
SingSimpleton

Definice

- Oficiální
  - Jediná instance třídy - Singleton se stará o svou životnost
  - Globální přístup k instanci
- Řeší četné problémy
- Proč jedna instance?
  - Přístup k jednomu zdroji (soubo
  - Více může být nežádoucí (datab é
- Globální přístup?
  - Ne globální proměnná
  - Vícero míst v programu

Ne světlé písmo na tmavém pozadí !!!

Příklad

```
class Database {
private:
    Database() { /* Establishing connection to database. */ }
    ~Database() { /* Ending the connection. */ }
    inline static Database* instance_ = nullptr;
public:
    static Database& getDatabase() {
        if (instance_ == nullptr)
            instance_ = new Database();
        return *instance_;
    }

    Database (const Database&) = delete;
    Database& operator=(const Database&) = delete;

    void processQuery(string SQLQuery) {
        // Sending query to database
    }
};
```

inline - specifikum jazyka (C++17)

Nová instance jen jednou

Nechceme kopírovat

Nechceme přiřazovat

Statické třídy?

- Podobný koncept
- Nejde líně načítat
  - Chybí kontrola nad životem
- Nejde dědit
  - Ani sama nemůže dědit
- Singleton jde předávat

```
class Database {
    Database() = delete;

public:
    static void processQuery(string SQLQuery) {
        // Sending query to database
    }
};
```

Simple?

Další problémy Singletonu

- Multithreading
- Dědičnost
- Destrukce

Multithreading

- Fungující implementace
  - C++
    - Meyersův singleton
  - C#
    - Statická proměnná ve statické třídě; nerozbije se, statické konstruktory jsou thread-safe

```
class Nested {
    public static Database instance = new Database();
}
public static Database getDatabase() {
    return Nested.instance;
}
```

    - Použití třídy Lazy<T> - konstrukce je thread-safe

```
static Lazy<Database> instance = new Lazy<Database> (() => new Database());
```

Multithreading – data race, zámky

```
class Database {
    Database() {};
    static Database* instance_;
public:
    static Database& getInstance() {
        if (instance_ == nullptr) {
            instance_ = new Database();
        }
        return *instance_;
    }

    void processQuery(...) { /* ... */ }
};
```

ve vícevláknových programech zde vznikne data race

```
class Database {
    Database() {};
    static Database* instance_;
    static std::mutex lock_;
public:
    static Database* getInstance() {
        lock_.lock();
        if (instance_ == nullptr) {
            instance_ = new Database();
        }
        lock_.unlock();
        return instance_;
    }
    void processQuery(...) { /* .. */ }
};
```

Možným řešením jsou standardní zámky:
…takhle ale zamykáme i pro již zkonstruované instance…
Tohle by šlo řešit pomocí double-checked locking pattern.

Multithreading – double-checked locking pattern

```
class Database {
    Database() {};
    inline static Database* instance_ = nullptr;
    static std::mutex lock_;
public:
    static Database& getInstance() {
        if (instance_ == nullptr) {
            lock_.lock();
            if (instance_ == nullptr)
                instance_ = new Database();
            lock_.unlock();
        }
        return *instance_;
    }
    void processQuery(string SQLQuery) { /* ... */ }
};
```

Vznikne opět data race, protože zápis do proměnné, která je typu pointer, v C++ NENÍ atomický

Multithreading – double-checked locking pattern, fixed

```
class Database {
    Database() {};
    inline static std::atomic< Database*> instance_ = 0;
    static std::mutex lock_;
public:
    static Database& getDatabase() {
        if (instance_.load() == nullptr) {
            lock_.lock();
            if (instance_.load() == nullptr)
                instance_.store(new Database());
            lock_.unlock();
        }
        return *instance_;
    }
    void processQuery(string SQLQuery) { /* ... */ }
};
```

Pointer na singleton zabalíme do `std::atomic`

Od C++20 podpora taky pro smart pointery, nebo lze použít jejich atomické operace

Multithreading – std::call_once

```
class Database {
private:
    Database() {};
    ~Database() {};
    inline static Database* instance_ = nullptr;
    static std::once_flag flag;
public:
    static Database& getDatabase() {
        std::call_once(flag, [&]() { instance_ = new Database(); });

        return *instance_;
    }

    Database(const Database&) = delete;
    Database& operator=(const Database&) = delete;

    void processQuery(string SQLQuery) { /* ... */ }
};
```

Je také možné použít třídu `mutex` a vytvořit tak singleton jenom jednou

Použijeme funkci `std::call_once`

Dědičnost

- Je potřeba si rozmyslet
- Různé způsoby použití
- Otázky
  - Má rodič vědět o všech potomcích?
  - Má existovat jedna instance celkově, nebo od každého?
  - Má být instance nahraditelná jinou (jiným potomkem)?
  - Základní třída abstraktní?

Dědičnost - první příklad

```
class Database {
protected:
    Database() { };
public:
    static Database& getDatabase() {
        if (instance_ == nullptr) {
            if (Config::DatabaseType == "MySQL")
                instance_ = new MySQLDatabase();
            else
                instance_ = new TSQLDatabase();
        }
    return *instance_;
    }
    virtual void processQuery() = 0;
};
class MySQLDatabase : public Database {
private:
    MySQLDatabase() : Database() { }
    friend class Database;
public:
    virtual void processQuery() override { /* ... */ }
};
```

Potomci potřebují konstruktor rodiče

Známí potomci

Rodič musí být označen jako `friend`, aby měl ke konstruktoru přístup

Dědičnost - druhý příklad

```
class Database {
    inline static Database* instance_ = nullptr;
    static std::map<std::string, Database*> registry_;
protected:
    Database() { };
    static Database* lookUp (const std::string& name);
    void register (const std::string & name, Database* s);
public:
    static Database& getDatabase() {
        if (instance_ == nullptr)
            instance_ = lookUp(Config::DatabaseName);
        return *instance_;
    }
    virtual void processQuery() = 0;
};
class MySQLDatabase : public Database {
    inline static MySQLDatabase instance_;
    MySQLDatabase() {
        register("MySQLDatabase", &instance_);
    };
};
```

Registr potomků

Opatrně

Potomci se instanciují vždy, aby se zaregistrovali

Dědičnost - třetí příklad

```
class Database {
    inline static Database* instance_ = nullptr;
protected:
    Database() { };
public:
    template <typename T>
    static Database& getDatabase() {
        if (instance_ == nullptr)
            instance_ = new T();
        return *instance_;
    }
    virtual void processQuery() = 0;
};
class MySQLDatabase : public Database {
    friend class Singleton;
    MySQLDatabase() { };
};
int main() {
    Database::getDatabase<MySQLDatabase>();
}
```

Destrukce, klíčové slovo static v C++

- Kdo má zodpovědnost za zrušení?
  - Mechanismy C++ nebo uživatel
  - Jak se postarat o (správné) zrušení objektu
  - Největším nepřítelem jsou resource a memory leaks
- Kdy je objekt zrušen a jaké je pořadí rušení objektů?
- Statická data
  - Kdy jsou inicializována?
  - Kdo se postará o následnou destrukci?

```
int counter = 0; //constant,
load-time, static (implicit)

class Log {};
Log l; //load-time, static (implicit)

class Keyboard {
    static Log l; //load-time
    void key_init() {
        /*constant, so load-time*/
        static int space = 32;
        /*first-pass init*/
        static Keyboard instance;
    }
}
```

Destrukce, ostrich singleton

- Problém destrukce budeme ignorovat
- Paměť se uvolní při ukončení procesu
- Je otázkou, co je a co není resource leak
- Destruuje se pouze pointer, ne však objekt

```
class OstrichData {
private:
    OstrichData() {};
    inline static OstrichData* instance = nullptr;

public:
    static OstrichData& getInstance();
    void doSomething();
};
```

Destrukce, funkce atexit()

- Statické (i lokální) proměnné se odstraňují od nejmladších, na principu LIFO
- Při vytváření objektu lze zaregistrovat funkci pro zrušení, tyhle funkce se pak zavolají při ukončení programu

```
Database& Database::getDatabase() {
   extern void __constructDatabase(void* memory);
   extern void __destroyDatabase();

   static bool __initialized = false;
   static char __buffer[sizeof(Database)];

   if (!__initialized) {
     __constructDatabase(__buffer);
     atexit(__destroyDatabase);
     __initialized = true;
   }
   return *reinterpret_cast< Database*>(__buffer);
}
```

Proměnné a funkce generované kompilátorem.

`__buffer` obsahuje Database

Destrukce se zaregistruje při konstrukci singletonu (Database)

Scott Meyers singleton

- Použije se statická lokální proměnná
- Klíčové slovo new se zahodí, funkce vracejí reference na statický objekt ve funkci, zanikne na konci programu
- Od C++11 (gcc C++03) je tenhle přístup thread safe

```
class Database {
public:
    static Database& getDatabase() {
        static Database instance;
        return instance;
    }

    Database(const Database&) = delete;
    Database& operator=(const Database&) = delete;
private:
    Database() { /* ... */ };
    ~Database() { /* ... */ };
};

int main() { Database& d = Database::instance(); }
```

2. Inicializace statického objektu, pouze při prvním přechodu

4. Vrácení (nově zkonstruovaného) objektu

3. Konstruktor objektu

6. Destruktor objektu

1. Zavolání metody pro získání instance

5. Konec programu, destrukce statických proměnných

Destrukce - nebezpečí

1. Inicializace Keyboard
2. Inicializace Display s chybou
3. Inicializace Log a zapsání chyby
4. Konec programu
5. Destrukce Log
6. Destrukce Display
7. Destrukce Keyboard s chybou
8. **Reference neexistujícího objektu Log**

- Problém mrtvé reference
  - Dáno designem, lazy přístup
- Log by se měl destruovat až jako poslední
- Destruktor je `noexcept`
  - Nutně nemusí pomoct

Detekce mrtvé reference

```
class Database {
public:
    static Database& getDatabase() {
        if (instance_ == nullptr)
            if (destroyed_) {
                onDeadRef();
            }
            else create();
        return *instance_;
    }
private:
    inline static Database* instance_ = nullptr;
    static void create() {
        static Database theInstance;
        instance_ = &theInstance;
    }
    static void onDeadRef() { throw /*...*/ }
    inline static bool destroyed_ = false;
    ~Database() {
        destroyed_ = true;
        instance_ = nullptr;
    }
};
```

- Statický příznak o destrukci
- Vyhodí výjimku při přístupu k destruovanému objektu
  - Nemusí nutně pomoct

Phoenix singleton

```
class Database {
private:
    static void onDeadRef() {
        create();
        new(instance_) Database;
        atexit(killPhoenix);
        destroyed_ = false;
    }
    static void killPhoenix() {
        instance_->~Database();
    }
};
```

Nová instance se vytvoří na místě té původní

Registrace destruktoru

- Objekt se po destrukci znovu vytvoří
- Příklad: Keyboard a Display singletony, Log -> Fénix
- C++: paměť statických objektů alokována do konce běhu programu
- Problém: vznikne úplně nový objekt

Destrukce - dlouhověkost

- Problémy Fénixe – ztráta stavu, uložení, uzavření...
- Dependency manager – nutnost konstrukce všech závislostí
- K Singletonu přidáme dlouhověkost
  - Při vytváření Singletonu se nastaví priorita destrukce
  - Log bude mít větší dlouhověkost než klávesnice
  - Jedná se o explicitní mechanismus destrukce objektů

```
class SomeThirdClass { /**/ };
class Keyboard { /**/ };
class Log { /**/ };
SomeThirdClass* third_object(new SomeThirdClass);

template < typename T>
void setLongevity(T* object, int longevity);

int main() {
    setLongevity(Keyboard::instance(), 5);
    setLongevity(Log::instance(), 6);
    setLongevity(third_object, 5);
}
```

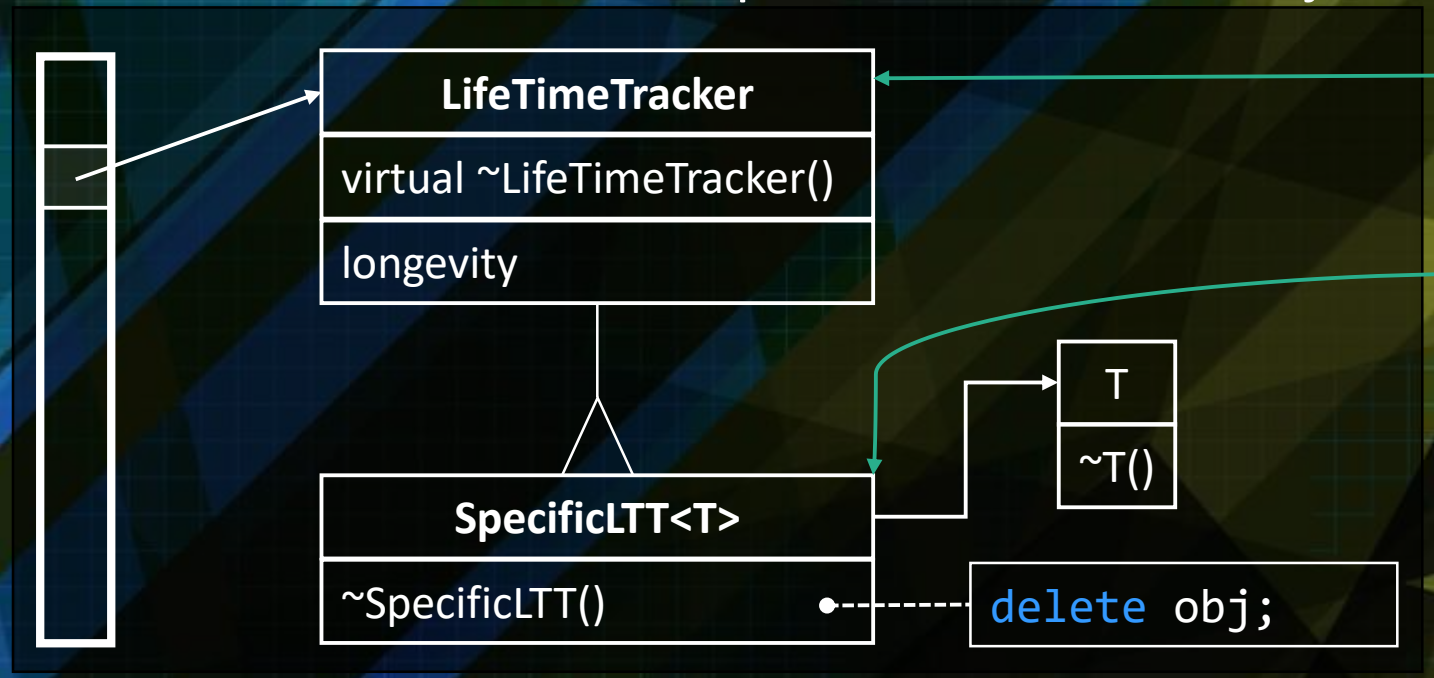
Mechanismus by měl být univerzální a fungovat pro jakékoli dynamické objekty

Šablona pro nastavení dlouhověkosti

Po ukončení programu se objekty zabíjejí v tomto pořadí (Log má větší prioritu a zabije se později)

Destrukce – implementace dlouhověkosti

- Prioritní fronta
- Stejné priority → LIFO (pravidla C++)
- Registrační funkce je šablona
- Ukazatel na abstraktního předka šablon obsahující ukazatel na objekt



Virtuální držák
Virtuální konstruktor

Šablona instanciována a objekt vytvořen za pomocí šablony `setLongevity`

Destrukce – implementace dlouhověkosti

```
template <typename T>
struct Deleter {
    static void DeleteIt(T* pObj) {
        delete pObj;
    }
};

class LifetimeTracker {
public:
    LifetimeTracker(unsigned int x): longevity(x) {}
    virtual ~LifetimeTracker() = 0;
    friend inline bool Compare(unsigned int longevity,
         const LifetimeTracker* p) {
        return p->longevity > longevity;
    }
private:
    unsigned int longevity;
};

inline LifetimeTracker::~LifetimeTracker() {}
using TrackerArray = LifetimeTracker**;
extern TrackerArray pTA;
extern unsigned int elements;
```

Možnost definovat si vlastní způsob zabíjení objektu

`LifetimeTracker` umí jak zabíjet, tak porovnávat stáří

`TrackerArray` by měl taky být singleton, kdo se o něj jako o singleton postará?

Fronta na zabití

Policies

- Při vytváření singletonu můžeme využít Policies třídy
- Creation policy
  - new, malloc, dědičnost
- Lifetime policy
  - podle C++ → LIFO
  - fénix, dlouhověkost, pštros
- Threading model policy
  - jedno vláknové
  - více vláknové
  - jiné, systémově specifické řešení

```
template
<
  class T,
  template <class> class CreationPolicy = CreateUsingNew,
  template <class> class LifetimePolicy = DefaultLifetime,
  template <class> class ThreadingModel = SingleThreaded
>
class SingletonHolder {
 public:
  static T& Instance();

 private:
  // Helpers
  static void DestroySingleton();

  // Protection
  SingletonHolder();

  // Data
  using InstanceType = ThreadingModel<T>::VolatileType; // For threading model purposes
  static InstanceType* pInstance_;
  static bool destroyed_;
};
```

Singleton

Policies pro jednotlivé „části“ singletonu

Použití žádaného threading modelu pro daný singleton

```
template <...>
T& SingletonHolder<...>::Instance() {
  if (!pInstance_) {
    typename ThreadingModel<T>::Lock guard;
    if (!pInstance_) {
      if (destroyed_) {
        LifetimePolicy<T>::OnDeadReference();
        destroyed_ = false;
      }
      pInstance_ = CreationPolicy<T>::Create();
      LifetimePolicy<T>::ScheduleCall(&DestroySingleton);
    }
  }
  return *pInstance_;
}
```

```
class A { /* ... */ };

using SingleA = SingletonHolder<A, CreateUsingMalloc, Phoenix, MultiThreaded>;
// dále se použije SingleA::Instance()
```

```
class KeyboardImpl { ... };
class DisplayImpl { ... };
class LogImpl { ... };

inline unsigned int GetLongevity(KeyboardImpl*) { return 1; }
inline unsigned int GetLongevity(DisplayImpl*) { return 1; }
inline unsigned int GetLongevity(LogImpl*) { return 2; }  // The log has greater longevity

using Keyboard = SingletonHolder<KeyboardImpl, SingletonWithLongevity>;
using Display = SingletonHolder<DisplayImpl, SingletonWithLongevity>;
using Log = SingletonHolder<LogImpl, SingletonWithLongevity>;
```

Kdy používat Singleton?

Pro

- Základní požadavky (jedna instance, globální přístup)
- Lazy přístup
- Jednoduchý na pochopení

Proti

- Může být na obtíž při unit testování
  - Porušuje jejich nezávislost
- Přílišné zneužití
  - "Simpleton"
- Nejde nastavit konstruktor
  - Skryté závislosti
- Není to až tak dobrý návrh
  - Stará se sám o sebe
  - "Jako globální proměnná"

Související návrhové vzory

- Facade
  - V případě potřeby pouze jednoho vstupu do systému
- State
  - Stav systému je singleton
- Abstract Factory, Builder, Prototype
  - Málokdy potřebujeme víc než jednu instanci