Singleton

První část

Zdeněk Pavlů

10.09.2016

Dokument byl zpracován podle návrhových vzorů Andrei Alexandrescu, nejedná se o doslovný překlad.

Při procházení knihy „Modern C++ Design: Generic Programming and Design Patterns Applied“ od autora Andrei Alexandrescu jsem narazil na zajímavý problém pro otázku implementace Singletonu. Knihu si můžeme stáhnout zde.(strana 113)
O co se jedná:

- Máme tři Singletony pro řízení *Klávesnice, Obrazovky a Deníku* implementované jako Meyersovy Singletony.
- První dva modelují své protějšky a třetí slouží pro zápis chyb (na příklad textový soubor).
- *Deník* pro svou činnost potřebuje nějaké prostředky a proto bude vytvořen pouze objeví-li se nějaká chyba. Pokud není chyba, nebude *Deník* vytvořen.
- Program *Deník* oznamuje jakoukoliv chybu jak *Klávesnice* nebo *Obrazovky* .
- Předpokládejme, že po úspěšné implementaci *Klávesnice* se nepodaří inicializovat třídu *Obrazovka*. Pak konstruktor třídy *Klávesnice* inicializuje a vytvoří *Deník*, chyba je zaznamenána a program Deník je ukončen spolu s aplikací. Při ukončení jsou odstraněny lokální statické objekty v obráceném pořadí jejich vzniku. *Deník* je odstraněn před instancí *Klávesnice*. Pokud se z nějakého důvodu *Klávesnice* neodstraní, pak zkusí oznámit chybu programu *Deník*.
- Deník::Instance nevědomky vrací odkaz na slupku odstraněného objektu Deník. Náš program se začíná chovat nedefinovaně. Vzniká problém mrtvého odkazu.
- Pořadí odstraňování objektů *Klávesnice, Deník a Obrazovka* není definován. Je potřeba, aby *Klávesnice a Obrazovka,* poslední vytvořený je odstraněn jako první (pravidlo C++) a zároveň musíme z tohoto pravidla vyjmout Deník. Nezávisle na tom, kdy byl vytvořen musí být odstraněn poslední, aby mohl zachytit chyby předchozích obou.

Používáme-li v aplikaci více interagujících Singletonů, nelze používat automatickou kontrolu doby jejich existence. Dobře navržený Singleton by měl mrtvý odkaz detekovat.

Chtěl bych ukázat jak autor řešil tento návrhový vzor a jak postupně dospěl a implementoval generickou šablonu třídy SingletonHolder. Existují různé implementace Singletonů, jejich vhodnost závisí na problému, který řešíme. Veškeré ukázky jsou použité z této knihy bez úprav, autor je dal volně k dispozici. Totéž platí pro soubor „LokiLibrary.lib“ z knihovny „Loki“.

Static Data + Static Functions != Singleton

Nejprve se podíváme, zda nároky dané na Singleton mohou být nahrazeny za použití statických členských funkcí a statických proměných.

```
class Font { ... };
class PrinterPort { ... };
class PrintJob { ... };
```

```
class MyOnlyPrinter
{
public:
	static void AddPrintJob(PrintJob& newJob)
	{
		if (printQueue_.empty() && printingPort_.available())
		{
			printingPort_.send(newJob.Data());
		}
		 else
		{
			printQueue_.push(newJob);
		}
	}
private:
	// All data is static
	 static std::queue<PrintJob> printQueue_;
	static PrinterPort printingPort_;
	static Font defaultFont_;
};


PrintJob somePrintJob("MyDocument.txt");
MyOnlyPrinter::AddPrintJob(somePrintJob)
```

Pokud použijeme toto řešení, zjistíme, že má řadu nevýhod. Problém je, že statické funkce nemohou být virtuální, tato skutečnost zabraňuje provádět změny v chování bez úprav kódu na příklad MyOnlyPrinter.
Menší problém způsobuje obtížná inicializace a úklid. Pro data třídy MyOnlyPrinter neexistuje žádný bod inicializace a úklidu.

The Basic C++ Idioms Supporting Singletons

Podívejme se na nejčastější návrh Singletonu.

```
// Hlavičkový soubor Singleton.h
class Singleton
{
        public:
                static Singleton* Instance() // Unikátní přístupový bod
                 {
                        if (!pInstance_) pInstance_ = new Singleton;
                        return pInstance_;
                 }
                ... operations ...
        private:
                Singleton(); // Neumožňuje vytvořit nový Singleton
                Singleton(const Singleton&); // Znemožňuje vytvořit kopii
                static Singleton* pInstance_; // Jediná instance
};
// Implementační soubor Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
```

Zde vidíme, privátní konstruktory, uživatelský program nemůže Singletony vytvořit. Unikátnost je zajištěna při kompilaci, a toto je podstatou implementování návrhového vzoru Singleton v C++. Pokud nezavoláme metodu Instance Singleton se nevytvoří. Malá cena za toto řešení je zanedbatelný test na začátku členské funkce Instance. Výhodou tohoto řešení, že se vytvoří při prvním použití.

Zjednodušení Singletonu, který se někdy vyskytuje, ukazuje nesprávnou konstrukci.

```
// Header file Singleton.h
class Singleton
{
        public:
                static Singleton* Instance()  // Unikátní přístupový bod
                {
                        return &instance_;
                }
                int DoSomething();
        private:
                 static Singleton instance_;
};
// Implementation file Singleton.cpp
Singleton Singleton::instance_;
```

Toto řešení je špatné i když Instance je statickým členem jako v předchozím příkladu mezi nimi je velký rozdíl. Instance je zde inicializován dynamicky, konstruktor je volán za běhu programu, zatímco v předchozím příkladu pInstance těží ze statické inicializace (typ bez konstruktoru inicializovaný při kompilaci).
Kompilátor provede statickou inicializaci dříve před spuštění prvního příkazu sestavení programu. Nahrání programu je vlastně inicializací. Musíme si uvědomit, že C++ nedefinuje pořadí inicializace dynamicky inicializovaných objektů, které jsou v různých jednotkách překladu. Toto je hlavním zdrojem problémů.

Podívejme se na tento kus kódu:

```
// SomeFile.cpp
#include "Singleton.h"
int global = Singleton::Instance()->DoSomething();
```

V závislosti na pořadí inicializací, který kompilátor zvolí, může Singleton::Instance vracet ještě nevytvořený objekt. V tomto případě se nemůžeme spolehnout na inicializace, pokud je používán jako externí objekt.

Enforcing the Singleton's Uniqueness

(Zajištění jedinečnosti singletonu)

Nyní se podíváme na pár technik, které zajišťuje jedinečnost singletonu. Přednastavený konstruktor a kopírovací konstruktor budeme vždy definovat jako privátní.

```
Singleton sneaky(*Singleton::Instance()); // error!
// Není možné provést plíživou kopii objektu
```

Pokud nebudeme definovat kopírovací konstruktor, pak to udělá kompilátor za nás, jedná se o veřejný konstruktor. Deklarování explicitního kopírovacího konstruktoru znemožní automatické vygenerování a umístění tohoto konstruktoru v private sekci a pří kompilaci je vyvolána chyba v definici sneaky.
Pokud zapomeneme při návrhu třídy na sématiku kopírování a přiřazení je to chybou, tuto chybu dělají nejen začátečníci. Je to běžné hlavně u malých tříd používaných pro RAII (Resource Acquisition Is Initialization). Třída by měla kopírování podporovat nebo zakázat.
V našem případě explicitně zakážeme kopírování a přiřazení.

```
Class T { //……
        private :
                T( const T& );  //není implementován
                T& operator = ( const T&); //není implementován
```

Dalším zlepšením je nechat Instance vracet odkaz místo ukazatele. Vzniká, ale problém možnosti tohoto ukazatele smazat. Proto zmenšíme tuto možnost zhruba takto.

```
// odkaz bude uvnitř třídy Singleton
static Singleton& Instance();
```

Poslední co nám zbývá udělat deklarovat destruktor jako privátní. Naše rozhraní třídy bude vypadat takto:

```
class Singleton
{
                Singleton& Instance();
                 ... operations ...
        private:
                Singleton();
                Singleton(const Singleton&);
                Singleton& operator=(const Singleton&);
                ~Singleton();
};
```

Destroying the Singleton

(Odstranění Singletonu)

Na začátku jsme řekli, že Singleton je vytvořen na žádost při zavolání Instance. Tím je definován okamžik vytvoření, ale zůstává problém odstranění. Tento problém je palčivý a je popsán v knize „Pattern Hatching“ Johna Vlissidese (pro někoho, kdo chce studovat déle).
Pokud nesmažeme Singleton nezpůsobíme ztrátu paměti. Ke ztrátě dojde tehdy pokud alokujeme získaná data a přijdeme na všechny odkazy na ně. V tomto případě nic neshromažďujeme a uchováváme informace o alokované paměti až do konce.
Potíž je, že zde nastává ztráta zdrojů, je to dáno tím, že konstruktor třídy Singletonu může získat neomezenou množinu zdrojů (síťová spojení, manipulátory mutexů operačního systému a podobně). Pokud se chceme vyhnout ztrátě zdrojů musí být Singleton odstraněn během ukončování aplikace. Nikdo se nesmí pokusit přistupovat k Singletonu po jeho odstranění.
Správný způsob jak se vyhnout ztrátám zdrojů, je odstranění objektu singletonu během ukončení aplikace. Problémem zůstává jak zvolit správný okamžik ukončení, aby nebylo možné přistoupit k objektu po jeho odstranění.
Zkusíme jiné řešení inicializace, nebudeme používat dynamickou alokaci a statický ukazatel, budeme se spoléhat na lokální statické proměnné.

```
Singleton& Singleton::Instance()
{
        static Singleton obj;
         return obj;
}
```

Tento způsob je elegantní implementace, která byla poprvé publikována Scottem Meyersem a proto se nazývá Meyersův Singleton.
Dá se říci, že je založen na magii kompilátoru. Statický objekt ve funkci je inicializován ve chvíli, kdy běh programu poprvé projde její definicí.
Pozor toto není statická proměnná inicializovaná v čase běhu aplikace s primitivními statickými proměnnými inicializovanými konstantou při kompilaci.

```
int Fun()
{
        static int x = 100;
        return ++x;
}
```

V tomto příkladu je proměnná x inicializována před vykonáním, nejpravděpodobněji již při kompilaci. Při prvním zavolání je skutečnost, že Fun po zavolání mělo hodnotu x = 100.
Úplně jiná situace nastane pokud není inicializátor konstantou při kompilaci, nebo je-li statická proměnná objektem s konstruktorem, pak je proměnná inicializována při prvním průchodu definicí funkce.
V tomto případě kompilátor generuje kód tak, aby po inicializaci podpora pro běh programu zaregistrovala proměnnou k odstranění.

Proměnné jejichž jména jsou na začátku se dvěma podtržítky jsou skryté a o jejich vygenerování se stará pouze kompilátor.

```
Singleton& Singleton::Instance()
{
        // Funkce, které generuje kompilátor
        extern void __ConstructSingleton(void* memory);
        extern void __DestroySingleton();
        // Proměnné generované kompilátorem
        static bool __initialized = false;
        // Vyrovnávací paměť, slouží k uchování Singletonu
        // (Budeme předpokládat, že je korektně zarovnaná)
        static char __buffer[sizeof(Singleton)];
        if (!__initialized)
        {
                // První volání – vytvoření objektu
                // Vyvolá  Singleton::Singleton
                // V paměti  __buffer memory
                __ConstructSingleton(__buffer);
                // Provede registraci odstranění pomocí atexit
                atexit(__DestroySingleton);
                __initialized = true;
        }
        return *reinterpret_cast<Singleton *>(__buffer);
}
```

Jako jádro programu je funkce C++ **atexit**. Pracuje na principu LIFO bližší popis na MSDN. Dříve uložené objekty jsou odstraněny později.
Toto neplatí pro objekty, které spravujeme sami pomocí new a delete.
Kompilátor vygeneruje funkci __DestroySingleton tato funkce odstraní objekt, který je uložen v paměti __buffer a předá adresu této funkce funkci atexit.
Jak tato funkce pracuje? Každé zavolání funkce atexit uloží její parametr do soukromého zásobníku, který je udržovaný knihovnou jazyka C. Během ukončování volá aplikace program funkce registrované pomocí atexit.
Funkce atexit je velmi důležitá, ale nešťastná pro implementaci a návrh vzoru Singletonu. Nezbavíme se jí i když se nám to líbit nebude. Můžeme vyzkoušet jakékoliv řešení návrhu a tato funkce vždy musí být použita.

Meyersův Singleton poskytuje ty nejjednodušší prostředky k odstranění Singletonu během ukončovací sekvence aplikace. V celé řadě aplikací funguje dobře.

The Dead Reference Problem

(Řešení problému mrtvého odkazu 1)

Tento příklad jsem již použil na začátku mého textu. Obstojný Singleton by měl mrtvý odkaz aspoň detekovat. Tento stav můžeme zajistit logickou statickou členskou proměnnou destroyed_ . Tato bude na počátku nastavena na false. Destruktor objektu Singletonu ji nastaví na true.
Provedeme zrevidování toho, co zatím bylo řečeno. Kromě vytváření a vracení odkazů na objekt Singleton má Singleton:Instance navíc zodpovědnost provádět detekci mrtvého odkazu. Budeme se řídit pravidlem „Jedna funkce – jedna zodpovědnost“.
Budeme definovat tři oddělené členské funkce: Create – funkce vytvoří objekt, OnDeadReference – ošetří chybové zpracování a Instance – tato zpřístupní objekt Singletonu.

```
// Singleton.h
class Singleton
{
	public:
		Singleton& Instance()
		{
			if (!pInstance_)
			{
				// Zkontrolujeme zda se nejedná o mrtvý odkaz
				if (destroyed_)
				{
					OnDeadReference();
				}
				else
				{
					// První volání - Inicializace
					Create();
				}
			}
			return pInstance_;
		}
	private:
		 // Vytvoříme nový Singleton a uložíme na něj pointer  pInstance_
		static void Create();
		{
			// úkol: initialize pInstance_
			 static Singleton theInstance;
			pInstance_ = &theInstance;
		}
		// Volá se pokud je detekován mrtvý odkaz
		static void OnDeadReference()
		{
			throw std::runtime_error("Dead Reference Detected"); //je detekován
```

```
        }
        virtual ~Singleton()
        {
                pInstance_ = 0;
                destroyed_ = true;
        }
        // Data
        Singleton pInstance_;
        bool destroyed_;
        ... vypnutý konstruktor, destruktor a operator= ...
};

// Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
bool Singleton::destroyed_ = false;
```

Tato konstrukce funguje, pokud dojde k ukončení aplikace, je zavolán destruktor objektu Singleton. Tento nastaví pInstance_ na nulu a destroyed_ ne true. Pokud nějaký objekt potom se pokusí přistoupit k Singletonu, řízení převezme OnDeadReference a je vyvolána výmka throw std::runtime_error(….) .
Tím se dostáváme k levnému a jednoduchému řešení.

Tato část byla poměrně jednoduchá ve druhé části budeme používat knihovnu STL a šablony .

- Addressing the Dead Reference Problem (I):The Phoenix Singleton
- Problems with atexit
- Addressing the Dead Reference Problem (II): Singletons with Longevity
- Implementing Singletons with Longevity
- Living in a Multithreaded World
- The Double-Checked Locking Pattern
- Putting It All Together
- Decomposing SingletonHolder into Policies
- Assembling SingletonHolder
- Stock Policy Implementations