LECTURE 4

A Simple Array Class (of float variables)

Let's write a simple array class of float variables. Later we will get much more sophisticated versions. Our goals are

1. Make use of the array safe from memory leaks.
2. Put in optional bounds checking.
3. Automate copying of arrays.
4. Simplify resizing of arrays.
5. Simplify assigning one value to all elements of the array.

This is the example given in chapter 4 (section 4.2) of Barton and Nackman.

```
array.h
-------
class Array
    {
public:
    Array(int n);                       // Constructor:
                                        //   Create array of n elements
    Array();                            // Default constructor:
                                        //   Create array of 0 elements
    ~Array();                           // Destructor:Destroy array(comment 1)
    Array(const Array &);               // Copy array (comment 2:
                                        //   Copy constructor)

    int numElts();                      // Number of elements
    Array & operator = (const Array &); // Array assignment
    Array & operator = (float);         // Scalar assignment (comment 3)
    void setSize(int n);                // Change size

    float &operator[] (int i);          // Subscripting (comment 4)

private:
    int num_elts;                       // Number of elements
    float *ptr_to_data;                 // Pointer to data
    void copy(const Array & a);         // Copy elements of a
    };

void error(const char *s);              // For bounds checking
```

```
array.cc
--------
#include "array.h"

Array::Array(int n)
    {
    num_elts = n;
    ptr_to_data = new float[n];
    }
Array::Array()
    {
    num_elts = 0;
    ptr_to_data = 0;
    }

Array::~Array()                         // Destructor
    {
    delete[] ptr_to_data;
    }

Array::Array(const Array& a)            // Copy constructor
    {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];
    copy(a);                            // Copy a's elements
    }

void Array::copy(const Array& a)
    {
    // Copy a's elements into the elements of *this
    float *p = ptr_to_data + num_elts;
    float *q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data)
         *--p = *--q;
    }

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

int Array::numElts()
    {
    return num_elts;
    }

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

void Array::setSize(int n)
    {
    if (n != num_elts)
       {
        delete[] ptr_to_data;              // Delete old elements,
        num_elts = n;                      // set new count,
        ptr_to_data = new float[n];        // and allocate new elements
       }
    }

Array& Array::operator=(float rhs)
    {
    float *p = ptr_to_data + num_elts;
    while (p > ptr_to_data)
        *--p = rhs;
    return *this;
    }

void error(const char *s)
    {
    cerr << endl << s << endl; //1st "endl" is in case program is printing
    cout << endl << s << endl; //something out when the error occurs
    cout.flush();              //write the output buffer to the screen
```

```
                                //or wherever the output of cout goes.
    abort();
    }
```

Comment 1: Destructor

We have seen that a class has a special member function called a constructor that creates an object of the class. It also has a special member function that destroys an object of a class. This is called a *destructor*. Why do we need a destructor? A class may allocate resources in the constructor; these resources need to be deallocated before the object ceases to exist. For example, if the constructor opens a file, the file needs to be closed by a destructor. Or if the constructor allocates memory from the heap, this memory must be freed before the object goes away. The destructor allows the class to do these cleanup tasks automatically without relying on the application to call the proper member functions. The destructor is invoked automatically when an object is destroyed, or in C++ parlance, when an object is *destructed*. Another way to say this is "when the object goes out of scope." A local object goes out of scope when the function returns. A global or static object goes out of scope when the program terminates.

The destructor member has the name as the class but with a tilde (`~`) added to the front. `~Array()` is the destructor of the class `Array`. (The tilde is the symbol for NOT in C.) Like a constructor, the destructor has no return type. If you don't explicitly provide a destructor for the class, the compiler will provide one for you. Often this is fine, but if you explicitly allocated resources, (e.g., using `new`), then you need to explicitly provide a destructor to deallocate these resources (e.g., using `delete`). In our `Array` example, the destructor frees up the memory occupied by the array:

```
Array::~Array()                             // Destructor
    {
    delete[] ptr_to_data;
    }
```

Comment 2: Copy Constructor

`Array(const Array &)` is an example of a copy constructor. In general the standard syntax for a copy constructor is `X::X(X&)`, where `X` is any class name. That is, it is the constructor of class `X` which takes as its argument a reference to an object of class `X`. The copy constructor makes a copy of an existing object. Since you usually do not want to change the original object, the copy constructor should be written as `X::X(const X&)`. When would you need a copy of an existing object? A function call is a good example.

```
  void func(Array sfa);
  main()
   {
   Array mymatrix(5);
   func(mymatrix);
   }
```

The statement `Array mymatrix(5)` calls the ordinary constructor `Array(int n)`. C++ calls the copy constructor to make a copy of `mymatrix` to pass to `func()`. The copy is destroyed at the return from `func()`. The original object, `mymatrix`, is destroyed at the end of `main`.

If you don't provide your own copy constructor, C++ provides one for you. The automatic copy constructor performs a member–by–member copy. So why bother ever writing your own copy constructor? When would you ever want to do anything but make a member–by–member copy? The answer has to do with asset allocation. Assets are things like memory off the heap, open files, ports, allocated hardware like printers, etc. Consider what happens if the constructor allocates an asset such as memory off the heap. If the copy constructor simply makes a copy of that asset without allocating its own, you end up with a troublesome situation: two objects thinking they have exclusive access to the same asset. This becomes nastier when the destructor is invoked for both objects and they both try to put the same asset back. What's needed is a copy constructor that allocates the new object its own assets. You can see that this is done by `Array's` copy constructor:

```
Array::Array(const Array& a)              // Copy constructor
    {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];    // Allocate new memory for copy
    copy(a);                              // Copy a's elements
    }
```

Another way to say this: Suppose a class has pointers as data members. Automatic copies of these pointers duplicate the pointer values, not the objects pointed to (e.g., the elements in an array). This is called a *shallow copy.* Programmer–defined copy constructors can copy the object pointed to if this is desired. This is called a *deep copy.* A good rule of thumb is that if your class requires a destructor to deallocate assets, it also requires a copy constructor.

In the copy constructor, look at the line:

```
    num_elts = a.num_elts;
```

Notice that the copy constructor has access to the private members of another object of the same class. In this case it has access to `a.num_elts`. In general an object of a class has access to the private members of another object of the same class.

Operator Overloading

Recall that we can overload functions that have the same name but take different arguments. For example,

```
   void func(int);
   void func(float);
```

Operators, such as $+, -$, and $=$, are also functions; they just have a peculiar syntax. The functional name of an operator is the operator symbol preceded by the keyword `operator` and followed by the appropriate argument types. For example, the + operator that adds an `int` to another `int` and returns an `int` is called `int operator+(int,int)`. The programmer can overload, or redefine, existing operators for newly defined types or classes. The only operators that can't be overloaded are `.` (member selection) ,`::` (scope resolution), and `.*` (member selection through pointer to function (see Stroustrup, 3rd ed., section 15.5)). The programmer cannot invent new operators. Nor can the precedence or format of the operators be changed. In addition, the operators cannot be redefined when applied to intrinsic types.

The above files have 3 examples of *operator overloading*:

```
    float &operator[] (int i);          // Subscripting

    Array & operator = (const Array &); // Array assignment
    Array & operator = (float);         // Scalar assignment
```

Comment 3: Overloading the assignment operator

The declaration `Array & operator = (const Array &)` overloads the assignment operator "=". The argument of this function `(const Array &)` will appear on the right hand side. In this case the argument is a reference to an object of the Array class. The assignment operator function returns a reference to the "current" Array. We could return a copy of the values in the array, but copying the values can take a lot of time and memory, so it is often better to return a reference to the array. Assignment member functions should work correctly when the left and right operands are the same object. Typically an assignment operator has two parts. The first part resembles a destructor in that it deletes the assets that the object already owns. The second part resembles a copy constructor in that it allocates new assets. In other words, you delete existing stuff before replacing it with new stuff. We can see this in the following example. Here array assignment is implemented by setting the size of the left operand to the size of the right operand and copying the elements.

```
Array& Array::operator=(const Array& rhs)
    {
    if (ptr_to_data != rhs.ptr_to_data)
        {
        setSize(rhs.num_elts);
        copy(rhs);
        }
    return *this;
    }
```

The test handles the case in which an array is assigned to itself. Without the test, `setSize()` might delete the elements of the left operand, which would also be the elements of the right operand, before the values are copied.

The size of an `Array` is set by deleting its elements, saving the new size, and allocating space in memory for an array with the appropriate number of elements.

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

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

void Array::copy(const Array& a)
    {
    // Copy a's elements into the elements of *this
    float *p = ptr_to_data + num_elts;
    float *q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data)
         *--p = *--q;
    }
```

Notice that the assignment operator is much like the copy constructor. In use the two look almost identical.

```
void fn(Array &matrix)
  {
    Array new_matrix=matrix;           //This is the copy constructor.
    new_matrix=matrix;                 //This is the assignment operator.
  }
```

The difference is that the copy constructor is used when a new object, that didn't already exist, is being created. The assignment operator is used if the left–hand object already exists. Like the copy constructor, the assignment operator should be provided whenever a shallow copy is not appropriate.

Comment 4: Overloading the subscripting operator

The declaration `float &operator[] (int i)` overloads the subscript operator `[ ]`. When this operator is used, the argument `i` is placed between the square brackets, as in `x[i]`. `float &` means that the operator function returns a reference to a float. In particular, x[i] returns a reference to the ith element of the array. This can be seen from the definition:

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

Array is a class dealing with a one dimensional array. But suppose we have a two dimensional array. How do we access the elements? The problem is that the operator [ ] can only take one argument. That's why two dimensional arrays in C have the form matrix[i][j]. The solution is to use the operator (), which can take more than one argument. Then we can write matrix(i,j) just like in Fortran. So in our class definition, we would write

```
float& operator()(int,int);
```

In the definition of the overloaded operator, we would write:

```
float& Array2D::operator()(int i, int j)
   {
   return ptr_to_data[i][j];             // indices start at 0

       or

   return ptr_to_data[i * num_columns + j];  // indices start at 0
   }
```

Other operators can be overloaded. We will discuss this further in lecture 6.

Error Function

We use the error function for bounds checking. If the subscript of the array goes beyond the bounds of the array, the error function goes into effect. error accepts a character string as an argument and writes it to the screen. By default, cout and cerr write to the screen. You can redirect the output of cout and input of cin in the command line of your program to files by writing:

```
a.out < infile > outfile
```

< and > are called *redirection operators*. Now the program will write output into outfile and get input from infile. But cerr will still write to the screen.

Returning by reference vs. returning by value

In the examples we have looked at so far, a reference to the "current" Array is returned. Overloaded operators don't always return references. Sometimes they return by value. For example, `int operator+(int a,int b)` returns by value. That is, `a + b` adds `a` and `b` and then generates a temporary object into which it can store the result of the addition. It then returns this object by value to the caller. What if we decided to return by reference instead? We would write `int& operator+(int a,int b)`. This would compile fine, but it would generate flaky results. The problem is that the returned reference refers to an object, call it `result`, whose scope is local to the function. `result` is out of scope by the time it can be used by the calling function. The moral of the story is that you should not return a reference to an object whose scope is local to a function and will soon disappear. Return references to objects that are going to be around for a while. Returning references to the "current" Array qualifies. Indeed, assignment operators usually return references. Another time when you would return a reference is when you pass an argument by reference and then the operator changes the value of that argument. For example, `int& operator++(int& i)`. Here the prefix operator takes a reference to an `int` as an argument, adds 1 to it, and returns a reference to the resulting `int`. The argument will last awhile since it came from the calling function. Input arguments can always be referential.