G52CPP
C++ Programming
Lecture 17

Dr Jason Atkin

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

Last Lecture

- Exceptions
  - How to ***throw*** (return) different error values as ***exceptions***
  - And ***catch*** the exceptions
  - Anything can be thrown
    - **But you should prefer to use Exception sub-classes**
  - Pointers and Objects/References are different
  - Sub-class gets caught by a base class catch

- RAII
  - Resource Acquisition Is Initialisation

This Lecture

- Operator overloading
  - Changing the meaning of an operator
- Some standard class library things
  - Illustrating operator overloading

String and stream classes

```
#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***

Operator overloading

Operator overloading

- Function overloading:
  - Change the meaning of a function according to the types of the parameters
- Operator overloading
  - Change the meaning of an operator according to the types of the parameters
- Change what an operator means?
  - Danger! Could make it harder to understand!
- Useful sometimes, do not overuse it
  - e.g. **+** to concatenate two strings

My new class: MyFloat

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

class MyFloat
{
public:
  // Constructors
  MyFloat( const char* szName, float f )
    : f(f), strName(szName) {}

  MyFloat( string strName, float f )
    : f(f), strName(strName) {}

private:
  float f;
  string strName;
};
```

char* and float

string and float

Internal string and float

Printing

```
// Constructors
MyFloat( const char* szName, float f )
  : f(f), strName(szName)
{}

// Print details of MyFloat
void print()
{
      cout << strName << " : " << f << endl;
}
```

```
Main function:
MyFloat f1("f1", 1.1f);
f1.print();
MyFloat f2("f2", 3.3f);
f2.print();
```

```
f1 : 1.1
f2 : 3.3
```

Conversion operators

```
// Print details of MyFloat
void print() { cout << strName << " : " << f << endl; }

// Conversion operators
operator string ()  { return strName; }
operator float ()   { return f; }
```

```
MyFloat f1("f1", 1.1f);
f1.print();
MyFloat f2("f2", 3.3f);
f2.print();

string s( f1 );
cout << "s: " << s << endl;
float f( f1 );
cout << "f: " << f << endl;
```

```
f1 : 1.1
f2 : 3.3
s: f1
f: 1.1
```

Non-member operator overload

```
MyFloat operator-( const MyFloat& lhs, const MyFloat& rhs )
{
   MyFloat temp(
            lhs.strName + "-" + rhs.strName, /* strName */
            lhs.f - rhs.f);      /* f, float value */
   return temp;
}
```

```
class MyFloat
{
  …
  // Non-member operator overload – friend can access private
  friend MyFloat operator-(
       const MyFloat& lhs, const MyFloat& rhs );
  …
}
```

Non-member operator overload

```
MyFloat operator-( const MyFloat& lhs, const MyFloat& rhs )
{
   MyFloat temp(
            lhs.strName + "-" + rhs.strName, /* strName */
            lhs.f - rhs.f);      /* f, float value */
   return temp;
}
```

```
MyFloat f3 = f1 - f2;

f3.print();


Output:      f1-f2 : -2.2
```

Or simplified version…

```
MyFloat operator-( const MyFloat& lhs, const MyFloat& rhs )
{
  return MyFloat(
              lhs.strName + "-" + rhs.strName,
              lhs.f - rhs.f );
}
```

```
MyFloat f3 = f1 - f2;

f3.print();
```

```
Output:       f1-f2 : -2.2
```

Member function version

```
MyFloat MyFloat::operator + ( const MyFloat& rhs ) const
{
  return MyFloat( this->strName + "+" + rhs.strName,
       this->f + rhs.f );
}
```

```
class MyFloat
{
public:
 // Member operator
 MyFloat operator+ (
        const MyFloat& rhs )
        const;
};
```

```
MyFloat f1("f1", 1.1f);
MyFloat f2("f2", 3.3f);
MyFloat f4 = f1 + f2;
f4.print();
```

```
f1+f2 : 4.4
```

Summary so far

```
int main()
{
MyFloat f1("f1", 1.1f);
   f1.print();
MyFloat f2("f2", 3.3f);
   f2.print();
MyFloat f3 = f1 - f2;
   f3.print();
MyFloat f4 = f1 + f2;
   f4.print();
string s( f4 );
cout << "s:" << s << endl;
float f( f4 );
cout << "f:" << f << endl;
}
```

```
class MyFloat
{
public:
  …

  // Member operator
  MyFloat operator+
    ( const MyFloat& rhs)
     const;

  // Non-member
  friend MyFloat operator-
    ( const MyFloat& lhs,
      const MyFloat& rhs );
};
```

Member vs non-member versions

```
// Member function:
MyFloat MyFloat::operator+ (const MyFloat& rhs) const

// Non-member function
friend MyFloat operator- (const MyFloat& lhs,
                          const MyFloat& rhs )
```

```
// These would work:
MyFloat f5 = f1.operator+( f2 );    f5.print();
MyFloat f6 = operator-( f1, f2 );   f6.print();

// These would not compile:
MyFloat f7 = operator+( f1, f2 );   f7.print();
MyFloat f8 = f1.operator-( f2 );    f8.print();
```

Operator overloading restrictions

- You cannot change an operator's precedence
  - i.e. the order of processing operators
- You cannot create new operators
  - Can only use the existing operators
- You cannot provide default parameter values
- You cannot change number of parameters (operands)
- You cannot override some operators:
  `::` `sizeof` `?:` or `.` (dot)
- You must overload `+`, `+=` etc separately
  - Overloading one does not overload the others
- Some can **only** be overloaded as member functions:
  `=` `,` `[]` and `->`
- Postfix and prefix `++` and `--` are different
  - Postfix has an unused `int` parameter

Post-increment vs pre-increment

```
MyFloat MyFloat::operator ++ ( int )
{
  MyFloat temp(
    string("(") + strName +")++", f );
  // NOW increment it
  f++;
  return temp;
}
```

```
MyFloat f9 = f5++;

cout << "Orig: ";
f5.print();
cout << "New : ";
f9.print();
```

```
MyFloat MyFloat::operator ++ ()
{
  ++f; // Increment f first
  strName =
    string("++(") + strName +")";
  return *this;
}
```

```
MyFloat f10 = ++f6;

cout << "Orig: ";
f6.print();
cout << "New : ";
f10.print();
```

Assignment and comparison

== vs = operators

```
class C
{
public:
   C( int v1=1, int v2=2 )
   : i1(v1), i2(v2)
   {}

   int i1, i2;
};

int main()
{
   C c1, c2;
   if ( c1 == c2 )
   {
       printf( "Match" );
   }
}
```

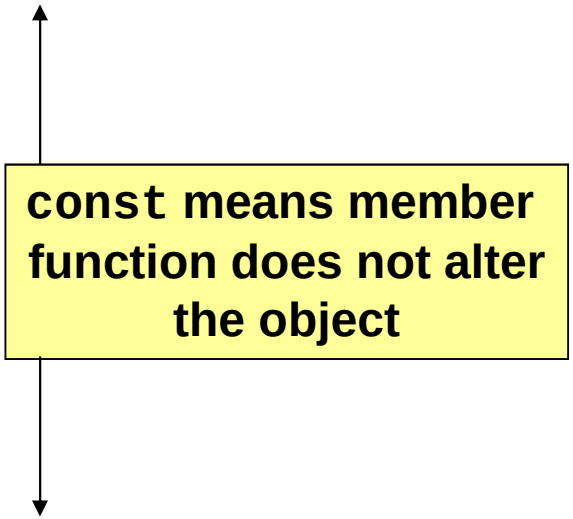
- The code on the left will NOT compile:

```
g++ file.cpp

In function `int main()':
file.cpp:17: error: no
   match for 'operator=='
   in 'c1 == c2'
```

- i.e. there is no **==** operator defined by **default**
- Pointers could be compared though, but not the objects themselves
- NB: Assignment operator IS defined by default (it is one of the four functions created by compiler when necessary)

!= can be defined using ==

```
bool MyClass::operator==
  (const MyClass &other) const
{
  // Compare values
  // Return true or false
}

bool MyClass::operator!=
  (const MyClass &other) const
{
  return !(*this == other);
}
```



+ and += are different

```
MyClass MyClass::operator+
  (const MyClass &other) const
{
  MyClass temp;
  // set temp.…  to be this->… + other.…
  return temp; // copy
}
```

const means member function does not alter the object
i.e. makes the this pointer constant

```
MyClass m1,m2,m3,m4;
m1 = m2 + m3 + m4;
```

```
MyClass& MyClass::operator+=
  (const MyClass &other)
{
  // set this->… to this->… + other.…
  return *this;
}
```

```
MyClass m1, m2, m3;
(m1 += m2) += m3;
```

Operator overloading summary

- Can define/change meaning of an operator, e.g.:

```
MyFlt operator-(const MyFlt&, const MyFlt&);
```

- You can make the functions member functions

```
MyFlt MyFlt::operator-(const MyFlt& rhs) const;
```

  - Left hand side is then the object it is acting upon
- Act like any other function, only syntax is different:
  - Converts `a-b` to `a.operator-(b)` or `operator-(a,b)`
- Access rights like any other function
  - e.g. has to be a `friend` or member to access `private`/`protected` member data/functions

- Also, parameter types can differ from each other, e.g.

```
MyFlt operator-( const MyFlt&, int );
```

  - Would allow an `int` to be subtracted from a `MyFlt`

Questions to ask yourself

- Define as a member or as a global?
  - If global then does it need to be a friend?
- What should the parameter types be?
  - References?
  - Make them `const` if you can
- What should the return type be?
  - Should it return `*this`?
  - Does it need to return a copy of the object?
    - e.g. post-increment must return a **copy**
- Should the function be `const`?

Operator overloading - what to know

- Know that you can change the meaning of operators
- Know that operator overloading is available as both member function version and global (non-member) function version
- Be able to provide the code for the overloading of an operator
  - Parameter types, `const`?
  - Return type
  - Simple implementations

More strings, streams and containers

Examples of operator overloading

Earlier example, again

```
#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;
}
```

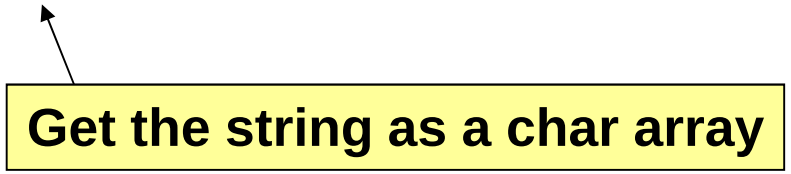
```
extern istream cin;
extern ostream cout;
extern ostream cerr;
```

>> is implemented for the **istream** class for each type of value on the left-hand side of the operator

Similarly for **ostream** and **<<**

My string comparison operator

```
bool operator==(  const std::string& s1,
                  const std::string& s2)
{
   return 0 == strcmp( s1.c_str(), s2.c_str() );
}

int main ()
{
   string str1( "Same" );
   string str2( "Same" );
   string str3( "Diff" );
   printf( "str1 and str2 are %s\n",
       (str1 == str2) ? "Same" : "Diff" );
   printf( "str1 and str3 are %s\n",
       (str1 == str3) ? "Same" : "Diff" );
   printf( "str2 and str3 are %s\n",
       (str2 == str3) ? "Same" : "Diff" );
}
```



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++’

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
- 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

Assume that the text loaded (and output using **cout**) matches what was written in the previous sample

```
file << "Hello file\n“ << 75;
```

SEM feedback

SEM Feedback – tick sheets

- Negative ticks were:

4	Size of the class is helpful
6	Module has helped your communication skills
3	Library resources helped
1	Module complements others I have studied
1	Method of assessment is appropriate (they thought it should be 100% coursework)

- One person disliked most of the above, also saying the pace was very wrong, the module did not help them to think critically and they had not had an opportunity to show what they had learned ☹
  - I guess it depends what 'show' means – I'd have thought the labs allowed this

- One person added extra 'peace' (I think) boxes to the assessment

More comments

- Multiple comments wanting more percentage on coursework (rather than an exam)
  - Problem is what I want to assess is whether you can be a ‘mechanic’ rather than a ‘driver’
  - Yes it’s practical but there are a lot of important underlying principles to test understanding of – hard to do in coursework
  - Also harder to differentiate in coursework
- Split it into multiple courseworks
  - Did that previously and many people did not submit the earlier ones – I don’t like people throwing away marks
  - If I split this on features, you need to decide early on what you want to do
  - Doing multiple smaller ones usually involves more work than one large one
  - You’ll get to practise good time management skills ☺

More comments

- Many things in lectures are of no use in the coursework or real world
  - I agree on the coursework part – I deliberately did not require a lot of the theory in the coursework – it's the 'can you drive' bit
  - The exam assesses a lot of the theory
  - *Almost* all of the things we cover in lectures I had to know at least at some point when programming in industry *(plus more!)*
- Give more walkthroughs of the framework, it's hard
  - A lot of it you can ignore, it will just work
  - I don't want to tell you everything – it would defeat the purpose
    - I want you to show that you can understand existing code
    - And adapt/reuse it in your own program
  - The demo lectures are aimed at helping you to do so if you want help, but there are limits to how far I want to go in that direction
  - Concentrate on the demo code not the framework

More comments

- Better not to have so much on Fridays
  - I agree ☹ I don't like it either
- Could change coursework to allow other than games
  - You don't have to do a game, but you do have to do something with animation etc.
  - By standardising some features which you have to provide, it makes it possible to assess you against each other, to see how much you understand of each of the key areas
  - Each requirement needs you to understand how to do a specific thing, which we can assess
- 60% Exam is daunting
  - Please take a look at the previous exams
  - They are not as bad as you may expect
  - Mainly tests understanding of concepts

What now…

What now

- Today: No demo lecture, but I'll go to the lecture room and answer any questions you have
- But you are probably better off going to the labs and finishing your group projects
- Have a good Easter Break
- Finish your coursework programs – I am encouraged by how far people have got
- Go through the slides and start thinking about the exam

Next lecture

- After Easter…
- Template functions
- Template Classes
- A **few** more, important, comments about the Standard Template Library (STL)
  - And the slicing problem