Templates

A Form of Polymorphism

Polymorphism: The ability of a variable, function, or class to take more than one form.

- **ad-hoc polymorphism**: function overloading
- **true polymorphism**: virtual member functions, dynamic binding
- **parametric polymorphism**:
  - controlled through a parameter
  - works at compile time

- C++ has two types of parametric polymorphism:
  - Function templates
  - Class templates

Function Templates

Suppose you want to write a generic function with two arguments of the same type that returns the largest of the two.

You could overload the function for each data type:

```
int maxInt (int a, int b);
int maxFloat (float a, float b);
…
Foo maxFoo(const Foo& a, const Foo& b);
```

Function Templates (con't)

The code in each would look the same:

```
if (a < b)
        return b;
else
        return a;
```

The algorithm does not rely on the data type. (Note: the < operator may have to be overloaded, e.g., if Foo is a class.)

Function Templates (con't)

Using templates, we can write one function to handle (almost) all data types and classes.

```
template <typename T>
T max (const T& a, const T& b)
{
    if ( a < b)
        return b;
    else
        return a;
}
```

T is called the **type parameter** and can have any legal identifier name.

max( ) for int

When compiler sees this code:

```
int i1 = 1, i2 = 2, i3;
i3 = max(i1, i2);
```

it creates this function:

```
int max (const int& a, const int& b) {
    if (a < b)
        return b;
    else
        return a;
}
```

max( ) for floats

When compiler sees this code:

```
float f1 = 1.2 f2 = 2.2, f3;
f3 = max(f1, f2);
```

it creates this function:

```
float max (const float& a, const float& b) {
    if (a < b)
        return b;
    else
        return a;
}
```

max( ) for Foo

When compiler sees this code:

```
Foo F1 = value1, F2 = value2, F3;
F3 = max(F1, F2);
```

it creates this function:

```
Foo max (const Foo& a, const Foo& b) {
    if (a < b)
        return b;
    else
        return a;
}
```

Calling a Template Function

- Note that the function call to max( ) was not changed.

```
i3 = max (i1, i2);
f3 = max (f1, f3);
etc.
```

- The caller doesn’t “know” it’s calling a template function.

Can max( ) Be Created for Any Data Type?

Answer -- Almost

Note that max ( ) includes the statement

if (a < b) . . .

- If the data type is a class, that class must support the < operator (i.e., it must be overloaded).

Can max( ) Be Created for Any Data Type? (con't)

When the compiler sees this code:

```
char *str1 = "abc";
char *str2 = "def";
cout << max(str1, str2) << endl;
```

it creates this function:

```
char* max (const char*& a, const char*& b) {
    if (a < b)        // What gets compared?
        return b;
    else
        return a;
}
```

A Workaround

We can work around this problem by overloading max() and providing an explicit function for char *.

```
char* max(char *a, char *b)
{
    if (strcmp(a,b) < 0)
        return b;
    else return a;
}
```

The compiler will look for an exact function signature match before using the template.

Class Templates

- Templates can be provided for classes as well as functions.
- This is how we create generic classes that work with any data type (predefined or programmer-defined).
- This concept is also the basis (in C++) for **container classes**, classes that hold collections of objects.

MyArray of ints

```
class MyArray {
   public:
        MyArray (int s = 100);
        // other members
   private:
        int size;
        int *theData;
   };
```

Nothing Special About a MyArray of ints

- Nothing in the code for the MyArray relies on the fact that this is an array of integers.
- We can create a class template for MyArray so that we use it for any primitive data type or class (assuming that the class has the necessary overloaded operators).

MyArray Template

```
template <typename T>
class MyArray {
   public:
        MyArray ( int s = 100 );
        // other members

   private:
        int size;
        T *theData;// array of any type
};
```

Default Constructor for MyArray Class Template

```
// Each member function is a function template
template <typename T>
MyArray<T>::MyArray (int s)
{
   size = s;
   theData = new T [size];          // an array of T's
   for (int j = 0; j < size; j++)   // just to be nice
        theData [ j ] = T( );       // default object
}
```

Copy Constructor for MyArray Class Template

```
template <typename T>
MyArray<T>::MyArray (const MyArray<T>& a)
{
   size = a.size;
   theData = new T [ size ];
   for (int j = 0; j < size; j++ )
        theData[ j ] = a.theData [ j ];
}
```

Another Example - operator [ ]

```
template <typename T>
T MyArray<T>::operator[ ] (int index)
{
   return theData [ index ] ;
}
```

Using MyArray

- In the main program,

  #include “MyArray.H”

- Define some MyArrays:

  MyArray**<float>** array1;
  MyArray**<myClass>** array2;

- When the compiler sees these definitions, it looks for the MyArray template to determine how to generate the needed class code.

Template .H and .C Files

- As usual, we put the class template in the .H file and the implementation in the .C file.
- Same for function templates

How Does the Compiler “find” the Implementation (.C) File?

- This varies from compiler to compiler.
- Some compilers assume that the code for the template found in *file.H* will be in *file.C* -- they assume the same “root” filename.
- The g++ compiler isn’t that smart.

Templates and g++

- For the g++ compiler to find the implementation of a template, we must #include the .C file inside the .H file.
- Also note that this means you DO NOT #include the .H file inside the .C file as you normally would.
- **This applies ONLY to templates**

MyArray.H

```
#ifndef MyArrayH
#define MyArrayH
template <typename T>
class MyArray {
    public:
        MyArray ( int s = 100 );
        // other members

    private:
        int size;
        T *theData;
};
#include “MyArray.C”
#endif
```

Compiling Templates

- Function and class templates are just that – templates (i.e., frameworks) that tell the compiler how to generate code. The compiler can't generate any meaningful code until it knows what type to use in place of the template type parameter.
- So compiling max.C (or MyArray.C) will never create a meaningful max.o (or MyArray.o), so we don't include rules for making max.o (or MyArray.o) in our makefile. The code will be generated when the compiler encounters the code that uses the class template.

More About Compiling

- We may want to compile max.C (MyArray.C) to check for syntax errors (using the -c switch), but this must be done manually and carefully.
- We just said that with g++, we #include max.C inside max.H. Then, what happens if we compile max.C? Since we don't include max.H in max.C, we can expect to get syntax errors.
- If we really want to do this, we must **temporarily** do the normal thing – #include the .H in the .C, but don't include the .C in the .H.

Writing Templates

- Often the best way is to write your function, or design your class to work just with ints.

- Then, replace all the appropriate ints (not ALL the ints) with the template type parameter.