G52CPP
C++ Programming
Lecture 9

Dr Jason Atkin

This lecture

- **`class`**es (and C++ **`struct`**s)
- Member functions
- **`inline`** functions

classes and structs

A brief introduction

Something to refer back to

Differences between C++ and Java

C-style structs

- Can still use **`structs`** in C++
- Everything for structs so far applies to both C and C++
  - We will call them C-style structs
  - These are usually called POD structs (Plain Old Data)
- **If you use only C features:**
  - **`structs`** 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 for POD structs**
- Note: C++11 uses the terms ‘trivially-copyable’ and ‘standard-layout’
  - The restrictions are loosened compared with C++98/C++03
  - Look up these terms if you are interested, and see:

    http://en.cppreference.com/w/cpp/types/is_trivially_copyable

    http://en.cppreference.com/w/cpp/types/is_standard_layout

C++ vs C structs

- In C++ you can add functions to `structs`
  - And inheritance, static members, etc, etc, etc
- **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)
- Implementing some of these features MAY require that some hidden data is created
  - You would have to set or copy this data if you create or copy the objects, but don't know what it is!
- Format and implementation NOT defined in the standard for non-POD structs
  - Implementation dependent

classes vs structs in C++

- **`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 (etc) then 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
    - These should really be '`const`' functions, see later lecture
- 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
    - C++11 provides alternatives and relaxes rules
- **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
  - Including parameters for 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

IMPORTANT

- 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!!!! **(I am warning you because the compiler won't!)**

```
class MyClass
{
public:
    int ai[4];
    short j;
};
```

- Member data of basic types will be uninitialised
- Use the initialisation list to initialise variable
- Use the constructor to set values for arrays
  - The default constructors do nothing

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. Probably do not actually exist as functions in executable – 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

Next lecture

- References
- new and delete