l

C,C++ Design Pattern

Linh La

Table of Contents

[I. Phần 1 1](#_Toc491299446)

[1. Policy-Based Class Design 1](#_Toc491299447)

## Cơ bản

Design pattern đại diện cho những kinh nghiệm thực tế rút ra từ các lập trình viên OOP lâu năm. Design pattern là giải pháp cho các vấn đề tổng quát mà lập trình viên gặp phải trong quá trình phát triển phần mềm.

Các giải pháp này đã trải qua một thời gian dài kiểm nghiệm.

### Tổng quan về Design Pattern

##### Gang of Four (GOF)

Vào năm 1994, Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides phát hành cuốn sách có tên Design Patterns – Các yếu tố lập trình hướng đối tượng có thể tái sử dụng, đã đưa ra nền tảng cho Design Pattern trong ngành phát triển phần mềm.

Những tác giả này được biết với tên Gang of Four (GOF). Theo các tác giả, design pattern được dựa trên cơ sở học thuyết thiết kế hướng đối tượng sau:

* Program to an interface not an implementation
* Favor object composition over inheritance

##### Các dùng Design Pattern

Design Patterns có 2 cách dùng trong ngành phát triển phần mềm.

##### Platform phổ biến cho lập trình viên

Design patterns cung cấp một định nghĩa chuẩn và đặc thù cho từng ngữ cảnh. Ví dụ, singleton design pattern đặc biệt sử dụng cho đối tượng đơn lẻ, mà chúng ta chỉ cần duy nhất một đối tượng được khởi tạo.

##### Thực tế nhất

Design patterns trải qua thời gian dài kiểm nghiệm và là giải pháp tốt nhất cho các vấn đề cụ thể trong ngành phát triển phần mềm. Học các pattern này giúp các lập trình viên chưa có kinh nghiệm học thiết kế phần mềm nhanh và dễ dàng hơn.

##### Phân loại Design Patterns

Theo cuốn Design Patterns - Elements of Reusable Object-Oriented Software, có tổng cộng 23 design pattern phần chia thành 3 nhóm pattern là :

|  |  |
| --- | --- |
| **S.N.** | **Pattern & Description** |
| 1. Patterns tạo đối tượng | Các pattern này cung cấp các cách tạo đối tượng trong khi che giấu logic về cách tạo, mà không dùng trực tiếp toán tử new.  Điều này giúp ứng dụng linh hoạt hơn khi quyết định đối tượng thế nào cần được tạo ra cho những nhu cầu khác nhau. |
| 2. Patterns về cấu trúc | Các pattern đề cấp tới kết cấu của đối tượng và class. Lý thuyết thừa kế được dùng nhiều trong triển khai interface và định nghĩa cấu trúc của đối tượng để có phương thức mới |
| 3. Patterns hành vi | Các pattern dùng cho giao tiếp giữa các đối tượng. |
| 4. J2EE Patterns | Các pattern đề cấp tới các tầng trình diễn, đưa ra bởi Sun Java Center. |

Ngoài ra chúng ta có thể thảo luận về nhóm 4 là : J2EE pattern

### Xu hướng thiết kế

##### Policy và policy class

##### Triển khai policy class là tham số template

##### Triển khai policy class là hàm thành viên template

## Pattern tạo Object

### Singleton Pattern

##### Tiêu chí

Singleton là một cải tiến của biến global: Singleton đảm bảo không tạo ra đối tượng thứ 2. Do đó nên dùng Singleton khi kiểu dữ liệu là độc nhất cho ứng dụng, như Keyborad, Display, PrintManager hay SystemClock.

Cung cấp phương tiện global để truy xuất tới đối tượng mà không cần phải khởi tạo đối tượng. Do đó đối tượng Singleton có trách nhiệm tự khởi tạo và hủy đối tượng, việc khởi tạo đối tượng sẽ chỉ diễn ra trong lần truy xuất đầu tiên.

Singleton thường được dùng để kiểm soát truy xuất tài nguyên như kết nối cơ sở dữ liệu hay sockets.

Chúng ta có một license cho 1 kết nối tới Database chẳng hạn, một đối tượng kết nối Singleton đảm bảo luôn chỉ có 1 kết nối trong mọi thời điểm.

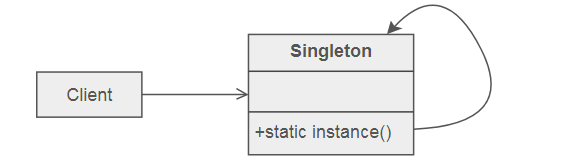
Điển hình nhất việc dùng Singleton là bộ logging.

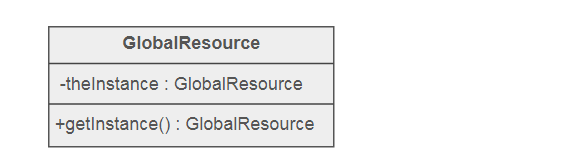
##### Cấu trúc

Có 3 điểm cần chú ý:

* Đối tượng duy nhất phải có thuộc tính static và private
* Hàm truy xuất phải có thuộc tính static và public
* Các hàm dựng phải có thuộc tính private, không cho phép tùy tiện tạo đối tượng ra bên ngoài.

Người dùng phải gọi hàm truy xuất bất ki khi nào cần tham chiếu tới đối tượng của class.





Nhờ cấu trúc như trên nên Singleton giúp giảm số namespace cho biến global.

##### Quan điểm thiết kế

1. Duy trì tính duy nhất

Ngoài hàm dựng, trình biên dịch còn tự sinh ra toán tử gán mặc định. Tính duy nhất nên thực hiện triệt để, đối với đối tượng Singleton chỉ có thể thực hiện gán với chính nó mà thôi, điều này là vô nghĩa.

Điều quan trọng cuối cùng là để hàm hủy là private, giúp đảm bảo người dùng không thể hủy con trỏ đối tượng Singleton.

class Singleton

{

Singleton& getInstance();

private:

Singleton();

Singleton(const Singleton&);

Singleton& operator=(const Singleton&);

~Singleton();

};

Chúng ta sẽ tạo class SingleObject, và phương thức static getInstance() để lấy đối tượng static ra bên ngoài.

Nếu không định nghĩa hàm dựng sao chép, trình biên dịch sẽ tự tạo ra phiên bản public, do đó phải tường minh khai báo tất cả hàm dựng là private, cũng tương tự với phép gán:

#include <iostream>

class SingleObject {

private:

//Đặt tất cả hàm dựng vào đầy. Không cho người dùng khởi tạo tùy ý

SingleObject (){};

SingleObject (const SingleObject& rhs);

public:

//Đối tượng singleton sẽ khai báo trong hàm với thuộc tính static

static SingleObject & getInstance(){

static SingleObject instance;

return instance;

}

void showMessage(){

std::cout<<"Welcome to singleton \n";

}

};

Nếu đối tượng Singleton không được dùng thì sẽ không phải khởi tạo, điều này giúp tối ưu chi phí khởi tạo, đặc biệt đối với đối tượng sử dụng nhiều tài nguyên khi khởi tạo.

Sử dụng:

#include "SingleObject.h"

void SingletonDemo(){

SingleObject::getInstance()->showMessage();

}

1. Singleton dùng lazy initialization

Có rất nhiều cách triển khai Singleton, thông thường sẽ khai báo con trỏ tới đối tượng Singleton là static và private, sau đó dùng lazy initialization trong phương thức global:

class SingletonObject

{

public:

static SingletonObject\* getInstance()

{

if (!pInstance\_)

pInstance\_ = new SingletonObject;

return pInstance\_;

}

private:

SingletonObject();

SingletonObject(const SingletonObject&);

static SingletonObject\* pInstance\_;

};

SingletonObject\* SingletonObject::pInstance\_ = 0;

1. Singleton dùng static

Ngoài ra có người còn muốn làm đơn giản quá trình khởi tạo bằng cách thay con trỏ tới đối tượng Singleton bằng biến như trường hợp dưới đây:

class SingletonObject

{

public:

static SingletonObject\* getInstance()

{

return &instance\_;

}

int DoSomething();

private:

static SingletonObject instance\_;

};

SingletonObject SingletonObject::instance\_;

Đối tượng static instance\_ trong class sẽ được khởi tạo động khi runtime, trong khi biến con trỏ static pInstance\_ hưởng lợi từ khởi tạo tĩnh (không gọi hàm dựng trong suốt thời gian biên dịch)

Trình biên dịch khởi tạo tĩnh trước khi mọi dòng lệnh thực thi, thông thường các biến khởi tạo tĩnh sẽ nằm ngay trên file thực thi, chúng được nạp khi bắt đầu thực thi.

C++ không tuân theo thứ tự khởi tạo khi khởi tạo động đối tượng trong các đơn vị biên dịch khác nhau. Xem đoan code sau:

// SomeFile.cpp

#include "Singleton.h"

int global = Singleton::getInstance()->DoSomething();

Tùy vào thứ tự khởi tạo mà trình biên dịch chọn instance\_ hay global, lệnh gọi tới Singleton::Instance có thể trả về đối tượng chưa khởi tạo. Nghĩa là bạn không thể chắc rằng instance\_ luôn được khởi tạo trước, nếu đối tượng bên ngoài đang dùng nó.

Một điểm cải thiện là phương thức global nên trả về tham chiếu thay vì con trỏ, vì nơi gọi hàm có xu hướng xóa con trỏ.

// bên static trong class Singleton

static Singleton& Instance();

1. Hủy Singleton

Chúng ta có một vấn đề: khi nào thì hủy đối tượng Singleton và hủy như thế nào?

Cuốn sách GoF không thảo luận về vấn đề này, nhưng thực tế Singleton không bị hủy trong suốt vòng đời ứng dụng.

Vùng nhớ chỉ rò rỏ khi nó không còn tham chiếu bởi con trỏ nào nữa, trường hợp Singleton sẽ tùy trường hợp.

Tất cả hệ điều hành hiện nay đều có cơ chế quản lý vùng nhớ cho từng process, vùng nhớ tự động thu hồi sau khi process kết thúc.

Có khả năng rò rỉ tài nguyên của đối tượng Singleton, khi hàm dựng của Singleton có thể yêu cầu một lượng tài nguyên không giới hạn: kết nối mạng, handle OS, mutex và các phương tiện IPC,…

Cách phù hợp nhất là hủy đối tượng Singleton trong quá trình ứng dụng thoát.

1. Atexit và biến static

Đối tượng static trong hàm chỉ được khởi tạo cho lần thực thi đầu tiền đi qua. Ngoài ra, trình biên dịch sinh ra code để sau khi khởi tạo, môi trường runtime hỗ trợ đang ký hàm hủy cho biến static, tức là biến static được quản lý bởi trình biên dịch.

Chúng ta có thể tưởng tượng đoạn code mà C++ hủy biến static như sau, lưu ý các biến có tiền tố “\_\_” sẽ là không có trên thực tế.

Singleton& Singleton::Instance()

{

// Hàm sinh ra bởi trình biên dịch

extern void \_\_ConstructSingleton(void\* memory);

extern void \_\_DestroySingleton();

// Biến sinh ra bởi trình biên dịch

static bool \_\_initialized = false;

// Bộ đệm chứa singleton, giả định rằng bộ đệm đủ vùng nhớ để chứa

static char \_\_buffer[sizeof(Singleton)];

if (!\_\_initialized)

{

// Cho lệnh gọi đầu tiên, hàm dựng đối tượng sẽ gọi Singleton::Singleton trong vùng nhớ

// \_\_buffer

\_\_ConstructSingleton(\_\_buffer);

// đăng kí hàm hủy

atexit(\_\_DestroySingleton);

\_\_initialized = true;

}

return \*reinterpret\_cast<Singleton \*>(\_\_buffer);

}

Điểm cốt lõi là lệnh gọi tới hàm **atexit**, một hàm trong thư viện chuẩn của C, cho phép bạn đăng kí hàm để được gọi tự động trong quá trình ứng dụng thoát ra, theo cơ chế FILO, tức đối tượng nào tạo trước sẽ hủy sau cùng (ngoại trừ đối tượng cấp phát động).

Trình biên dịch sinh ra hàm \_\_DestroySingleton() làm nhiệm vụ hủy đối tượng Singleton nằm trong vùng nhớ của \_\_buffer, và truyền địa chỉ hàm \_\_DestroySingleton() vào **atexit**.

Cú pháp atexit:

int atexit(void (\*pFun)());

* pFun : con trỏ hàm
* Return 0 nếu thất bại, ngược lại trả về giá trị khác zero là thành công

Hàm pFun sẽ được gọi trong quá trình hủy đối tượng statics, theo cơ chế stack: tức nếu hàm A đăng ký trước hàm B, thì hàm B được gọi trước hàm A, trật tự này cũng được áp dụng trong các hàm dựng đối tượng static gọi tới atexit (C++11)

Mỗi lệnh gọi tới atexit sẽ đẩy tham số vào stack dùng riêng của C runtime library. Trong quá trình ứng dụng thoát, môi trường runtime sẽ gọi các hàm đã đăng ký thông qua atexit.

Tiến trình atexit có tầm quan trọng với triển khai Singleton Pattern trên C++.

Meyers Singleton hoạt động ổn định trong hầu hết trường hợp. Chúng ta sẽ tìm hiểu các vấn đề và cải thiện cho từng trường hợp đặc biệt.

1. Singleton Meyer

Scott Meyers đưa ra giải pháp đơn giản là dựa vào cơ chế ngôn ngữ, Meyers không dùng cấp phát động và static pointer, mà dùng biến static local:

//triển khai dưới gọi là Meyers singleton

Singleton& Singleton::Instance()

{

static Singleton obj;

return obj;

}

Đối tượng static trong hàm chỉ được khởi tạo cho lần thực thi đầu tiền đi qua. Ngoài ra, trình biên dịch sinh ra code để sau khi khởi tạo, môi trường runtime hỗ trợ đang ký hàm hủy cho biến static, tức là biến static được quản lý bởi trình biên dịch.

Meyer Singleton hoạt động ổn định trong hầu hết trường hợp. Chúng ta sẽ tìm hiểu các rắc rối và cải thiện cho từng trường hợp đặc biệt cho Meyer Singleton.

1. Tham chiếu invalid

Tình huống:

1. Triển khai class Log dùng Singleton: nếu hàm dựng của Log tốn nhiều chi phí, thì chỉ khởi tạo khi có lỗi xảy ra.
2. Giả sử ứng dụng triển khai thêm 2 Singleton Keyboard và Display: trong khi Singleton Keyboard khởi tạo thành công thì Singleton Display khởi tạo thất bại, hàm dựng Display tạo Log, lỗi được ghi lại, và sau đó ứng dụng thoát ngay.
3. Tại thời điểm thoát, theo cơ chế ngôn ngữ: môi trường runtime hủy các đối tượng static local theo trình tự FILO, như vậy Singleton Log sẽ bị hủy trước Keyboard.
4. Giả sử hàm hủy Singleton Keyboard xảy ra lỗi và gọi tới Singleton Log::getInstance(), phương thức sẽ trả về tham chiếu tới phần xác đã hủy dẫn tới hành vi không xác định.

Đây là vấn đề tham chiếu chết, thứ tự gọi hàm dựng và hàm hủy của Keyboard, Log và Display không biết trước được.

Điều phải làm là phải để Singleton Log không theo rule của C++ (LIFO), Singleton Log phải luôn hủy sau cùng mọi đối tượng để có thể thu thập lỗi trong giai đoạn kết thúc ứng dụng.

Nếu ứng dụng dùng singleton tương tác đa điểm thì không thể có cách tự động kiểm soát vòng đời của nó.

Cơ chế kiểm tra tham chiếu:

* Dùng một biến private static kiểu Boolean được khởi tạo false.
* Hàm hủy Singleton: set biến Boolean thành true

Ngoài việc tạo và trả về tham chiếu tới biên Singleton, Singleton::getInstance() phải kiểm tra tham chiếu còn hợp lệ không.

Hướng dẫn về design có nói rằng một hàm chỉ nên làm một nhiệm vụ, chúng ta sẽ tách ra thành 3 hàm:

* Private: Singleton::Create(): Tạo đối tượng Singleton
* Private: Singleton:: OnDeadReference() : xử lí lỗi khi tham chiếu invalid
* Public: Singleton:: Instance () : truy xuất tới đối tượng Singleton  
  Nếu tham chiếu invalid thì gọi tới Singleton:: OnDeadReference()  
  Ngược lại gọi Singleton::Create()

Đối tượng Singleton:

* Biến static local đối tượng Singleton đặt trong hàm Singleton::Create()
* Con trỏ private static Singleton: khởi tạo null, sẽ được trỏ tới đối tượng Singleton trong Singleton::Create().   
  Set về null khi hàm hủy Singleton được gọi.

Triển khai:

#include <exception>

#include <iostream>

class Singleton

{

public:

Singleton& getInstance()

{

if (!pInstance\_)

{

// Kiểm tra tham chiếu

if (destroyed\_)

OnDeadReference();

else

Create();

}

return \*pInstance\_;

}

private:

static Singleton \*pInstance\_;

static bool destroyed\_;

// Tạo mới đối tượng Singleton và set con trỏ trỏ tới nó

static void Create()

{

// Task: khởi tạo pInstance\_

static Singleton theInstance;

pInstance\_ = &theInstance;

}

// gọi hàm OndeadReference khi tham chiếu bị xóa

static void OnDeadReference()

{

throw std::runtime\_error("Dead Reference Detected");

}

virtual ~Singleton(){

pInstance\_ = 0;

destroyed\_ = true;

}

};

// Singleton.cpp

Singleton\* Singleton::pInstance\_ = 0;

bool Singleton::destroyed\_ = false;

Nếu đối tượng có vòng đời lâu hơn Singleton truy xuất tới, ngoại lệ std::runtime\_error sẽ ném ra.

Giải pháp này không tốn nhiều chi phí đơn giản và hiệu quả. Tuy nhiên, nếu áp dụng giải pháp trên cho KDL ((Keyboard, Display, Log) thì không ổn.

Ngoại lệ ném ra từ Singleton::OnDeadReference() không cho phép hàm hủy Display ghi báo cáo vào Log.

Kết quả là hành vi không xác định xảy ra trong giai đoạn thoát ứng dụng.

1. Phoenix Singleton

Singleton Log phải sẵn sàng trong mọi thời điểm, kể cả khi thoát ứng dụng, đây là ý tưởng của Phoenix Singleton Pattern:

* Phoenix Singleton ám chỉ tới khả năng tái sinh sau khi hủy: đối tượng có thể tạo lại nếu tham chiếu bị chết
* Tính duy nhất của đối tượng Singleton vẫn được đảm bảo: đối tượng mới vẫn nằm trên vùng nhớ cũ

Với Phoenix Singleton pattern, Keyboard và Display vẫn là một Singleton bình thường, trong khi Log dùng Phoenix Singleton.

Triển khai Phoneix Singleton với biến static khá đơn giản: Khi tham chiếu bị chết, chúng ta tạo một đối tượng Singleton mới trong cùng một biến (C++ cho phép) : vùng nhớ đối tượng static còn duy trì khi ứng dụng chưa hoàn tất việc thoát.

class Singleton

{

// …implement…

static void KillPhoenixSingleton(); // Thêm mới

};

// Singleton.cpp

Singleton\* Singleton::pInstance\_ = 0;

bool Singleton::destroyed\_ = false;

void Singleton::OnDeadReference()

{

// Dùng Create() để trỏ pInstance\_ tới vùng nhớ của đối tượng Singleton đã hủy

Create();

// Tạo đối tượng Singleton mới ngay tại vùng nhớ cũ

new(pInstance\_) Singleton;

// Đưa hàm hủy đối tượng mới tạo vào Queue

atexit(KillPhoenixSingleton);

// Reset trạng thái cho destroyed\_

destroyed\_ = false;

}

void Singleton::KillPhoenixSingleton()

{

// Gọi thủ công hàm hủy

pInstance\_->~Singleton();

// pInstance\_ bị set zero và destroyed\_ set true

}

Toàn tử new phiên bản đang sử dụng không cấp phát vùng nhớ mà khởi tạo đối tượng vào địa chỉ vùng nhớ cung cấp bởi con trỏ pInstance\_

Vì dùng toán tử new để hồi sinh đối tượng nên chúng ta phải đăng ký hàm hủy đối tượng mới vào **atexit** trong Singleton:: OnDeadReference() : truyền địa chỉ hàm KillPhoenixSingleton vào **atexit**()

Trong quá trình ứng dụng thoát, hàm hủy Singleton được gọi. Khi đối tượng global cố truy xuất tới Singleton lần nữa, Singleton::getInstance() gọi tới OnDeadReference() để tái sinh.

Phoenix Singleton đảm bảo các đối tượng global và Singleton khác có thể truy xuất tới thể hiện hợp lệ của nó bất kì lúc nào, tất phù hợp cho Log

Chúng ta có thể sử dụng Macro #ifdef để đăng ký hàm hủy Phoenix Singleton cho atexit:

#ifdef ATEXIT\_FIXED

// Queue this new object's destructor

atexit(KillPhoenixSingleton);

#endif

1. Trạng thái Phoenix Singletons

Việc dùng atexit chỉ khởi tạo lại đối tượng Singleton tại vùng nhớ cũ nhưng không bảo toàn trạng thái cũ cho đối tượng.

Trở lại với ví dụ của KDL, nếu Log được tạo, vòng đời của Log phải lâu hơn so với Keyboard và Display.

Chúng ta cần một cách để kiểm soát vòng đời của nhiều singleton, ngoài ra còn có thể kiểm soát cả các đối tượng global cấp phát động.

Cơ chế kiểm soát vòng đời đối tượng độc lập với học thuyết về singleton:

// Singleton class

class SomeSingleton {

//...

};

// Class thông thường

class SomeClass

{

//...

};

SomeClass\* pGlobalObject(new SomeClass);

int main()

{

SetLongevity(&SomeSingleton().Instance(), 5);

// Đảm bảo pGlobalObject sẽ bị xóa sau đối tượng SomeSingleton

SetLongevity(pGlobalObject, 6);

//...

}

Hàm SetLongevity() nhận tham chiếu tới đối tượng bất kì cấp phát động và tham số thứ hai kiểu integer đại diện cho vòng đời đối tượng:

template <typename T>

void SetLongevity(T\* pDynObject, unsigned int longevity);

SetLongevity() đảm bảo pDynObject sẽ sống lâu hơn những đối tượng có vòng đời ngắn hơn. Khi ứng dụng thoát, tất cả đối tượng đăng ký với SetLongevity() sẽ được xóa theo thứ tự vòng đời giảm dần.

Bạn không thế áp dụng SetLongevity() cho những đối tượng có vòng đời kiểm soát bởi trình biên dịch như:

* Đối tượng global
* Đối tượng static
* Đối tượng automatic

Vì trình biên dịch đã tự sinh ra code hủy chúng rồi, việc gọi SetLongevity() cho các đối tượng trên sẽ hủy chúng lần thứ 2

SetLongevity() chỉ áp dụng các đối tượng cấp phát động với new, và luôn đảm bảo không bao giờ gọi delete cho các đối tượng này.

Class DependencyManager làm nhiệm vụ kiểm soát tính lệ thuộc giữa các đối tượng:

class DependencyManager

{

public:

template <typename T, typename U>

void SetDependency(T\* dependent, U& target);

//...

};

Hàm hủy DependencyManager sẽ hủy các đối tượng theo trật tự dependent rồi mới tới target

GIải pháp DependencyManager có một nhược điểm lớn: cả hai đối tượng dependent và target phải tồn tại.

Nghĩa là nếu bạn cố thiết lập tính lệ thuộc giữa Keyboard và Log, bạn vẫn sẽ phải tạo đối tượng Log, dù không cần.

Không thể thiết lập mối liên hệ giữa Keyboard và Log trong hàm dựng của Log vì Keyboard lệ thuộc vào định nghĩa class của Log (Keyboard dùng Log) và Log lệ thuộc vào định nghĩa class Keyboard (Log set dependency), đây là mô hình lệ thuộc vòng tròn (circular dependency) nên tránh.

Trở về với học thuyết về vòng đời: SetLongevity() có tương tác với atexit(), ví dụ định nghĩa chuỗi hàm hủy được gọi:

class SomeClass {

//...

};

int main()

{

//Tạo đối tượng pObj2 và đăng ký vòng đời

SomeClass\* pObj1 = new SomeClass;

SetLongevity(pObj1, 5);

// Tạo đối tượng static có vòng đời tuân theo luật C++

static SomeClass obj2;

//Tạo đối tượng pObj3 có vòng đời lâu hơn pObj1

SomeClass\* pObj3 = new SomeClass;

SetLongevity(pObj3, 6);

// Các đối tượng sẽ hủy thế nào

}

Hàm main tạo đối tượng cấp phát động và đăng ký vòng đời theo luật C++. Định nghĩa trật tự hủy hợp lý cho 3 đối tượng trên là rất khó vì bên cạnh việc dùng atexit(), chúng ta không có bất kì phương tiện nào để xử lí stack.

Chúng ta nên áp dụng chính sách thiết kế an toàn sau:

* Mỗi hàm gọi tới SetLonggevity luôn phải gọi tới atexit
* Hàm hủy của đối tượng có vòng đời thấp hơn phải đặt trên hàm hủy đối tượng sống lâu hơn.
* Hàm hủy các đối tượng có cùng vòng đời tuân theo luật C++ : khởi tạo sau cùng thì hủy trước

Hàm hủy 3 đối tượng trong hàm main có thứ tự: \*pObj1, obj2,\*pObj3.

1. Hàm gọi SetLongevity() đầu tiên: đăng ký atexit() cho hàm hủy \*pObj3
2. Hàm gọi SetLongevity() thứ 2: đăng ký atexit() cho hàm hủy \*pObj1

SetLongevity cho phép lập trình viên quản lý vòng đời đối tượng và là cách nhanh và tốt nhất để áp dụng luật quản lý vòng đời của C++.

Tuy nhiên, vì quá mạnh mẽ nên sẽ nguy hiểm. Điều quan trọng là luôn tuân theo luật:

**Đối tượng dùng chung phải sống lâu hơn các đối tượng sử dụng.**

1. Kiểm soát life time Object

SetLongevity duy trì một queue ẩn chứa các con trỏ hàm truyền vào

* Điểm cốt lõi là thứ tự các phần tử trong queue tuân theo chỉ số vòng đời
* Queue hoạt động giống như stack: phần tử cuối pop ra trước
* Các đối tượng cùng chỉ số vòng đời tuân theo luật tạo sau thì hủy trước.
* Không thể dùng std::priority\_queue vì nó không đảm bảo thứ tự của phần tử có cùng **longevity**

Các phần tử có cấu trúc dữ liệu gồm con trỏ giám sát vòng đời LifetimeTracker: triển khai hàm hủy ảo và toán tử so sánh, các subclass sẽ phải override hàm hủy.

namespace Private

{

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\_;

};

virtual LifetimeTracker::~LifetimeTracker() {}

}

Mục đích của hàm friend LifetimeTracker::Compare là để so sánh các phẩn tử trong hàng đợi theo chỉ số **longevity**.

Chúng ta chỉ duy trì duy nhất một đối tượng kiểu Tracker, để tránh vấn đề hiện tại của Singleton, chúng ta chỉ dùng mảng cấp phát động cho LifetimeTracker :

namespace Private

{

typedef LifetimeTracker\*\* TrackerArray; // định nghĩa hàng đợi TrackerArray

extern TrackerArray pTrackerArray; // khai báo hàng đợi ưu tiên pTrackerArray

extern unsigned int elements; // số phần tử hàng đợi

}

SetLongevity quản lý hàng đợi ưu tiên pTrackerArray :

* Dùng cấp phát head của C để mở rộng queue như vẫn giữ được trạng thái dữ liệu trên queue.
* Nhiệm vụ của SetLongevity: tạo một đối tượng giám sát, thêm vào stack và đăng ký vào atexit

Thực ra SetLongevity chỉ dùng std::realloc thay cho cả std::malloc và std::free, vì std::realloc có hành vi giống với std::malloc khi gọi cho con trỏ null và giống với std::free khi gọi với vùng nhớ kích thước zero.

Đoạn code sau triển khai đối tượng hàm gọi tới hàm hủy đối tượng và subclass ConcreteLifetimeTracker :

//Phương thức template trợ giúp gọi hàm hủy của mọi đối tượng

template <typename T>

struct Deleter

{

static void Delete(T\* pObj) { delete pObj; }

};

// class template giám sát vòng đời cho mọi đối tượng

template <typename T, typename Destroyer>

class ConcreteLifetimeTracker : public LifetimeTracker

{

public:

ConcreteLifetimeTracker(T\* p, unsigned int longevity, Destroyer d):

LifetimeTracker(longevity),

pTracked\_(p),

destroyer\_(d)

{}

virtual ~ConcreteLifetimeTracker()

{

destroyer\_(pTracked\_);

}

private:

T\* pTracked\_; // con trỏ tới đối tượng

Destroyer destroyer\_; // Đối tượng hàm gọi hàm hủy cho con trỏ pTracked

};

SetLongevity() tạo ConcreteLifetimeTracker tương ứng và cho vào queue pTrackerArray:

static void AtExitFn();

template <typename T, typename Destroyer>

void SetLongevity(T\* pDynObject, unsigned int longevity, Destroyer d = Private::Deleter<T>::Delete)

{

TrackerArray pNewArray =

static\_cast<TrackerArray>(std::realloc(pTrackerArray, sizeof(T) \* (elements + 1)));

if (!pNewArray)

throw std::bad\_alloc();

pTrackerArray = pNewArray;

LifetimeTracker\* p = new ConcreteLifetimeTracker<T, Destroyer>(pDynObject, longevity, d);

TrackerArray pos = std::upper\_bound(pTrackerArray, pTrackerArray + elements, longevity,Compare);

std::copy\_backward(pos, pTrackerArray + elements, pTrackerArray + elements + 1);

\*pos = p;

++elements;

std::atexit(ConcreteLifetimeTracker::AtExitFn);

}

Sử dụng **std::upper\_bound** và **std::copy\_backward** trong thư viện **<algorithm>**giúp đoạn code dễ đọc hơn.

Đối tượng có longevity lớn hơn sẽ nằm đầu mảng, nếu cùng longevity thì theo nguyên tắc tạo sau hủy trước.

ConcreteLifeTimeTracker có **longevity** nhỏ nhất (cuối mảng) sẽ hủy trước. Khi đó, hàm hủy của ConcreteLifeTimeTracker sẽ gọi hàm hủy của đối tượng.

Triển khai của AtExitFn:

#include <cassert>

using namespace Private;

template <typename T, typename Destroyer>

static void ConcreteLifetimeTracker::AtExitFn()

{

assert(elements > 0 && pTrackerArray != 0);

// Lấy phần tử đầu tiên trên stack

LifetimeTracker\* pTop = pTrackerArray[elements - 1];

// Xóa đối tượng khỏi stack, Don't check errors-realloc with less memory can't fail

pTrackerArray = static\_cast<TrackerArray>

(std::realloc(pTrackerArray, sizeof(T) \* --elements));

// Hủy phần tử

delete pTop;

}

Mẹo vặt ở đây là AtExitFn() sẽ pop phần tử đầu stack, dùng std::realloc giảm kích thước queue và **elements** trước khi xóa vì quá trình hủy có thể sinh ra đối tượng nào đó đẩy vào hàng đợi.

Cấu trúc dữ liệu và AtExitFn được che giấu trong Private namespace, người dùng chỉ tương tác với SetLongevity().

Singleton dùng cơ chế kiểm soát life time triển khai theo một cách chung như sau:

class Log

{

public:

static void Create()

{

// Tạo đối tượng

pInstance\_ = new Log;

// Set vòng đời

SetLongevity(\*this, longevity\_);

}

// … Bỏ qua triển khai còn lại …

private:

// Định nghĩa giá trị cố định cho vòng đời

static const unsigned int longevity\_ = 2;

static Log\* pInstance\_;

};

Nếu triển khai Keyboard và Display theo cách trên với chỉ số longevity\_ là 1, thì Singleton Log đảm bảo hủy sau Keyboard và Display.

##### Môi trường đa luồng

1. Race condition khi khởi tạo

Giả sử Singleton được truy xuất trong môi trường đa luồng với triển khai như sau:

Singleton& Singleton::Instance()

{

if (!pInstance\_) // 1

{

pInstance\_ = new Singleton; // 2

}

return \*pInstance\_; // 3

}

Race condition xảy ra tại #1 và #2, tiến trình đọc và khởi tạo tài nguyên cho con trỏ Singleton không đồng bộ với nhau.

Trong khi thread #2 đang khởi tạo đối tượng thì thread #1 có thể check pInstance đang bằng null tại #1 và tiếp tục khởi tạo cho pInstance lần thứ 2.

Đối tượng Singleton là tài nguyên chia sẻ global và những tài nguyên như vậy luôn gặp rắc rối với race condition.

1. Đồng bộ quá trình khởi tạo

Giải pháp chống race condition khá đơn giản là đặt khóa ngay đầu hàm getInstance:

Singleton& Singleton::getInstance()

{

std::unique\_lock<std::mutex> lk(mutex\_);

if (!pInstance\_)

{

pInstance\_ = new Singleton;

}

return \*pInstance\_;

}

Tất cả các thread sẽ phải dừng tại hàm dựng của std::unique\_lock để pInstance có thể khởi tạo.

Dẫu vậy giải pháp trên đúng nhưng lại thiếu hiệu quả, quá trình đồng bộ chỉ cần khi khởi tạo. Quá trình truy xuất sau đó chỉ có một thread lấy được đối tượng Singleton.

Tiến trình truy xuất tài nguyên có dùng khóa và mở khóa thường tốn nhiều chi phí (thời gian), ý tưởng là nếu đối tượng đã khởi tạo thì không cần khóa khi truy xuất nữa:

Singleton& Singleton::Instance()

{

if (!pInstance\_)

{

std::unique\_lock<std::mutex> lk(mutex\_); // #1 Race condition

pInstance\_ = new Singleton;

}

return \*pInstance\_;

}

Vấn đề chi phí thời gian đã mất, nhưng race condition sẽ xảy ra tại #1 khi đối tượng Singleton chưa được khởi tạo.

1. Double-Checked Locking Pattern

Pattern Double-Checked Locking đưa ra bởi Doug Schmidt và Tim Harrison để giảm thời gian truy xuất tài nguyên dùng chung.

Ý tưởng: Kiểm tra tài nguyên, khóa và kiểm tra tài nguyên lần nữa.

Triển khai:

Singleton& Singleton::Instance()

{

if (!pInstance\_) // #1

{ // #2

std::unique\_lock<std::mutex> lk(mutex\_); // #3

if (!pInstance\_) // #4

{

pInstance\_ = new Singleton;

}

}

return \*pInstance\_;

}

Tình huống đối tượng singleton chưa khởi tạo : thread đầu tiên sẽ chiếm khóa tại #3, trong khi các thread còn lại sẽ phải dừng tại #2.

Đối tượng Singleton khởi tạo thành công và các thread còn lại sẽ rẽ nhánh tại #4 để trả về ngay.

Thống kê các lần test cho pattern double-checking:

* Lần 1: Đối tượng Singleton đã có sẵn thì chạy khá nhanh
* Lần 2: Cần khởi tạo Singleton thì hơi chậm.

Kết luận: Double checking đáp ứng tốc độ truy xuất Singleton, không có race condition.

Tuy nhiên nhiều lập trình viên đa luồng có kinh nghiệm sẽ nhận ra ngay rằng pattern này thực tế sẽ không hiệu nghiệm.

Trong môi trường đa nhân, hành vi ghi dữ liệu vào vùng nhớ của nhân này sẽ không thấy được bởi nhân khác.

Cụ thể là phép gán của pInstance\_ thực hiện bởi một nhân khi đối tượng Singleton được khởi tạo thành công có thể không thấy được và một số thread sẽ tiếp tục khởi tạo lại Singleton.

Double-checking pattern đang có lỗi cho hệ thống đa nhân, bạn phải đọck kỹ tài liệu trình biên dịch trước khi dùng Double-checking pattern.

Thông thường một platform sẽ cung cấp một cơ chế thay thế để truy xuất tuần tự tới vùng nhớ. Tối thiểu nên đặt thuộc tính **volatile** cho con trỏ pInstance\_, trình biên dịch tốt sẽ sinh ra code chuẩn cho các đối tượng **volatile**

##### Chính sách cho SingletonHolder

Chính sách Singleton sẽ tập trung vào các vấn đề tạo đối tượng, lifetime và thread:

1. Creator: có nhiều cách triển khai, thường dùng toán tử new. Tách riêng quá trình khởi tạo là điều cốt lỗi, nhất là khi dùng đối tượng abstract.
2. Lifetime: định nghĩa các chính sách sau:
   1. Theo luật C++: Tạo sau thì hủy trước.
   2. Tính tuần hoàn: Phoenix singleton
   3. Kiểm soát: singleton với longevity
   4. Infinite ( "leaking" singleton, đối tượng không bao giờ bị hủy)
3. ThreadingModel: Dùng đơn luồng hay đa luồng (dùng mutex và Checked Locking pattern), hoặc dùng các thread model không portable (WinAPI, Boost…).

Triển khai Singleton phải duy trì tính duy nhất, các chính sách trên đảm bảo định nghĩa cho Singleton

1. Chính sách tạo đối tượng

Chính sách tạo phải bao gồm hàm tạo và hảm hủy. Class Creator<T> sẽ đại diện cho chính sách tạo object, Createor<T> phải hỗ trợ:

T\* pObj = Creator<T>::Create();

Creator<T>::Destroy(pObj);

Lưu ý rằng Create và Destroy phải là 2 thành viên static của Creator. Singleton không giữ Creator – nếu không sẽ gặp rắc rồi với vấn đề vòng đời Singleton.

1. Chính sách Lifetime

Chính sách Lifetime quan trọng phải kiểm soát được quá trình hủy đối tượng Singleton tạo bởi chính sách Creation,

Tính năng của Liftime policy giúp hủy Singleton object tại thời điểm định trước trong suốt vòng đời của ứng dụng. Ngoài ra, Lifetime quyết định hành vi phù hợp nếu ứng dụng vi phạm luật vòng đời đối tượng Singleton. Tóm lại:

* Nếu bạn cần hủy Singleton theo luật C++, Lifetime dùng cơ chế tương tự như atexit
* Đối với Phoenix singleton, Lifetime vẫn dùng cơ chế kiểu atexit nhưng có ngoại lệ khi tái sinh đối tượng Singleton
* Đối với Singleton với chỉ số longevity, Lifetime gọi tới SetLongevity như phần trước đã trình bày
* Đối với Singleton có lifetime vô hạn thì không cần triển khai gì thêm.

Tóm lại, Lifetime policy thể hiện qua hai hàm:

* ScheduleDestruction: đảm bảo Singleton được hủy theo đúng kế hoạch
* OnDeadReference: đưa ra hành vi phù hợp khi tham chiếu Singleton chết

Class Lifetime<T> đại diện cho chính sách vòng đời:

void (\*pDestructionFunction)();

//con trỏ hàm làm tác vụ hủy đôi tượng

template<class T>

void Lifetime<T>::ScheduleDestruction(pDestructionFunction);// nhận tham số là con trỏ hàm thực hiện tác vụ hủy đối tượng

template<class T>

void Lifetime<T>::OnDeadReference();//ném ngoại lệ trong tất cả trường hợp, riêng Phoenix Singleton thì không cần.

Với triển khai này, chúng ta có thể kết hợp Lifetime policy với Createion policy, đừng quên rằng Lifetime không đề cập tới phương thức hủy, đây là đặc quyền của Creator, Lifetime chỉ quan tâm tới thời điểm nào thì phương thức hủy phải được thực thi.

1. Chính sách ThreadModel

The ThreadingModel policy is the one described in the appendix. SingletonHolder không hỗ trợ locking ở tầng-đối tượng, chỉ áp dụng ở tầng-class, vì chúng ta chỉ luôn có một đối tượng mà thôi.

##### Triển khai SingletonHolder

SingletonHolder sẽ là một class template, như đã thảo luận trong phần thiết kế,  [**mỗi policy là một tham số template**](#_Triển_khai_policy) .

Tất cả tham số liên quan tới policy sẽ dùng macro SingletonPolicy cho đơn giản, ngoài ra có tham số T đại diện cho kiểu đối tượng dùng dịch vụ Singleton.

Bản thân SingletonHolder không phải là một Singleton.SingletonHolder chỉ cung cấp hành vi và sự quản lý cho đối tượng Singleton.

#define SingletonPolicy template<class> class CreationPolicy = CreateUsingNew,\

template<class> class LifetimePolicy = DefaultLifetime,\

template<class> class ThreadingModel = SingleThreaded

template<class T, SingletonPolicy>

class SingletonHolder

{

public:

static T& getInstance();

private:

// Helpers

static void DestroySingleton();

// Protection

SingletonHolder();

...

// Data

typedef ThreadingModel<T>::VolatileType InstanceType;

static InstanceType\* pInstance\_;

static bool destroyed\_;

};

Kiểu **pInstance**\_ không phải T\*, thay vào đó là kiểu ThreadingModel<T>::VolatileType\*.

ThreadingModel<T>::VolatileType mở rộng định nghĩa vừa là T hay vừa là volatile T, tùy thuộc vào mô hình thread sử dụng.

Từ khóa **volatile** áp dụng cho kiểu hàm ý nói với trình biên dịch rằng giá trị của kiểu đó có thể bị thay đổi bởi nhiều threads, nên trình biên dịch sẽ tránh các thao tác tối ưu (như giữ giá trị trong thanh ghi nội bộ của nó), giúp code đa luồng chạy ổn định.

Việc định nghĩa **pInstance**\_ thành kiểu volatile T\*: để làm việc với code đa luồng (tham khảo thêm tài liệu trình biên dịch) và không bị đụng chạm không code đơn luồng.

Mặt khác, trong chế độ đơn luồng, bạn nên tận dụng ưu thế của việc tối ưu thì nên chọn kiểu T\* chop **pInstance**\_. Đó là lý do tại sao kiểu của **pInstance**\_ quyết định bởi chính sách ThreadingModel.

Nếu ThreaddingModel là chính sách đơn thread, nó đơn giản chỉ định nghĩa VolatileType như sau:

template <class T>

class SingleThreaded

{

// ...

public:

typedef T VolatileType;

};

Chính sách đa luồng phải định nghĩa T với **volatile**.

Triển khai của getInstance thể hiện sự liên kết của 3 chính sách:

template<class T, SingletonPolicy>

T& SingletonHolder<class T, SingletonPolicy>::getInstance()

{

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\_;

}

SingletonHolder chỉ có một phương thức truy xuất global là getInstance(), phương thức này là một cái vỏ cho

CreationPolicy, LifetimePolicy, và ThreadingModel.

ThreadingModel<T> policy class public lớp lồng Lock. Chừng nào Lock chưa hủy thì tất cả thread khác đang cố tạo đối tượng Lock sẽ bị khóa.

**DestroySingleton** đơn giản chỉ hủy đối tượng Singleton, dọn dẹp tài nguyên và set **destroyed**\_ thành true.

SingletonHolder không bao giờ gọi tới **DestroySingleton**() trừ khi nó truyền địa chỉ hàm tới LifetimePolicy<T>::ScheduleDestruction:

template<class T, SingletonPolicy>

void SingletonHolder<class T, SingletonPolicy>::DestroySingleton()

{

assert(!destroyed\_);

CreationPolicy<T>::Destroy(pInstance\_);

pInstance\_ = 0;

destroyed\_ = true;

}

LifetimePolicy cần đủ thông tin để định nghĩa hành vi theo quan điểm:

* C++ rules: LifetimePolicy<T>::ScheduleDestruction truyền địa chỉ của DestroySingleton tới atexit. OnDeadReference ném ngoại lệ std::logic\_error.
* Hồi sinh (Phoenix singleton): Tương tự như C++ rule, ngoại trừ OnDeadReference không ném ngoại lệ mà tái tạo lại đối tượng
* Người dùng tự kiểm soát (singleton với longevity): LifetimePolicy<T>::ScheduleDestruction gọi tới SetLongevity((GetLongevity(pInstance)).
* infinite: LifetimePolicy<T>::ScheduleDestruction không làm gì cả

SingletonHolder trao vấn đề xử lí tham chiếu chết cho LifetimePolicy: nếu SingletonHolder::getInstance phát hiện tham chiếu chết, nó gọi LifetimePolicy::OnDeadReference. Khi OnDeadReference return, getInstance tiếp tục với đối tượng tái tạo.

Tóm lại, OnDeadReference nên ném ngoại lệ hoặc ngắt ứng dụng nếu bạn không muốn dùng Phoenix Singleton. Ngược lại, nếu dùng Phoenix Singleton thì OnDeadReference không làm gì cả.

Phần tiếp theo sẽ giới thiệu cơ chế ủy nhiệm cho 3 policy

##### Stock Policy Implementations

Việc phân tích để triển khai theo policy nào là khá khó. Giai đoạn triển khai thì dễ dàng hơn, tóm gọn trong bảng sau:

***(Predefined Class Template: là các tham số template mặc định đại diện cho 1 policy)***

|  |  |  |
| --- | --- | --- |
| **Predefined Policies cho SingletonHolder** | | |
| ***Policy*** | ***Predefined Class Template*** | ***Comment*** |
| Creation | **CreateUsingNew** | Tạo đối tượng dùng toán tử new và hàm dựng mặc định. |
| CreateUsingMalloc | Tạo đối tượng dùng std::malloc và hàm dựng mặc định. |
| CreateStatic | Tạo đối tượng dùng vùng nhớ static. |
| Lifetime | **DefaultLifetime** | Quản lý lifetime theo luật C++. |
| PhoenixSingleton | Tương tự DefaultLifetime, nhưng cho phép tái tạo đối tượng Singleton |
| SingletonWithLongevity | Gán longevity cho đối tượng Singleton.  Gọi GetLongevity chop Instance\_ để trả về longevity của Singleton |
| NoDestroy | Không hủy đối tượng Singleton. |
| ThreadingModel | **SingleThreaded** | Tham khảo appendix về threading Models. |
| ClassLevelLockable |  |

##### Sử dụng SingletonHolder

The SingletonHolder class template does not provide application-specific functionality. It merely

provides Singleton-specific services over another class—T in the code in this chapter. We call T the client

class.

The client class must take all the precautionary measures against unattended construction and destruction:

The default constructor, the copy constructor, the assignment operator, the destructor, and the address-of

operator should be made private.

If you take these protective measures, you also have to grant friendship to the Creator policy class that

you use. These protective measures and the friend declaration are all the changes needed for a class to

work with SingletonHolder. Note that these changes are optional and constitute a trade-off between

the inconvenience of touching existing code and the risk of spurious instances.

The design decisions concerning a specific Singleton implementation usually are reflected in a type

definition like the following one. Just as you would pass flags and options when calling some function, so

you pass flags to the type definition selecting the desired behavior.

class A { ... };

typedef SingletonHolder<A, CreateUsingNew> SingleA;

// from here on you use SingleA::Instance()

Providing a singleton that returns an object of a derived class is as simple as changing the Creator policy

class:

class A { ... };

class Derived : public A { ... };

template <class T> struct MyCreator : public CreateUsingNew<T>

{

static T\* Create()

{

return new Derived;

}

};

typedef SingletonHolder<A, StaticAllocator, MyCreator> SingleA;

Similarly, you can provide parameters to the constructor or use a different allocation strategy. You can

tweak Singleton along each policy. This way you can largely customize Singleton, while still

benefiting from default behavior when you want to.

The SingletonWithLongevity policy class relies on you to define a namespace-level function

GetLongevity. The definition would look something like this:

inline unsigned int GetLongevity(A\*) { return 5; }

This is needed only if you use SingletonWithLongevity in the type definition of SingleA.

The intricate KDL problem has driven our tentative implementations. Here's the KDL problem solved by

making use of the Singleton class template. Of course, these definitions go to their appropriate header

files.

class KeyboardImpl { ... };

class DisplayImpl { ... };

class LogImpl { ... };

...

inline unsigned int GetLongevity(KeyboardImpl\*) { return 1; }

inline unsigned int GetLongevity(DisplayImpl\*) { return 1; }

// The log has greater longevity

inline unsigned int GetLongevity(LogImpl\*) { return 2; }

typedef SingletonHolder<KeyboardImpl, SingletonWithLongevity> Keyboard;

typedef SingletonHolder<DisplayImpl, SingletonWithLongevity> Display;

typedef SingletonHolder<LogImpl, SingletonWithLongevity> Log;

It's a rather easy-to-grasp and self-documenting solution, given the complexity of the problem.

##### Thảo luận

Chương này đề cập tới triển khai phổ biến nhất của Singleton trên C++. Nhờ khả năng hỗ trợ tốt của ngôn ngữ nên rất dễ để duy trì tính độc nhất của đối tượng. Vấn đề phức tạp nhất là quản lý vòng đời cho đối tượng Singleton, đặc biệt là thời điểm để hủy Singleton.

Việc phát hiện sự truy xuất Singleton sau khi bị hủy là dễ dàng và không tốn kém, kiểm tra tham chiếu bị chết bắt buộc là một phần của bất kì triển khai Singleton nào.

Chúng ta đã đề cập tới 4 hình thái là:

* Compiler-controlled singleton
* Phoenix singleton
* Singleton với longevity
* Leaking singleton

Mỗi loại có những ưu và nhược điểm khác nhau.

Xảy ra một vấn đề nghiêm trọng về đa luồng trong Singleton pattern, Double-Checked Locking pattern là một cách hữu hiệu để triển khai thread-safe singleton.

Triển khai Singleton rất đa dạng, chúng ta nên dựa vào 3 policy: Creation, Lifetime và ThreadingModel. Tất cả được đóng

Abstract Factory, Builder,Prototype , Facade và State Pattern có thể triển khai dùng Singleton.

Singleton Pattern cũng là một trong pattern bị hiểu sai nhiều, Singleton không phải để dùng thay cho biến global.

Singleton là cách truy xuất và quản lý truy xuất tới tài nguyên dùng chung.

##### Check list

* Thiết lập thuộc tính private và static cho đối tượng duy nhất của class
* Định nghĩa phương thức truy xuất là public và static
* Dùng "lazy initialization" để khởi tạo đối tượng lần đầu tiên trong hàm truy xuất trong trường hợp dùng con trỏ đối tượng Singleton.
* Tất cả hàm dựng phải là protected hay private.
* Người dùng chỉ được phép dùng hàm truy xuất để tương tác với class singleton.

### Factory Pattern

Factory pattern là một trong pattern tạo đối tượng dùng nhiều nhất trên Java.

Với Factory pattern, việc tạo đối tượng sẽ bị che giấu tới người sử dụng và sử dụng một interface gọi là Factory để tạo đối tượng cần dùng.

Factory quan niệm đối tượng có thể tái sử dụng, hạn chế dùng toán tử **new**.

##### Factory class

1. Tiêu chí

Một framework cần phải chuẩn hóa kiến trúc model cho khoảng ứng dụng nào đó, nhưng phải cho phép bản thân ứng dụng định nghĩa được phạm vi đối tượng của chính nó để khởi tạo.

Cách Factory tạo đối tượng giống hệt như Template. Một superclass xác định tất cả hành vi có tính tổng quá và tiêu chuẩn (sử dụng một “place holders” thuần ảo cho bước tạo), sau đó ủy nhiệm chi tiết cách tạo đối tượng cho subclass – cung cấp bởi người dùng.

Factory giúp các thiết kế tùy biến hơn và ít phức tạp. Các pattern khác đòi hỏi new class, trong khi Factory chỉ cần dùng toán tử new.

Người ta thường dùng Factory như cách chuẩn để tạo đối tượng, nhưng nó sẽ không cần thiết nếu:

* Class được khởi tạo không bao giờ thay đổi.
* Việc khởi tạo có thể dễ dàng override bởi subclass

Factory khá giống với Abstract Factory nhưng không có nhấn mạnh về phân cấp thừa kế

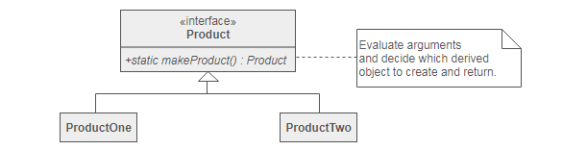
Factory thường được nói rõ trong thiết kế framework, sau đó triển khai bởi người dùng framework này.

1. Cấu trúc

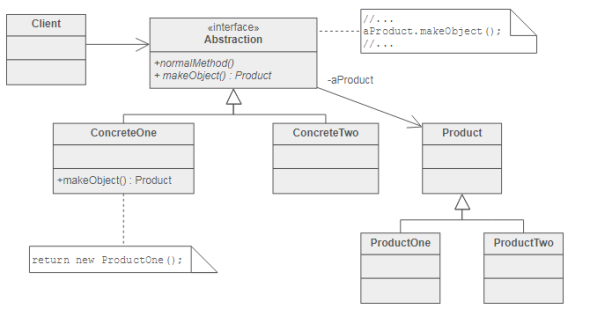
Việc triển khai Factory sẽ chồng lấn nhiều tới khái niệm của Abstract Factory, nhưng chúng ta sẽ chỉ tập trung vào khía cạnh của Factory mà thôi.

Sự phổ biến ngày càng lớn của Factory tập trung bởi yếu tố: nhu cầu một phương thức static của class trả về đối tượng của class.

Không giống như hàm dựng, một đối tượng đang tồn tại có thể tái sử dụng, thay vì dùng **new** tạo ra đối tượng mới, Factory có tên mô tả dễ liên tưởng hơn



Người dùng hoàn toàn tách biệt với hoạt động khởi tạo của lớp dẫn xuất. Việc khởi tạo trở nên đa dạng hơn



Factory Method định nghĩa một interface cho việc đối tượng, nhưng chi tiết khởi tạo thế nào thì ủy nhiệm cho subclass

Chúng ta sẽ triển khai Factory để tạo ra các đối tượng dẫn xuất của class Shape:



1. Tạo Interface Shape

Tạo Interface Shape

class Shape {

public:

virtual void draw() = 0;

};

1. Tạo concreate class

Tạo ra các phiên bản concrete class triển khai từ interface Shape

class Rectangle : public Shape {

public:

virtual void draw() {

std::cout<<"Inside Rectangle::draw() method.";

}

};

Square

class Square : public Shape {

public:

virtual void draw() {

std::cout<<"Inside Square::draw() method.";

}

};

Circle

class Circle : public Shape {

public:

virtual void draw() {

std::cout<<"Inside Circle::draw() method.";

}

};

1. Tạo ShapeFactory

Tạo class ShapeFactory làm nhiệm vụ tạo đối tượng theo thông tin đầu vào là kiểu enum eSHAPE, dùng phương thức getShape() để lấy đúng đối tượng của Shape.

Chúng ta khai báo thuộc tính static để có thể gọi hàm mà không cần khởi tạo đối tượng ShapeFactory

enum eSHAPE{

CIRCLE = 0,

RECTANGLE,

SQUARE

};

class ShapeFactory {

//use getShape method to get object of type shape

public:

static Shape\* getShape(eSHAPE shape){

Shape\* ret = nullptr;

if(shape == eSHAPE::CIRCLE){

ret = new Circle();

} else if(shape == eSHAPE::RECTANGLE){

ret = new Rectangle();

} else if(shape == eSHAPE::SQUARE){

ret = new Square();

}

return ret;

}

};

1. Sử dụng Factory tạo object

int main(std::string args) {

//get an object of Circle and call its draw method.

std::shared\_ptr<Shape> shape1(ShapeFactory::getShape(eSHAPE::CIRCLE));

//call draw method of Circle

shape1->draw();

//get an object of Rectangle and call its draw method.

std::shared\_ptr<Shape> shape2(ShapeFactory::getShape(eSHAPE::RECTANGLE));

//call draw method of Rectangle

shape2->draw();

//get an object of Square and call its draw method.

std::shared\_ptr<Shape> shape3(ShapeFactory::getShape(eSHAPE::SQUARE));

//call draw method of Square

shape3->draw();

return 0;

}

Kết quả:

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

1. Checklist

* Nếu bạn đang có cây phân cấp thừa kế có sử dụng tính đa hình, thì hãy cân nhắc thêm vào khả năng tạo đối tượng đa hình, bằng cách định nghĩa phương thức satic factory trong lớp cơ sở.
* Thiết kế các đối số cho phương thức factory để cung cấp đầy đủ thông tin cho việc quyết định đúng lớp dẫn xuất nào cần phải khởi tạo.
* Cân nhắc tới “object pool”, cho phép các đối tượng có thể tái sử dụng thay vì tạo ra và hủy một cách ngẫu hứng
* Cân nhắc để tất hàm dựng là private hay protected.

1. Thảo luận

Abstract Factory thường triển khai cùng với Factory, nhưng chúng có thể triển khai cùng với Prototype.

Factory Methos thường được gọi bên trong Template Methods

Factory Method: tạo đối tượng qua thừa kế trong khi Prototype tạo đối tượng qua ủy nhiệm.

Thông thường, ý tưởng thiết kế thường bắt đầu từ Factory Method (ít phức tạp, tùy biến, subclass nhiều) và mở rộng tới Abstract Factory, Prototype, hay Builder (tùy biến hơn, phức tạp hơn).

Prototype không cần phải có subclassing, nhưng nó yêu cầu phải có tiến trình khởi tạo. Factory yêu cầu subclassing nhưng không cần khởi tạo.

Điểm tốt của Factory là nó có thể return cùng một đối tượng nhiều lần, hoặc return subclass mà không phải chính xác đối tượng cần.

Một số Factory gợi ý rằng tất cả hàm dựng tuyệt đối nên là private hay protected: tức không cho phép các phương tiện nào khác ngoài Factory có quyền tạo đối tượng hay thu hồi đối tượng cũ và tái sử dụng.

Toán tử new được xem là có hại: có sự khác biệt giữa yêu cầu đối tượng và tạo mới, toán tử new luôn tạo mới, và thất bại để bao đóng quá trình tạo. Factory đảm bảo cho tính bao đóng, cho phép một đối tượng được yêu cầu mà không cần phải móc nối với hành vi tạo.

##### Abstract Factory Pattern

Mục đích là cung cấp interface Abstract Factory để tạo ra các subclass Factory, Abstract Factory còn được gọi là factory của các factory

Người dùng sử dụng mỗi factory để tạo đối tượng tương ứng với Factory pattern của nó.

Cây thừa kế sẽ bao đóng nhiều platforms, và xây dựng nên một bộ sản phẩm.

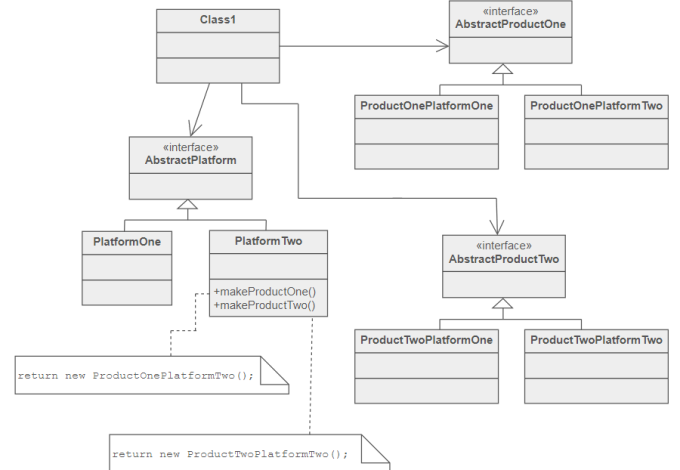
1. Tiêu chí

Nếu một ứng dụng nhắm tới portable, nó cần bao đóng sự lệ thuộc platform, platform ở đây bao gồm: windowing system, operating system, database, ...

Việc xác định platform thường không được xác định từ trước, nên rất nhiều câu lệnh #ifdef nhằm hỗ trợ cho tất cả platform mọc như nấm trong source code.

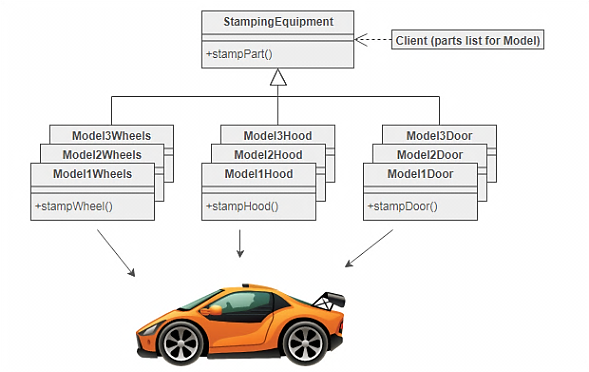
1. Cấu trúc

Abstract Factory định nghĩa Factory Method cho từng dòng sản phẩm. Mỗi Factory Method bao đóng toán tử new và concrete class, platform chỉ định, class sản phẩm. Mỗi platform sau đó được làm mẫu bởi lớp con của Factory



Mục đích của Abstract Factory là cung cấp interface cho việc tạo ra các dòng đối tượng, mà không quan tâm tới concrete class nào bên dưới.

Pattern Abstract Factory được sử dụng điển hình trong công nghệ làm nhãn hiệu thép tấm xe hơi tại Nhật bản



Trang bị nhãn là một Abstract Factory tự động tạo ra các phần thân, công nghệ này cũng dùng cho nhãn cửa phải, trái, chắn bùn trái, phải, nắp xe…

Nhờ concrete class mà việc thay đổi nhãn thép tự động diễn ra trong vòng 3 phút.

Chúng ta sẽ triển khai Abstract Factory cho công nghệ sản xuất stamp.

1. Abstract product và concrete class

Chúng ta triển khai interface và các concrete của StampDoor, StampHood và StampWheel:

StampDoor.h

//interface StampDoor

class StampDoor{

public:

virtual void stampInfo() = 0;

};

//concrete class for StampDoor

class StampDoorModel1: public StampDoor{

public:

virtual void stampInfo(){

std::cout<<"Stamp door model 1 is equipped" << "\n";

}

};

class StampDoorModel2: public StampDoor{

public:

virtual void stampInfo(){

std::cout<<"Stamp door model 2 is equipped" << "\n";

}

};

class StampDoorModel3: public StampDoor{

public:

virtual void stampInfo(){

std::cout<<"Stamp door model 3 is equipped" << "\n";

}

};

StampHood.h

//interface StampHood

class StampHood{

public:

virtual void stampInfo() = 0;

};

//concrete class for StampHood

class StampHoodModel1: public StampHood{

public:

virtual void stampInfo(){

std::cout<<"Stamp hood model 1 is equipped" << "\n";

}

};

class StampHoodModel2: public StampHood{

public:

virtual void stampInfo(){

std::cout<<"Stamp hood model 2 is equipped" << "\n";

}

};

class StampHoodModel3: public StampHood{

public:

virtual void stampInfo(){

std::cout<<"Stamp hood model 3 is equipped" << "\n";

}

};

StampWheel.h

//interface StampWheel

class StampWheel{

public:

virtual void stampInfo() = 0;

};

//concrete class for StampWheel

class StampWheelModel1: public StampWheel{

public:

virtual void stampInfo(){

std::cout<<"Stamp wheel model 1 is equipped" << "\n";

}

};

class StampWheelModel2: public StampWheel{

public:

virtual void stampInfo(){

std::cout<<"Stamp wheel model 2 is equipped" << "\n";

}

};

class StampWheelModel3: public StampWheel{

public:

virtual void stampInfo(){

std::cout<<"Stamp wheel model 3 is equipped" << "\n";

}

};

1. Abstract Factory

Khai báo enum class qui định model và Abstract Factory

StampFactory.h

#include "stampDoor.h"

#include "stampHood.h"

#include "stampWheel.h"

//enum Model

enum class eMODEL{

MODEL1 = 0,

MODEL2,

MODEL3

};

//Abstract Factory for StampWheelFactory, StampDoorFactory and StampHoodFactory

class StampFactory{

public:

virtual StampWheel\* getStampWheel(eMODEL choice) = 0;

virtual StampDoor\* getStampDoor(eMODEL choice) = 0;

virtual StampHood\* getStampHood(eMODEL choice) = 0;

};

1. Triển khai Factory từ AbstractFactory

StampFactory.h

//Factory for StampWheel, StampDoor and StampHood

class StampWheelFactory : public StampFactory{

public:

virtual StampWheel\* getStampWheel(eMODEL choice){

StampWheel \*ret = nullptr;

switch (choice)

{

case eMODEL::MODEL1:

ret = new StampWheelModel1();

break;

case eMODEL::MODEL2:

ret = new StampWheelModel2();

break;

case eMODEL::MODEL3:

ret = new StampWheelModel3();

break;

default:

break;

}

return ret;

}

virtual StampDoor\* getStampDoor(eMODEL choice){return nullptr;}

virtual StampHood\* getStampHood(eMODEL choice){return nullptr;}

};

class StampDoorFactory : public StampFactory{

public:

virtual StampWheel\* getStampWheel(eMODEL choice){return nullptr;}

virtual StampDoor\* getStampDoor(eMODEL choice){

StampDoor \*ret = nullptr;

switch (choice)

{

case eMODEL::MODEL1:

ret = new StampDoorModel1();

break;

case eMODEL::MODEL2:

ret = new StampDoorModel2();

break;

case eMODEL::MODEL3:

ret = new StampDoorModel3();

break;

default:

break;

}

return ret;

}

virtual StampHood\* getStampHood(eMODEL choice){return nullptr;}

};

class StampHoodFactory : public StampFactory{

public:

virtual StampWheel\* getStampWheel(eMODEL choice){return nullptr;}

virtual StampDoor\* getStampDoor(eMODEL choice){return nullptr;}

virtual StampHood\* getStampHood(eMODEL choice){

StampHood \*ret = nullptr;

switch (choice)

{

case eMODEL::MODEL1:

ret = new StampHoodModel1();

break;

case eMODEL::MODEL2:

ret = new StampHoodModel2();

break;

case eMODEL::MODEL3:

ret = new StampHoodModel3();

break;

default:

break;

}

return ret;

}

};

1. Tạo Factory Producer

Triển khai class StampEquipment với phương thức thành viên static getStampFactory () để lấy đối tượng factory tương ứng với tham số enum truyền vào:

StampEquipment.h

#include "stampFactory.h"

//Factory producer - StampEquiment

enum class eSTAMP\_FACTORY{

WHEEL = 0,

HOOD,

DOOR

};

class StampEquipment{

public:

static StampFactory\* getStampFactory(eSTAMP\_FACTORY choice){

StampFactory\* ret = nullptr;

switch (choice)

{

case eSTAMP\_FACTORY::WHEEL:

ret = new StampWheelFactory();

break;

case eSTAMP\_FACTORY::HOOD:

ret = new StampHoodFactory();

break;

case eSTAMP\_FACTORY::DOOR:

ret = new StampDoorFactory();

break;

default:

break;

}

return ret;

}

};

1. Sử dụng

Ứng dụng demo sẽ sử dụng FactoryProducer để tạo ra đối tượng AbstractFactory, sau đó truyền các tham số cần thiết (*eSTAMP\_FACTORY* và *eMODEL* ) để lấy đối tượng cần dùng.

#include <memory>

#include "StampEquipment.h"

void StampEquipment\_AbstractFactoryDemo(eMODEL model){

//get stamp wheel factory

std::shared\_ptr<StampFactory> wheelFactory(StampEquipment::getStampFactory(eSTAMP\_FACTORY::WHEEL));

//get stamp wheel model 1

std::shared\_ptr<StampWheel> wheelModel1(wheelFactory->getStampWheel(model));

//get stamp info for wheel model

wheelModel1->stampInfo();

//get stamp door factory

std::shared\_ptr<StampFactory> doorFactory(StampEquipment::getStampFactory(eSTAMP\_FACTORY::DOOR));

//get stamp door model 1

std::shared\_ptr<StampDoor> doorModel1(doorFactory->getStampDoor(model));

//get stamp info for Door model

doorModel1->stampInfo();

//get stamp hood factory

std::shared\_ptr<StampFactory> hoodFactory(StampEquipment::getStampFactory(eSTAMP\_FACTORY::HOOD));

//get stamp hood model 1

std::shared\_ptr<StampHood> hoodModel1(hoodFactory->getStampHood(model));

//get stamp info for Hood model

hoodModel1->stampInfo();

}

Kết quả:

Stamp wheel model 1 is equipped

Stamp door model 1 is equipped

Stamp hood model 1 is equipped

1. Thảo luận

Cung cấp một tầng điều hướng, trừu tượng việc khởi tạo các đối tượng liên quan, lệ thuộc, cùng cây thừa kế mà không phải trực tiếp chỉ ra các concrete class của chúng.

Đối tượng factory có trách nhiệm cung cấp dịch vụ tạo đối tượng cho toàn bộ platform family.

Người dùng không giờ tạo đối tượng platform trực tiếp mà thông qua Factory

Cơ chế này khiến sự thay đổi các dòng sản phẩm dễ dàng vì các class của đối tượng Factory chỉ xuất hiện trong ứng dụng – nơi nó được khởi tạo

Ứng dụng có thể thay thế toàn bộ 1 dòng sản phẩm đơn giản bằng cáchkhởi tạo một concrete Factory khác từ Abstract Factory.

Factory thường được triển khai với Singleton.

### Object Pool Pattern

##### Mục đích

Object pool giúp tăng hiệu suất đáng kể ứng dụng, nó đặc biệt hiệu quả khi chi phí khởi tạo class là đáng kể, tần suất tạo đối tượng cao nhưng vòng đời của đối tượng lại thấp.

##### Tiền đề

Object pools (còn biết tới là resource pools) được dùng để quản lý đối tượng, một client truy xuất tới Object pool có thể tránh việc khởi tạo mới đối tượng bằng cách yêu cầu đối tượng tới pool, và pool sẽ kiểm tra xem đối tượng đã tồn tại chưa để trả về, nếu chưa thì sẽ tạo mới.

Thường pool sẽ luôn mở rộng, pool mặc định sẽ tự tạo đối tượng nếu nó đang rỗng, chúng ta có thể giới hạn số đối tượng tối đa cho pool.

Ngoài ra, Pool được thiết kế để thu hồi đối tượng không còn dùng vào trong đúng pool của nó, để tái sử dụng lại. Như vậy, Pool tái chế nên được thiết kế là singleton.

##### Thảo luận

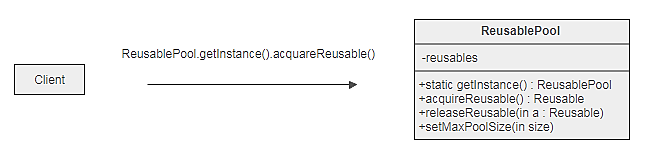
Object Pool cho phép lấy đối tượng từ pool của nó, khi những đối tượng này không còn dùng trong process, chúng sẽ được trả về pool để tái sử dụng

Tuy nhiên, chúng ta không thể để một process phải đợi lâu khi không còn đối tượng nào trong pool hoặc chờ đối tượng được giải phóng, Object pool nên khởi tạo đối tượng ngay.

Do đó phải có cơ chế để Object pool kiểm tra số lượng đối tượng và giải phóng những đối tượng không dùng theo chu kỳ.

##### Cấu trúc

Mục đích khi kết nối tới Pool là tránh việc khởi tạo đối tượng khi đã có đối tượng phù hợp trong Pool.



* **Reusable** - Instances of classes in this role collaborate with other objects for a limited amount of time, then they are no longer needed for that collaboration.
* **Client** - Instances of classes in this role use Reusable objects.
* **ReusablePool** - Instances of classes in this role manage Reusable objects for use by Client objects.

Usually, it is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the ReusablePool class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its getInstance method to get the one instance of the ReusablePool class.

A Client object calls a **ReusablePool** object's acquireReusable method when it needs a Reusable object. A ReusablePool object maintains a collection of Reusable objects. It uses the collection of Reusable objects to contain a pool of Reusable objects that are not currently in use.

If there are any Reusable objects in the pool when the acquireReusable method is called, it removes a Reusable object from the pool and returns it. If the pool is empty, then the acquireReusable method creates a Reusable object if it can. If the acquireReusable method cannot create a new Reusable object, then it waits until a Reusable object is returned to the collection.

Client objects pass a Reusable object to a ReusablePool object's releaseReusable method when they are finished with the object. The releaseReusable method returns a Reusable object to the pool of Reusable objects that are not in use.

In many applications of the Object Pool pattern, there are reasons for limiting the total number of Reusable objects that may exist. In such cases, the ReusablePool object that creates Reusable objects is responsible for not creating more than a specified maximum number of Reusable objects. If ReusablePool objects are responsible for limiting the number of objects they will create, then the ReusablePool class will have a method for specifying the maximum number of objects to be created. That method is indicated in the above diagram as setMaxPoolSize.

##### Cơ chế Object Pool

Reusable - Instances of classes in this role collaborate with other objects for a limited amount of time, then they are no longer needed for that collaboration.

Client - Instances of classes in this role use Reusable objects.

ReusablePool - Instances of classes in this role manage Reusable objects for use by Client objects.

Usually, it is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the ReusablePool class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its getInstance method to get the one instance of the ReusablePool class.

A Client object calls a ReusablePool object's acquireReusable method when it needs a Reusable object. A ReusablePool object maintains a collection of Reusable objects. It uses the collection of Reusable objects to contain a pool of Reusable objects that are not currently in use.

If there are any Reusable objects in the pool when the acquireReusable method is called, it removes a Reusable object from the pool and returns it. If the pool is empty, then the acquireReusable method creates a Reusable object if it can. If the acquireReusable method cannot create a new Reusable object, then it waits until a Reusable object is returned to the collection.

Client objects pass a Reusable object to a ReusablePool object's releaseReusable method when they are finished with the object. The releaseReusable method returns a Reusable object to the pool of Reusable objects that are not in use.

In many applications of the Object Pool pattern, there are reasons for limiting the total number of Reusable objects that may exist. In such cases, the ReusablePool object that creates Reusable objects is responsible for not creating more than a specified maximum number of Reusable objects. If ReusablePool objects are responsible for limiting the number of objects they will create, then the ReusablePool class will have a method for specifying the maximum number of objects to be created. That method is indicated in the above diagram as setMaxPoolSize.

##### Ví dụ

Object pool pattern is similar to an office warehouse. When a new employee is hired, office manager has to prepare a work space for him. She figures whether or not there's a spare equipment in the office warehouse. If so, she uses it. If not, she places an order to purchase new equipment from Amazon. In case if an employee is fired, his equipment is moved to warehouse, where it could be taken when new work place will be needed.

Object Pool example

##### Check list

* Đặt Create ObjectPool class là private array of Objects inside
* Triển khai phương thức Create acquire và release trong ObjectPool class
* Đảm bảo rằng ObjectPool là Singleton

##### Rules of thumb

Factory Method dùng để bao đóng tính logic khi tạo đối tượng, tuy nhiên nó không quản lý đối tượng sau khi tạo, trong khi object pool có thể giám sát đối tượng sau khi tạo.

Object pool thường triển khai với Singletons.

## Phần 1

### Policy thiết kế class

Chương này giới thiệu về kỹ thuật thiết kế class và chính sách class quan trọng cho phép tạo ra thư viện tái sử dụng hiệu quả, linh hoạt – như thư viện Loki.

Chính sách dựa trên một interface liên quan tới từng vấn đề cụ thể cần giải quyết. Bạn có thể triển khai policy theo nhiều cách.

Vì bạn có thể trộn nhiều policy khác nhau hay chỉ dùng một, tức có thể kết hợp nhiều hành vi bằng cách sử dụng một lượng nhỏ các components.

Policy được dùng trong nhiều chương cuốn sách này. Class template SingletonHolder ở chương 6 dùng policy để quản lý vòng đời và thread safety.

SmartPtr ở chương 7 triển khai dùng policy trên toàn miền.

Cơ chế double-displatch ở chương 11 dùng policy để hoán đổi ưu nhược điểm.

Triển khai Factory Abstract ở chương 9 dùng policy khi chọn phương thức tạo.

Chương này sẽ giải thích vấn đề mà policy đang nhắm đến để giải quyết, thế nào là thiết kế class theo policy, và đưa ra lời khuyên khi triển khai class theo policy.

##### Thiết kế phần mềm theo nhiều policy

Bạn có thể đạt mục tiêu theo nhiều cách, có rất nhiều sắc thái vô tận giữa cái sai và cái đúng. Mỗi con đường lại mở ra một chân trời mới.

Khi bạn đã chọn ra một giải pháp, vô vàn khả năng sau đó lại xuất hiện, ở từng cấp độ một.

Thiết kế hệ thống phần mềm là phải đưa ra quyết định chọn ra không gian giải pháp kết hợp.

Ví dụ, ở mức độ thiết kế tầng thấp (LLD-low level design): con trỏ thông minh ở chương 7

* có thể đơn luồng hay đa luồng
* có thể dùng nhiều cách sở hữu
* có thể trao đổi giữa 2 đặc tính an toàn hay tốc độ
* có hay không hỗ trợ ép kiểu cho con trỏ bên trong.

Tất cả tinh năng trên có thể kết hợp tự do với nhau để tạo nên 1 giải pháp phù hợp cho ứng dụng.

Không gian thiết kế nhiều policy luôn gây bối rối cho những người mới làm quen. Khi gặp phải vấn đề thiết kế, đâu là giải pháp tốt cho nó? Sự kiện? Đối tượng? Mục tiêu? Callbacks? Virtuals? Template? Tùy theo mức độ, mà nhiều giải pháp khác nhau đều đáp ứng được.

Điểm khác biệt quan trọng nhất giữa người có kinh nghiệm và người mới là kinh nghiệm về cái gì sẽ khả thi và không. Đối với bất kì vấn đề thiết kế, có nhiều cách khá cạnh tranh để giải quyết, sẽ có một tập hợp các điểm thuận lợi và bất lợi, phù hợp hay không phù hợp.

Một giải pháp xuất hiện để chấp nhận được trên giấy nhiều khi không khả thi với thực tế.

Thiết kế hệ thống phần mềm rất khó vì nó đòi hỏi bạn phải luôn lựa chọn một cách khó khăn.

Đối với những người mới làm quen, mỗi quyết định thiết kế mở ra cánh cửa mịt mù. Những nhà thiết kế lão luyện thì giống như người chơi cờ: có thể nhìn thấy các bước đi tiếp theo.

##### Sự thất bại của Interface Do-It-All

Triển khai mọi thứ dưới cái ô do-it-all interface không phải giải pháp tốt, dù nguyên nhân gì đi nữa, vì nó tốn hiệu suất, kích thước lớn.

Các class quá lớn sẽ không hiệu quả, làm tốn hiệu suất so với các phiên bản thủ công.

Nhưng vấn đề lớn nhất đó là thiếu sự an toàn kiểu (static type safety)

Lập trình viên có thể tạo ra vô vàn hàm dựng hợp lệ về cú pháp nhưng vô nghĩa.

##### Đa thừa kế

Đa thừa kế cho phép kết hợp nhiều giải pháp thiết kế với nhau từ một vài class cơ sở, chúng ta có thể có class smart pointer đa luồng – refcount từ class BaseSmartPtr với 2 class : MultiThreaded và RefCounted.

Dù tạo ra những thiết kế linh hoạt, mềm dẻo nhưng các chuyên gia đều biết rằng thiết kế như vậy sẽ thất bại, nguyên nhân:

1. Cơ chế: chỉ có sự trợ giúp từ ngôn ngữ lập trình, trình biên dịch để liên kết các lớp cở sở với nhau. Tốn nhiều công để xử lí hành vi của đối tượng
2. Thông tin kiểu: lớp cơ sở không có đủ thông tin kiểu để gánh vác nhiệm vụ của nó. Ví dụ, triển khai deep copy cho class smartpointer bằng cách cho nó kế thừa base class DeepCopy, vấn đề là interface DeepCopy sẽ không thể biết được kiểu đối tượng mà nó sẽ phải xử lí
3. Các class phải dùng phương thức ảo để xử lí cùng một vấn đề, khiến thiết kế trở nên phức tạp.

Mặc dù việc kết hợp như trên là tự nhiên, nhưng đa thừa kế không bao giờ định hình một triết lý thiết kế nào cả.

##### Lợi ích của Templates

Template là ứng cử viên tốt cho việc kết hợp các hành vi vì nó tự sinh ra code tại thời điểm biên dịch dựa trên kiểu dữ liệu được sử dụng.

Class template tùy biến không như class thông thường, nếu cần xử lí trường hợp đặc biệt, chỉ cần chuyên hóa bất kì hàm thành viên template nào cần.

Ngoài ra, class template hỗ trợ nhiều tham số template, bạn có thể dùng chuyên hóa từng phần cho một vài tham số template của nó.

Ví dụ:

template <class T, class U> class SmartPtr { ... };

có thể chuyên hóa SmartPtr<T, U> cho Widget và cho bất kì kiểu nào khác với cú pháp:

template <class U> class SmartPtr<Widget, U> { ... };

Do bản tính tự nhiên nên template luôn được dùng trong các thiết kế.Dù vậy template cũng có vài hạn chế:

1. Không thể chuyên hóa cấu trúc của class (thành viên của class) khi chỉ dùng riêng lẻ template, chỉ có thể chuyên hóa hàm mà thôi
2. Thiếu tính mở rộng: chỉ chuyên hóa được hàm thành viên với 1 tham số template, ví dụ:

template <class T>

class Widget

{

void Fun() { .. generic implementation ... }

};

// OK: chuyên hóa hàm thành viên của Widget

template <>

void Widget<char>::Fun()

{

//... specialized implementation ...

}

template <class T, class U>

class Gadget

{

void Fun() { .. generic implementation ... }

};

// Error! Không thể chuyên hóa hàm thành viên class Gadget

template <class U>

void Gadget<char, U>::Fun()

{

//... specialized implementation ...

}

1. Thư viện không thể cung cấp nhiều giá trị mặc định. Một class template có thể hỗ trợ một tham số mặc định cho mỗi hàm thành viên, không thể nhiều tham số mặc định cho hàm thành viên template.

Đa thừa kế có một cơ chế khá tồi, template thể hiện tính đa hình với nhiều cơ chế. Đa hình mất thông tin về kiểu, trong khi template thì không.

Chuyên hóa template không có tính mở rộng, nhưng đa hình thì co giãn tốt hơn.

Bạn chỉ có thể khai báo một template mặc định cho hàm thành viên mà thôi, nhưng có thể viết không giới hạn các lớp cơ sở

Do đó, sự kết hợp giữa template và đa thừa kế là ý tưởng rất hay để tạo ra các thiết kế tốt.

##### Policy và class policy

Policies và policy classes giúp triển khai an toàn, hiệu quả và duy trì các yếu tố thiết kế tùy biến.

Policy định nghĩa interface class hay interface template class.

Interface bao gồm một hay tất cả các đối tượng sau: định nghĩa kiểu bên trong, hàm thành viên và biến thành viên.

Policy không nhấn mạnh về kiểu, thiên về hành vi nhiều hơn.

Ví dụ về policy về tạo đối tượng: Creator policy quy định class template kiểu T.

For example, let's define a policy for creating objects. The Creator policy prescribes a class template of

type T. This class template must expose a member function called Create that takes no arguments and

returns a pointer to T. Semantically, each call to Create should return a pointer to a new object of type T.

The exact mode in which the object is created is left to the latitude of the policy implementation.

Let's define some policy classes that implement the Creator policy. One possible way is to use the new

operator. Another way is to use malloc and a call to the placement new operator (Meyers 1998b). Yet

another way would be to create new objects by cloning a prototype object. Here are examples of all three

methods:

template <class T>

struct OpNewCreator {

static T\* Create(){ return new T; }

};

template <class T>

struct MallocCreator{

static T\* Create() {

void\* buf = std::malloc(sizeof(T));

if (!buf) return 0;

return new(buf) T;

}

};

template <class T>

struct PrototypeCreator

{

PrototypeCreator(T\* pObj = 0):pPrototype\_(pObj) {}

T\* Create() {

return pPrototype\_ ? pPrototype\_->Clone() : 0;

}

T\* GetPrototype() { return pPrototype\_; }

void SetPrototype(T\* pObj) { pPrototype\_ = pObj; }

private:

T\* pPrototype\_;

};

For a given policy, there can be an unlimited number of implementations. The implementations of a policy

are called policy classes.[2] Policy classes are not intended for stand-alone use; instead, they are inherited

by, or contained within, other classes.

[2] This name is slightly inaccurate because, as you will see soon, policy implementations can be class

templates.

An important aspect is that, unlike classic interfaces (collections of pure virtual functions), policies'

interfaces are loosely defined. Policies are syntax oriented, not signature oriented. In other words, Creator

specifies which syntactic constructs should be valid for a conforming class, rather than which exact

functions that class must implement. For example, the Creator policy does not specify that Create must

be static or virtual—the only requirement is that the class template define a Create member function.

Also, Creator says that Create should return a pointer to a new object (as opposed to must). Consequently,

it is acceptable that in special cases, Create might return zero or throw an exception.

You can implement several policy classes for a given policy. They all must respect the interface as defined

by the policy. The user then chooses which policy class to use in larger structures, as you will see.

The three policy classes defined earlier have different implementations and even slightly different

interfaces (for example, PrototypeCreator has two extra functions: GetPrototype and

SetPrototype). However, they all define a function called Create with the required return type, so

they conform to the Creator policy.

Let's see now how we can design a class that exploits the Creator policy. Such a class will either contain or

inherit one of the three classes defined previously, as shown in the following:

// Library code

template <class CreationPolicy>

class WidgetManager : public CreationPolicy

{

...

};

The classes that use one or more policies are called hosts or host classes.[3] In the example above,

WidgetManager is a host class with one policy. Hosts are responsible for assembling the structures and

behaviors of their policies in a single complex unit.

[3] Although host classes are technically host class templates, let's stick to a unique definition. Both host

classes and host class templates serve the same concept.

When instantiating the WidgetManager template, the client passes the desired policy:

// Application code

typedef WidgetManager< OpNewCreator<Widget> > MyWidgetMgr;

Let's analyze the resulting context. Whenever an object of type MyWidgetMgr needs to create a Widget,

it invokes Create() for its OpNewCreator<Widget> policy subobject. How ever, it is the user of WidgetManager who chooses the creation policy. Effectively, through its design, WidgetManager

allows its users to configure a specific aspect of WidgetManager's functionality.

This is the gist of policy-based class design.

1. Implementing Policy Classes with Template Template Parameters

Often, as is the case above, the policy's template argument is redundant. It is awkward that the user must

pass OpNewCreator's template argument explicitly. Typically, the host class already knows, or can easil

deduce, the template argument of the policy class. In the example above, WidgetManager always

manages objects of type Widget, so requiring the user to specify Widget again in the instantiation of

OpNewCreator is redundant and potentially dangerous.

In this case, library code can use template template parameters for specifying policies, as shown in the

following:

// Library code

template <template <class Created> class CreationPolicy>

class WidgetManager : public CreationPolicy<Widget>

{

...

};

In spite of appearances, the Created symbol does not contribute to the definition of WidgetManager.

You cannot use Created inside WidgetManager—it is a formal argument for CreationPolicy (not

WidgetManager) and simply can be omitted.

Application code now need only provide the name of the template in instantiating WidgetManager:

// Application code

typedef WidgetManager<OpNewCreator> MyWidgetMgr;

Using template template parameters with policy classes is not simply a matter of convenience; sometimes,

it is essential that the host class have access to the template so that the host can instantiate it with a

different type. For example, assume WidgetManager also needs to create objects of type Gadget using

the same creation policy. Then the code would look like this:

// Library code

template <template <class> class CreationPolicy>

class WidgetManager : public CreationPolicy<Widget>

{

...

void DoSomething()

{

Gadget\* pW = CreationPolicy<Gadget>().Create();

...

}

};

Does using policies give you an edge? At first sight, not a lot. For one thing, all implementations of the

Creator policy are trivially simple. The author of WidgetManager could certainly have written the

creation code inline and avoided the trouble of making WidgetManager a template.

But using policies gives great flexibility to WidgetManager. First, you can change policies from the

outside as easily as changing a template argument when you instantiate WidgetManager. Second, you

can provide your own policies that are specific to your concrete application. You can use new, malloc,

prototypes, or a peculiar memory allocation library that only your system uses. It is as if WidgetManager

were a little code generation engine, and you configure the ways in which it generates code.

To ease the lives of application developers, WidgetManager's author might define a battery of often-used

policies and provide a default template argument for the policy that's most commonly used:

template <template <class> class CreationPolicy = OpNewCreator>

class WidgetManager ...

Note that policies are quite different from mere virtual functions. Virtual functions promise a similar effect:

The implementer of a class defines higher-level functions in terms of primitive virtual functions and lets

the user override the behavior of those primitives. As shown above, however, policies come with enriched

type knowledge and static binding, which are essential ingredients for building designs. Aren't designs full

of rules that dictate before runtime how types interact with each other and what you can and what you

cannot do? Policies allow you to generate designs by combining simple choices in a typesafe manner. In

addition, because the binding between a host class and its policies is done at compile time, the code is tight

and efficient, comparable to its handcrafted equivalent.

Of course, policies' features also make them unsuitable for dynamic binding and binary interfaces, so in

essence policies and classic interfaces do not compete.

1. Implementing Policy Classes with Template Member Functions

An alternative to using template template parameters is to use template member functions in conjunction

with simple classes. That is, the policy implementation is a simple class (as opposed to a template class)

but has one or more templated members.

For example, we can redefine the Creator policy to prescribe a regular (nontemplate) class that exposes a

template function Create<T>. A conforming policy class looks like the following:

struct OpNewCreator

{

template <class T>

static T\* Create()

{

return new T;

}

};

##### Enriched Policies

The Creator policy prescribes only one member function, Create. However, PrototypeCreator

defines two more functions: GetPrototype and SetPrototype. Let's analyze the resulting context.

Because WidgetManager inherits its policy class and because GetPrototype and Set-Prototype

are public members of PrototypeCreator, the two functions propagate through WidgetManager and

are directly accessible to clients.

However, WidgetManager asks only for the Create member function; that's all WidgetManager

needs and uses for ensuring its own functionality. Users, however, can exploit the enriched interface.

A user who uses a prototype-based Creator policy class can write the following code:

typedef WidgetManager<PrototypeCreator>

MyWidgetManager;

//...

Widget\* pPrototype = //...;

MyWidgetManager mgr;

mgr.SetPrototype(pPrototype);

//... use mgr ...

If the user later decides to use a creation policy that does not support prototypes, the compiler pinpoints the

spots where the prototype-specific interface was used. This is exactly what should be expected from a

sound design.

The resulting context is very favorable. Clients who need enriched policies can benefit from that rich

functionality, without affecting the basic functionality of the host class. Don't forget that users—and not

the library—decide which policy class to use. Unlike regular multiple interfaces, policies give the user the

ability to add functionality to a host class, in a typesafe manner.

##### Destructors of Policy Classes

There is an additional important detail about creating policy classes. Most often, the host class uses public

inheritance to derive from its policies. For this reason, the user can automatically convert a host class to a

policy and later delete that pointer. Unless the policy class defines a virtual destructor, applying

delete to a pointer to the policy class has undefined behavior,[4] as shown below.

[4]

In Chapter 4, Small-Object Allocation, you can find a discussion on exactly why this happens.

typedef WidgetManager<PrototypeCreator>

MyWidgetManager;

...

MyWidgetManager wm;

PrototypeCreator<Widget>\* pCreator = &wm; // dubious, but legal

delete pCreator; // compiles fine, but has undefined behavior

Defining a virtual destructor for a policy, however, works against its static nature and hurts performance.

Many policies don't have any data members, but rather are purely behavioral by nature. The first virtual

function added incurs some size overhead for the objects of that class, so the virtual destructor should be

avoided.

A solution is to have the host class use protected or private inheritance when deriving from the policy clas

However, this would disable enriched policies as well (Section 1.6).

The lightweight, effective solution that policies should use is to define a nonvirtual protected destructor:

template <class T>

struct OpNewCr

{

static T\* C{return new T;}

protected:

~OpNewCreator() {}

};

Because the destructor is protected, only derived classes can destroy policy objects, so it's impossible for

outsiders to apply delete to a pointer to a policy class. The destructor, however, is not virtual, so there is

no size or speed overhead.

##### Optional Functionality Through Incomplete Instantiation