University of Hertfordshire

7COM1025
Programming for Software Engineers
Lecture 9

Dr. Renato C. de Amorim

THE ONE HUNDRED UNDER FIFTY

PIMPL IDIOM

Pointer to implementation (pimp) is a technique used to avoid exposing private details in your header file.

We normally have all declarations in a header file (.h) and the definitions in a c++ file (.cpp). We can easily hide the implementation of the cpp file by compiling it. What about the .h?

We want our clients to see the public methods of our classes, but not really the private members.

Vector3.h

```
class Vector3
{
    public:
        float GetSum();
        void SetX(float _x);
        void SetY(float _y);
        void SetZ(float _z);
    private:
     float mX, mY, mZ;
};
```

Vector3.cpp

```
#include "Vector3.h"

float Vector3::GetSum()
{
    return mX + mY + mZ;
}

void Vector3::SetX(float _x)
{mX=_x;}
 void Vector3::SetY(float _y)
 {mY=_y;}
void Vector3::SetZ(float _z)
{mZ=_z;}
```

Myprogram.cpp

```
#include <iostream>
#include "Vector3.h"
using namespace std;
int main()
{
 Vector3 Vec;
 Vec.SetX(3.0);
 Vec.SetY(2.0);
 Vec.SetZ(5.0);
 cout<<Vec.GetSum()<<endl;
 return 0;
}
```

PIMPL IDIOM (IMPLEMENTATION)

Vector3.h

```
class Vector3
{
    public:
        Vector3();
        ~Vector3();
        float GetSum();
        void SetX(float _x);
        void SetY(float _y);
        void SetZ(float _z);
    private:
        class Impl;
        Impl *mImpl;
};
```

Vector3.cpp

```
#include "Vector3.h"

class Vector3::Impl{
    public:
    float mX, mY, mZ;
};

Vector3::Vector3()
{mImpl = new Impl();}

Vector3::~Vector3()
{delete mImpl;}

float Vector3::GetSum()
{return mImpl->mX + mImpl->mY
+ mImpl->mZ;}

void Vector3::SetX(float _x)
{mImpl->mX = _x;}
 void Vector3::SetY(float _y)
 {mImpl->mY = _y;}

void Vector3::SetZ(float _z)
{mImpl->mZ = _z;}
```

Myprogram.cpp

```
#include <iostream>
#include "Vector3.h"
using namespace std;
int main()
{
 Vector3 Vec;
 Vec.SetX(3.0);
 Vec.SetY(2.0);
 Vec.SetZ(5.0);
 cout<<Vec.GetSum()<<endl;
 return 0;
}
```

University of Hertfordshire

THE ONE HUNDRED UNDER FIFTY

PIMPL IDIOM

Don't forget to allocate and deallocate the implementation object (mImpl in the previous example).

Advantages of using Pimpl
- Information hiding. Private members (variables or methods) can be completely hidden.
- Reduced coupling. You can move dependences (think headers) from .h to .cpp files.
- Faster compile time. It **may** lead to a shorter compile time because of the dependences.
- Greater binary compatibility. The size of the pimpl object never changes (its a pointer), changes in implementation change only the size of the implementation class. Major implementation changes can be done without changing the binary representation of your object.
- Lazy allocation. The mImpl class can be constructed on demand (when you actually need it).

Disadvantage
- You must allocate (and free!) an additional implementation object for each object created.
- Some inconvenience as you need to access all private members by `mImpl->` (this is not exposed to the user of your API).
- Compiler will no longer catch changes to private members in const methods. This is because private members live now in a different object

DESIGN PATTERNS

A design pattern is a general solution to a common software design problem. There are three main categories: (i) creational patterns; (ii) structural patterns; (iii) behavioural patterns.

Singleton (creational pattern)
It ensures a class has only ever one instance, providing a global point of access to this single instance.

It offers many advantages over the use of a global variable: (i) Enforces that only one instance of a class can be created; (ii) provides control over the allocation and destruction of the object; (iii) allows support for thread-safe access to the object; (iv) avoids polluting the global namespace.

Examples of possible use: (i) a system clock; (ii) the global clipboard; (iii) the keyboard, etc...
Or a manager class providing a single point of access to multiple resources (database example).

SINGLETON (CREATIONAL DESIGN PATTERN)

Implementation:

1) Create a static public GetInstance(), it will return the same instance every time it's called.

2) Prevent clients to create new instances, declare the constructor (even if default) as private.

3) You want it to be non-copyable. Declare a private copy constructor and a private assignment operator.

4) Prevent clients from deleting it. Declare the destructor as private.

```
Class Singleton
{
   public:
      Static Singleton &GetInstance();
  private:
      Singleton();
      ~Singleton();
      Singleton(const Singleton &);
      const Singleton &operator=(const Singleton &);
};
```

Declaring the constructor and destructor as private also means that clients cannot create subclasses.
If you wish to allow this simply declare them as protected.

SINGLETON

The Singleton design pattern is designed to hold and control access to a global state.

In an interview authors of the original design patterns book stated that the only pattern they would consider removing from the original list is Singleton.

- This is because the pattern is essentially a way to store global data and this tends to be an indicator of poor design.

In your programs do think if Singleton is really the pattern you need. Requirements change and code evolve. In the future you may find that you will need multiple instances of the class.

eg. Web-browser. Initially one could use a Singleton to hold a webpage, now we have tabs, each with (possibly) a different webpage.

Use Singleton to model objects that are truly singular in their nature.

FACTORY (CREATIONAL DESIGN PATTERN)

It allows you to create objects without having to specify the specific C++ type of the object to create.

In essence this is a generalisation of a constructor. The constructor for MyClass is:
MyClass();
Of course you can overload the above with a different list of parameters, but note that it doesn't return anything.

C++ constructors have the following limitations:
- No return result. You simply cannot return a result from a constructor. So you cannot (for example) signal an error during the initialisation of an object.
- Constrained naming. A constructor must have the same name as the class. By consequence you cannot have two constructors with the same list of parameters.
- Statically bound creation. When constructing an object you must specify a name of a class known at compiler time. Eg: MyClass *mc = new MyClass();
- No virtual constructors. You cannot declare a virtual constructor in C++.

The factory design pattern circumvent all these limitations!

FACTORY

A factory method is simply a normal method call that can return an instance of a class.

Often used together with inheritance. A derived class can override the factory method and return an instance of that derived class.

```
//MyClassFactory.h
#include "myclass.h"
#include <string>
Using namespace std;
class MyClassFactory
{
public:
    MyClass *CreateMyClassObject(const string &type)
};
```

Use Factory method to prove more powerful class construction semantics and to hide subclass details.

EXERCISE

In groups discuss:
Can we mix the Singleton and the Factory design patterns? If you believe you can, can you think of an example? Otherwise, explain why you can't.