Prostor pro jména

Myšlenka prostorů jmen (namespace) je v C++ poměrně stará; ovšem v průběhu standardizace tohoto jazyka prošla několika proměnami.

Prostory jmen se objevily již v neoficiálním standardu jazyka C++
[1] a byly poměrně jednoduché. Přesto je komerční překladače tohoto jazyka implementovaly dost pozdě, až v polovině devadesátých let. Příčinou byly zřejmě mimo jiné nejasnosti kolem jedné zdánlivé drobnosti, kterou standard přidal - vyhledávání jmen volaných funkcí v závislosti na parametrech (tzv. Koenigova vyhledávání). Podívejme se tedy, jak to s prostory jmen v C++ je.

Příliš mnoho identifikátorů

K čemu jsou prostory jmen dobré? Začneme zeširoka, od počítačové prehistorie.

Problémy se jmény

Identifikátory prvních verzí jazyka FORTRAN - tehdy se psal se všemi písmeny velkými - mohly mít nejvýše 6 znaků; k dispozici bylo 26 písmen anglické abecedy a číslice 0 - 9. (Malá a velká písmena se nerozlišovala, většina počítačů znala pouze velká písmena.) Identifikátor musel začínat, jak je dodnes obvyklé, písmenem. Teoreticky měl tedy programátor k dispozici 26 jednoznakových identifikátorů, 26 x 36 dvouznakových atd. až 26 x 365 šestiznakových identifikátorů. To je více než 16 miliard různých jmen, a to se autorům jazyka FORTRAN zdálo dostatečné.

Nicméně brzy se ukázalo, že identifikátory jako X235B1 nevedou k nejpřehlednějším programům a pro smysluplná jména je 6 znaků obvykle málo. Proto pozdější jazyky - a hlavně jejich překladače umožnily používat podstatně delší identifikátory.

S růstem rozsahu projektů a také s růstem rozsahu programových knihoven se však brzy ukázalo, že "smysluplné identifikátory" snadno působí problémy jiného druhu - dochází ke konfliktům jmen. Poměrně často se stane, že různí členové vývojového týmu přidělí různým proměnným nebo funkcím stejný identifikátor, nebo že pro ně použijí identifikátor, který zároveň označuje funkci, typ nebo proměnnou z některé z programových knihoven. Stejně dobře se může stát, že v jednom projektu použijeme dvě různé knihovny od dvou různých dodavatelů, které shodou okolností používají týž identifikátor pro dvě různé věci - a problém je na světě.

Řešení v C

Někteří programátoři v jazyce C prý používali poměrně jednoduchý trik: své globální proměnné deklarovali jako složky struktury. Například takto:

```
struct { // Globální struktura
int proud1;
// a další proměnné
} Vstup;
// ...
int f() {
if(Vstup.proud1) ZpracujTo();
// ... a další příkazy
}
```

Můžeme si představovat, že Vstup je označení části programu, který vyvíjí jeden člen týmu. To může docela dobře fungovat, pokud se členové tohoto týmu dohodnou, jak se budou jejich struktury s proměnnými jmenovat.

Toto řešení ovšem funguje pouze pro identifikátory proměnných, nikoli pro identifikátory funkcí, neboť ty nemohou být v jazyce C součástí struktury.

V C++ bychom mohli do struktur ukrýt i funkce a datové typy. Tento jazyk nám ale nabízí elegantnější řešení - prostory jmen. Na prostor jmen se můžeme dívat jako na příjmení, které připojíme k identifikátoru a tím zajistíme jeho jednoznačnost v rámci programu.

Deklarace prostoru jmen

Syntaxe deklarace prostoru jmen je jednoduchá. Deklarace začíná klíčovým slovem namespace, za nímž následuje identifikátor prostoru jmen. Pak následují ve složených závorkách deklarace složek prostoru jmen. Například takto:

```
// Deklarace prostoru jmen
namespace vstup {
int proud1;
void f();
class X;
}
```

Zde jsme v prostoru jmen vstup deklarovali globální proměnnou, funkci a třídu. Plné jméno (tzv. kvalifikované jméno) této proměnné je vstup::proud1, plné jméno funkce je vstup::f(), plné jméno třídy je vstup::X. (Říkáme, že identifikátory složek prostoru jmen kvalifikujeme jménem prostoru jmen; k tomu používáme operátor ::.)

Uvnitř prostoru jmen můžeme deklarovat datové typy, funkce, proměnné, ale také další prostory jmen.

Definice složek

Funkci vstup::f() stejně jako třídu X můžeme definovat někde dále; v tom případě musíme jejich identifikátory kvalifikovat jménem prostoru jmen.

```
void vstup::f(){
// ...
}
class vstup::X{ /* ... */ };
```

Nic nám ovšem nebrání zapsat jejich definice celé do prostoru jmen:

```
namespace vstup {
int proud1;
void f() { /* ... */ }
class X { /* ... */ };
}
```

Mimo prostor jmen vstup můžeme v témže programu deklarovat funkci s prototypem void f(); nebo třídu X niž by došlo ke konfliktu.

Použití složek prostoru jmen

Složky prostoru jmen můžeme mimo "jejich" prostor jmen používat, pokud je kvalifikujeme jménem prostoru jmen (nebo pokud je nezpřístupníme pomocí deklarace či direktivy using, o nichž si povíme dále).

To znamená, že ve funkci g(), která neleží v prostoru jmen Vstup, můžeme napsat

```
void g() {
// Voláme funkci f() z prostoru jmen vstup vstup::f();
// Voláme funkci f(), která neleží v prostoru
// jmen vstup f();
vstup::proud1 = 6589;
// Deklarujeme instanci třídy X
vstup::X x;
}
```

Uvnitř prostoru jmen můžeme jeho složky používat bez kvalifikace. To znamená, že v těle funkce vstup::f() může vypadat např. takto: void vstup::f() { // použije vstup::proud1 scanf("%d", &proud1);
X x; }

Přitom je jedno, zda zapíšeme definici funkce f() uvnitř prostoru jmen, nebo zda ji v prostoru jmen vstup pouze deklarujeme a definici zapíšeme někde jinde. Tělo funkce, deklarované v prostoru jmen, je vždy jeho součástí.

Vnořené prostory jmen

Uvnitř prostoru jmen můžeme deklarovat další prostor jmen. Například takto:

```
namespace vnejsi{
namespace vnitrni{
void h(){/* ... */}
}void h(){
vnitrni::h();
}
}
```

Jméno vnořeného prostoru jmen se skládá z identifikátoru tohoto prostoru, ke kterému připojíme operátorem :: jméno vnějšího prostoru jmen. Vnořený prostor jmen v předchozím příkladu se tedy jmenuje vnejsi::vnitrni.

Ve vnějším prostoru jmen stačí jména objektů, deklarovaných ve vnořeném prostoru jmen, kvalifikovat pouze jménem vnitřního prostoru jmen. Například funkci h(), deklarovanou v prostoru jmen vnejsi::vnitrni, můžeme ve funkci vnejsi::h() volat zápisem vnitrni::h().

Hloubka vnořování prostorů jmen není omezena.

Proč tolik psát

Zavedení prostorů jmen znamená mnoho psaní navíc a programátoři jsou, jak známo, líní - ostatně jako většina lidí. Proto nabízí C++ několik možností, jak si ušetřit práci.

Alias prostoru jmen

Má-li prostor jmen dlouhé jméno, může být jeho opakované vypisování nepohodlné. Proto nabízí C++ možnost prostor jmen přejmenovat - definovat pro něj alias. Například prostor jmen vnejsi::vnitrni přejmenujeme na vv deklarací namespace vv = vnejsi::vnitrni;

Potom budou zápisy vnejsi::vnitrni::h() a vv::h() ekvivalentní.

Poznamenejme, že alias můžeme definovat pro jakýkoli prostor jmen, nejen pro vnořený. Alias umožňuje také pracovat s abstraktním prostorem jmen. Jedinou deklarací aliasu určíme, o jaký prostor jmen jde. (Podobně např. deklarace typedef umožňuje pracovat s abstraktním datovým typem.)

Direktiva using

Používáme-li často jména z některého prostoru jmen, můžeme překladači říci, že budeme kvalifikaci jménem prostoru jmen vynechávat. K tomu slouží tzv. direktiva using, jež má tvar using namespace jméno;

Za touto direktivou můžeme používat identifikátor z prostoru jmen jméno bez kvalifikace.

Například všechny identifikátory ze standardní knihovny jazyka C++ leží v prostoru jmen std. To znamená, že napíšeme-li

```
#include <iostream>
#include <list>
using namespace std;
```

můžeme psát

```
list<int> l;
for(int i = 0; i < 10; i++) l.push_front(i);
for(list<int>::iterator i = l.begin();
i != l.end(); i++)
{
cout << *i << endl;
}
```

Kdybychom vynechali direktivu using, museli bychom psát
std::list<int>, std::list<int>::iterator, std::cout a std::endl.

Deklarace using

Některé prostory jmen jsou značně rozsáhlé a my z nich potřebujeme jen některá jména; v takovém případě může direktiva using napáchat více škody než užitku. Proto nabízí jazyk C++ ještě deklaraci using, jež umožňuje specifikovat jednotlivá jména z prostoru jmen, která chceme používat bez kvalifikace. Pokud bychom chtěli např. používat bez kvalifikace jen identifikátory z konce předchozího odstavce, mohli bychom napsat

```
using std::list; // list je jméno šablony
using std::list<int>::iterator;
using std::cout;
using std::endl;
```

Podobně budeme-li chtít používat bez kvalifikace jméno funkce

Vstup::f(), napíšeme using vstup::f; // Jen jméno

V deklaraci using uvádíme vždy jen jeden identifikátor. Deklarace using std::list; umožňuje používat bez kvalifikace šablonu list s jakýmikoli parametry.

using je tranzitivní

Jednou ze zajímavých vlastností direktivy i deklarace using je, že jsou tranzitivní. To znamená: Uvedeme-li v prostoru jmen B direktivu using namespace A; a v prostoru jmen C direktivu using namespace B; můžeme v prostoru jmen C používat bez kvalifikace všechna jména z prostoru jmen A. Podobně zpřístupníme-li v prostoru jmen B nějaká jména z prostoru jmen A deklarací using a uvedeme-li v prostoru jmen C direktivu using namespace B;budou v prostoru jmen C tato jména přístupná.
Pokud vám to připadá nesrozumitelné, příklad to jistě vyjasní.

```
#include <iostream>
#include <list>

namespace pomocny {
using namespace std;
// ... další deklarace
}
namespace hlavni {
using namespace pomocny;
int Fufu()
{ // To je OK list<int> l; // tato jména cout << "ahoj" // netřeba kvalifikovat << endl;
// ...
}
}
```

V prostoru jmen pomocny jsme mj. uvedli direktivu using namespace std;, která v něm zpřístupnila všechny identifikátory z prostoru jmen std. V prostoru jmen hlavni jsme si zpřístupnili všechna jména z prostoru jmen pomocny direktivou using namespace pomocny;, a proto je můžeme v těle funkce hlavni::Fufu() použít bez kvalifikace. Kdybychom změnili deklaraci prostoru jmen pomocny následujícím způsobem,

```
namespace pomocny {
using std::cout;
using std::endl;
using std::list;
using std::list<int>::iterator;
// ... a další deklarace
}
```

mohli bychom v prostoru jmen hlavni používat bez kvalifikace jen uvedená jména.

Jména vnesená do nějakého prostoru jmen pomocí deklarace nebo direktivy using se chovají, jako kdyby byla jeho součástí. To znamená, že platí-li výše uvedená deklarace prostoru jmen pomocny, můžeme např. ve funkci main(), jež leží mimo jakýkoli prostor jmen, napsat Pomocny::list<double> dl;a bude to znamenat totéž, jako kdybychom napsali std::list<double> dl;

Deklarace po částech

Další zajímavou a užitečnou vlastností prostorů jmen v jazyce C++ je, že je můžeme deklarovat po částech. Napíšeme-li někde v programu

```
namespace jupi {
void fupi();
}
```

nic nám nebrání napsat někde jinde

```
namespace jupi {
void gupi();
}
```

a překladač si obě části prostoru jmen jupi spojí do jednoho celku. Přitom tyto části nemusí ležet v témže souboru. Překladač ovšem může vždy pracovat jen se jmény, která už byla deklarována, nedokáže se "podívat dopředu". To znamená, že kdybychom napsali

```
namespace jupi {
void fupi(){/* ... */}
}
void dupy(){
jupi::gupi();// Nelze
}
namespace jupi {
void gupi(){/* ... */}
}
```

ohlásil by překladač, že používáme funkci gupi(), která není součástí prostoru jmen jupi.

Hlavičkové soubory

Příslušnost k prostoru jmen je třeba pochopitelně vyznačit i v hlavičkových souborech. V nich ovšem zapisujeme pouze deklarace, nikoli definice - jak je v C++ obvyklé. Deklarujeme-li hlavičkový soubor

```
// Souboru jupi.h
namespace jupi {
void fupi();
void gupi();
}
```

a vložíme-li tento soubor direktivou #include "jupi.g" přijme překladač příklad z předchozího oddílu bez námitek.

Knihovny

Skutečnost, že deklaraci prostoru jmen lze rozdělit na několik částí, umožňuje rozdělit velké prostory jmen do několika souborů. Typickým příkladem je standardní knihovna jazyka C++, která leží, jak víme, v prostoru jmen std. Tato knihovna je popsána v řadě hlavičkových souborů. Skutečnost, že překladač vidí jen ta jména, o nichž se dozví z deklarací, které si přečte, umožňuje pracovat s menší množinou jmen, nikoli s celým prostorem jmen najednou. Napíšeme-li ve svém programu #include <iostream> bude překladač znát jen jména deklarovaná v tomto hlavičkovém souboru iostream, nikoli však jména deklarovaná v hlavičkových souborech list, queue, map a dalších. (Pozor ovšem na součásti zpřístupněné díky tranzitivitě direktiv a deklarací using. Jazyk C++ bohužel nespecifikuje vzájemné závislosti hlavičkových souborů.)

Poznamenejme, že v tomto ohledu se výrazně liší pojetí prostorů jmen v C++ od pojetí prostorů jmen v jazyce C#.

Anonymní prostory jmen

V deklaraci prostoru jmen nemusíme uvést jeho jméno; pak dostaneme tzv. anonymní (nepojmenovaný) prostor jmen. Můžeme např. napsat

```
namespace {
void huhu() {/* ... */ }
int x;
}
```

Na identifikátory deklarované v anonymním prostoru jmen se můžeme odvolávat prostřednictvím samotného identifikátoru, případně identifikátoru, před který připojíme unární operátor :: - ale pouze v rámci samostatně překládané části programu, v němž je tento prostor jmen deklarován.

Překladač totiž spojí všechny anonymní prostory jmen v jednom samostatně překládaném modulu v jeden prostor jmen a tomu přidějí jakýsi vnitřní identifikátor. V důsledku toho identifikátory deklarované v tomto prostoru jmen nejsou vidět mimo daný modul. Proměnné a funkce deklarované v anonymním prostoru jmen se tedy vlastně chovají jako statické (deklarované s modifikátorem static).

Vyhledávání podle parametrů

Podívejme se na jednu z mnoha variant proslulého programu "Hello, world":

```
// Hello, world - po kolikáté už
include <iostream>

using std::cout;
using std::endl;

int main()
{
cout << "Ahoj, lidi" << endl;
return 0;
}
```

O objektech cout a endl jsme překladači řekli, že patří do prostoru jmen std. Neuvedli jsme ale deklaraci using std::operator<<; která by zpřístupnila přetížené operátory <<. Přesto překladače, které odpovídají současnému standardu jazyka C++, tento program bez problémů přeloží - operátory << najdou a použijí. Za to vděčíme pravidlu, které říká, že operátory a funkce se vyhledávají nejen v kontextu jejich použití (tj. v prostoru jmen, v němž jsou volány), ale i v prostorech jmen svých operandů, resp. parametrů. (Toto pravidlo se obvykle označuje jako Koenigovo vyhledávání.)

Operátory << v tomto příkladu jsou sice použity mimo jakýkoli prostor jmen, ale jejich operandy - cout a endl - leží v prostoru jmen std, a proto je bude překladač hledat i tam, a tam je také najde.

Podívejme se ještě na jeden příklad: namespace prvni
```
{
class X{};
void f(X x){}
}
int main()
{
Prvni::X xx;
f(xx); // Ok
return 0;
}
```

Také zde použije překladač vyhledávání závislé na parametrech. Funkci f()najde bez problémů, i když jsme její jméno nekvalifikovali jménem prostoru jmen prvni, neboť ji bude hledat nejen mimo prostory jmen, ale i v prostoru jmen prvni, v němž je deklarován typ X skutečného parametru.

K významu Koenigova vyhledávání se vrátíme příště.

C++ a knihovny jazyka C

Jazyk C++ převzal standardní knihovny jazyka C, a tedy také hlavičkové soubory, v nichž jsou makra, funkce, typy a konstanty z této knihovny deklarovány. Názvy těchto souborů se ovšem v C++ změnily. Odpadla přípona .h a před jméno se připojil znak c. To znamená, že např. hlavičkový soubor, známý v jazyce C pod názvem stdio.h, se v C++ jmenuje cstdio.

Chceme-li některou z konstrukcí z knihovny jazyka C použít v C++, máme dvě možnosti:

Můžeme použít jméno hlavičkového souboru podle pravidel jazyka C++. V tom případě budou identifikátory z něj ležet v prostoru jmen std.

Můžeme použít hlavičkový soubor z jazyka C tak, jak jsme byli v C zvyklí. V tom případě budeme identifikátory z tohoto hlavičkového souboru používat bez kvalifikace. (Tuto možnost standard připouští, ale pokládá ji za zastaralou.)

To znamená, že napíšeme-li #include <cstdio> musíme psát std::printf("Ahoj, lidi"); nebo použít direktivu či deklaraci using. Na druhé straně napíšeme-li #include <stdio.h> můžeme napsat printf("Ahoj, lidi");

Hlavičkové soubory jazyka C++ podle standardu nemají příponu .h. Nicméně ve starších verzích jazyka, které neobsahovaly prostory jmen, tyto příponu měly, a proto řada překladačů pro ně používá podobnou konvenci jako pro hlavičkové soubory z jazyka C: Uvedeme-li v jejich jménu příponu .h, nemusíme jména z nich kvalifikovat jménem prostoru jmen std.

Implementace

V úvodu jsme si řekli, že prostory jmen jsou jedním z posledních velkých rysů jazyka C++, které překladače implementovaly. Doplňme, že mnohé s nimi mají problémy dodnes. Například Visual C++ .NET implementuje Koenigovo vyhledávání pouze pro operátory volané operátorovým zápisem. Pro funkce, a dokonce ani pro operátory volané funkčním zápisem, je nepoužívá.

Příště

Tolik o prostorech jmen a o Koenigově vyhledávání. Příští pokračování, v němž se podíváme na důsledky této konstrukce pro práci s třídami v C++, najdete nikoli v tištěném časopise, ale na redakčním Chip CD. Pro mnohé z vás to bude mít výhodu snadnější práce s textem a s výpisy programů.

Miroslav Virius, autor@chip.cz

Odkazy: B. Stroustrup, M. A. Ellis: The Annotated C++ Reference Manual. Addison-Wesley 1991. International Standard ISO/IEC 14882-1998. Programming Languages - C++.