

Oddělení rozhraní a implementace

PV173 Programování v C++11

Vladimír Štill, Jiří Weiser

Fakulta Informatiky, Masarykova Univerzita

20. října 2014

Bývá dobrým zvykem mít v .h jen rozhraní a implementaci mít v .cpp

- rychlejší kompilace
- .cpp nemusí být dodáváno ve zdrojové formě (může to být zkompilovaná knihovna)

Ale:

- šablonované funkce/metody musí být celé v .h, protože kompilátor potřebuje vidět tělo funkce
- šablonované třídy musí být celé v .h
- kompilátor nemůže tak dobře optimalizovat
- změna privátního rozhraní pořád vyvolá rekompilaci

Celkově, modularita v C++ je jeden z velkých problémů jazyka.

Představení problému

Co chceme?

- implementovat funkcionalitu s dobře definovaným rozhraním
- skrýt vše, co je možné
- nemuset překompilovávat soubory používající naši třídu, pokud se změní něco v implementaci
- nemuset includovat tolik věcí do .h

Nápad I

Definujeme v `.h` souboru pouze hlavičky veřejných funkcí a atributy. Vše ostatní definujeme v `.cpp`.

- + relativně jednoduchý koncept
- ± hodí se pro třídy s málo atributy
- pokud budeme potřebovat pomocné funkce, nemohou to být metody třídy → musíme si nějak předat data, s nimiž budeme pracovat
- při přidání/odebrání privátních atributů musíme měnit `.h`
- typy všech atributů musí být přítomné v `.h` → neřeší `include`

Problém s includey

Příklad

```
#include <sys/types.h>
#include <sys/socket.h>

struct Socket {
    Socket( std::string address );
    /* ... */
private:
    int fd;
    struct sockaddr *addr;
};
```

Problém s includey

Co je problém?

- chceme rozhraní, které skryje implementační detaily jako typ `struct sockaddr`
- zároveň ale musíme znát tento typ v `.h`
- tím dostáváme do `.h` systémové include obsahující makra
- makra jsou problém – nerespektují namespace a může docházet ke konfliktům s jinými makry nebo proměnnými/funkcemi
- proto se v C++ snažíme makrům vyhnout

Nápad II

socket.h:

```
struct SocketData;  
struct Socket : SocketData {  
    Socket( std::string address );  
    /* ... */  
};
```

socket.cpp:

```
#include <sys/types.h>  
#include <sys/socket.h>  
struct SocketData {  
    protected:  
        int fd;  
        struct sockaddr *addr;  
};  
Socket::Socket(std::string addr) { /* ... */ }  
/* ... */
```

Nápad II

ehm,

```
socket.h:2:17: error: base class has incomplete
      type
```

```
struct Socket : SocketData {
                ^~~~~~
```

```
socket.h:1:8: note: forward declaration of
      'SocketData'
```

```
struct SocketData;
      ^
```

Tudy cesta nevede. SocketData je nekompletní typ v .h, a proto jej nelze použít jako předka pro Socket.

Nápad III

socket.h:

```
struct Socket {  
    Socket( std::string address );  
    /* ... */  
private:  
    struct Data;  
    Data *_data;  
};
```

socket.cpp:

```
#include <sys/types.h>  
#include <sys/socket.h>  
struct Socket::Data {  
    int fd;  
    struct sockaddr *addr;  
};  
/* ... */
```

Nápad III

Ano, tohle je řešení, označované jako PImpl („Pointer to implementation“) idiom, tak jak se používá v C++98.

- pointer na nekompletní typ je sám o sobě považován za typ kompletní (označujeme někdy jako *opaque pointer*)
- musíme definovat destruktory `~Socket` (v `.cpp`)

PImpl idiom

Princip

- všechna privátní data a metody jsou umístěny v implementačním objektu (impl)
- třída si drží ukazatel na impl
- pokud metody impl potřebují volat metody třídy, dostanou na ni referenci v parametru
- protected data jsou součástí rozhraní a tedy v .h
- musíme napsat kopírovací konstruktor!

V C++11

- použijeme `std::unique_ptr` namísto běžného pointeru
- destruktory musíme definovat v .cpp, protože musí mít v okamžiku definice k dispozici typ impl (vyžaduje `std::unique_ptr` aby se dokázal destruovat)

PImpl idiom

Schéma, class.h

```
struct A {  
    A();  
    A( const A& );  
    A( A && );  
    ~A();  
  
    A &operator=( A );  
  
    /* další veřejné rozhraní */  
private:  
    struct Impl;  
    std::unique_ptr< Impl > _imp;  
};
```

PImpl idiom

Schéma, class.cpp

```
struct A::Impl {  
    /* privátní metody a data */  
};  
  
A::A() : _imp( new Impl() ) { /* ... */ }  
// musí být v .cpp kvůli dtor unique_ptr  
A::~~A() = default;  
// musí být explicitní - _impl není kopírovatelný  
A::A( const A &o ) : _imp( new Impl(*o._imp) ) {}  
A::A( A && ) = default;  
  
A &operator=( A o ) {  
    _impl = std::move( o._impl );  
} /* ... */
```

PImpl idiom

Implementace

Kopírovací konstruktor & copy-assign operátor musí být explicitně definovaný, protože `std::unique_ptr` není kopírovatelný.

Move konstruktor & move-assign operátor

- pokud by nebyly definovány, byly by skryté kvůli explicitně specifikovanému kopírovacímu konstrukturu, respektive copy-assign operátoru
- můžeme použít default definice, po
`A a; A b(std::move(a));` je `a` v nevalidním stavu, protože jeho `_impl` je `nullptr`
- to ale nevadí, protože `a` by stejně neměl být použit, musí být pouze možné zavolat jeho destruktork (což je OK)

PImpl idiom

Problémy s dědičností

- potomci nemají přístup k obsahu `_impl`, pokud nejsou definováni ve stejném `.cpp`
- pokud jsou ve stejném `.cpp`, nemůžeme rozlišit `private` a `protected` data
- těžko se přidávají další privátní data/metody

Možné obejít

- definovat rozhraní abstraktní třídou
- neobsahuje atributy
- může obsahovat i implementace některých veřejných metod, ale jen pokud nepotřebují přístup k atributům
- až teprve potomci používají PImpl idiom
- **nevýhoda:** nemůžeme mít hlubší dědičnost

PImpl idiom

Problémy s dědičností

Alternativní (částečné) řešení

- definovat `Impl` v samostatném `.h`, který budou používat jen implementace třídy a jejích potomků (to však nelze vynutit!)
- potomci použijí pro `_impl` třídu, která dědí `Impl`

Problémy:

- konzistence! (`Impl` předka v potomkovi, ...)
- potomci musí dělat `dynamic_cast` pro přístup k dodatečným datům/metodám
- pokud chceme označit obsah `Impl` jako `private`, musíme použít `friend`
- lze zneužít k přístupu k implementaci mimo potomky

PImpl idiom

Další problémy

- komplikovanější, vyžaduje explicitně specifikovat všechny konstruktory a přiřazovací operátory
- vyšší overhead (alokace, přístup přes pointer), **ale**:
 - chtěli jsme mít oddělené rozhraní od implementace
 - pokud by nám šlo tolik o výkon, měli bychom mít vše v .h a umožnit tak inlinování (pravděpodobně větší rozdíl než má jeden pointer)

Další informace a podrobnosti na
http://herbsutter.com/gotw/_100/.

Kdy tedy používat?

- chceme mít robustní rozhraní
- potřebujeme minimalizovat čas na kompilaci
- neočekáváme složitou objektovou hierarchii
- nesnažíme se vyždímat z toho maximální výkon

Cvičení: Paměťově mapované soubory

Naprogramujte třídu, která namapuje soubor do paměti v zapisovatelném režimu. K mapování použijte POSIX funkci `mmap` (map 2 `mmap`, délku souboru můžete zjistit pomocí funkce `fstat`).

Použijte PImpl idiom ke skrytí systémových include souborů a implementačních detailů.

Rozhraní:

```
struct MMap {  
    MMap( std::string filepath ); // map file  
    MMap( MMap && );  
    MMap( const MMap & ) = delete;  
    ~MMap(); // end mapping and close file  
  
    char *data();  
};
```

Domácí úloha

C++ rozhraní pro UDP sockety

viz cecko.eu