Templates

What are templates?

- Templates provide support for *generic* programming.
- What this means is that the same code can be used for different data types.
- So, its a style of programming that is type independent, in the sense that the type of an object is taken in as a parameter to a template class, or to a function template.
- C++ has *class* templates and *function* templates, i.e., we can write classes that take types as parameters, and we can write functions that take types as parameters.
- For example, STL is designed for use in generic programming.
  - vector <int> or vector<double> or vector< MyClass> or vector< MyClass*>

Class Templates

- In our examples and assignments so far, we have used classes specific to a particular data type.
  - Example:
    - Stack. We can have a Stack of integers.
      - What if we wanted a stack of doubles or floats, or characters.
      - We would have to re write the Stack class and make it store the data type we wanted. So we may have classes like IntStack, FloatStack, CharStack and so on.
      - However, if we could pass in the data type as a parameter to the Stack class definition, then we would be able to write the code for Stack class once, and give it different data types for creating a Stack of those types.
      - Templates let use do exactly that.

Example

```
template <typename T>          ← NOTE: Can also say class instead of typename
class Stack
{
 public:
             Stack ( int size );

     void    Push ( const T & e );     ← Accepts type T. We do not specify a
                                           specific type like int or double.

     T       Pop ();                   ← Returns type T

     int     GetNumElements() const;

     bool    IsEmpty() const;

private:
     T *     m_Array;          ← We used to have an array of int, now array of T
      int    m_Size;
     int     m_NumElements;
};
```

Function templates

- Templates can also be used to pass types as parameters to functions.
- Function templates ensure that the function call semantics are preserved.
- As an example, macros are used in C to define some operations that are common across different types.
- An example of this is the min or max macro which looks like

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

- Notice there is no type checking here, and one of the arguments will be evaluated twice, which means something like the following is a problem:
  - max(value1 ++, value2 ++). Oops! One of these will get incremented twice.

Contd.

- Lets look at a very simple example:

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

This is limited to taking an integer, and now if we want one for double, we need to write something like:

```
double MaxDouble (double a, double b)
{
  return  a > b  ?  a : b;
}
```

And one for some class, say, MyString

```
MyString MaxMyString ( const MyString & a, const MyString & b)
{
  return  a > b  ?  a : b;
}
```

QUESTION: What is the requirement in class `MyString` if function `MaxMyString` has to compile?

Contd.

- Instead of having to define so many different Max functions, we can pass the type as a parameter, so that would look something like:

```
Type Max (Type a, Type b)
{
  return  a > b  ?  a : b;
}
```

This is syntactically incorrect, but lets not worry about that yet.

- So what we have done is, pass in the type itself as a parameter. This is telling the compiler:
  - We want a Max function that:
    - Compares two objects that are taken in as arguments,
    - Returns larger of the two objects by value.

    Note that if they are equal, the second argument is returned.
- And the requirement on Type is that the > operator is overloaded.
- Of course, if Type is a built in type (int, char, double, pointer), then the operator > is already defined by the language, and if it is a class, operator > has to be overloaded in that class, or …?

Function max() and min()

```
template <typename T>    ← NOTE: Can also say class instead of typename
inline const T& max (const T & a, const T & b )
{
   return  a  >  b  ?  a : b;
}



template <typename T> ← NOTE: Can also say class instead of typename
inline const T& min (const T& a, const T& b )
{
   return  a  <  b  ?  a : b;
}
```

Function Search()

```
template < typename T >
int Search ( const T * array,  int length,  const T & item )
{
   for ( int ii = 0; ii < length;  ++ ii )
        if (array[ii] == item)   ← NOTE that == should be defined on the type T passed in
           return ii;
   return -1;
}
```

Now, we can do something like:

```
int   intArray[10];
char charArray[10];
MyClass  myArray[10];
MyClass  mcObj;

Search( intArray, 10, 4);              ← Search intArray for 4
Search( charArray, 10, 'A');           ← Search charArray for 'A'
Search( myArray, 10,  mcObj );  ← Search for MyClass object which is "equal to" mcObj
```

NOTE: == operator should be defined in class MyClass

Compile time, not run time

- Templates are a compile time hit, not a run time hit.
- This means that when you compile and link your program, the templates are instantiated as needed, and at runtime, the needed code is just there as any other non-template code.
- FYI:
  - C++ standard requires that template member functions be instantiated only when needed, i.e., if a class has 4 template member functions, and only 3 are used by the client code, then only those 3 will be instantiated. Older C++ systems may not follow this, though, i.e., they may instantiate all 4 methods in this example.

Compile time

- Templates are compiled once when the template code is checked for any errors.
- At instantiation time, the template is compiled again to check for any errors with respect to the type(s) being instantiated for.
- This means that at instantiation time, the compiler will need to see the template definition (template code) again.
- This is the reason templates are usually defined in header files.
- More on this later.

Argument deduction

- Argument deduction is used to figure out which template to use, based on the type of the arguments being passed.

- Example:

```
template <typename T>
inline const T& max (const T & a, const T & b );

max( 101, 77 );   // This is fine, as both arguments are int
max( 101.1, 77 ); // This will give an error since first argument is a
    double, second is an int. Keep in mind that 77 will not get auto converted to
    a double here… the argument match has to be exact.
```

To fix this:

```
max<double>( 101.1, 77 ); OR
max( 101.1, static_cast<double>(77) );

OR, see next slide
```

Argument deduction

- To solve the issue with passing different types as shown in the previous example, you can write the template to accept different types:

```
template <typename T1, typename T2>
inline T1 max (const T1 & a, const T2 & b );
```

Now I can make the following calls:

```
max( 101.1, 77 );
max( 77, 101.1 );
```

Argument deduction

Cons of this approach:

- You have to declare the return type as one of T1 or T2.
- This means that the return value can be different depending on the order of arguments passed in.
  - For example, in the `max` calls example:
    - `max( 101.1, 77 ); // This will return 101.1`
    - `max( 77, 101.1 ); // This will return 101`
- This means that a temporary object may get created (int 101 in the example above).
- Which then means that the return type of `max` cannot be by reference, but has to be by value:

```
template <typename T1, typename T2>
inline T1 max (const T1 & a, const T2 & b );
```

Argument deduction

- So, argument deduction is the ability to deduce the template parameters (like T or T1 or T2) based on the types of the arguments being passed in.
- If argument deduction is not possible, then you have to specify the type(s).
- Example:

```
template <typename RET, typename T1, typename T2>
inline RET max (const T1 & a, const T2 & b );
```

RET is the return type, and is by value.

Argument deduction

Given this template and a call to max such as

`max(77, 101 )` or `max(77, 101.1 )` or `max(77.7, 101.1 )`

- the compiler cannot deduce the type for **`RET`**.
- It can deduce the type for `T1` and `T2`
- This means we have to specify the type for template parameter **`RET`**.

- So the call will look like:
  - `max<double>( 77, 101.1 ) // return type is double`
    - OR
  - `max<int>( 77, 101.1 )  // return type is int`
    - Note that the compiler will give a warning here since the second argument is a double which will need to be converted to an int. So:

      - `max<int>( 77, static_cast<int>( 101.1 ) )`

Argument deduction

- You can also specify the rest of the parameters, but that is not needed:
  - `max< double, int, double >( 77, 101.1 );`
    - OR
  - `max< double, int >( 77, 101.1 );`
  - But this is unnecessary. You only need to specify the first template parameter for the return value.
  - **Note**:
    - These examples are for illustrating template usage, and if you have to write a min/max template, you should try to stay with the one parameter template

Templates and inline

```
template < typename T >
class MyTemplateClass
{ public:
        MyTemplateClass( const T & t ) : mValue(t) {}

        void Add( int count )     // inlined because defined inside class
        { mValue += count; cout << mValue; }

        void Subtract(int count );
        void Mult( int count );
private:
        T       mValue;
};

template < typename T >           // inlined because of keyword inline
inline void Subtract(int count )
{ mValue -= count; cout << mValue; }

template < typename T >           // not inlined
void Mult( int count )
{ mValue *= count; cout << mValue; }
```

Templates and inline

- So, templates are not inline by default.
- They will be inlined by the compiler if
  - Implementation defined inside the class(member `Add` in our example)
  - OR
  - Keyword **`inline`** is used (member `Subtract` in our example).
- Keep in mind that the keyword inline is only a hint/suggestion to the compiler.
- The compiler is free to ignore it if the function does not meet "inlining" requirements for that compiler.

Zero initialization

- You can initialize template variables as follows:

```
template < typename T >
void MethodA() {
        T   t = T();
}
```

If `T` is, say, int or bool, `t` gets initialized to 0 or false, respectively.

In a template class:

```
template < typename T >
class ABC {
public:
        ABC() : mValue() {}   // mValue initialized in initializer list
// if T is a class, default constructor is called.
// If T is a built in type, it gets the zero value for that type.
// Note: For class type, default constructor would get called even
// if we didn't call it, but this is needed if T is a built in type.
private:
        T       mValue;
};
```

Syntax and semantic requirements

- As we have seen, template code has to specify *syntax* requirements.
- This is needed such that the template class or template function code will compile when instantiated for a given type(s).
- However, there is also the *semantic* requirement, which the compiler cannot verify. This is up to the developers to verify.
  - An obvious example could be:
    - The template requires operator < to be defined for the types instantiated with.
    - Lets say the type does overload operator <.
    - This satisfies the syntactic requirement and the code will compile.
    - However, the implementation of operator < actually has to do the right thing, and this is what the semantic requirement is.

Disadvantages of templates

- Use of templates does increase the amount of code → *Code Bloat*

- For example:

```
vector < int > vi;
vector < double > vdbl;
vector < string > vs;
list < MyClass> lm;
list < MyString> lms;
```

  - All of the above are very convenient to use, but it does lead to increased amount of code generated at compile time.
  - This can be an issue when memory is at a premium, for e.g., in embedded systems programming.
  - Note that if you absolutely need all containers shown above, you may need to pay the price for memory anyway, but the code bloat should be kept in mind.

  - A technique used to mitigate code bloat is **template hoisting**

Template Hoisting

- Template hoisting is a technique where the type independent code is moved up into a base class.
- This base class is NOT a template class (since it contains type independent code). And therefore it is NOT instantiated for different types.
- A derived class is created. This is a template class, and all type dependent code is placed in here.
- When the derived class is used, only the code in this class is instantiated for different types (and not the base class).
- See example on next slide.

Example of template hoisting

```
class StackBase
{
public:
                StackBase( int size );
    int         GetNumElements() const;
    bool        IsEmpty() const;
private:
     int        m_Size;
    int         m_NumElements;
};
template <typename T>
class Stack : public StackBase
{
 public:
                Stack ( int size );
    void        Push ( const T & e );
    T           Pop ();
 private:
    T *         m_Array;
};
```

You won't see T being used in here, as this class is not dependent on type T, which is not even mentioned here as it is NOT a template class.

This is a template class, and derives from the StackBase class. This class is instantiated once for each type that it is used for, but its base class is NOT instantiated multiple times.

E.g.: Stack<int>, Stack <double>, Stack< XYZ>

Deriving from a template class

- You can also derive a class from a template class:

```
template <typename T>
class Derived : public Base <T>
{
    ... Class stuff here, independent of T
};
```

So, the base class is the one that uses the template information. Derived class does not.

Of course, you can have the Derived class use T also:

```
template <typename T>
class Derived : public Base <T>
{
    ... Class stuff here, can be dependent on T
private:
    T      m_Value;
};
```

In the above two cases, objects of Derived type are defined as:

```
    Derived < int >  d1;        // i.e., you have to use the template syntax and
                                // specify the type.
```

Contd.

Or you can pass in the type explicitly:

```
class Derived : public Base <int *>
{
    ...
};
```

- Or

```
class Derived : public Base <XYZ>       // where XYZ is some class or type
{
    ...
};
```

- In the above two cases, objects of Derived class are declared without the template syntax:

```
Derived  d1;
```

Member templates

- Member methods can also be templates. This is so that these methods can be instantiated for different types (as needed).

- Example:

```
Stack <int> si;
Stack <double> sd;
Stack <Complex> scmp;

scmp = si;          // type error
scmp = sd;          // type error
```

The two assignments above will generate a compilation error. This is because the type Stack <int> is **different** from Stack <Complex>, even though there is a conversion from int to Complex (via the conversion constructor we had in our Complex class example).

Member templates

So, whereas the following statements are OK:

```
int i = 10;
double d = 15;
Complex c1 (i), c2(d);          // uses the conversion constructor
```

- The following are NOT OK:

```
scmp = si;          // type error
scmp = sd;          // type error
```

- If this kind of assignment is needed, then we need to use member templates:

```
template < typename T2 >
Stack<T> & operator = (const Stack<T2> & other );
```

Look at next slide for class definition using the member template

Member templates

```
template <typename T>        ← NOTE: Can also say class instead of typename
    class Stack
    {
     public:
                    Stack ( int size );
         void       Push ( const T & e );
         T          Pop ();
         int        GetNumElements() const;
         bool       IsEmpty() const;

         template < typename T2 >
         Stack<T> & operator = (const Stack<T2> & other );
      private:
         T *        m_Array;
         int        m_Size;
         int        m_NumElements;
    };
```

```
template <typename T>
template <typename T2>
Stack<T> & Stack<T>::operator = (const Stack<T2> & other )
{
... Code to assign from other to this.
}
```

If this gives an error (it shouldn't) , define it inline within the class definition

Member templates

- Note that if it does not make sense to assign one type A to another type B, (i.e., there is no conversion constructor to convert from type A to type B), then the member template will also fail to instantiate, and you get a compilation error.

- Example:

```
Stack <int *> siptr;
Stack <Complex> scmp;
Stack <XYZ> sxyz;    // XYZ is some class that has nothing to do with the Complex class

// The following are errors in spite of Stack class having a template member
// assignment operator.
scmp = siptr;        // error, because we can't assign an int pointer to a Complex class object
siptr = scmp;        // error, because we can't assign an Complex class object to an int *
scmp = sxyz;         // error, because we can't assign an XYZ object to a Complex class object
sxyz = scmp;         // error, because we can't assign a Complex class object to an XYZ class
                     // object.
```

So, adding template members assignment operator does not give you the capability to do nonsense assignments.