G52CPP
C++ Programming
Lecture 18

Dr Jason Atkin

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

Welcome Back

Last lecture

- Operator Overloading
- Strings and streams

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`**?

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

streams for input/output

- C++ input/output **classes** use streams
- Three standard stream **objects** 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 are overloaded for input and output
- `endl` sent to a stream will output `\n` and flush

Example of output

```
#include <iostream>

using namespace std;

int main()
{
   const char* str = "Test string";
   int i = 1;

   cin >> i;

   cout << str << " " << i << endl;

   cerr << str << endl;

   return 0;
}
```

Header file for input AND output

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

Overloaded operator - input

Overloaded operator - output

This lecture

- Template functions
- Template Classes
- Standard Template Library
  - Only and introduction/overview

The plan now…

- No more labs, so use the 2pm lectures for questions
- Friday 26th April, 10am, Lecture 19
  - When is a duck an instrument? (Multiple Inheritance)
- Friday 26th April, 2pm – Optional
  - How to create programs fast
- Thursday 2nd May, 4pm, Lecture 20
  - Wrapping up (slicing problem, smart pointers)
- Friday 3rd May, 10am, Revision Lecture
  - Revision and exam strategy
- Friday 3rd May, 2pm - Optional
  - Any questions / examples
  - What do you want to 'revise'?

Function Overloading

- We can use function overloading to have multiple versions of the same function
- Consider the following functions:

```
int mymax( int a, int b )
  { return a > b ? a : b; }
float mymax( float a, float b )
  { return a > b ? a : b; }
char mymax( char a, char b )
  { return a > b ? a : b; }
```

- It would be nice to create just the one

Template version

```
int mymax( int a, int b )
  { return a > b ? a : b; }
float mymax( float a, float b )
  { return a > b ? a : b; }
char mymax( char a, char b )
  { return a > b ? a : b; }
```

```
template < typename T >
T mymax( T a, T b )
{ return a > b ? a : b; }
```

Template functions

- Templates specify how to **create** functions of a certain format, if they are ever needed, e.g.:

```
template < typename T >
T mymax( T a, T b )
{ return a > b ? a : b; }
```

- Note: you can use keyword **`class`** or **`typename`** :

  i.e. **`template < class T >`**

- Type placeholders are used, and are replaced implicitly
- Could use it as any type, e.g.:

```
int i1 = 4, i2 = 14;
int i3 = mymax( i1, i2 );
```

What templates do

- The compiler will **actually generate the functions which are needed**, according to the parameters
- i.e. at **compile time**, new functions are created
- If there are any problems, it will not compile
  - e.g. if new template class needs a function or operator which is not supported by the type
- This is **NOT** something done at runtime

Example for mymax

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

template < typename T >
  T mymax( T a, T b )
  { return a > b ? a : b; }


int main()
{
  int i1 = 4, i2 = 14;
  int i3 = mymax( i1, i2 );
  cout << "mymax(" << i1 << ","
     << 12 << ") = " << i3 << endl;
}
```

So that compiler knows what **cout** is when we use it later (and what **endl** is)

Compiler generates a function…

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

template < typename T >
  T mymax( T a, T b )
  { return a > b ? a : b; }


int main()
{
  int i1 = 4, i2 = 14;
  int i3 = mymax( i1, i2 );
  cout << "mymax(" << i1 << ","
     << 12 << ") = " << i3 << endl;
}
```

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

How to create template functions

- The easy way to create these template functions:
  - First manually generate a function for ***specific*** types
  - Next replace **all** copies of the types by an identifier
  - Then add the keyword **`template`** at the beginning and put the type(s) in the **`<>`** with keyword **`typename`** (or **`class`**)
- For example, an “addition with casting” function:

```
int addcast( int a, float b )
{ return a + static_cast<int>(b); }
```

- Becomes:

```
template <typename T1, typename T2>
T1 addcast( T1 a, T2 b )
{ return a + static_cast<T1>(b); }
```

- And can be used as:

```
int val = addcast( 12, 4.65 );
```

Example of addcast<T1,T2>

```
#include <iostream> // cout

using namespace std;

template <typename T1, typename T2>
   T1 addcast( T1 a, T2 b )
   { return a + static_cast<T1>(b); }

int main()
{
   int val = addcast( 12, 4.65 );

   cout << 12 << "+" << 4.65 << "=" << val << endl;

   return 0;
}
```

Creates a version which changes the float to an int and adds them

Question: Will this compile?

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

template <typename T1, typename T2>
T1 addcast( T1 a, T2 b )
{ return a + static_cast<T1>(b); }

class MyFloat
{
public:
  MyFloat( float f )
  : f(f)   {}

  float f;
};
```

Code in main:

```
MyFloat f1(1.1);
float f2 = 2.2;
MyFloat f3 = addcast(f1,f2);
cout << f3.f << endl;
```

The compilation error

```
template <typename T1, typename T2>
T1 addcast( T1 a, T2 b )
{ return a + static_cast<T1>(b); }

class MyFloat
{
public:
  MyFloat( float f )
  : f(f)   {}

  float
};
```

```
MyFloat f1(1.1);
float f2 = 2.2;
MyFloat f3 = addcast(f1,f2);
cout << f3.f << endl;
```

```
template1.cpp: In function “T1 addcast(T1, T2)
      [with T1 = MyFloat, T2 = float]”:
template1.cpp:32:28:   instantiated from here
template1.cpp:7:31: error: no match for
      “operator+” in “a + MyFloat(b)”
```

Add an operator+

```
template <typename T1, typename T2>
T1 addcast( T1 a, T2 b )
{ return a + static_cast<T1>(b); }
```

```
class MyFloat
{
public:
  MyFloat( float f )
  : f(f)   {}
```

```
MyFloat f1(1.1);
float f2 = 2.2;
MyFloat f3 = addcast(f1,f2);
cout << f3.f << endl;
```

```
MyFloat operator+( const MyFloat& f1, const MyFloat& f2)
{
  MyFloat f( f1.f + f2.f );
  return f;
}
```

Template classes

Template class

- You can make template forms of entire classes as well as individual functions
- Again the **`typename`** placeholder name (e.g. **`T`**) is replaced throughout the class
- You need to use it in both the class declaration and the member function implementations
- To alter class definition:
  - Add **`template <typename T>`** at the start, as for template functions
  - Then replace the templated type throughout the code

Template class : linked list

```
class MyLinkedList
{
   struct Entry
   {
        struct Entry* pNext;
        int iData;
   };

   Entry* _pHead;

public:
   MyLinkedList()
   : _pHead(NULL)
   {}

   void InsertHead( int iData );

   void List();
};
```

```
template < typename T >
class MyLinkedList
{
   struct Entry
   {
        struct Entry* pNext;
        T tData;
   };

   Entry* _pHead;

public:
   MyLinkedList()
   : _pHead(NULL)
   {}

   void InsertHead( T tData );

   void List();
};
```

How to alter member functions

- Add prior to each member function definition:

```
template <typename T>
```

- Add the `<T>` to the end of the class name in the member function implementation/definition:
- Example member function implementation:

```
template <typename T>
void MyLinkedList<T>::Store(T tData)
{ … }
```

- Find **each** occurrence of the **templated type** and replace it by the templated type name
  - e.g. replace `int` with `T` in the example
- Note: '`typename`' can be replaced by '`class`'

The member functions

```
void MyLinkedList::
InsertHead(int iData)
{
   Entry* pNewEntry
        = new Entry();
   pNewEntry->iData = iData;
   pNewEntry->pNext = _pHead;
   _pHead = pNewEntry;
}

void MyLinkedList::List()
{
   Entry* pEntry = _pHead;
   while( pEntry != NULL )
   {
        cout << pEntry->iData
             << endl;
        pEntry = pEntry->pNext;
   }
}
```

```
template <typename T>
void MyLinkedList<T>::
   InsertHead(T tData)
{
   Entry* pNewEntry = new Entry();
   pNewEntry->tData = tData;
   pNewEntry->pNext = _pHead;
   _pHead = pNewEntry;
}

template <typename T>
void MyLinkedList<T>::List()
{
   Entry* pEntry = _pHead;
   while( pEntry != NULL )
   {
        cout << pEntry->tData
             << endl;
        pEntry = pEntry->pNext;
   }
}
```

Using the template class

```
int test1()
{
   MyLinkedList<float> oList;
   oList.InsertHead( 1.1 );
   oList.InsertHead( 2.2 );
   oList.InsertHead( 3.3 );
   oList.InsertHead( 4.4 );
   oList.InsertHead( 5.5 );
   oList.InsertHead( 6.6 );
   oList.List();
}
```

```
int test2()
{
   MyLinkedList<string> oList;
   oList.InsertHead( "Adam" );
   oList.InsertHead( "Brian" );
   oList.InsertHead( "Carl" );
   oList.InsertHead( "Dave" );
   oList.InsertHead( "Eric" );
   oList.InsertHead( "Fred" );
   // Following line would not
   compile:
   //oList.InsertHead( 1.2 );
   oList.List();
}
```

The class name is qualified with a type in angled brackets.
Once specified, the type is fixed.
Instantiations of the class are generated by the compiler as needed.

Exam: do I need to know all of this?

- Template functions
  - Be able to recognise them
  - Know what they do
  - Be able to convert from a normal function to a template version
  - Know the difference between a template function and a macro (`#define`)
    - And the dangers of using #define
- Template classes
  - Recognise them
  - Be able to understand code which uses them
  - Understand code using STL classes …

STL container classes

`vector`
`string`
`map`
`list`
`set`
`stack`
`queue`
`deque`
`multimap`
`multiset`

- In `std` namespace
- Know that Standard Template Library exists
  - If you go for C++ job interview, learn basics
- These are template classes
  e.g. `vector<int>` for `vector` of `int`s
- Also have iterators
  - Track position/index in a container
  - e.g. to iterate through a container
- And algorithms (over 70 of them)
  - Apply to containers
  - e.g. min(), max(), sort(), search()

Example of using **vector**

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

using namespace std;

int main()
{
   vector<char> v(10);
   // 10 elements

   int size = v.size();

   cout << "Size " << size
        << endl;

   // Set each value
   for( int i=0 ; i < size ; i++ )
       v[i] = i;

   // Iterate through vector
   vector<char>::iterator p
               = v.begin();

   for( ; p != v.end() ; p++ )
       *p += 97;

   // Output the contents
   for( int i=0 ; i < size ; i++ )
       cout << v[i] << endl;

   return 0;
}
```

Arrays of pointers

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

class Base
{
public:
  Base() : i(1) {}
  virtual void out()
    { cout <<"Base" <<i <<endl; }
  int i;
};

class Sub : public Base
{
public:
  Sub() { i = 2; }
  void out()
    {cout <<"Sub" <<i <<endl; }
};
```

Lec18a.cpp

```
int main()
{
  Sub* arrayp1[3] = { new Sub,
        new Sub, new Sub };
  Base* arrayp2[3] = { new Base,
        new Base, new Base };
  // Output Array 1
  for ( int i = 0 ; i < 3 ; i++ )
    arrayp1[i]->out();
  // Output Array 2
  for ( int i = 0 ; i < 3 ; i++ )
    arrayp2[i]->out();
  // Copy Array 1 elements to 2
  for ( int i = 0 ; i < 3 ; i++ )
    arrayp2[i] = arrayp1[i];
  // Output Array 2
  for ( int i = 0 ; i < 3 ; i++ )
    arrayp2[i]->out();
}
```

Objects are not pointers

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

class Base
{
public:
  Base() : i(1) {}
  virtual void out()
    { cout <<"Base" <<i <<endl; }
  int i;
};

class Sub : public Base
{
public:
  Sub() { i = 2; }
  void out()
    {cout <<"Sub" <<i <<endl; }
};
```

Lec18b.cpp

```
int main()
{
Sub array1[3];
Base array2[3];
// Output Array 1
for ( int i = 0 ; i < 3 ; i++ )
  array1[i].out();
// Output Array 2
for ( int i = 0 ; i < 3 ; i++ )
  array2[i].out();
// Copy array 1 to 2
for ( int i = 0 ; i < 3 ; i++ )
  array2[i] = array1[i];
// Output Array 2
for ( int i = 0 ; i < 3 ; i++ )
  array2[i].out();
}
```

Next lecture and beyond…

- No more labs, so use the 2pm lectures for questions
- Friday 26th April, 10am, Lecture 19
  - When is a duck an instrument? (Multiple Inheritance)
- Friday 26th April, 2pm – Optional
  - How to create programs fast
- Thursday 2nd May, 4pm, Lecture 20
  - Wrapping up (slicing problem, smart pointers)
- Friday 3rd May, 10am, Revision Lecture
  - Revision and exam strategy
- Friday 3rd May, 2pm - Optional
  - Any questions / examples
  - What do you want to 'revise'?