Jazyk C++

Nová paměť podruhé

V květnovém čísle Chipu jsme se při povídání o operátorech new a delete seznámili především s pozadím jejich fungování a s některými novinkami, které v tomto ohledu přinesl standard ISO/ANSI jazyka C++. Dnes se podíváme především na problémy, na které může programátor při jejich použití narazit.

Jako vždy i při používání operátorů new a delete můžeme udělat chyby a "zadělat" si tak na slušnou porci problémů. Podívejme se teď na některé obzvláště pěkné. Následující příklady pocházejí nejen z programů začínajících céčkařů, ale bohužel i z knih – našich i zahraničních. Některé z nich dokonce nesly označení "učebnice"...

Kontrola výsledku

Operátor new nemusí uspět. Paměť počítače může být sice velká, ale je vždy konečná. Proto je třeba výsledek operátoru new kontrolovat. To znamená podle okolností buď testovat, zda je výsledek (vrácená adresa) různý od 0, nebo uzavřít alokační výraz do bloku try.

Nedávno jsem v jedné zahraniční knize našel tvrzení, že testovat výsledek operátoru new vlastně není nutné – díky mechanismu virtuální paměti prý dnes mají programy k dispozici tolik paměťového prostoru, že ho prakticky nelze vyčerpat. Nemohu se ubránit dojmu, že se tím řada programátorů opravdu řídí. Uvedené tvrzení vypadá věrohodně, neboť 4 GB jsou opravdu hodně, nebo alespoň nám to tak připadá. Nikde však není psáno, že náš program poběží vždy v prostředí s dostatečně velkým diskovým prostorem nebo že zároveň s ním nepoběží další programy konzumující obrovské množství paměti. Takže zmíněné tvrzení přece jen příliš rozumné není.

Ostatně výroky tohoto typu zastarávají velice rychle. Vzpomeňme jen, jak Bill Gates roku 1981 prohlašoval, že 640 KB operační paměti by mělo být dost pro každého...

Předefinování globálních operátorů

Na samotném předefinování globálních operátorů new a delete ve skutečnosti není nic špatného. Musíme ale mít stále na paměti, že náhrada standardních funkcí operator new(size_t) a dalších platí po celou dobu běhu programu, že začíná ještě před spuštěním funkce main() a trvá i po jejím ukončení. To znamená, že se uplatní i při vytváření globálních instancí knihovních tříd (např. proudů cin, cout atd.) a při jejich uvolňování.

Je tedy třeba takovou náhradu pečlivě uvážit, neboť může mít nepříjemné následky. Například pokusy s alokací paměti do "arény", vyhrazeného pole, mohou způsobit zhroucení programu, neboť se nemusí podařit alokovat dostatečné množství paměti pro objektové datové proudy.

Existují ovšem i subtilnější chyby, které může předefinování globálních operátorů new a delete způsobit. Podívejme se na příklad. Chceme – například kvůli ladění – zajistit, aby operátor new inicializoval přidělenou paměť určitou hodnotou,. aby například uložil do všech bitů hodnotu 1. Napíšeme tedy následující funkci:

```
#include <stdlib.h>
#include <memory.h>
#include <new>
void* operator new(size_t s) throw(std::bad_alloc)
{
void *p = malloc(s);
if(!p) throw std::bad_alloc();
memset(p,0xff,s);
return p;
}
```

Bude to v pořádku? Téměř. Tato funkce se chová podobně jako standardní operátor new, až na to, že nespolupracuje s funkcí set_new_handler(). Pokud by na to některá část programu spoléhala, vzniknou chyby, které se těžko hledají.

Ke svérázným problémům může vést použití některých standardních objektů v předefinovaných funkcích operator new() nebo operator delete(). Kdybychom například vytvořili funkci operator delete(), která bude kromě uvolňování paměti informovat o tom, že je volána, dejme tomu takto:

```
void operator delete(void*p)
{
  std::cout << "volá se operátor delete" << std::endl;
  free(p);
}
```

dočkali bychom se nejspíš nepříjemného překvapení. V některých překladačích by program po ukončení ohlásil nedefinovanou chybu, v některých by vznikl při použití operátoru delete nekonečný cyklus. Proč?

Standardní proudy si mohou při použití alokovat pomocnou paměť a k tomu využívají operátory new a delete. To ale znamená, že po vstupu do funkce operator delete() se použije operátor new, v zápětí pak operátor delete, který zavolá funkci operator delete(), ta použije opět new a delete atd. Program pak skončí vyčerpáním zásobníku.

Podobné problémy se mohou objevit také při použití objektových datových proudů ve funkci operator new(), která nahrazuje standardní verzi.

To znamená, že předefinování standardních operátorů se – pokud to jde – vyhneme. Nic nám totiž nebrání funkci operator new()přetěžovat, tj. definovat vlastní verze s dodatečnými parametry. Tyto přetížené verze použijeme jen tam, kde je opravdu potřebujeme, a pro standardní objekty ponecháme standardní new.

Dvojí volání konstruktoru

Následující chyba může vypadat neuvěřitelně, našel jsem ji však v jedné německé knize, která se tvářila jako referenční příručka jazyka C++. Autor předváděl operátor new definovaný jako metodu takto:

```
class {
  public:
  X();
  void* operator new(size_t s);
  };
void* X::operator new(size_t s)
{
  X* x = ::new X;
  // Nějaká úprava vytvořené instance
  return x;
}
```

Zde autor v operátoru new nejprve vytvoří pomocí globálního operátoru novou instanci, nějak ji upraví a ukazatel na ni vrátí. Vypadá to docela dobře, ale je tu nejméně jeden problém: Pro tuto instanci se bude dvakrát volat konstruktor, a to může mít podobně zhoubné následky, jako když se konstruktor vůbec nezavolá. Jestliže totiž někde v programu napíšeme např.

```
X* ux = new X;
```

proběhnou obvyklé operace – nejprve se zavolá metoda X::operator new(), která by měla vyhradit paměť. Ta ji opravdu vyhradí, ovšem použije k tomu globální operátor new, a ten pro tuto paměť ihned zavolá konstruktor třídy X. Pak X::operator new()ukazatel na vytvořenou instanci vrátí. Po návratu pro ni zavolá operátor new znovu konstruktor. Kdyby konstruktor třídy X například alokoval dynamickou paměť, otevíral soubory apod., mohou nastat problémy.

Pokud by programátor chtěl podobným způsobem postupovat, měl by v metodě X::operator new() použít zápis operátorové funkce:

```
void* X::operator new(size_t s)
{
  X* x = ::operator new(s);
  // Nějaká úprava alokované paměti
  return x;
}
```

Takto definovaný operátor new však vlastně nahrazuje konstruktor, a to je zbytečné. Pokud nám tedy nejde o nějakou “preventivní” inicializaci, která má třeba usnadnit hledání chyb, je lepší ponechat inicializaci konstruktoru – to je přece jeho vlastní úloha.

Zděděné delete

Deklarujeme-li funkce operator new() a operator delete() jako metody, budou statické, i když klíčové slovo static neuvedeme. To znamená, že nemohou být virtuální – a to může občas vést k

problémům. Podívejme se na příklad:

```
int a[1000];
class X
{
public:
  void *operator new(size_t s){
        cout << "new X" << endl;
        return a;
  }
  void operator delete(void* p) {
        cout << "delete X" << endl;
  }
};
class Y: public X
{
public:
  void *operator new(size_t s){
        cout << "new Y" << endl;
        return a;
  }
  void operator delete(void* p){
        cout << "delete Y" << endl;
  }
};
```

Zde jsme deklarovali třídu Y jako potomka třídy X. Jak předek, tak potomek obsahují vlastní verze operátorů new a delete. (Jejich implementace zde má především za úkol vypsat upozornění – na něm bude totiž nejsnáze vidět, oč jde.)

Při konstrukci nové instance většinou problémy nenastanou. Napíšeme-li v programu

```
X* ux = new Y;
```

zavolá se metoda Y::operator new(), jak očekáváme, a vypíše řetězec new Y. Jestliže ale napíšeme

```
delete ux;
```

zavolá se metoda předka, X::operator delete(), která vypíše delete X – a to je špatně (jinak bychom nemuseli definovat v potomkovi novou verzi této funkce).

Řešení je ovšem jednoduché: Stačí v předkovi, ve třídě X, definovat virtuální destruktor. Přidáme-li tedy do deklarace třídy X řádek

```
virtual ~X(){}
```

bude vše v pořádku; příkazem

```
delete ux;
```

zavoláme totiž opravdu operátor delete pro třídu Y.

Alokace vícerozměrného pole

O této chybě jsem v Chipu už kdysi psal. V začátečnických programech se však objevuje s úpornou pravidelností, a proto prokládám za účelné se k ní vrátit.

Podívejme se na následující příklad:

```
int** m = (int**)new int[2][3];         // !!!
```

Problém je, že pokud něco takového napíšete, v některých prostředích – např. ve stále ještě žijícím operačním systému DOS – může váš program dlouhou dobu běžet, aniž by se cokoli špatného dělo. Pak se ovšem zhroutí, neboť si přepíše část paměti – data, kód programu, část operačního systému, podle toho, co může napáchat větší škody.

Dokonce i v prostředích s ochranou paměti – například pod Win32 – může tato konstrukce za jistých okolností chvíli fungovat, pak ovšem skončí výjimkou, porušením ochrany paměti.

Jak to tedy má vypadat? Pokud chceme alokovat pole, musíme použít ukazatel na první prvek. Dvourozměrné pole se skládá z jednorozměrných polí, takže potřebujeme ukazatel na pole, nikoli ukazatel na ukazatel. Přesněji, pole vytvořené výrazem new int[2][3] je pole o dvou prvcích složené z polí o třech prvcích typu int. Potřebujeme ukazatel na jeho první prvek, tedy ukazatel na pole o třech prvcích typu int:

```
int (*mat)[3] = new int[2][3];          // OK
```

S takto alokovaným polem lze zacházet jako s “normálním” polem, můžeme např. napsat

```
for(int i = 0; i < 2; i++)
```

```
  for(int j = 0; j < 3; j++)
        mat[i][j] = 10*i+j;
```

Zmíněná chyba nesporně pochází z oblíbeného tvrzení mnoha autorů učebnic jazyků C a C++, že pole a ukazatele jsou v těchto jazycích jedno a totéž. (Nevím, jak může někdo něco takového vůbec napsat, nicméně nejde o nijak vzácné tvrzení.) Odtud je již jen krok k představě, že tedy dvourozměrné pole je totéž co ukazatel na ukazatel. Navíc překladač tuto chybu nezachytí, neboť ukazatel na ukazatel opravdu lze dvakrát indexovat – význam je ovšem poněkud jiný než dvakrát indexovaný identifikátor pole nebo ukazatel na pole.

Je-li M ukazatel na ukazatel na int, očekává překladač, že jde o ukazatel na první prvek pole typu int a dovolí nám ho indexovat. Podobně je-li m ukazatel na ukazatel na int, očekává překladač, že jde o první prvek pole složeného z ukazatelů na int. Pak m[i] bude znamenat i-tý prvek tohoto pole, tedy ukazatel na int, a tedy ukazatel na první prvek pole typu int. Nakonec m[i][j] je prvek v poli, na které tento ukazatel ukazuje. Názorněji je to vidět na obrázku 1.

Na druhé straně je-li mat ukazatel na jednorozměrné pole, očekává překladač, že jde o první prvek pole složeného z polí, mat[i] je i-tý prvek tohoto pole a mat[i][j] je j-tý prvek i-tého prvku (obr. 2).

Podrobnější rozbor najdete v článku Když se céčkaři s plusy nedaří (4) v Chipu 11/95 nebo v mé knize Pasti a propasti jazyka C++ (Grada 1997, ISBN 80-7169-607-2).

Ve skutečnosti zde narážíme ještě na jeden problém: Proč je v zápisu označeném třemi vykřičníky přetypování? Protože překladač odmítl tento příkaz přeložit s odůvodněním, že nedokáže konvertovat ukazatel na pole na ukazatel na ukazatel. Už to mělo programátora varovat, že je něco v nepořádku – operátor new vrací vždy ukazatel na typ, jaký si autor poručil. Zde ovšem programátor ignoroval upozornění a prosadil svou, aniž o věci přemýšlel.

Pole objektů

Podívejme se na následující deklaraci třídy Z:

```
class Z
{
public:
  void* operator new(size_t s);
  Z();
  // ... a další složky
};
```

Tato třída obsahuje operátor new pro alokaci jednoduchých proměnných, nikoli pro alokaci pole. To znamená, že napíšeme-li

```
Z* uz = new Z;
Z* upz = new Z[10];
```

použije se v prvním případě pro alokaci paměti metoda Z::operator new(), avšak ve druhém případě se použije globální funkce operator new[](). Pokud chceme řídit také alokaci polí třídy Z, musíme doplnit odpovídající metodu. Obvykle stačí, když se “polní” alokační funkce odvolá na “obyčejnou”:

```
void* Z::operator new[](unsigned s)
{
  return operator new(s);
}
```

Poznamenejme, že takto je zpravidla implementována i standardní globální funkce operator new[]().

Při implementaci “obyčejné” alokační funkce, tj. metody operator new(size_t s), musíme počítat s tím, že bude volána i s hodnotou parametru s, která není rovna velikosti instance třídy Z. V případě alokace pole o N prvcích může mít parametr s obecně hodnotu N*sizeof(Z)+k, kde k představuje jakousi režii (třeba místo, do kterého si program uloží počet prvků pole pro pozdější orientaci, například při volání destruktorů).

New má mít své delete

Podívejme se znovu na třídu Z z předchozího odstavce. Jestliže alokujeme instanci příkazem

```
Z* uz = new Z;
```

a pak ji uvolníme příkazem

```
delete uz;
```

použije se k alokaci metoda Z::operator new(), avšak k uvolnění globální funkce operator

delete(). To je nejspíš chyba: Pokud operátor new používá při alokaci nějaký zvláštní postup, například přiděluje paměť ve zvláštní haldě, je nezbytné paměť stejným způsobem i uvolňovat, tedy definovat také metodu operator delete(). (Totéž platí i pro "polní" verze těchto operátorů.)

Ještě jednou pole objektů

Občas také zapomeneme, že při uvolňování pole je třeba použít operátor delete[], nikoli jen delete. Pokud pracujeme s neobjektovými poli, většinou to projde. V případě polí objektových typů je situace horší, liší se však překladač od překladače. Je-li X třída a napíšeme-li

```
X* ux = new X[N];
delete ux; // Má být delete[] ux;
```

obvykle se nezavolá správný destruktor pro všechny instance. Může však dojít i k porušení ochrany paměti.

Zápis typu

Operátor new má nižší prioritu než například operátor volání funkce. Proto může překladač odmítnout některá komplikovanější označení typu za klíčovým slovem new. Jestliže chceme alokovat dynamickou proměnnou typu "ukazatel na funkci typu void bez parametrů" a napíšeme

```
void (** v)() = new void (*)();
```

ohlásí překladač nejspíš řadu podivných chyb.

Tato situace má několik řešení. Stačí třeba označení typu uzávorkovat:

```
void f(void);
void (** v)(void) = new (void (*)())(f);
(**v)();                // Volání funkce f()
```

Zde jsme nově vytvořené proměnné přiřadili jako počáteční hodnotu adresu funkce f() a vzápětí jsme tuto funkci zavolali.

Asi nejpřehlednější je pojmenovat požadovaný typ pomocí deklarace typedef, například

```
typedef void (*funkce)(void);
```

a pak nově zavedené použít v alokačním výrazu:

```
funkce* u = new funkce(f);
```

Třída je obor viditelnosti

Následující příklad skončí chybou při překladu, méně zkušení programátoři pak ovšem obviňují překladač, že obsahuje chybu (to jsem si kdysi myslel i já).

```
Class X
{
  public:
  void* operator new(size_t s, int a);
  // ... a další složky
};

X* ux = new X;    // Chyba
```

Třída X obsahuje operátor new deklarovaný jako metodu s jedním dodatečným parametrem, nicméně v následujícím příkazu používáme operátor new bez dodatečných parametrů. I když se zdá, že by překladač měl podle počtu a typu parametrů zjistit, že chceme použít globální operátor new, nepozná to a ohlásí, že ve třídě X operátor new s požadovanými parametry neexistuje. Důvod je zřejmý: třída je totiž také "oblast viditelnosti" a v ní je globální operátor new zastíněn lokální definicí. Pokud chceme použít globální operátor new, musíme si o něj explicitně říci pomocí rozlišovacího operátoru ::, pak bude vše v pořádku:

```
X* ux = ::new X;  // OK
```

Konstruktory, destruktory a skalární typy

V obou dílech povídání o operátorech new a delete jsme stále hovořili o konstruktorech a destruktorech, jako kdybychom nealokovali nic jiného než instance objektových typů. Ve skutečnosti lze vše, co jsme řekli, přenést i na skalární datové typy. Standardní C++ totiž dovoluje i pro tyto typy používat zápisy jako int() nebo a.~int(), kde a je proměnná typu int ("konstruktor" nebo "destruktor" typu int). Tento "konstruktor" inicializuje zpravidla hodnotou 0, "destruktor" skalárního typu nedělá nic. Proto můžeme také s klidem hovořit o inicializaci dynamicky alokované skalární proměnné

pomocí konstruktoru.

I když to vypadá podivně, má uvedené pravidlo dobrý důvod: Umožňuje používat naprosto stejným způsobem objektové typy a skalární typy v šablonách a v některých dalších situacích.

Ještě není konec...

Operátory new a delete nejsou jediné nástroje pro alokaci paměti v C++. Vedle funkcí malloc(), calloc()a free(), zděděných po jazyku C, přinesl standard jazyka i tzv. alokátory. To jsou třídy, které zapouzdřují nástroje pro alokaci paměti a které se hojně využívají především ve standardní šablonové knihovně C++. O nich si povíme někdy jindy v samostatném článku.

Miroslav Virius