G52CPP
C++ Programming
Lecture 9

Dr Jason Atkin

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

Last lecture

- const
  - Constants, including pointers
- The C pre-processor
  - And macros
- Compiling and linking
  - And multiple header files
- Linkage and visibility

- Demos of these things later today

Avoiding multiple inclusion

- Code to include the contents of a file only once:

```
#ifndef UNIQUE_DEFINE_NAME_FOR_FILE
#define UNIQUE_DEFINE_NAME_FOR_FILE
… include the rest of the file here …
#endif
```

- To work, the name in the **`#define`** has to be unique throughout the program
  - E.g. you probably should include the path of the header file, not just the filename
  - Example: mycode/game/graphics/screen.h could be called **`MYCODE_GAME_GRAPHICS_SCREEN_H`**
  - By convention, **`#defines`** are in upper case

Example coursework file

```
#ifndef DISPLAYABLEOBJECT_H  ←  If not already marked as included
#define DISPLAYABLEOBJECT_H  ← Mark it as included now, by setting the #define

#include "BaseEngine.h"  ←  Includes the header files it needs

class DisplayableObject
{
public:
       // Constructor
       DisplayableObject(BaseEngine* pEngine);
       // Destructor
       virtual ~DisplayableObject(void);
private:
       // True if item is visible
       bool m_bVisible;
};

#endif  ←  End of the #ifdef around the contents
```

Three rules for header files

1. Ensure that the header file **`#include`**s everything that it needs itself
   – i.e. **`#include`** any headers it depends upon
2. Ensure that it doesn’t matter if the header file is included multiple times
   – See previous slides
3. Ensure that header files can be included in any order
   – A consequence of the first two rules

#define and macro definitions

- You can use **`#define`** to define a **macro**:

```
#define max(a,b) (((a)>(b)) ? (a) : (b))

int v1 = max( 40, 234 );
int v1 = (((40)>(234)) ? (40) : (234))

int v2 = max( v1, 99 );
int v2 =(((v1)>(99)) ? (v1) : (99))

int v3 = max ( v1, v2 );
int v3 =(((v1)>(v2)) ? (v1) : (v2))
```

- **Remember: done by the pre-processor!**
  - **NOT a function call**

What is the output here?

MyHeader.h

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

#define max(a,b) (((a)>(b)) ? (a) : (b))

#endif
```

MyTest.cpp

```
#include <cstdio>
#include “MyHeader.h”
int main( int argc, char* argv[] )
{
  int a = 1, b = 1;
  while ( a < 10 )
  {
      printf( “a = %d, b = %d “, a, b );
      printf( “max = %d\n”, max(a++,b++) );
  }
}
```

The (surprise?) output

```
printf( “a = %d, b = %d “, a, b );
printf( “max = %d\n”, max(a++,b++) );
```

- **The output is:**

```
a = 1, b = 1 max = 2
a = 2, b = 3 max = 4
a = 3, b = 5 max = 6
a = 4, b = 7 max = 8
a = 5, b = 9 max = 10
a = 6, b = 11 max = 12
a = 7, b = 13 max = 14
a = 8, b = 15 max = 16
a = 9, b = 17 max = 18
```

- **Why?**

  `max( a++, b++ )` expands to:

  ```
      ((a++)>(b++)) ? (a++) : (b++)
  ```

- So, whichever number is greater will get incremented twice, and the lesser number only once

Warning about macros

- Do not use a macro where the evaluation of the parameters may have a side-effect

- E.g.

```
max(a++,b++)
```

- Evaluating these parameters alters a value
  - A side-effect

This lecture

- `class`es (and C++ `struct`s)

- Member functions

- `inline` functions

classes and structs

A very quick introduction

Something to refer back to

Differences between C++ and Java

C++ vs C structs

- Can still use `struct`s in C++
- Everything for structs so far applies to both C and C++
  - We will call them C-style structs
- **If you use only C features**, `struct`s in C++ work as for C, i.e. you can predict `sizeof()`, can `malloc()` space for them, etc
  - Everything we have seen so far is valid
- In C++ you can add functions to `struct`s
- **If you use ANY C++ only features** (e.g. add functions or use inheritance), **their behaviour may change**
  - If you have used ANY C++ only features, **DO NOT** try to treat them as C structs – you may get a nasty surprise
  - e.g. size may grow or it may have hidden parts (see later lectures)

classes and structs

- `class`es and `struct`s are (almost) the same thing in C++
- The difference is **(ONLY!!!)** in encapsulation
  - `struct` defaults to public, `class` to private
- **Everything you do with a `class` in C++ could also be done with a `struct`**
- Common coding practice in C++:
  - Data only and no member functions: use a `struct`
    - You get the guarantees about size and positions of member data that you get in a C `struct`
  - If you add member functions, use a `class`
  - Advice: use `struct` only for C-type `struct`s

Methods / member functions

- In C++, functions can be made **`class`**/**`struct`** members
  - Just like Java functions

```
#include <cstdio>

struct Print
{
    void print() { printf( "Test\n" ); }
};

int main()
{
    Print p;
    p.print();
}
```

Create a **`struct`** on stack as in C

Call a method on the **`struct`**
If we had a **`struct*`** we would use **`p->print();`**

Hiding data inside classes (or structs)

- Data and methods in a class have either **`public`** or **`private`** access
  - There is also **`protected`** – we will see later
- **`public`** methods and data can be accessed by anything
  - Like non-static global functions/data in a file
- **`private`** methods and data can only be accessed by other members of the **SAME class**
  - Like static global functions/data in a file
- **Note: There is no ‘package only’ access**
- **`class`** members **default** to private access
- **`struct`** members **default** to public access

Methods/functions and data

- Data **should** (usually) be private
  - If it is not, then have a VERY good justification
- Methods (functions) **should** be:

  `private` for internal use only

  `public` for the external class interface
- **The values of the data members comprise the state of the object**
- Interface methods can be:

  Mutators – change the ‘state’ of the object

  Accessors – only query values, no changes
- Note: inline functions (see later) for methods ensure that it is no slower at runtime to use accessors than to use the variable names

`public` and `private`

- Keyword `private:` will change access to private from **then onwards**
- Keyword `public:` will change access to public

```
class DemoClass
{
public:
  int GetValue() { return m_iValue; }
  void SetValue( int iValue )
                 { m_iValue = iValue; }
private:
  int m_iValue;
};
```

public: for the interface
Public from this point onward

private: for data and internal functions
Private from this point onwards

Member functions and data

- Member data should be private
- Accessor and mutator functions could be public including ‘getters and setters’

```
class DemoClass
{
public:
  int GetValue() { return m_iValue; }
  void SetValue( int iValue )
             { m_iValue = iValue; }
private:
  int m_iValue;
};
```

Methods/member functions/operations

Member data/attributes/state

Semi-colon at the end

Some advance knowledge...

- You can use inheritance, e.g.:

  ```
  class SubClass : public BaseClass
  { <Data and methods> }
  ```

  - Like extends in Java

- Member functions can access the data in **`class`**es or **`struct`**s
  - There is a hidden **`this`** pointer
    - Like the hidden **`this`** object reference in Java
  - Use **`this->`** not **`this.`**

- **`static`** member data and functions work as per Java, shared between instances, no **`this`** pointer

Constructors and destructors

Constructors and Destructors

- **Constructor**
  - Called when an object is created
  - Has function name same as class name
  - And no return type (none/empty, NOT **`void`**!)
  - Adding a constructor makes it impossible to provide a C-style initialiser. e.g. **`= {0,1,2};`**
    - Look back at the slides on initialisers for structs in C
    - No constructor => you **can** use the C-style initialiser
- **Destructor** (*similar* to Java finalize)
  - Called when an object is destroyed
  - A function with name ~ then class name
    - E.g.: **`~DemoClass()`**
  - And no return type

Example C++ class

```
class DemoClass
{
public:
   DemoClass()
   {  }

   ~DemoClass()
   {  }

   int GetValue() const { return m_iValue; }

   void SetValue( int iValue ) { m_iValue = iValue; }

private:
   int m_iValue;
};
```

Constructor
No return type

Destructor
No return type

Accessor
Access only, no changes
Ideally label the function with keyword ‘**const**’
(see later lecture for why)

Mutator. Mutates/changes the object

Data member/member variable/attribute

Constructor parameters

- You can pass parameters to constructors
- You can have multiple constructors
  - Which differ in which parameter types they expect
  - The compiler will consider which parameters are passed in order to determine which constructor to use
    - In the same way as functional overloading
  - You are probably used to this from Java
- General C++ rule: **if your code introduces ambiguity** (i.e. this could mean A or B) **then it will not compile**
  - If the constructor that the compiler should call is ambiguous, the code will not compile!

Passing parameters to constructors

1. Create a constructor which takes parameters
   - e.g. a constructor which takes an `int`:

```
DemoClass(int iValue)
{ … } // In class DemoClass
```

2. To create an object **on the stack**, passing values to constructor use:

```
DemoClass myDemoClass(4);
```

Default parameters

- In C++, parameters can have default values
  - So can parameters in constructors
- Use the '**`= <value>`**' syntax following the parameter declaration
- e.g.:

```
DemoClass( char* dummy,
                int iValue = -1)
{ /*Nothing*/ }
```

- Will match any of the following:

```
DemoClass myDemoClass3( "Temp", 3 );
DemoClass myDemoClass4( "Temp" );
```

- Default values appear only in the function **declaration**, not any **separate** definition

Default Constructor

- The ‘Default Constructor’ is a constructor which **can be called** with no parameters
  - e.g. one which **has no** parameters
  - **or** has **default** values for **all** parameters
  - A class can only have one default constructor
    - More would introduce ambiguity
- When you create arrays ***of objects***, the default constructor is used (because no parameters are provided):

  e.g.: `DemoClass myDemoArray[4];`

Constructor parameters or not?

- Create an object, using default constructor

  `DemoClass myDemoClass1;`

- Or create an object, passing values to the constructor (selects the constructor to use)

  `DemoClass myDemoClass3( "Temp" );`

IMPORTANT: Do **NOT** add empty brackets `()` **when constructing on the stack** if there are no parameters!

  – Compiler thinks you are ***declaring* a function**

  – e.g. `DemoClass myDemoClass1(); // WRONG!!!`

Basic types

- **Basic types can be initialised in the same way as classes (using the brackets)**

- Create an `int` (we have seen this a lot)

```
int iVal = 4; // Initialisation!
```

- The `()` form can also be used for basic types

```
int iVal(4); // Initialisation!
```

- Both do exactly the same thing

Initialisation list

- Initialisation list allows you to pass values to:
  - Data member constructors
  - Base class constructors

- Uses the `()` form of initialisation
  - i.e. initialisation values to use are inside `()`
- Uses the `:` operator following the constructor parameters (before the opening brace):

```
DemoClass(int iValue)
: m_iValue(iValue)
{}
```

Initialisation list, comma separated

Example Initialisation List

```
class DemoClass
{
public:
  DemoClass( int iValue )
      : m_iValue(iValue)
      { … }

  ~DemoClass() { … }

  int GetValue()             { return m_iValue; }

  void SetValue(int iValue) { m_iValue = iValue; }

private:
  int m_iValue;
};
```

Two ways to set member values

- With an `int` type data member called `m_iValue`
- Compare the following:

```
DemoClass(int iValue)
: m_iValue(iValue)
{}
```

- With the following:

```
DemoClass(int iValue)
{
  m_iValue = iValue;
}
```

- **Question: Are these the same?**

Initialisation vs Assignment (1)

```
class DemoClass
{
public:
    DemoClass( int iValue )
        : m_iValue(iValue)
        { … }

    DemoClass(int iValue)
    {
      m_iValue = iValue;
    }

    …
```

Note: You could only have ONE of the following in a class, since they have the same parameters

`m_iValue` is **initialised** with value `iValue`

`m_iValue` is created but **not** initialised

then the value of `iValue` is **assigned** to it

If it was an object (of type struct/class) it would be initialised using default constructor, then assigned i.e. value would be set twice!

Initialisation vs Assignment (2)

- Compare the following:

1)
```
int i = 4; // Initialisation
```

2)
```
int j; // Uninitialised
j = 4; // Assignment
```

- Initialisation lists are used a LOT in C++
  - Should be used in preference to member **assignment**
- **Not available in Java!**
- In Java you use super() to pass parameters to base class constructor, and then just ***assign*** values to members in the constructor
  - In C++ you use the initialisation list for both
- Initialisation list can be faster in some cases
  - Avoids work from an unnecessary default constructor

Member data initialisation

- Member data is **NOT always** initialised
  - ***Basic types and pointers*** (e.g. `int`, `short` or `char*`) are ***NOT initialised***
    - You should always initialise them
  - ***Default constructor*** is called ***for members of type class/struct*** unless you say otherwise (using initialisation list)

BIG WARNING TO YOU!!!!

(I am warning you because the compiler won’t!)

Inline functions
Member functions and data

Inline functions

- Inline functions act **exactly** like normal functions but no function call is made (code is put in caller function)
- Use the keyword '`inline`', e.g.:

```
inline int max( int a, int b )
{ return a>b ? a : b; }

printf("%d\n", max(12,34) );
```

- Similar to a 'safe' macro expansion
  - **Safely** replaces the function call with the code
    - Unlike a macro (`#define`)
  - Avoids the overhead of creating a stack frame
  - Code gets included in EVERY file/function which calls it
- VERY useful for small, fast functions
- It is advice only: compiler can decide to ignore you

Function **definitions** ‘outside’ the class

- IN professional code, member functions are usually **defined** outside of the class declaration
  - In ***Java*** they are always defined within the class declaration, with one class per file
- In C++ you **usually** have:
  - Function **declaration** inside class declaration
  - Function **definition** somewhere else
    - With a ‘label’ to say it is a class member
    - We use the scoping operator `::` to label it
  - Reason: allows hiding of the implementation
    - Good program design, that Java’s policy makes very hard to do
- Defining functions **within** the class declaration **implicitly** makes them **`inline`**
  - As if they had ‘**`inline`**’ on them

Class Declaration and Definition

```
class DemoClass
{
public:
  DemoClass( int iValue = -1 )
      : m_iValue(iValue)
      { … }

  ~DemoClass() { … }

  int GetValue() { return m_iValue; }

  void SetValue(int iValue) { m_iValue = iValue; }

private:
  int m_iValue;
};
```

These are all inline functions.
Do not actually exist in executable as functions – their code is included INLINE in the caller

Defining class member functions

DemoClass.h

```
class DemoClass
{
public:
   DemoClass
      ( int iValue = -1 );

   ~DemoClass();

   int GetValue();

   void SetValue(
        int iValue);

private:
   int m_iValue;
};
```

DemoClass.cpp

```
#include "DemoClass.h"

DemoClass::DemoClass
   ( int iValue )
: m_iValue(iValue)
{ … }

DemoClass::~DemoClass()
{ … }

int DemoClass::GetValue()
{ return m_iValue; }

void DemoClass::SetValue(
      int iValue )
{ m_iValue = iValue; }
```

Note: Default value

DemoClass.h

```
class DemoClass
{
public:
   DemoClass
      ( int iValue = -1 );

   ~DemoClass();

   int GetValue();

   void SetValue(
        int iValue);

private:
   int m_iValue;
};
```

DemoClass.cpp

```
#include "DemoClass.h"

DemoClass::DemoClass
   ( int iValue )
: m_iValue(iValue)
{ … }

…
```

Functions in C++ can have default values for parameters. Specify these in the function declaration, not the definition

Demo lecture

- **Please try the coursework lab A BEFORE the demo lecture**
- The coursework framework
  - Class files and header files
  - Function definitions outside the header file
  - Constructors and destructors
    - Initialisation list
  - Inline functions
  - Mainfunction.cpp
- Also, inheritance (single) – base classes
  - Base class : BaseGameEngine

Next lecture

- new and delete
- Inheritance
- Virtual functions