G52CPP
C++ Programming
Lecture 13

Dr Jason Atkin

http://www.cs.nott.ac.uk/~jaa/cpp/g52cpp.html

Last lecture

- Function pointers
  - Arrays of function pointers
- Virtual and non-virtual functions
  - vtable and vptr
  - Virtual functions are slower to call, more work

What to know about vpointers

- Some equivalent of a **`vpointer`** exists in objects with virtual functions
  - Just one pointer is needed in each object
- Only virtual functions appear in **`vtables`**
  - No need to record non-virtual functions
- Looking up which function to call is slower than calling a non-virtual function
  - But can be done!

1. Go to the object itself
2. Retrieve the **`vtable`** (following the **`vpointer`**)
3. Look up which function to call from index
4. Call the function

Example: virtual functions

```
#include <cstdio>
class BaseClass
{
public:
   virtual char* bar() { return "BaseBar"; }
   char* bar2() { return this->bar(); }
};

class SubClass : public BaseClass
{
public:
   char* bar() { return "SubBar "; }
};
```

BaseClass vtable:
`BaseClass::bar()`

SubClass vtable:
`SubClass::bar()`

- Virtual-ness is inherited
- Calling a function from a base class function is like using a base class pointer to call it

Should a function be `virtual`?

- If member function is called from a base class function or through a base class pointer **AND** the behaviour should depend on class type **then** the member function has to be `virtual`
  - Otherwise when it is called by the base class, or through a base-class pointer, the base-class version will be called, not your modified version

- Utility functions will often **not** be `virtual`
  - Do not expect to change functionality in sub-classes
  - Faster to call these functions – no look-up needed
  - Makes it easier for function to be `inline,` which can make code even faster: no function call needed

To be clear … `sizeof()`

- Functions act on objects of the class type
  - Even member functions
- They are not actually in the objects
  - The just have a hidden '**`this`**' parameter saying which object they apply to
- The object is the collection of data
  - But also includes any hidden data and pointers that the compiler adds to implement features (e.g. **`vtable`**)
- Adding a member function to an existing class will not **usually** make the **objects** bigger
  - Exception: adding the first virtual function may add a **`vtable`** pointer (or equivalent)
  - Understand why this is the case!

This lecture

- Automatically created methods:
  - Default Constructor
  - Copy Constructor
  - Assignment operator
  - Destructor
- Conversion constructors
- Conversion Operators
- Friends

Reminders about object creation

Reminders

- If you want to **create** objects **in dynamic memory** then you **must** go through **`new`** (NOT **`malloc()`**)
  - You **cannot** construct **correct objects** yourself
    - e.g. You cannot set the hidden data, e.g. **`vpointer`** to **`vtable`**
    - You cannot call the constructor manually
- Function overrides
  - Provide an alternative implementation in a sub-class
  - Non-virtual functions : type of pointer or calling function determines which function to call
  - Virtual functions : type of object the function applies to determines the function to call
    - Has to look up which function to call (e.g. in **`vtable`**?)
- You can use **`new`** on basic types (e.g. **`int`**)
  - By default they are NOT initialised
- Array **`new []`** uses default constructor for objects, and does not initialise basic types

Where to put objects?

- Objects can be created on the stack

```
MyClass ob1; // Use default constructor
MyClass ob2(3); // Provide initial value
MyClass obarray[4]; // Array of 4 elements
```

- Or in dynamic memory

```
MyClass* pOb1 = new MyClass;
MyClass* pOb2 = new MyClass(5);
MyClass* pObArray = new MyClass[6];
```

- In which case they need deleting

```
delete pOb1;
delete pOb2;
delete [] pObArray;
```

- A good rule of thumb (or heuristic):
‘**create things on the stack if you can, so that you don’t need to worry about deleting them**’
  – So, when the stack frame is destroyed, the objects will be destroyed

Default member functions

Automatically generated functions

- 4 functions created by default **if needed**
  - You can make them unavailable (e.g. private)
  - Or change their behaviour
- If they are needed, you will get:

1. A default constructor (no parameters needed)
2. A copy constructor (copy one object to another)
3. An assignment operator (**=** operator)
   - We will see general operator overloading later
4. A destructor

1: A default constructor

- A constructor which takes no parameters
  - Automatically created **if and only if** you do NOT create any other constructors
  - If you create **any** constructor, compiler will not create a default one for you
- The generated default constructor is empty
  - Does nothing: lets members construct themselves (using their default constructors)
- This is why you can still create objects, even when classes appear to have no constructors
- To prevent this, create your own (private?)

2: The Copy Constructor

- The **copy constructor** is used to **initialise** one object from another **of the same type**
- **This includes when a copy is implicitly made:**
  - **Passing object as a parameter into a function**
  - **Returning object *by value* from a function**
- **A copy constructor is created by default**
  - **Unless** you create your own
- Default behaviour copies each member in turn
  - **i.e. calls copy constructor for each member**
- Note: To avoid having a copy constructor, **declare** a **private** one **without implementation** (so the linker causes an error if it is used)

Creating a copy constructor

- You can define your own copy constructor, for example:

```
MyClass( const MyClass& rhs )
{ … }
```

- Takes a **constant reference** to the object to copy from (or a non-constant reference)
- Has to be a reference! (to avoid needing to copy)
  - If not a reference then copying parameter value (to pass in) means copying the object
  - i.e. would need to have copy constructor to implement the copy constructor

All of these are initialisation

- All five of these are initialisation:

```
MyClass ob1( 1, 2, 3 );
MyClass ob2 = MyClass( 1, 2, 3 );
```

Identical: call `(int,int,int)` constructor

```
MyClass ob3( ob2 );
MyClass ob4 = ob2;
MyClass ob5 = MyClass( ob2 );
```

Identical: call `(const MyClass&)` constructor

- First two are same. Last three are same.
- The last three all use copy constructor!
- Why?
  - Because it is ***defined*** in the standard to be so
  - It is faster to initialise than initialise+assign

Example : Copy(ish) constructor

```
class Example
{
public:
   Example( int iVal = 1)
   : m_iVal(iVal)  {}

   // WARNING – not exact copy!
   Example( const Example& rhs)
   : m_iVal(rhs.m_iVal+1)
   { }

   void print()
   {
        printf("%d\n", m_iVal);
   }
private:
   int m_iVal;
};
```

```
int main()
{
   Example eg1;
   Example eg2(2);
   // Initialisation:
   Example eg3 = eg2;
   Example eg4;

   // Assignment
   eg4 = eg2;

   eg1.print();
   eg2.print();
   eg3.print();
   eg4.print();
   return 0;
}
```

3: Assignment operator

- Used when value of one object is assigned to another
- Assignment operator will be created by default, if needed
  - Unless you create one yourself
- Default one does member-wise assignment
  - i.e. calls assignment operator for each member
  - To prevent this, declare private one without implementation

- Create your own using **operator overloading**:

```
MyClass& operator=( const MyClass& rhs )
{
    /* Assign the members here */
    return *this;
}
```

- Takes a reference to the one we are getting values from
- Returns a reference to `*this`, so we can chain these
  - e.g.: `ob1 = ob2 = ob3 = ob4;`

Example : Assignment operator

```
class Example
{
public:
  Example( int iVal = 1)
  : m_iVal(iVal)  {}

  Example& operator=(
      const Example& rhs)
  {
      m_iVal = rhs.m_iVal + 10;
      return *this;
  }

  void print()
  { printf("%d\n", m_iVal); }
private:
  int m_iVal;
};
```

```
int main()
{
  Example eg5(4);
  Example eg6(5);
  Example eg7(6);

  // Assignment
  eg7 = eg6 = eg5;

  eg5.print();
  eg6.print();
  eg7.print();

  return 0;
}
```

4: Destructor

- A destructor is created if you do not create one yourself
- Default destructor does nothing
  - Member destructors get called as members get destroyed
  - Destructors for objects are called
  - Basic data types (e.g. `int`) just get destroyed
    - No need for destructors
  - Pointers just get destroyed
    - **The thing they point to will NOT**!

A ‘default’ implementation

```
class MyClass
{
public:

    // Constructor
    MyClass()
    {
    }

    // Destructor
    ~MyClass()
    {
    }

    // Copy constructor
    MyClass( const MyClass& rhs )
        // Initialise each member
        : i( rhs.i )
    {
    }

    // Assignment operator
    MyClass& operator=(
        const MyClass& rhs )
    {
        // Copy each member
        return *this;
    }
};
```

General rule (rule of three)

- If you need to create one of:
  - a **copy constructor**
  - or an **assignment operator**

  then you probably need to create the other, plus a **destructor** as well
- Decide: Do you need to implement them?
  - If you control resources (or memory on the heap that you need to free) then you probably do
- Decide: Should users be able to copy the objects at all and, if so, then will the default copy mechanism be adequate?

Conversion constructors

Reminder: implicit functions

```
class MyClass
{
private:
   int i;

public:
   // Constructor
   MyClass()
   { }

   // Destructor
   ~MyClass()
   { }

   // Copy constructor
   MyClass( const MyClass& rhs )
      // Initialise each member
      : i( rhs.i )
   {
   }
   // Assignment operator
   MyClass& operator=(
      const MyClass& rhs )
   {
      // Copy each member
      i = rhs.i;
      return *this;
   }
};
```

Conversion constructor

- A conversion constructor is a constructor with one parameter. e.g. Constructor for **`MyClass`**:

```
MyClass( char c )
{ … do something with c … }
```

  – **Then you can use the following code:**

```
MyClass ob = 'h';
```

- Conversion constructor converts from one type of object to another
  – Can be used **implicitly** to convert between types (unless you say otherwise)
- The conversion constructor is very similar to the copy constructor, i.e.

```
MyClass( const MyClass& rhs )
{ … Copy the members … }
```

Conversion constructor

```
class Converter
{
public:
   // Conversion constructor
   // Convert INTO this class
   Converter( int i = 4 );

private:
   int _i;
};
```

```
int main()
{
   int i = 4;
   // Construction from int
   Converter c1(5);
   Converter c2 = i;
}
```

```
// Conversion constructor
Converter::Converter(int i)
   : _i(i) // Set value
{
   cout << "Constructing from int\n";
}
```

Forcing explicit construction

- Providing a **one-parameter** constructor provides a conversion constructor
- This allows compiler to use it to convert to the type whenever it wants/needs to do so
- To avoid this, use the keyword explicit
  - Constructor can then ONLY be used explicitly

```
class MyClass
{
public:
   explicit MyClass( int param );
};
```

Example of ‘explicit’

```
struct MyClass
{
   MyClass( int );
};

MyClass::MyClass( int i )
{
   cout << "Constructor M "
        << i << endl;
}

struct ExplicitClass
{
   explicit
        ExplicitClass( int );
};
```

struct defaults to public

```
ExplicitClass::
   ExplicitClass( int i )
{
   cout << "Constructor E "
        << i << endl;
}

int main()
{
   // Call constructor
   MyClass m1(1);
   MyClass m7 = 5;
   // Call constructor
   ExplicitClass e1(100);
   // Cannot do this:
   ExplicitClass e7 = 300;
}
```

Next Lecture

- Inheritance and constructors
  - Virtual destructors
- Namespaces and scoping
- Some standard class library classes
  - String
  - Input and output
  - Container classes