklad si ukážeme definici tohoto operátoru pro typ `int`:

```
inline ostream& ostream::operator<< (int _i)
{return *this << (long) _i;
}
```

Zde konvertujeme levý operand na hodnotu typu `long` a použijeme operátor << pro tento typ; pak vrátíme odkaz na proud, pro který jsme jej zavolali.

Ve třídě `ostream` jsou také definovány metody

```
long ostream::tellp()
ostream& ostream::seekp(long);
ostream& ostream::seekp(long, seek_dir);
```

První z nich zjistí aktuální pozici v proudu, druhá a třetí umožňuje aktuální pozici změnit. Pro jejich parametry platí podobné podmínky jako pro metody `istream::tellg()` a `istream::seekg()`.

Třída iostream

Třída `iostream` obsahuje prostředky pro vstup i pro výstup. Ke zděděným vlastnostem nepřidává nic nového.

Třídy ostream_withassign, istream_withassign a iostream_withassign

V těchto třídách je navíc definován přiřazovací operátor, který umožňuje sdružit objekt této třídy s objektem typu `streambuf`. Tím, že se změní sdružený objekt třídy `streambuf`, se proud přesměruje.

V hlavičkovém souboru `iostream.h` jsou definovány standardní instance

```
extern ostream_withassign cout;
extern ostream_withassign cerr;
```

```
extern ostream_withassign clog;
extern istream_withassign cin;
```

Proud `cout` slouží k formátovanému výstupu do souboru `stdout`, proudy `cerr` a `clog` představují standardní chybový výstup. První z nich nemá vyrovnávací paměť, druhý má. Proud `cin` slouží k formátovanému vstupu ze souboru `stdin`.

Poznámka

V některých překladačích jsou objekty `cout`, `cerr` a `clog` instance třídy `ostream` a objekt `cin` je instancí třídy `istream`.

Třídy fstream, ifstream a ofstream

Třída `fstream`, definovaná v hlavičkovém souboru `fstream.h`, obsahuje prostředky pro formátovaný vstup a výstup do externích souborů. Po třídě `iostream` zdědila operátory >> a << a další metody. Konstruktor

```
fstream::fstream(const char* jmeno, int rezim)
```

vytvoří nový proud, napojí jej na soubor `jmeno` a tento soubor otevře v režimu `rezim` s atributem `pristup`. Jako `rezim` můžeme použít hodnoty výčtového typu `ios::openmode` (tab. 1.3). Jako třetí parametr můžeme použít některou z hodnot `S_IREAD` (pouze čtení) a `S_IWRITE` (pouze zápis), definovaných v souboru `sys\stat.h`. Implicitní hodnota třetího parametru `filebuf::openprot` odpovídá `S_IREAD | S_IWRITE`.

Můžeme také použít konstruktor bez parametrů. Ten pouze vytvoří proud, nic více. Tento proud můžeme později sdružit se souborem pomocí metody `open()`, která má stejné parametry jako první uvedený konstruktor.

Po použití můžeme soubor sdružený s proudem uzavřít (a odpojit od proudu) pomocí metody `close()` (zděděné po třídě `filebuf`).

Pokud potřebujeme soubor pouze pro vstup resp. pouze pro výstup, můžeme použít tříd `ifstream` resp. `ofstream`. Jejich konstruktory lze volat s jediným parametrem, jménem souboru.

Příklad

Nejprve napíšeme program, který vytvoří v aktuálním adresáři textový soubor DATA.DTA a zapíše do něj čísla od 0 do 9, na každý řádek jedno. Použijeme `fstream`, i když bychom vystačili s proudem `ofstream`.

```
#include <fstream>
void chyba(){/*...*/}
fstream F;                              // Deklarujeme proud

void main()
{
  F.open("data.dta", ios::out);    // Otevřeme soubor
  if(!F) chyba();                  // Test, zda se to podařilo
  for(int i = 0; i < 10; i++) F << i << endl;      // Výpis
}
```

Pomocí operátoru `!` testujeme, zda se podařilo otevřít soubor. O jeho uzavření se v tomto jednoduchém případě nemusíme starat, udělá to za nás destruktor proudu `F`.

Dále napíšeme funkci, která otevře soubor DATA.DTA pro čtení, přečte z něj všechna čísla a vypíše je do standardního výstupu:

```
#include <fstream.h>

void ctipis()
{
  ifstream F("data.dta");
  if(!F) chyba();
```

```
    int i;
    while(F >> i)
    {
      cout << i << endl;
    }
}
```

Výraz `F >> i` má hodnotu proudu po provedeném čtení. Díky operátoru přetypování na `void*` jej můžeme použít v podmínce. Jakmile narazíme při čtení na konec souboru, vrátí tento operátor 0, neboť je nastaven příznak `eofbit`.

Pokud bychom chtěli tento proud dále používat, museli bychom tento příznak pomocí metody `clear()` vynulovat.

Paměťové proudy strstream, istrstream a ostrstream

Tyto třídy najdeme v hlavičkovém souboru `strstrea.h`. Slouží pro zápis do znakových řetězců a pro čtení z nich. Konstruktor třídy `strstream` má tvar

```
strstream::strstream(char* pole, int delka, int rezim).
```

Parametr `pole` je ukazatel na znakový řetězec (pole znaků), ke kterému bude proud připojen. Toto pole musí být dostatečně dlouhé, aby se při výstupních operacích nepřekročila jeho horní mez.

Je-li parametr `delka` kladný, určuje délku daného pole; 0 znamená řetězec, zakončený `'\0'`, a záporné číslo znamená pole nekonečné délky. Parametr `rezim` musí být jedna z hodnot výčtového typu `ios::openmode` (tab. 1.4); jejich význam pro paměťové proudy shrnuje tabulka 1.5.

příznak	**význam**
`ios::in`	Vstupní proud, čte se od počátku pole
`ios::out`	Výstupní proud, zapisuje se od počátku pole
`ios::ate`	Výstupní proud. Pole je řetězec ukončený `'\0'`, zapisuje se od tohoto znaku
`ios::app`	Znamená totéž co `ios::ate`

Tabulka 1.5 Režimy otevření paměťového proudu

Potřebujeme-li paměťový proud pouze pro vstup resp. pouze pro výstup, můžeme použít proudů `istrstream` resp. `ostrstream`. V jejich konstruktorech neuvádíme parametr `rezim`.

Třída constream

Třída *constream* je borlandské rozšíření knihovny datových proudů; najdeme ji spolu s řadou speciálních manipulátorů v hlavičkovém souboru `constrea.h`. Obsahuje proudy orientované na konzolu PC a poskytuje podobné služby jako knihovna `conio.h`, o níž budeme hovořit v oddílu 20.6.

Formátované vstupy a výstupy

Formátování vstupů a výstupů určují především příznaky nastavené v atributech `x_precision`, `x_width` a `x_fill` a dalších. Pro práci s nimi můžeme používat nejen metody, o kterých jsme hovořili výše, ale především řadu manipulátorů; jejich přehled najdeme v tabulce 1.6.

Základ číselné soustavy

Celá čísla implicitně vystupují v desítkové soustavě; při vstupu se implicitně používá konvence jazyka C, tj. čísla začínající `0x` se berou jako šestnáctková, čísla začínající `0` se berou jako osmičková a ostatní jako desítková.

Pokud chceme explicitně předepsat číselnou soustavu, použijeme některý z manipulátorů `dec`, `hex` nebo `oct`, které specifikují po řadě desítkovou šestnáctkovou nebo osmičkovou soustavu. Můžeme také použít manipulátor `setbase(n)`, kde `n` je jedno z čísel 0, 8, 10, 16. Hodnota 0 znamená obnovení implicitního stavu.

Šířka výstupního pole

Šířka výstupního pole udává nejmenší počet znaků, které vystoupí. Pokud má vystupující hodnota více znaků, výstup „přeteče“, zabere tolik, kolik je nezbytné.

Šířku pole lze nastavit pomocí manipulátoru `setw(int n)` nebo pomoci metody `width()`. Tato metoda umožňuje také šířku pole zjistit.

Manipulátor	užití	význam
`dec`	*ios*	Následující vstupy nebo výstupy v tomto proudu budou probíhat v desítkové soustavě
`oct`	*ios*	Následující vstupy nebo výstupy v tomto proudu budou probíhat v osmičkové soustavě
`hex`	*ios*	Následující vstupy nebo výstupy v tomto proudu budou probíhat v šestnáctkové soustavě
`setbase(n)`	*ios*	n = 0, 8, 10, 16 specifikuje číselnou soustavu nebo implicitní stav
`endl`	*o*	Vloží znak pro přechod na nový řádek a vyprázdní vyrovnávací paměť, tj. řádek se vypíše do souboru
`ends`	*o*	Vloží znak '\0' na konec řetězce (jen paměťové proudy)
`flush`	*o*	Vyprázdní vyrovnávací paměť proudu (spláchne)
`ws`	*i*	Přikazuje přeskočit bílé znaky ve vstupním proudu
`setprecision(int n)`	*ios*	Určuje přesnost n
`resetiosflags(long n)`	*ios*	Vloží 0 do specifikovaných bitů složky `ios::x_flags`, která určuje formátování dat — jako parametry používáme hodnoty výčtového typu z tab. 1.2, definovaného ve třídě *ios*
`setiosflags(long n)`	*ios*	Vloží 1 do specifikovaných bitů složky `ios::x_flags`, která určuje formátování dat — jako parametry používáme konstanty z tab.18.2
`setfill(int n)`	*ios*	Definuje vyplňovací znak
`setw(int n)`	*ios*	Definuje šířku pole

Tabulka 1.6 Přehled manipulátorů; i, o resp. ios znamená, že jde o manipulátor definovaný na proudu istream, ostream resp. ios

Pozor

Po každém formátovaném výstupu se šířka výstupního pole automaticky nastaví na implicitní hodnotu 0. Musíme ji tedy musíme nastavovat pro každou vystupující hodnotu zvlášť.

Přesnost

Přesnost znamená počet míst za desetinnou tečkou při výstupu reálných čísel (nic víc — na rozdíl od „přesnosti“ při výstupu pomocí funkce `printf()`, o níž budeme hovořit v oddílu věnovaném standardním vstupům a výstupům jazyka C). Zadáme ji pomocí manipulátoru `setprecision(int n)`, kde `n` je počet desetinných míst.

Vyplňovací znak

Potřebuje-li výstup ve skutečnosti méně místa než předepisuje šířka, vyplní proud tyto nadbytečné pozice vyplňovacím znakem. Implicitním vyplňovacím znakem je mezera; jinou hodnotu nastavíme manipulátorem `setfill(int n)`. Jako parametr zadáme požadovaný znak (buď číselně, jako hodnotu kódu, nebo jako znakovou konstantu). Chceme-li zjistit aktuální hodnotu vyplňovacího znaku, použijeme metodu `fill()`.

Další nástroje

Manipulátory `setiosflags(long n)` a `resetiosflags(long n)` umožňují nastavovat nebo nulovat příznaky z tabulky 1.2. Fungují takto: Manipulátor `setiosflags(n)` vezme parametr `n` a ve stavovém slově `x_flags` uloží hodnotu 1 do všech bitů, které mají hodnotu 1 v `n`, a ostatní bity ponechá beze změny. Udělá tedy operaci

```
x_flags |= n;
```

Manipulátor `resetiosflags(n)` uloží hodnotu 0 do bitů, v nichž je 1 v `n`, a ostatní ponechá beze změny. Udělá tedy operaci

```
x_flags &= (~n);
```

Hodnotu parametru poskládáme jako bitový součet příznaků z tabulky 1.2. Aktuální hodnotu atributu `x_flags` zjistíme pomocí metody `flags()`.

Manipulátor `flush` „spláchne“ proud, to znamená, že přenese všechna data z vyrovnávací paměti do souboru. (Připomeňme si, že „spláchnutí“ dělá i manipulátor `endl`, který kromě toho vloží do proudu znak pro přechod na nový řádek.)

Příklad

Napíšeme funkci `tab()`, která vytvoří tabulku nějaké matematické funkce. Parametry funkce `tab()` budou ukazatel na tabelovanou funkci, meze intervalu, počet kroků tabelace, proud, do kterého chceme výsledek zapisovat, a jméno tabelované funkce.

```
#include <string.h>
#include <fstream.h>
#include <iomanip.h>
#include <math.h>

void tab(double (*F)(double), double Od, double Do,
          int Kolik, ostream& Proud, const char *Jmeno)
{
  const w = 10;
  Proud << "Tabulka funkce " << Jmeno << endl;
  Proud << "---------------";
  for(int i = 0; i < strlen(Jmeno); i++) Proud << '-';
  Proud << endl;
  double krok = (Do-Od)/Kolik;
  long flags = Proud.flags();
  long prec = Proud.precision();
  Proud  << setprecision(5)
```

```
        << setiosflags(ios::showpoint|ios::right);
  Proud << setw(w) << "x" << setw(w-3) << Jmeno
        << "(x)" << endl;
  double x = 0d;
  for(i=0; i < Kolik+1; i++)
  {
    Proud << setw(w) << x << setw(w) << F(x) <<endl;
    x += krok;
  }
  Proud << setprecision(prec);
  Proud.flags(flags);
}
```

Datový proud předáváme odkazem jako parametr typu `ostream&`. Skutečným parametrem pak může být libovolný potomek, tj. např. proud typu `ofstream`, `ostream_withassign` atd.
Na počátku `vytvoříme` hlavičku tabulky; tu poskládáme z pevně zadaného textu a ze jména tabelované funkce (parametr `Jmeno`).
Pak si zapamatujeme aktuální hodnotu formátovacích příznaků a přesnosti, abychom mohli před návratem uvést proud do původního stavu, a nastavíme přesnost. Protože chceme vypisovat vždy všech 5 desetinných míst, nastavíme také příznak `ios::showpoint`.
V následujícím cyklu postupně vypisujeme hodnoty `x` a `F(x)`. Všimněte si, že šířku nastavujeme před každým výstupem. Nakonec nastavíme původní hodnoty příznaků a přesnosti.
Zavoláme-li tuto funkci příkazem

```
ofstream F("data.dta");
tab(atan, 0, 1, 10, F, "arctg");
```

najdeme v souboru DATA.DTA výstup

```
Tabulka funkce arctg
--------------------
 x         arctg(x)
 0.00000   0.00000
 0.10000   0.09967
 0.20000   0.19740
 0.30000   0.29146
 0.40000   0.38051
 0.50000   0.46365
 0.60000   0.54042
 0.70000   0.61073
 0.80000   0.67474
 0.90000   0.73282
 1.00000   0.78540
```

Předáme-li této funkci jako parametr `cout`, dostaneme týž výstup na obrazovku. O další vylepšení vzhledu tabulky se můžete pokusit sami.

Paměťové proudy

Paměťové proudy jsou podobné souborovým proudům, až na to, že zdrojem nebo spotřebičem dat jsou pole znaků. (První parametr konstruktoru je vždy typu `char *`.) Nelze je otevírat nebo uzavírat pomocí metod `open()` nebo `close()`, neboť ty zde nejsou k dispozici. Ke zjištění, resp. nastavení aktuální pozice v řetězci můžeme použít metody `tellg()`, resp. `seekg()` (pro čtení) a `tellp()`, resp. `seekp()` (pro zápis).

Při zápisu vložíme do výstupu znak `'\0'` ukončující řetězec pomocí manipulátoru `ends`.

Poznámka

Proud `strstream` slouží pro vstup i pro výstup, ale ze znakového řetězce, na který ho napojíme, umí přečíst jen to, co do něj zapíšeme. Pokud chceme pomocí paměťových proudů zpracovat data uložená v řetězci nějakým jiným způsobem, musíme k tomu použít vstupní proud `istrstream`.

Znakově orientované vstupy a výstupy

Prostředky pro znakově orientované (tj. neformátované) vstupní a výstupní operace jsou definovány ve třídách `istream`, resp. `ostream`, takže jsou k dispozici i v odvozených třídách. Formátovací příznaky nastavené v atributech proudu nemají pro tuto operace význam.

Pro výstup jednoho bajtu slouží metoda

```
ostream& ostream::put(char c).
```

která vloží do výstupního proudu znak `c`.

Potřebujeme-li zařídit výstup většího množství bytů, použijeme metodu

```
ostream& ostream::write(const char *p, int n);
```

která do výstupního proudu okopíruje `n` bytů z paměti od adresy `p`. Tato metoda neskončí, jestliže narazí bajt s hodnotou 0 (není určena jen pro výstup znaků).

Pro neformátované čtení můžeme použít některou z metod

```
int istream::get();
istream& istream::get(char& c);
istream& istream::get(char *p, int i, char c = '\n');
```

První z těchto funkcí přečte následující znak ze vstupního proudu a vrátí ho, druhá přečte následující znak ze vstupního proudu, uloží ho do proměnné `c` a vrátí odkaz na proud po čtení (to lze využít např. k testování konce souboru).

Poslední z těchto funkcí přečte ze vstupního proudu nejvýše `i` znaků a uloží je do pole `p`. Čtení může skončit i dříve, pokud se ve vstupním proudu narazí na znak `c` (implicitně konec řádku). Ukončovací znak (omezovač) se již nepřečte a zůstane ve vstupním proudu.

Chceme-li přečíst celý řádek, můžeme použít metodu

```
istream& istream::read(char *p, int i, char c = '\n');
```

která přečte znaky až po omezovač `c`, nejvýše však `i` znaků, uloží je do pole `p` a připojí `'\0'`. Omezovač odstraní z proudu, ale neuloží ho do `p`.

Vedle toho můžeme pro neformátované čtení užít metodu

```
istream& istream::read(char *p, int i)
```

která přečte ze vstupního proudu `i` bytů (pokud nenarazí na konec souboru).

Chceme-li zjistit hodnotu následujícího znaku v proudu, aniž bychom ho vyňali z proudu, použijeme metodu

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

Chceme-li přečíst určitý počet znaků, aniž by nás zajímala jejich skutečná hodnota, poslouží nám metoda

```
istream& istream::ignore(int n=1, int zar=EOF);
```

která přeskočí `i` znaků ve vstupním proudu. Přeskakování může skončit i dříve, pokud tato funkce narazí na znak `zar` (implicitně konec souboru).

Někdy také potřebujeme vrátit přečtený znak zpět do vstupního proudu. K tomu slouží metod

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

Vrácený znak bude první, který se přečte při následující vstupní operaci. Počet znaků, které můžeme vrátit, ovšem není neomezený — např. v Borland C++ 3.1 lze vrátit maximálně 4 znaky; pokusíme-li se vrátit pátý, nastane chyba (nastaví se příznak `failbit`) a další operace s proudem se nepodaří, pokud příznak chyby nevynulujeme voláním metody `clear()`.

Rozšiřování možností vstupů a výstupů

Knihovna objektových datových proudů je založena na objektech jazyka C++, a proto ji můžeme poměrně snadno rozšířit. Nejčastěji se setkáme s nutností definovat vlastní vstupní a výstupní operátory pro naše objektové nebo výčtové typy nebo s definicí vlastních manipulátorů. S definicí vlastních zásadně nových proudů se setkáváme podstatně méně (pro něco takového potřebujeme nejen podrobné znalosti konstrukce datových proudů, ale i systému, pro který je nový proud určen). Proto se vlastní definicí proudů nebudeme zabývat; případné zájemce odkazuji na podrobný popis ve druhé části knihy [9]

Vlastní vstupní a výstupní operátory

Pro formátované vstupy a výstupy slouží přetížené operátory >> a <<. Pro vestavěné typy jsou definovány jako metody tříd `istream` a `ostream`; pro uživatelské typy je musíme definovat jako obyčejné funkce.

Má-li vstupní operátor pro typ `X` fungovat podobně jako vestavěné operátory, musíme ho definovat takto:

```
istream& operator >>(istream& Proud, X& x)
{
  // Čtení údaje typu X a uložení do x
  return Proud;
}
```

Náš operátor musí mít první parametr typu `istream&` (*předáváme odkazem!*) a musí vracet odkaz na tento proud. Druhý parametr je odkaz na proměnnou, do které se uloží přečtená hodnota.

Podobně pro výstupní operátor pro typ `X` je v podstatě závazný prototyp

```
ostream& operator <<(ostream& Proud, X& x)
{
  // ...zápis údajů z parametru x do proudu Proud
  return Proud;
}
```

Proud musíme předávat i vracet vždy odkazem, neboť jinak by se překladač snažil vytvořit jeho kopii (a to je nejen zbytečné, ale i nežádoucí a dokonce zakázané, neboť kopírovací konstruktor proudů je deklarován jako chráněný.) Vystupující hodnotu typu `X` předáváme hodnotou nebo odkazem — zde si můžeme vybrat.

Operátory << a >> definujeme vždy na proudech `ostream`, resp. `istream`. Pravidla jazyka sice nezakazují definovat je na jejich potomcích, např. na třídě *fstream*, ale způsobíme si tím problémy — překladač se např. bude za jistých okolností tvářit, že nezná tyto operátory pro vestavěné typy. (Podrobnější rozbor najdete v [6].)

Příklad

Už ve 2. kapitole jsme se setkali s objektovým typem pro komplexní čísla. Nyní definujeme šablonu `complex`, která nám umožní pracovat s komplexními čísly se složkami různých numerických typů, a pro tyto typy definujeme vstupní a výstupní operátory.

```
#include <iostream.h>

template <class T> class complex
{ // Komplexní čísla
  T re, im;
public:
  complex(T r, T i):re(r), im(i){}
  // ..a další metody
  friend ostream& operator<<(ostream& P, complex<T>& c);
  friend istream& operator>>(istream& P, complex<T>& c);
};

template<class T> ostream& operator<<(ostream& P, complex<T>& c)
{ return P << c.re << ' ' << c.im;
}

template<class T> istream& operator>>(istream& P, complex<T>& c)
{
  return P >> c.re >> c.im;
}
```

Náš operátor << vypisuje komplexní čísla jako dvojice reálných čísel. Jeho zjevnou nevýhodou je, že nebude spolupracovat s manipulátorem `setw(n)`: Nastavená šířka se použije pro první složku, ale druhá vystoupí s implicitní šířkou. Proto upravíme definici operátoru << takto:

```
template<class T> ostream& operator<<(ostream& P, complex<T>& c)
{
  int w = P.width();             // Zjistíme šířku
  return P << c.re << ' ' << setw(w) << c.im;
}
```

Nyní se použije zadaná šířka na obě složky komplexních čísel. Můžete si zkusit navrhnout jiné tvary operátoru, které budou např. zapisovat komplexní čísla ve tvaru `a+bi`, které budou chápat šířku jako počet znaků pro celé komplexní číslo apod.

Vlastní manipulátory

V tomto oddílu si ukážeme, jak manipulátory vlastně fungují a jak si můžeme definovat vlastní. Výklad vychází z vlastností překladačů Borland C++ 3.1, 4.0 a 4.5.

Manipulátory bez parametrů

Manipulátor bez parametrů na proudu `ostream` je identifikátor funkce, která vrací hodnotu typu `ostream&` a má jeden parametr typu `ostream&`; podobně manipulátor bez parametrů na proudu `istream` je identifikátor funkce, která vrací hodnotu typu `istream&` a má jeden parametr typu `istream&`.

Ve třídě `ostream`, resp. `istream` je přetížen operátor <<, resp. >>, který tuto funkci spustí a předá jí jako skutečný parametr aktuální proud:

```
inline ostream& ostream::operator<<(ostream & (* _f) (ostream&))
{ return (* _f)(*this);
}
```

Tato funkce dostane odkaz na proud a odkaz na něj také vrátí. Díku tomu může měnit stav proudu, vkládat do něj data pod.

Příklad

Napíšeme manipulátor `mez10`, který vloží do výstupního proudu 10 mezer.

```
ostream& mez10(ostream& Ostr)
{ return Ostr << "          ";
}
```

Příkaz

```
cout << "ahoj" << mez10 << "lidi";
```

způsobí nyní výstup

```
ahoj          lidi
```

Manipulátory s jedním parametrem

Manipulátor s parametrem je identifikátor, za kterým následuje parametr v závorkách. Ze syntaktických pravidel jazyka C++ plyne, že to musí být buď volání funkce nebo instance objektového typu, na kterou aplikujeme operátor volání funkce.

Standardní manipulátory s jedním parametrem jsou zpravidla implementovány jako funkce, které vracejí hodnotu speciálního objektového typu. V této instanci je uložen parametr manipulátoru a ukazatel na „výkonnou“ funkci, tj. funkci, která provede to, co se od manipulátoru požaduje.

Ve třídách `istream` a `ostream` jsou přetíženy operátory >> a <<, které z této instance vyjmou potřebná data a spustí výkonnou funkci.

Podívejme se na implementaci manipulátoru `setfill(c)` v Borland C++. V hlavičkovém souboru `iomanip.h` je definována obyčejná funkce (není to metoda žádného objektového typu)

```
smanip_int setfill(int);
```

Tato funkce vrátí instanci třídy `smanip_int`, používané pro manipulátory definované na proudu `ios`, tedy společné pro všechny proudy.

Implementace funkce `setfill()` může vypadat takto:

```
smanip_int setfill(int n)
{ return smanip_int(sfill, n);
}
```

Tato funkce prostě zavolá konstruktor třídy `smanip_int`, jenž vytvoří instanci, do které uloží adresu výkonné funkce `sfill()` a parametr `n`.

Funkce `sfill()` je definována takto:

```
ios& sfill(ios& i, int n)
{
  i.fill(n);
  return i;
}
```

Také zde je důležité, že se výkonné funkci předává proud odkazem a že tato funkce proud odkazem také vrací. Postup provedení příkazu

```
cout << setfill('*');
```

je následující:

- Nejprve funkce `setfill()` vytvoří instanci třídy `smanip_int` a uloží do ní adresu funkce `sfill()` a hodnotu `'*'`.
- Pak operátor <<, přetížený pro typ `smanip_int`, z této instance vyjme adresu výkonné funkce zavolá ji. Jako parametry jí předá odkaz na aktuální proud a hodnotu `'*'`.
- Výkonná funkce nastaví v daném proudu vyplňovací znak a vrátí odkaz na tento proud. Tento odkaz pak vrátí i operátor <<.

Poznámky

Jména pomocných tříd se mohou v různých implementacích lišit. V Borland C++ 3.1 to je `smanip_int`, `omanip_int` a `imanip_int` pro manipulátory na proudech `ios`, `ostream` a

istream; pro manipulátory s parametrem typu long jsou jména pomocných tříd podobná, končí však _long.

V novějších překladačích jsou tyto třídy generovány pomocí šablon, takže se jmenují smanip<int>, omanip<int> atd.

Ve Visual C++ 5.0 Se tyto třídy jmenují __SMANIP_int apod.

Definice vlastního manipulátoru

Z předchozího povídání plyne, že chceme-li si definovat vlastní manipulátor s jedním parametrem typu int na proudu ostream, musíme

- definovat výkonnou funkci, která bude mít prototyp

```
ostream& vf(ostream& ostr, int n);
```

- definovat funkci, která se bude jmenovat jako náš manipulátor a která vytvoří pomocnou instanci třídy omanip_int (či jiné, podle proudu a překladače).
  Podobný postup platí i v případě manipulátorů na jiných proudech.

Příklad

Napíšeme manipulátor, který vloží do výstupního proudu zadaný počet přechodů na nový řádek. Použijeme Visual C++ 5.0; zde se pomocná třída pro proud ostream jmenuje __OMANIP_int. Výkonná funkce bude

```
#include <iomanip.h> // Nikoli IOMANIP, ale IOMANIP.H!
ostream& _linky(ostream& os, int m)
{
  while (m--) os << endl;
  return os;
}
```

Vlastní definice manipulátoru je už jednoduchá:

```
__OMANIP_int linky(int m)
{ return __OMANIP_int (_linky, m);
}
```

Nyní můžeme napsat

```
cout << "Tady" << linky(5) << "a tam"<< endl;
```

a mezi tato dvě slova se vloží 4 přechody na nový řádek. Pokud bychom chtěli přeložit týž program v Borland C++ 4.0 nebo pozdějším, museli bychom místo __OMANIP_int psát omanip<int>.

Manipulátory s parametry jiných typů

Podobně lze definovat i manipulátory s parametry jiných typů. V hlavičkových souborech iomanip.h jsou připravena makra nebo šablony, které umožňují generovat pomocné třídy a operátory << nebo >>. Jejich popis by však přesáhl možnosti této učebnice. Podrobnosti pro borlandské překladače lze najít v [9] a [10].

Hlavní rozdíly mezi standardem a klasickou implementací proudů

Verze objektových datových proudů popsaná ve standardu jazyka [1] se na první pohled příliš neliší od „klasické“ verze, o níž jsme hovořili v předchozím oddílu, a proto si budeme povídat především o rozdílech mezi standardní a klasickou verzí. I když se po technické stránce jsou rozdíly mezi klasickou a standardní implementací poměrně hluboké, z hlediska běžného použití jsou zřejmé především tyto odlišnosti:

- Standardní proudy mohou za jistých okolností vyvolávat výjimky.
- Hierarchie standardních proudů využívá šablon, což umožňuje mít jak proudy pro „úzké“ znaky, tak proudy pro „široké“ znaky.

- Standardní objektové datové proudy leží spolu s ostatními součástmi standardní knihovny v prostoru jmen `std`.
- Součástí standardu nejsou proudy `strstream`, `istrstream` a `ostrstream`, orientované na pole znaků. Nahradily[3] je proudy `stringstream`, `istringstream` a `ostringstream`, využívající standardní třídu `string` (v níž nehrozí překročení mezí pole). V souvislosti s tím také odpadl hlavičkový soubor `strstream.h` a nahradil ho `sstream`.

Vedle objektů `cin`, `cout` atd. máme k dispozici nové knihovní objekty `wcin`, `wcout` atd. určené pro práci se „širokými“ znaky (v kódování UNICODE). Přibyly také nové manipulátory.

Práce s obrazovkou v textovém režimu

Tento oddíl obsahuje základní informace o funkcích pro práci s obrazovkou v textovém režimu, které jsou k dispozici pouze v borlandských překladačích. Jsou deklarovány v hlavičkovém souboru `<conio.h>`.

Tyto funkce umožňují specifikovat okno pro výstup textu, mazat ho, přemísťovat kurzor, měnit barvu textu a pozadí apod. Následující popis není úplný.

Funkce

```
void window(int Levy, int Horni, int Pravy, int Dolni)
```

definuje okno, do kterého bude zapisován výstup funkcí `putch()`, `cputs()` a `cprintf()`. Parametry udávají souřadnice levého horního a pravého dolního rohu okna. Přitom levý horní roh obrazovky má souřadnice (1,1) a pravý dolní (80, 25) nebo (40,25) podle zvoleného režimu.

Implicitní nastavení odpovídá zpravidla volání `window(1,1,80,25)`, tj. oknem je celá obrazovka. Parametry této funkce se vztahují vždy k celé obrazovce, na rozdíl od většiny dalších funkcí, kde jsou relativní k právě platné definici okna.

Chceme-li vymazat aktuální textové okno (přesněji vyplnit ho zadanou barvou pozadí), zavoláme funkci

```
void clrscr(void)
```

Textový kurzor (tj. místo, kde začne příští zápis na obrazovku) lze přemístit voláním funkce

```
void gotoxy(int x, int y)
```

Tyto souřadnice `x` a `y` jsou relativní vzhledem k aktuálnímu textovému oknu.

Chceme-li zjistit aktuální souřadnice kurzoru vzhledem k současnému oknu, použijeme funkce

```
int wherex(void),
int wherey(void)
```

Barvu vystupujícího textu a barvu pozadí nastavíme pomocí funkcí

```
void textcolor(int color)
void textbackground(int color)
```

Po nastavení barvy budou výstupy pomocí funkcí `putch()`, `cputs()` a `cscanf()` používat předepsanou barvu textu a pozadí.

barva	identifikátor	hodnota	barva	identifikátor	hodnota
černá	BLACK	0	tmavě šedá	DARKGRAY	8

[3] Některé překladače nabízejí proudy orientované na řetězce jako rozšíření.

modrá	BLUE	1	světle modrá	LIGHTBLUE	9
zelená	GREEN	2	světle zelená	LIGHTGREEN	10
modrozelená	CYAN	3	světle modrozelená	LIGHTCYAN	11
červená	RED	4	světle červená	LIGHTRED	12
purpurová	MAGENTA	5	světle purpurová	LIGHTMAGENTA	13
hnědá	BROWN	6	žlutá	YELLOW	14
světle šedá	LIGHTGRAY	7	bílá	WHITE	15
blikání	BLINK	128			

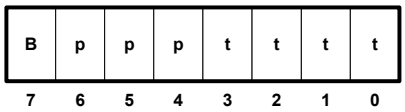
Tab. 1.22 Předdefinované konstanty pro označení barev

Pro barvy jsou definovány symbolické konstanty, které ukazuje tabulka 1.22. Pro pozadí můžeme používat pouze barvu 0 – 7. Přičteme-li k barvě textu konstantu `BLINK` (blikání), bude zobrazený text blikat.

Barvu textu i pozadí můžeme nastavit zároveň pomocí funkce

```
void textattr(int atr)
```

Tato funkce nastaví nové hodnoty atributů (barvu textu a pozadí, blikání). Nahrazuje tedy funkce `textcolor()` a `textbackground()`. Atributy musí být zakódovány v parametru `atr` po bitech tak, jak to ukazuje obr. 1.3.



Obr. 1.3 Atributy textu: tttt jsou 4 bity určující barvu textu, ppp jsou 3 bity určující barvu pozadí a B je bit určující, zda bude text blikat

Chceme-li vzít obsah textového okna na obrazovce a uložit ho do paměti nebo obsah takto uloženého okna opět zobrazit, použijeme funkce

```
void gettext(int L,int H, int P, int D, void *S)
void puttext(int L,int H, int P, int D, void *S)
void movetext(int L,int H, int P, int D, int L1,int H1)
```

(*L*, *H*), resp. (*P*, *D*) jsou souřadnice levého horního, resp. pravého dolního rohu okna na obrazovce (vzhledem k okrajům obrazovky, nikoli vzhledem k výstupnímu oknu). *S* je adresa prvního prvku pole, do kterého se obsah okna uloží. Přitom na každou znakovou pozici na obrazovce musíme počítat dva byty (znak + atributy).

Funkce `movetext()` přesune dané textové okno na novou pozici danou levým hormím rohem (L1, H1).

Funkce

```
void delline(void)
void insline(void)
```

odstraní, resp. vloží v textovém oknu řádku na místě, kde je kurzor.