**Classes and Dynamic Arrays**

A dynamic array can have a base type that is a class and a class can have a member variable that is a dynamic array. One of our programming projects is to write a new string type object – similar to the string type object which already exists. A string is a dynamic array of characters, we can use what we now know of pointers and dynamic arrays to extend the functionality of c-strings by creating a new variable called ‘StringVar’. Fortunately I have started things for you (done the complicated part), you’ll just need to add some further functionality.

StringVar has 2 member variables – a pointer to a dynamic array called ‘value’, and an integer for the max\_length of the array. It has appropriate constructors to create instances given different start points and has a few member functions for input and output to the array.

There are some further complications which arise when classes and dynamic arrays are combined, which means we need to create some further key function types. You should by now be comfortable with constructors, mutators and accessors. Now lets look at destructors and copy constructors.

**Destructors**

We have discussed the issues arising from using dynamic variables and arrays using the heap – dynamic variables don’t go away unless there is a suitable ‘delete’ call. Even if a local pointer disappears at the end of
a function call, the dynamic variable remains, continuing to occupy the computers memory space. If we embed dynamic variables in a class (for instance having a dynamic array of characters as a private member of the StringVar class), anyone using the class may not even know the dynamic variable exists, let alone know the need to make a call to delete. Fortunately C++ has another member function that can deal with this problem for us, a ***‘destructor***‘ function. While the constructor function is called when an instance of the class is declared, a destructor function is called whenever the instance passes out of scope. We can call delete from the destructor to tidy up after us, deleting any dynamic variables created by the object.

StringVar::~StringVar()

{
 delete [] value;
}

The ’tilde’ symbol ‘**~**‘ is used to indicate a destructor, followed by the name of the class. Like constructors there is no returned type (not even void), there are also no parameters. A class can only have one destructor function, beyond this the function is just like any other member function. There is no need to explicitly make a call to the destructor function as it is automatically called for us by the program.

 **Copy Constructors**

We now need to deal with a further complication, remember what was said about destructor functions; *a destructor function is called whenever the instance passes out of scope*. Consider this code, which sends a object as a call by value parameter;

void show\_string(StringVar the\_string)

{
 cout << “The string is: ” << the\_string << endl;
}

int main
{
 StringVar greeting(“Hello”);
 show\_string(greeting);
 cout << “After call: ” << greeting << endl;
}

In this example int main begins by creating an instance of the StringVar object with the value member variable set to “Hello”. We then send this object as a call by value parameter to the show\_string function. Sending as a call by value parameter makes a copy of the object. However, the value member variable is a pointer variable, so a copy of the pointer variable is created – pointing to the same data.



When the show\_string function finishes ‘the\_string’ goes out of scope, so the destructor function is called, deleting the dynamic array it points towards. Remember that both pointers are pointing to the same memory
space, so they both become undefined.



The final line of int main tries to call the greeting again, which gives a problem as there is no value for it to display.

To avoid this problem we can write a copy constructor. The copy constructor is called automatically under 3 circumstances;

1) When a class object is declared and is initialised by another object of the same type.
2) When a function returns a value of the class type.
3) Whenever an argument is plugged in for a call-by-value parameter (as in the case above).

Essentially when the computer needs to make a copy of an existing object. For the StringVar object, the copy constructor looks like this;

StringVar::StringVar(const StringVar& string\_object) : max\_length(string\_object.length())
{

 value = new char[max\_length + 1];
 strcpy(value, string\_object.value);
}