G52CPP
C++ Programming
Lecture 21

Dr Jason Atkin

Last lecture

- Operator Overloading

This lecture

- Macros
- Template functions
- Template Classes

#define and macros

#define

- A semi-**intelligent** '*find and replace*' facility
- Often considered ***bad*** in C++ code (useful in C)
  - `const` is used more often, especially for members
  - Template functions are better than macros
- Example: define a 'constant':
  - `#define MAX_ENTRIES 100`
  - Replace occurrences of "`MAX_ENTRIES`" by the text "`100`" (without quotes), e.g. in:

```
if ( entry_num < MAX_ENTRIES ) { … }
```

- **Remember:** Done by the pre-processor!
  - E.g. **NOT** actually a **definition** of a **constant**
- 'Constant' `#define`s usually written in CAPITALS

#define and macro definitions

- You can use **`#define`** to define a **macro**:

```
#define max(a,b) (((a)>(b)) ? (a) : (b))

int v1 = max( 40, 234 );
int v1 = (((40)>(234)) ? (40) : (234))

int v2 = max( v1, 99 );
int v2 = (((v1)>(99)) ? (v1) : (99))

int v3 = max ( v1, v2 );
int v3 = (((v1)>(v2)) ? (v1) : (v2))
```

- **Remember: done by the pre-processor!**
  - **NOT a function call**

What is the output here?

MyHeader.h

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

#define max(a,b) (((a)>(b)) ? (a) : (b))

#endif
```

MyTest.cpp

```
#include <cstdio>
#include “MyHeader.h”
int main( int argc, char* argv[] )
{
   int a = 1, b = 1;
   while ( a < 10 )
   {
       printf( “a = %d, b = %d “, a, b );
       printf( “max = %d\n”, max(a++,b++) );
   }
}
```

The (surprise?) output

```
printf( “a = %d, b = %d “, a, b );
printf( “max = %d\n”, max(a++,b++) );
```

- **The output is:**

```
a = 1, b = 1 max = 2
a = 2, b = 3 max = 4
a = 3, b = 5 max = 6
a = 4, b = 7 max = 8
a = 5, b = 9 max = 10
a = 6, b = 11 max = 12
a = 7, b = 13 max = 14
a = 8, b = 15 max = 16
a = 9, b = 17 max = 18
```

- **Why?**

The (surprise?) output

```
printf( “a = %d, b = %d “, a, b );
printf( “max = %d\n”, max(a++,b++) );
```

- **The output is:**

```
a = 1, b = 1 max = 2
a = 2, b = 3 max = 4
a = 3, b = 5 max = 6
a = 4, b = 7 max = 8
a = 5, b = 9 max = 10
a = 6, b = 11 max = 12
a = 7, b = 13 max = 14
a = 8, b = 15 max = 16
a = 9, b = 17 max = 18
```

- **Why?**

  `max( a++, b++ )` expands to:

  `((a++)>(b++)) ? (a++) : (b++)`

- So, whichever number is greater will get incremented twice, and the lesser number only once

Warning about macros

- Do not use a macro where the evaluation of the parameters may have a side-effect

- E.g.

```
max(a++,b++)
```

- Evaluating these parameters alters a value
  - A side-effect
- **`inline template`** functions are better

Template Functions

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 …

Reminder STL container classes

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

- **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
  - Iterators are also templates/parameterised
    - Know what they iterate across

Next Lecture

- The Slicing Problem