

# G52CPP

## C++ Programming

### Lecture 20

Dr Jason Atkin

[http://www.cs.nott.ac.uk/~jaa/cpp/  
g52cpp.html](http://www.cs.nott.ac.uk/~jaa/cpp/g52cpp.html)

# Wrapping up...

Slicing Problem

Smart pointers

More C++ things

Exams

# The slicing problem

# Objects are not pointers

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base() : i(1) {}
    virtual void out()
    { cout <<"Base" <<i <<endl; }
    int i;
};

class Sub : public Base
{
public:
    Sub() { i = 2; }
    void out()
    {cout <<"Sub" <<i <<endl; }
};
```

Lec18b.cpp

```
int main()
{
    Sub array1[3];
    Base array2[3];
    // Output Array 1
    for ( int i = 0 ; i < 3 ; i++ )
        array1[i].out();
    // Output Array 2
    for ( int i = 0 ; i < 3 ; i++ )
        array2[i].out();
    // Copy array 1 to 2
    for ( int i = 0 ; i < 3 ; i++ )
        array2[i] = array1[i];
    // Output Array 2
    for ( int i = 0 ; i < 3 ; i++ )
        array2[i].out();
}
```

# Why?

- In that example, we stored base class objects
- So when we tried to assign sub-class objects only the base class part was stored
- We sliced off the sub-class part
- Things can get worse than this...

# What is the slicing problem?

- Passing an object type parameter **by value** uses the copy constructor – i.e. copies it
- Assigning an object to another object copies the values, using the assignment operator
- If the thing you are copying to is a base class object (***or thinks it is!***) the base class assignment operator is used
- **Neither the default copy constructor nor assignment operator are virtual**
  - The base class version gets used!!!
  - (Just making them virtual would not help anyway – see later)
- The slicing problem occurs when you treat a sub-class as the base class for a copy/assignment
  - Only the base class parts get copied
  - i.e. the sub-class parts are sliced off

# How can it happen?

- By using references or pointers, e.g.:

```
class BaseClass
{
public:
    int MyParam;
};

class SubClass
    : public BaseClass
{
public:
    int MySubClassParam;
};
```

```
int main()
{
    SubClass s1, s2;

    BaseClass& rs1 = s1;
    BaseClass& rs2 = s2;
    rs1 = rs2;

    BaseClass* ps1 = &s1;
    BaseClass* ps2 = &s2;
    *ps1 = *ps2;
}
```

- The sub-class part will not be copied

# Why?

- The slicing problem occurs when you treat a sub-class as the base class for a copy/assignment
  - Only the base class parts get copied
  - i.e. the sub-class parts are sliced off
- A function like this was created and used:

```
Base& operator=(const Base& rhs )
{
    this->MyParam = rhs.MyParam;
    return *this;
}
```

- Rather than using the sub-class version:

```
SubClass& operator=(const SubClass& rhs )
{
    this->MySubClassParam = rhs.MySubClassParam;
    this->MyParam = rhs.MyParam;
    return *this;
}
```



# Advanced 'stuff'

- Thing to remember:
  - Use assignment with base class pointers and references with care (or not at all)
  - You may end up slicing off the sub-class part
- But some of this you ***could*** actually fix if you REALLY wanted to...
- The following slides are complicated stuff, just to show you what you COULD do if you really wanted to, and give you a taste of the power of operator overloading...

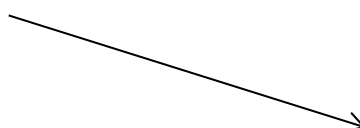
# 1: virtual assignment operator

- **First make the assignment operator in the base class **virtual****
  - I.e. implement it yourself so that you can make it virtual

```
virtual Base& operator=(const Base& rhs )
{
    this->i = rhs.i;
    return *this;
}
```

## 2: Sub-class assignment operator

- Add assignment operator in the sub-class
- Note: This is **NOT** an overload of the base class operator= since it takes a sub-class object reference



```
Sub& operator=(const Sub& rhs )
{
    this->Base::operator=(rhs);
    this->j = rhs.j;
    cout << "copied sub" << endl;
    return *this;
}
```

Use scoping to call base class version, not the sub-class one (virtual function!)

## 3: Override base class version

- In the **SUB CLASS** also provide an overload of the **BASE CLASS assignment operator**, which checks for sub-class objects and if so calls the sub-class assignment operator

```
virtual Base& operator=(const Base& rhs )
{
    try
    { // If object really is a sub (sub-class object)
        const Sub& rsub = dynamic_cast<const Sub&>(rhs);
        *this = rsub; // Assign using 2 subclass objs
    }
    catch( bad_cast )
    { // Object is NOT a sub (sub-class object)
        this->Base::operator=(rhs); // Use base class
    }
    return *this;
}
```

# Smart Pointers

# auto\_ptr

- Provides one way of avoiding memory leaks
  - As we have already seen
- And illustrates the way in which many of the facilities which we have seen can be used, e.g.:
  - Operator overloading
  - Copy constructor
  - Conversion constructor
  - Template classes and template functions
- It is really useful to know about `auto_ptr` and smart pointers when you develop large programs

# Using the `auto_ptr` class

```
void foo2()
{
    // Create objects
    MyClass* pObj1 = new MyClass;

    // Wrap them in auto-pointers
    cout << "Create auto-pointers" << endl;
    auto_ptr<MyClass> a(pObj1);

    cout << "Finished creation" << endl;

    bar(); // throws exception

    // Other code?
}
```

The `auto_ptr` is an object on the stack  
i.e. it gets destroyed when the function ends  
When destroyed the `auto_ptr` will call `delete` on any pointer it holds

Template class, wraps a pointer of the specified type

Function ends here  
Stack objects are destroyed  
`auto_ptr` objects destroy the objects whose pointers they wrap

# simple\_auto\_ptr template class

```
template<class T>
class simple_auto_ptr
{
private:
    T* _MyPtr;
```

**simple\_auto\_ptr** is my cut-down version of **auto\_ptr**, to demonstrate the key principles.

```
public:
    // construct from object pointer
    explicit simple_auto_ptr(T* _Ptr = NULL) throw()
    : _Myptr(_Ptr)
    {
        cout << "\t\t simple_auto_ptr created" << endl;
    }

    // destroy the object
    ~simple_auto_ptr()
    {
        cout << "\t\t simple_auto_ptr destroyed" << endl;
        if ( _Myptr )
            delete _Myptr; ← Key part!
    }
};
```

Key part!



# How (simple\_)auto\_ptr works

```
template<class T>
class simple_auto_ptr
{
private:
    T* _Myptr;
```

Template class – can wrap **any** object type

## The wrapped object pointer

## public:

```
// construct from object pointer
```

```
explicit simple_auto_ptr(T* _Ptr = NULL) throw()
```

```
:_MyPtr(_Ptr)
```

**Constructor stores the pointer passed in**

```
cout << "\t\tsimple_auto_ptr created" << endl;
```

}

## Destructor calls `delete` through the pointer

```
// destroy the object
```

## ~simple\_auto\_ptr()

{

```
cout << "\t\tsimple_auto_ptr destroyed" << endl;
```

```
if ( __MyPtr )
```

```
delete  Myptr;
```

**Has to be a template so that `delete` knows the type of object**

# Using the (simple\_)auto\_ptr

`simple_auto_ptr` is my simpler version of `auto_ptr`

```
// Create objects
```

```
MyClass* pObj1 = new MyClass; ←
```

Test code

Construct object

```
// Wrap them in auto-pointers
```

```
simple_auto_ptr<MyClass> a(pObj1); ←
```

Construct `auto_ptr`

```
cout << "Finished creation" << endl;
```

```
bar(); ←
```

Throws an exception  
The `auto_ptr`s destroy  
the objects

```
MyClass constructor 1
```

```
simple_auto_ptr created
```

```
Finished creation
```

```
simple_auto_ptr destroyed
```

```
MyClass Destructor 1
```

Messages on object  
creation  
and  
deletion

**‘Smart’ish pointer functionality**

`get()` and `release()`

```
// Return wrapped pointer
```

```
T* get() const throw()
```

{

```
cout << "\t\tsimple_auto_ptr get()" << endl;
```

```
return _Myptr;
```

}

**get ( )** just gets the stored pointer – i.e. a copy of the value of the pointer which is in the object, so function can be **const**

```
// Return wrapped pointer and give up ownership
```

## T\* release() throw()

{

```
cout << "\t\tsimple_auto_ptr release()" << endl;
```

```
T *__Tmp = __MyPtr;
```

```
_Myptr = NULL;
```

```
return (__Tmp);
```

}

`release()` detaches the pointer from the object (i.e. it alters the pointer, so not `const`) and returns the pointer (`delete` will no longer be called on it)

# Other functionality

- `reset()` is a function to replace the stored pointer with another pointer (of same type)
  - Note: it **deletes** any existing pointer
- `operator*()` overloaded
  - Access the thing which *internal pointer* points to
- `operator->()` overloaded
  - Access members of a class/struct which the *internal pointer* points to
- Also implements assignment operator and copy constructor to **take ownership** of internal pointer
- Objects of this class are smart(ish) pointers
  - Act like pointers, with a little more intelligence

# Example of using \* and ->

```
// Create objects and wrap them in auto-pointers
cout << "Create auto-pointers" << endl;
simple_auto_ptr<MyClass> a(new MyClass);
```

```
// get() will get the wrapped pointer
cout << "Use get()" << endl;
a.get()->print();
```

Object acts  
like a pointer

```
// * has been overloaded
cout << "Use overloaded *" << endl;
(*a).print();
```

\* gets the object *pointed at* by *contained* pointer  
Calls `print()` on the object

```
// -> has also been overloaded
cout << "Use overloaded ->" << endl;
a->print();
```

-> acts on the *contained* pointer  
i.e. Calls `print()` through the contained pointer

# \* and -> apply to the contained pointer

```
// Access the thing wrapped pointer points at  
// Can then do . on the result
```

```
T& operator*() const throw()  
{  
    return *_Myptr;  
}
```

```
// Like doing -> on wrapped pointer  
// Tell it what to actually do -> on  
// Note: I think this is unusual syntax!
```

```
T* operator->() const throw()  
{  
    return _Myptr; // -> will be used on this  
}
```

# Another example using \*

```
// Create objects and wrap them in auto-pointers
cout << "Create auto-pointers" << endl;
simple_auto_ptr<MyClass> a(new MyClass);

cout << "Use the * operator to dereference:" << endl;
MyClass& rmc = *a; // Gets the object inside

// rmc is just another name for the object wrapped
// by the simple_auto_ptr 'a'
rmc.print();

// Assume that b is another pointer like 'a'
cout << "Now use the assignment operator on MyClass:";
rmc = *b; // Assignment operator on objects!
rmc.print();
```



# auto\_ptr summary

- `auto_ptr` is a template class
  - It knows the type of object pointer that it wraps (so that it can call `delete` on it)
- When destroyed, the destructor for `auto_ptr` calls `delete` on the wrapped object
  - `auto_ptr` is NOT suitable for storing arrays since it calls `delete p;` not `delete[] p;`
  - Could easily create an array version though
- Call `release()` to detach the wrapped object
- Call `get()` to access the wrapped object pointer
  - To potentially use it
- Operators `*` and `->` are overloaded

# Smart pointers

- Smart pointers: objects which act like pointers, but are 'smarter'
- True smart pointers can be much smarter than `auto_ptr`
- e.g. perform reference counting
  - Delete the resource when the last smart pointer object which points to it is destroyed
  - Check for memory leaks?
- The principles are the same as `auto_ptr`
- These are really useful things to have
- If used properly, they can simplify memory management – doing the `deletes` for you

Moving on from here

# We have only looked at the basics

- We have covered only the basics of C++
  - None of the class libraries
  - None of the recent additions
- We could cover many more modules with content, but my hope is that by understanding the underlying principles you can work out the rest (read about it?)
- If you want to go for a job be aware of:
  - The Standard Template Library (STL)
  - C++ 11 and Boost

# C++ 11

- C++ 11 (newly-ish standardised version)
  - See page maintained by Bjarne Stroustrup:  
<http://www.stroustrup.com/C++11FAQ.html>
- Quotes from Bjarne Stroustrup:
  - “C++11 feels like a new language”
    - I agree, new and optional replacement features
  - “Currently shipping compilers (e.g. GCC C++, Clang C++, IBM C++, and Microsoft C++) already implement many C++11 features. For example, it seems obvious and popular to ship all or most of the new standard libraries.”

# Boost

- Web page: <http://www.boost.org/>

**Quotes from the web page (reformatted only):**

- **Boost provides free peer-reviewed portable C++ source libraries.**
- We emphasize libraries that work well with the C++ Standard Library.
- Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications.
- The Boost license encourages both commercial and non-commercial use.
- We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization.
- Ten Boost libraries are included in the C++ Standards Committee's Library Technical Report (TR1) and in the new C++11 Standard.
- C++11 also includes several more Boost libraries in addition to those from TR1. More Boost libraries are proposed for TR2.

Some exam comments

# Exam Structure

- Answer ANY three questions out of five
  - There is no compulsory question this year!!!
  - Each question is worth 20 marks
- Exam is 60% of course mark (and out of 60)
- 1.5 hour exam not 2 hours
  - Exam is a bit shorter than last year
  - 30 minutes per question
  - Many people left early last year, but still did well
  - Possible to do REALLY well (e.g. 99% or 100%)
- Similar structure and type of questions to last year
  - Possibly more “write code to...”
- Question heading will give you an idea what the question is about, where that won't give away the answer
  - However, most questions have a part asking you to work out what some code sample outputs or what is wrong with it (~40% marks)



# Resit exam

- Ignore this slide if you are not doing a resit
- Note that the resit exam is different
  - For people sitting resit now, and for people who may need to resit it later
- No coursework component
- Answer any **4** out of **6** questions (not 3/5)
- 25 marks and 25% each question (not 20!)
- 2 hours rather than 1.5 hours

# Exam Content

- Most concepts appear somewhere on exam
- Standard class library not needed, except:
  - Recognise that `cout << v` means output/print the value of `v` and be able to make simple examples of this
  - Know **basics** of the string and vector classes
    - As seen in lectures, i.e. understand lecture samples
- Know the **common C-library functions**
  - What a function does, not parameter details
  - File access, string functions, input and output
- Ensure that you can create a **template function** and **operator overload**
  - And a macro (`#define`) and understand the difference
- Understand about conversion constructors and operators, copy constructors and assignment operators

# Things to know

- You need to know `char*` type C-strings
  - Where the `char*` points to an array of chars with a 0 at the end
- Be aware of array bounds issues and pointer arithmetic

Know about:

- Pass/return by value vs by reference/pointer
- Inheritance, `virtual` and non-virtual functions
- Exceptions and exception handling
- Function pointers
- Casting (static vs dynamic, `const` and `reinterpret`)
- `const` members, parameters, references, pointers
- `static` local variables, `static` members
- `struct` vs `class` vs `union`

# Hints

- Pick and choose your questions according to what you are good at
  - Obvious? Why do so many people do Q1,2,3?
- Take some time to work out what each question (part) is asking
  - Is it something that you know how to do?
- Check the rest of the paper if you are stuck – sometimes it may jog your memory
  - e.g. does a code sample answer something?

# What now?

- Note: If there are any specific old exam questions that you want me to go through then please tell me in advance
- Revision lecture tomorrow, 10am
- Chance to ask questions at 2pm Friday
- Go through example code type questions
  - What is the question asking?
  - What do I need to know to answer it?
  - Are there any tricky bits?
  - What is the answer?