University of Hertfordshire

7COM1025
Programming for Software Engineers
Lecture 26

Dr. Renato C. de Amorim

THE ONE HUNDRED UNDER FIFTY

REQUIRED READING

This class relates to chapter 12: Extensibility

Book:
Reddy, M. (2011). API Design for C++. Elsevier.

Pages: 361 - 390

EXTENSIBILITY

API extensibility:
Relates to the ability of our clients to modify the behaviour of your interface without requiring you to evolve the API for their specific needs.

It allows you to deliver flexible systems that lets your users solve problems that you had never anticipated.

An API should be open to extension, but close for modification

Can you think of examples of APIs that are extensible?

EXTENDING VIA PLUGINS

A plugin is a dynamic library that is discovered and loaded at run time. This is not a dynamic library that is linked at build time.

Users can write plugins using a well-defined plugin API that you will need to provide.

You will need:

- The plugin API: This is the API your users must compile and link against in order to create a plugin.
- The plugin manager: This is an object (often a singleton) in the core API code that manages the lifecycle of all plugins.

There are, of course, a number of design issues that you will need to consider, including:

C vs C++?
versioning, what if the plugin was created with a previous version of your API?
Security (how much are you going to trust the plugins)?, etc.

PLUGIN IMPLEMENTATION

When a Core API loads a plugin, it needs to know which functions to call or symbols to access. This way the plugin can do its job.

The above means that you need to define specifically named entry points in the plugin your users must provide.

Example: when writing a GIMP plugin you must define a variable called PLUG_IN_INFO that lists the various callbacks defined in the plugin.

```
#include <libgimp/gimp.h>
GimpPlugInInfo PLUG_IN_INFO =
{
    NULL, //called when GIMP starts
    NULL, //called when GIMP exists
    query, // procedure registration and arguments definition
    run, // perform the plugin's operation
};
```

EXTENDING VIA INHERITANCE

The primary object-oriented mechanism for extending a class is inheritance. It can be used to allow clients to create new classes that build upon and modify the existing classes.

Inheritance can only be done safely if the base class has been designed to be inherited from. The primary indicator for this is a virtual destructor.

Sometimes the above will not be an issue, for instance if you do not allocate any memory that must be freed.

You can:
- Add new methods
- Overload any virtual function

THE VIRTUAL DESTRUCTOR ISSUE

This code will call the base constructor, the derived constructor, but only the base destructor.

```
#include <iostream>
using namespace std;
class base{
    public:
    base(){cout<<"Base constructor"<<endl;}
    ~base(){cout<<"Base destructor"<<endl;}};
class derived : public base{
    public:
    derived(){cout<<"Derived constructor"<<endl;}
    ~derived() {cout<<"Derived destructor"<<endl;;}
};
int main(){
    base *obj = new derived;
    delete obj;
    return 0;
}
```

If the destructor of base was virtual, you would have the four couts on the screen.

INHERITANCE AND THE STL

Programmers new to C++ often try to inherit from a STL container (such as vector, list, etc)

STL containers classes do not provide virtual destructors. In fact, they do not provide any virtual method.

As an alternative you can use composition to add functionality to an STL container in a safe manner

```
#include <string>
class MyString{
    public:
        MyString() : mSTR(""){}
        bool empty() const {return mSTr.empty();}
        void clear() {mStr.clear();}
        void push_back (char c) {mStr.push_back(c);}
        ...
    private:
        std::string mStr;
};
```

The STL does provide a few classes designed for inheritance, the most obvious is exception.