G52CPP
C++ Programming
Lecture 20

Dr Jason Atkin

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

Examples

```
#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;
}
```

Overloaded operator - input

Overloaded operator - output

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;
};
```

Wraps up a float
And a string describing it

char* and float

string and float

Internal string and float

Printing – member functions

```
// Constructor
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
```

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;
}
```

Calls the two-parameter constructor

Uses + on the strings

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

Needs to be a friend in this case, to access the private parts (no 'getters')

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 f1("f1", 1.1f);
MyFloat f2("f2", 3.3f);

MyFloat f3 = f1 - f2;

f3.print();
```

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

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

MyFloat f3 = f1 - f2;

f3.print();
```

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

```
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();
```

Output:
```
f1+f2 : 4.4
```

Member vs non-member versions

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

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

Member function:

('this' is lhs)

```
              MyFloat
            MyFloat::
          operator+ (
 const MyFloat& rhs
                    )
                const
```

```
Return type
LHS
RHS
```

Global function:

Explicitly state lhs

```
MyFloat
operator- (
const
   MyFloat& lhs,
const MyFloat& rhs
)
```

Differences ?

```
// 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();
```

Reminder : 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
```

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 );
};
```

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( // Make a copy
    string("(") + strName +")++", f );
  // NOW increment original
  f++;
  return temp; // Return the copy
}
```

```
MyFloat f9 = f5++;

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

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

```
MyFloat f10 = ++f6;

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

Assignment vs comparison

And + vs +=

== 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);
}
```

const means member function does not alter the object

+ 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;
```

Summary

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

Streams use 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;
}
```

Cin, cout and cerr are objects
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

i.e. different function called depending upon object type so it can change its behaviour

The stream object is returned as return value, to allow chaining together

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" );
}
```

Get the string as a char array

Does not need to be a 'friend'

Some exam comments

Operator overloading - what to know

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

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`**?

Next lecture

- Template functions
- Template Classes