LECTURE 5

Templates

We have written a simple array class of float variables. But suppose we want to have arrays of integers, or doubles, or something else. It's a pain to write a separate array class for each new case. Templates make it easy to have arrays of anything; not just ints, floats, and doubles, but *anything*. In general, templates make it easy to make a family of classes of closely related objects. To make a template, write the class for something definite, like floats. Then turn it into a class template. As an example, let's do this with our array class.

```
#include <iostream.h>
#include "arraytemplate.h"

void main()
  {
    // Create arrays with the desired number of elements
    int n;
    cin >> n;
    Array<float> x(n);
    Array<int> y(n);

    // Read the data points
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }

   ...
   }
```

File "arraytemplate.h"

A class template declaration consists of the keyword `template`, followed by a list of *template arguments* enclosed in angular brackets (`< >`), followed by a class declaration. These template arguments may be type–arguments preceded by the keyword `class` and/or numeric argument declarations. For example:

```
template<class T> class Array;             // Class template declaration

or
template<class T, int nL> class vector;    // For a vector of length nL.
```

Let's stick with the first case.

```
template<class T> class Array;            // Class template declaration

template<class T>                         // T="type",e.g.,int or float
class Array {
public:

    Array(int n);                         // Create array of n elements
    Array();                              // Create array of 0 elements
    Array(const Array<T>&);               // Copy array
    ~Array();                             // Destroy array
    T& operator[](int i);                 // Subscripting
    int numElts();                        // Number of elements
    Array<T>& operator=(const Array<T>&);  // Array assignment
    Array<T>& operator=(T);               // Scalar assignment
    void setSize(int n);                  // Change size
private:
    int num_elts;                         // Number of elements
    T* ptr_to_data;                 // Pointer to built-in array of elements

    void copy(const Array<T>& a);         // Copy in elements of a
};
```

Notice that the declarations of the constructor and destructor member functions do not include the parameter `T`.

Definitions of Template Member Functions

Notice that member function definitions for class templates are preceded by the `template` keyword with `<class T>`, but are otherwise analogous to ordinary member function definitions.

```
template<class T>
Array<T>::Array(int n) {
    num_elts = n;
    ptr_to_data = new T[n];
}

template<class T>
Array<T>::Array() {
    num_elts = 0;
    ptr_to_data = 0;
}

template<class T>
Array<T>::Array(const Array<T>& a) {
```

```
    num_elts = a.num_elts;
    ptr_to_data = new T[num_elts];
    copy(a); // Copy a's elements
}

template<class T>
void Array<T>::copy(const Array<T>& a) {
    // Copy a's elements into the elements of *this
    T* p = ptr_to_data + num_elts;
    T* q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = *--q;
}

template<class T>
Array<T>::~Array() {
    delete [] ptr_to_data;
}

template<class T>
T& Array<T>::operator[](int i) {
    #ifdef CHECKBOUNDS
       if(i < 0 || i > num_elts)
          error("out of bounds");
    #endif
    return ptr_to_data[i];
}

template<class T>
int Array<T>::numElts() {
    return num_elts;
}

template<class T>
Array<T>& Array<T>::operator=(const Array<T>& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}

template<class T>
```

```
void Array<T>::setSize(int n) {
    if (n != num_elts) {

        delete [] ptr_to_data;     // Delete old elements,
        num_elts = n;              // set new count,
        ptr_to_data = new T[n];    // and allocate new elements

    }
}

template<class T>
Array<T>& Array<T>::operator=(T rhs) {
    T* p = ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = rhs;
    return *this;
}
```

We can see from the `main()` program how templates are instanced. `Array<float> x(n)` is an instance of the template. Here `T` has become `float`.

Comments on Templates

1. No code is generated for a template. Code is generated after the template is converted into a concrete class or function. Thus the `.cc` source file is almost never associated with a template class. The entire template class definition, including all the member functions, should be contained in the `.h` include file so that it can be available for the compiler to expand.

   If you do decide to put the template declarations in a `.h` file and the template definitions in a `.cc` file, then put `#include "Array.cc"` at the very bottom of the `Array.h` file and in the `Array.cc` file, remove `#include "Array.h"`. Don't try to compile `Array.cc`. Just compile the rest of the source files, like the file with the `main` program. The compiler will do the rest. (This works with g++, but I don't guarantee it for other compilers.)

2. A template class does not consume any memory. But every instance of a template class does take up memory.

3. A template class cannot be compiled and checked for errors until it has been converted into a real class. Thus, a template class `Array` might compile fine even though it contains obvious syntax errors. The errors won't appear until a class such as `Array<float> x(n)` is created. Even if an error does appear when instancing a template class, it does not necessarily mean the template class has a problem. The problem may be in the instance itself.

Const in Classes

Class objects may be declared `const` in the same way as intrinsic types. Like an intrinsic object, a user–defined object must be assigned a value when it is created. For example, suppose we have defined a `class Student` where the constructor just requires the student's name as an argument (`Student::Student(char* name)`). Then we could say create a constant object in `main`:

```
const Student Michael("Michael");
```

A `const` object can't be changed after initialization. The compiler will declare an error if you try to pass a `const` object to a function that might try to change the object. One important use of `const` is to prevent bugs by safeguarding objects that you know shouldn't be changed. Consider the following example which uses our Array template. Recall that `strings` are arrays of `chars`.

```
    char day0[]="Sunday";
    Array<char>d(7);
    int i;
    for(i=0;i<7;i++)
      d[i] = day0[i];
    const Array<char>Day_zero=d;    //const Array can't be changed after
                                    //declaration. Notice that this
                                    //calls the copy constructor.

    Array<char>aday = Day_zero;     //It's ok to set a nonconst object equal
                                    //to a const one. Calls copy constructor.

    aday[0] = Day_zero[0];          //Get compiler warning.

    int n = Day_zero.numElts();     //Get compiler warning.
```

The problem is that `Day_zero` is declared `const`, but the `operator[]()` and the function `numElts()` are not declared `const`. In particular, these functions take nonconstant arguments, so the compiler is worried that the functions might change the `const` Array. So we need to change the definitions.

```
template<class T>                            // T="type",e.g.,int or float
class Array {
public:

    Array(int n);                            // Create array of n elements
    Array();                                 // Create array of 0 elements
    Array(const Array<T>&);                  // Copy array
    ~Array();                                // Destroy array
```

```
    T& operator[](int i);                   // Subscripting
//******************* Note the word "const"************************
    int numElts() const;                    // Number of elements
    const T& operator[](int i) const;       // Subscripting
//*****************************************************************
    Array<T>& operator=(const Array<T>&);    // Array assignment
    Array<T>& operator=(T);                 // Scalar assignment
    void setSize(int n);                    // Change size
private:
    int num_elts;                           // Number of elements
    T* ptr_to_data;                   // Pointer to built-in array of elements

    void copy(const Array<T>& a);           // Copy in elements of a
};
// Function definitions
template<class T>
int Array<T>::numElts() const{
    return num_elts;
}

// New subscript operator, returns const T
template <class T>
const T & Array <T>::operator[](int i) const
   { return ptr_to_data[i]; }


// Old supscript operator, not const
template <class T>
T & Array <T>::operator[] (int i)
   { return ptr_to_data[i]; }
```

When we put `const` after a function declaration, as in

```
int numElts() const
```

we don't mean that this function can't change. We mean that this is a function that can't change the class object that it belongs to. This is in contrast to a non`const` function which can change the object it belongs to. You can pass a `const` function nonconstant arguments, especially since these arguments often do not refer to `*this` object. If these arguments are not to be changed, then declare them `const`:

```
void fn(const Array& A2);
```

When `const` appears in front of a function declaration, we mean that it returns a `const` value or reference. For example, the first `const` in

```
const T& operator[](int i) const;
```

means that the subscripted variable returns a reference to a `const` object. The compiler is smart enough to choose between the `const` and nonconstant operator function:

```
const T& operator[](int i) const;
 T& operator[](int i);
```

It chooses according to whether the current object is constant or not. This is a case of operator overloading. You can also overload functions with respect to the `const`ness of their explicit arguments. Thus, the following two functions are not ambiguous:

```
  void fn(Array& A);       //used for non-const objects
  void fn(const Array& A); //used for const objects
```

Another solution to our problem is to write

```
T operator[](int i) const;              //does not return a reference
T & operator[](int i);                  //returns a reference
```

Notice that `T operator[](int i) const` does not return a reference. Rather it returns the value of the ith element of the array, so that the ith element can't be changed and is kept `const`. `T & operator[](int i)` returns a reference that can later be changed.

Why bother with `const`? Because it is a good way to safeguard your program and keep out bugs. Recall that the external function

```
  void fn(Array A);
```

is safe in that it can't overwrite the Array object because it's passed a copy of `A`. But suppose `Array A` is maintained in a large relational database so that making a copy requires a lot of time and effort. In that case it's easier to pass a reference to `Array A`, thus enhancing the efficiency of your program. So you write

```
  void fn(Array& A);
```

But now there's the danger that `fn()` will overwrite `Array A`. Even if you think that `fn()` doesn't change `Array A`, when debug time comes, you can't exclude the possibility. So you write

```
  void fn(const Array& A);
```

So the moral of the story is that you should put in `const` wherever you can. If you know that the member function doesn't change `*this`, i.e., the object of the class to which it belongs, add `const` to the end of the declaration.

(By the way, you only put `const` at the end of a declaration of a function that's a member of a class. It doesn't make sense to put it at the end of the declaration of an external function.)

Template Functions

We have seen how classes can be made into templates. You can also have templates for functions that are not members of classes. For example:

```
#include <iostream.h>

template <class T>
T MAX(T num1, T num2)                  //template function
 {
   return (num1 > num2 ? num1 : num2);
 }

template <class T>
T SQR(T num)                           //template function
 {
   return (num * num);
 }

int main()
  {
    float x, y, z1, z2, z3;
    long m;
    cin >> x >> y;
    z1 = MAX(x,y);
    z2 = SQR(x+y);
    cout << z1 <<" "<< z2 << endl;
    cin >> m;
    z3 = MAX(m,x);                 //won't work, template won't convert
                                   //argument types
    return 0;
  }
```

You can also use template functions with user-defined types (classes), as long as the operators in the template function make sense for the class. For example if you wanted to use `MAX` with `Student`, the "less than" symbol (`<`) would have to be defined in the class `Student`.

In C one can define macros that act like template functions, but macros are more error prone. For example if you wrote:

```
#define sqr(x) (x*x)

 func(float x, float y, float z)
  {
    z = sqr(x + y);              //z=x+y*x+y=x+(y*x)+y due to operator
                                 //precedence
    cout << z << endl;
  }
```

In addition, macros don't provide type checking.