G52CPP
C++ Programming
Lecture 14

Dr Jason Atkin

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

Last Lecture

- Automatically created methods:
  - A default constructor – so that objects can be created without defining a constructor
  - A copy constructor – used to copy objects
  - An assignment operator – an example of operator overloading : changing the meaning of an operator (i.e. **=**)
  - A destructor – calls member destructors
- Conversion constructors

This Lecture

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

Inheritance and constructors

Construction and destruction (1)

```
struct Base
{
    Base()
   {
        printf("Base constructed\n");
   }

   ~Base()
   {
        printf("Base destroyed\n");
   }
};

struct Derived : public Base
{
   …
}

int main()
{
        Derived d;
}
```

Construction and destruction (1)

lec14a.cpp

```
struct Base
{
   Base()
   {
       printf( "Base constructed\n" );
   }

   ~Base()
   {
       printf( "Base destroyed\n" );
   }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
{ Derived d; }
```

Purpose:

Create object d, allow it to be destroyed as stack frame exits.

Output:

?

Construction and destruction (1)

```
struct Base
{
   Base()
   {
       printf( "Base constructed\n" );
   }

   ~Base()
   {
       printf( "Base destroyed\n" );
   }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

lec14a.cpp

Source Code:

```
{ Derived d; }
```

Purpose:

Create object d, allow it to be destroyed as stack frame exits.

Output:

```
Base constructed
Derived constructed

Derived destroyed
Base destroyed
```

Construction and destruction (1)

```
struct Base
{
     Base()
    {
          printf("Base constructed\n");
    }

    ~Base()
    {
          printf("Base destroyed\n");
    }
};

struct Derived : public Base
{
    …
}

int main()
{
          Derived* pD = new Derived;
          delete pD;
}
```

Construction and destruction (2)

```
struct Base
{
   Base()
   {
       printf( "Base constructed\n" );
   }

   ~Base()
   {
       printf( "Base destroyed\n" );
   }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

lec14b.cpp

Source Code:

```
Derived* pD =
      new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it

Output:

?

?

Construction and destruction (2)

lec14b.cpp

```
struct Base
{
   Base()
   {
       printf( "Base constructed\n" );
   }

   ~Base()
   {
       printf( "Base destroyed\n" );
   }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
Derived* pD =
      new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it

Output:

```
Base constructed
Derived constructed

Derived destroyed
Base destroyed
```

Constructors and destructors

- Construction occurs in the order:
  - Base class first, then derived class
- Destruction occurs in the order:
  - Derived class first, then base class
- Effects:
  - **Derived class part of the object can always assume that base class part exists**
    - Derived class can assume that the base class has been constructed when the derived class is constructed
    - Derived class can assume that the base class has not yet been destroyed at the point the derived destructor is used
  - Derived class will NOT exist/be initialised when the base class constructor/destructor is called, so:
  - **Do not call virtual functions from the constructor or destructor**

Construction and destruction (3)

```
struct Base
{
    Base()
   {
       printf("Base constructed\n");
   }

   ~Base()
   {
       printf("Base destroyed\n");
   }
};

struct Derived : public Base
{
   …
}
```

```
Base* pD = new Derived;
delete pD;
```

Construction and destruction (3)

```
struct Base
{
   Base()
   {
       printf( "Base constructed\n" );
   }

   ~Base()
   {
       printf( "Base destroyed\n" );
   }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

lec14c.cpp

Source Code:

```
Base* pD =
      new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it through a **base class pointer**

Output:

?

Construction and destruction (3)

lec14c.cpp

```
struct Base
{
   Base()
   {
       printf( "Base constructed\n" );
   }

   ~Base()
   {
       printf( "Base destroyed\n" );
   }
};

struct Derived : public Base
{
   Derived()
   { printf("Derived constructed\n"); }

   ~Derived()
   { printf("Derived destroyed\n"); }
};
```

Source Code:

```
Base* pD =
      new Derived;
delete pD;
```

Purpose:

Create object d, then destroy it through a **base class pointer**

Output:

```
Base constructed
Derived constructed
Base destroyed

NOT Derived destroyed
```

Construction and destruction (4)

```
struct VirtualBase
{
   VirtualBase()
   {
       printf("Base constructed\n");
   }

   virtual ~VirtualBase()
   {
       printf("Base destroyed\n");
   }
};

struct VirtualDerived : public virtualBase
{
   …
}
```

Virtual Destructor

```
VirtualBase* pD = new VirtualDerived;
delete pD;
```

Construction and destruction (4)

```
struct VirtualBase
{
   VirtualBase()
   { printf("Base constructed\n"); }

   virtual ~VirtualBase()
   { printf("Base destroyed\n"); }

};

struct VirtualDerived : public
   VirtualBase
{
   VirtualDerived()
{printf("Derived constructed\n");}

   ~VirtualDerived()
{ printf("Derived destroyed\n");}

};
```

`lec14d.cpp`

Source Code:

```
VirtualBase* pD =
      new VirtualDerived;
delete pD;
```

Purpose:

Create object d, then destroy it through base class pointer.

Output:

?

Construction and destruction (4)

```
struct VirtualBase
{
   VirtualBase()
{printf("Base constructed\n");}

   virtual ~VirtualBase()
{printf("Base destroyed\n");}

};

struct VirtualDerived : public
   VirtualBase
{
   VirtualDerived()
{printf("Derived constructed\n");}

   ~VirtualDerived()
{ printf("Derived destroyed\n");}

};
```

Source Code:

```
VirtualBase* pD =
      new VirtualDerived;
delete pD;
```

Purpose:

Create object d, then destroy it through base class pointer.

Output:

```
Base constructed
Derived constructed
Derived destroyed
Base destroyed
```

Virtual destructors

- **If destructor is NOT `virtual` then it will NOT be called if the object is destroyed through a base class pointer, reference or function**
  - **Since type of pointer/reference/function will determine the destructor to call**

- But, if you make destructor **`virtual`** then the objects of that class will have a (hidden) vtable pointer (or equivalent)
  - i.e. they grow

Virtual destructors: Question

- **Do we make the destructor virtual or not?**
- My advice: (only advice!!!)
  - Make it virtual if and only if there are **ANY other** virtual functions
    - No loss since vtable pointer already exists anyway
    - Probably using object through a base class pointer/reference, so object potentially COULD be destroyed that way too
  - If there are no other virtual functions **AND** you do **not** expect the object to be **`delete`**d through a pointer or reference to the base class **THEN** do **not** make your destructor **`virtual`**
    - Otherwise you add an unnecessary vtable pointer (or equivalent) to objects

Scoping

Calling base-class functions

- If a function is virtual, you can still call the base class version from the sub-class version
  - Useful so that you don't need to repeat code
- From Java you can call the (immediate) super-class version of a method from within a method
  - Uses the **`super.foo()`** notation
- The C++ version is more flexible…
  - You can call any base-class version, not just the *immediate* base-class
- C++ uses the scoping operator **`::`**
  - Example…

Example of scoping operator

```
class Base
{
public:
    virtual void DoSomething()
    { x = x + 5; }
private:
    int x;
};
class Derived : public Base
{
public:
    virtual void DoSomething()
    {
        y = y + 5;
        Base::DoSomething();
    }
private:
    int y;
};
```

Base class version of **DoSomething()** adds 5 to x

Derived class version of **DoSomething()** adds 5 to y THEN calls the base class version, which will add 5 to x

This EXPLICITLY calls the base-class version

Namespaces and scoping

Namespaces

- Namespaces are used to avoid name conflicts
  - Only the name is affected
- To put code in a namespace use:

```
namespace <NamespaceName>
{
  <put code for classes or functions here>
}
```

- Can use scoping to specify a namespace to ‘look in’:

```
  <namespace>::<class>::<function>
```

e.g. `MyNameSpace::MyClass::foo();`

```
  <namespace>::<globalfunction>
```

e.g. `MyNameSpace::bar();`

Namespaces

- Can avoid needing to keep saying **`<namespace>::`** specify '**`using namespace <namespace>`**'
  - **From that point onwards** the namespace will be checked when resolving names

- The standard class library is in the **`std`** namespace
  - The C-type functions are also in the global (unnamed) namespace, so we have been able to ignore namespaces so far
  - A common line near to the start of C++ programs:

```
using namespace std;
```

Example of namespace

lec14e.cpp

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

namespace cfj
{
   void MyPrintFunction1()
   {
       // Do something
   }

// Function in cfj namespace,
// so can use MyPrintFunction1
// without '::' or 'using'
   void MyPrintFunction2()
   {
       MyPrintFunction1();
   }
}
```

```
// Not in cfj namespace!
void MyPrintFunction3()
{
   cfj::MyPrintFunction1();
}

using namespace cfj;
// From this point onwards,
// cfj namespace will be
// checked

int main()
{
   string s1( "Test string" );
   int i = 1;

   MyPrintFunction1();
   MyPrintFunction2();
   MyPrintFunction3();
}
```

The scoping operator

- You can use the scoping operator to call global functions or access global variables
  - use `::` with nothing before it
- Also used to denote that a function is a class member in a definition, e.g.

```
void Sub::modify() { … }
```

- Left of scoping operator is
  - **blank** (to access a global variable/function)
  - **class name** (to access member of that class)
  - **namespace name** (to use that namespace)

Using scoping to access data

```
#include <cstdio>
int i = 1; // Global

struct Base
{
   int i;

   Base()
   : i(3)
   {}
};

struct Sub : public Base
{
   int i;

   Sub()
   : i(2)
   {}
```

lec14f.cpp

```
   void modify()
   {
       int i = 7;    // Local
       ::i = 4;      // Global
       Sub::i = 5;   // Sub's i
       Base::i = 6;  // Base's i
   }
};

int main()
{
   Sub s;
   printf( "%d %d %d\n",
       i, s.i, s.Base::i );
   s.modify();
   printf( "%d %d %d\n",
       i, s.i, s.Base::i );
   return 0;
}
```

Standard class library classes

An introduction

We will see more later

string and std namespace

- The string class is in the **`std`** namespace
- Can be accessed as **`std::string`**
- Three of the string constructors:

```
string();
```

  - Default empty string

```
string(const char* str);
```

  - From a **`char*`** type string

```
string(const string& str);
```

  - From another string – the copy constructor
- **`#include <string>`** for declarations

string class – for reference

- **string class has many member functions**

`append()`	concatenate more text to the string
`substr()`	return a substring of some size
`insert()`	insert some text into the string
`replace()`	replace part of a string
`erase()`	delete/remove part of a string
`assign()`	replace content of string
`compare()`	lexically compare two strings
`find()`	search for some text in the string
`rfind()`	find, starting at the end
`c_str()`	obtain a const char* for the string

- **And overloads a number of operators**

Assignment:	`=`					
Comparison:	`==`	`!=`	`<`	`<=`	`>`	`>=`
Concatenation:	`+`	`+=`				
Character at:	`[]`					

streams for input/output

- C++ input/output **classes** use streams
- Three standard streams exist already
  - **`istream cin;`** (matches **`stdin`**)
  - **`ostream cout;`** (matches **`stdout`**)
  - **`ostream cerr;`** (matches **`stderr`**)
- Header file includes the declarations:
  - **`#include <iostream>`**
- They are in **`std`** namespace
  - Use **`std::cin`**, **`std::cout`**, etc
- **`>>`** and **`<<`** operators overloaded for input / output
- **`endl`** sent to a stream will output **`\n`** and flush

Example

```
#include <string>
#include <iostream>

using namespace std;

int main()
{
   string s1( "Test string" );
   int i = 1;

   cin >> i;

   cout << s1 << " " << i << endl;

   cerr << s1.c_str() << endl;
}
```

Example

```
#include <string>
#include <iostream>

using namespace std;

int main()
{
   string s1( "Test string" );
   int i = 1;

   cin >> i;

   cout << s1 << " " << i << endl;

   cerr << s1.c_str() << endl;
}
```

Header files for string and i/o

Look in **`std`** namespace for the names which follow e.g. **`cin, cout, string`**

Overloaded operator - input

Overloaded operator - output

Convert **`string`** to **`const char*`**

File access using streams

- **`ifstream`** object - open the file for input
- **`ofstream`** object - open the file for output
- **`fstream`** object – specify what to open file for
  - Takes an extra parameter on open (input/output/both)

- Use the **`<<`** and **`>>`** operators to read/write
- In the same way as for **`cin`** and **`cout`**

- Simple examples follow, for reference
- Read the documentation for more information

File output example

```
#include <fstream>
using namespace std;
int main()
{
  ofstream file;
  // Open a file
  file.open("file.txt");
  // Write to file
  file << "Hello file\n“ << 75;
  // Manually close file
  file.close();
  return 0;
}
```

Since the `ofstream` object is destroyed (with the stack frame) the file would close anyway

File input example

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
  ifstream file;
  char output[100];
  string str;
  int x;
  file.open("file.txt");
  file >> output;
  file >> str;
  file >> x;
  file.close();
  cout << output << endl;
  cout << str << endl;
  cout << x << endl;
}
```

Note that the array has enough space to hold the loaded data

Text loaded (and output using **`cout`**) matches what was written in the previous sample

Those people struggling with **`char*`**s may want to consider string for the CW

stringstream

```
#include <iostream>
#include <sstream>

using namespace std;

int main()
{
   stringstream strstream;
   string str;

   short year = 1996;
   short month = 7;
   short day = 28;

   strstream << year << "/";
   strstream << month << "/";
   strstream << day;

   strstream >> str;

   cout << "date: " << str
                    << endl;

   return 0;
}
```

Send data to the **stringstream** object, a bit at a time

Extract it out again afterwards, as one string

I prefer **sprintf()**, for easier formatting, but this is 'more C++'

`string/stream` comments

- **You may use the standard C++ classes in the coursework if you wish**
  - **Including the string or stream classes**
- `wstring` is a wide-character version
  - `basic_string` is a template class
  - `string` and `wstring` are instantiations
- `string` is also a container class
  - Can be treated as a container
  - e.g. use `size()`
- Know these exist, and how they are used

Exam: do I need to know all of this?

- I will expect you to be able to understand code that you have seen in lectures
  – i.e. if you can understand the lecture slides and samples and what the code does then you meet that criterion
  – e.g. if you see the code `cout << x` then know that it sends x to the output (e.g. screen) and that it uses operator overloading to do this

- **I will expect you to know the basics of the standard C++ class library**
  – **i.e. things we cover in lectures**
  – **We will see something about the STL later**

Next Lecture

- Conversion Operators
- Friends

- Casting
  - static cast
  - dynamic cast
  - const cast
  - reinterpret cast