**Week 2 - Arrays, Vectors & Strings**

Topics:  
1.0 Introducing Arrays  
 1.1 Arrays in Memory  
 1.2 Arrays and Functions  
 1.3 Arrays and Classes  
2.0 Multidimensional Arrays  
3.0 Vectors  
4.0 Strings  
 4.1 C-Strings  
 4.2 String CLASS

1.0 Introducing Arrays

*“An array is used to process a collection of data all of which is of the same type, such as a list of temperatures or a list of names.”*

We can use arrays to make code simpler when dealing with a group of similar pieces of data. If we had 5 scores, we could declare and use 5 separate variables (of type int). If we then wanted to perform a calculation on each of these values (while storing their values in memory) we would have to write 5 almost identical lines of code. While 5 is manageable, consider if there were 100 scores, the program will become harder to manage. It would be better to store these values in an array. If we wanted to declare an array of 5 integer type values, we would include square brackets after the variable name.

int score[5];

The square brackets indicate to the compiler that it is an array, and the number tells the number of elements. Each element is then given an index number, from 0 to (the number declared – 1), so in our case, score[0], score[1], score[2], score[3] and score[4]. This array of variables can now be used in much the same way as normal variables.

score[0] = 18;

cout << score[0];

A useful bonus now is that we can use a for loop to iterate through the array, and perform some task to each value, for instance;

for (i=0; i<5; i++)

{  
 cout << score[i];  
}

The loop above works for all cases where there are 5 scores to be output, but sometimes it is useful to make the program more versatile by using a constant for the size of the array;

const int number\_of\_students = 5;  
int score[number\_of\_students];  
for(i=0; i<5; i++)  
{  
 cin >> score[i];  
}

***1.1 Arrays in Memory***

When we declare a variable normally, a location in the computer’s memory is set aside to store the data we assign to the variable. The size (number of bytes) of the memory location depends on the type of the variable. When you declare an array a memory location to store all the elements of the array one after another is allocated. The location of the first element of the array is the address of the array. When you later refer to elements of the array they are found by counting forward from the first element – so if you want to use the 4th element of an array, the computer will go to the first element and count the amount of space needed for the first three elements and then know the next area is the 4th element.

In the previous example there are 5 elements in the array (numbered 0 to 4). If a different index number is given in error, the computer will continue, but the results returned could cause problems. For instance, if we tried to place a value into score[5] – it would actually try to place some value into the memory location after the array, which might store some other piece of information. The computer only knows the address of score[0] and then counts forward 5 spaces. It is worth checking that all loops iterate the correct number of times.

***1.2 Arrays and Functions***

Elements of an array can be sent as parameters to a function, in the same way you would send a normal variable, simply adding the index number of the array element. It is also possible to send a whole array as a parameter for a function – this isn’t a call by value parameter and it isn’t quite a call by reference parameter, it’s an ***array parameter***. When using arrays the computer knows the address (in memory) of the first element of the array, the type of each element in the array (base type) and hence the size of each element of the array, and also the size of the array. However, when an entire array is sent as a parameter, the computer only knows part of this information. It knows the address of the first element of the array, but it doesn’t know the size of the array. It can deduce the size of each element of the array, the array type is given. So, when you send an array to a function, you could also send another integer variable for the size of the array.

void output(int score[], int size\_of\_array)

***1.3 Arrays and Classes***

A class is a data type just like any other, so it is perfectly allowed to have an array of a class. It is also allowed to have an array as a class member – private member array variable. (This will make more sense once we have discussed classes in more detail.)

**2.0 Multi-dimensional Arrays**

Arrays can have more than one dimension. So far we have introduced one dimensional arrays – arrays with one index number. An array with 2 dimensions is similar to a spreadsheet, or a grid of values. You can declare a 2 dimensional array by having 2 different indexes after the array name;

int score[10][5];

This would create a 2 dimensional array, with 10 places for integers in one direction and 5 in the other;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Midterm | Coursework | Assignment | Final | Total |
| Student 1 |  |  |  |  |  |
| Student 2 |  |  |  |  |  |
| Student 3 |  |  |  |  |  |
| Student 4 |  |  |  |  |  |
| Student 5 |  |  |  |  |  |
| Student 6 |  |  |  |  |  |
| Student 7 |  |  |  |  |  |
| Student 8 |  |  |  |  |  |
| Student 9 |  |  |  |  |  |
| Student 10 |  |  |  |  |  |

Such an array could then be used to store the scores for 10 students in 5 different disciplines, as in the table above;

score[0][0]=10;

The above line could then set the midterm mark for student 1 to be 10.

for (i=0; i<4; i++)  
{  
 score[0][4] = score[0][4] + score[0][i];  
}

What would the above code do?

A two dimensional array could be used to store information that you can visualize in a grid or table. 3 dimensional arrays can be created by adding a third index such as;

Score[0][0][0];

### 3.0 Vectors

So far we have looked at arrays with a fixed size – we declare the size of the array when we declare the array, and then it remains that size. However, often the size of the array needs to change size during the running of the program – for instance if a student joins or leaves a course, or if I decide partway through my round of golf that I want to play more holes than the array is programmed to store. Vectors can be thought of as arrays that can grow (and shrink) in length while the program is running.

Much of the functioning of a vector is the same as that of an array. It has a series of elements each with an index number beginning from 0. The key difference is that you don’t need to specify the size of the vector, it is automatically done for you.

The definition of a vector is stored in the <vector> library and the std namespace. So you need to include the following lines within your code;

#include <vector>

using namespace std;

You then declare an instance of a vector using notation similar to this;

vector<int> v;

This creates a vector with elements of type ‘int’, and the vector is called v. Including the base type of the vector in <> essentially fills in the needed information into a ***template class***with the member variable of the class being declared as the specified type. Once defined, vector elements can be read or changed using the same square bracket notation;

cout << v[0];

While you can use this square bracket notation to read or change the elements you can’t use it to initialise an element – i.e. you can use the square brackets once the element is in use, but not to create it. To add new elements to a vector use the ‘push\_back’ member function.

v.push\_back(5);

v.push\_back(3);

This assigns values to the next elements in the vector (in this case v[0] and v[1]).

At any point you can find the ***‘size***’ of the vector by calling the member function ‘size’.

v.size();

Note that the size function returns a value of type ‘***unsigned int***‘. Unsigned int’s are a special type of int which only allows non-negative integer values. Most compilers will automatically convert this to an int when you need to use an int.

As well as a size, each vector will have a ‘***capacity***‘. While the size will return the number of elements in the vector, the capacity will return the amount of memory space that has been allocated for the vector. Whenever the size reaches the capacity of the vector, more memory space is automatically allocated. The amount of space which is allocated depends on the compiler, but normally the capacity is doubled each time it is reached. You can find out the capacity of a vector by calling the ‘capacity’ member function.

v.capacity();

Technically you can ignore the capacity function, as it is controlled automatically by the computer. However, in a large program if the vector capacity is continually doubled it could cause memory efficiency problems. Therefore it could be useful to be able to manually manipulate the capacity and size of a vector.

v.reserve(32); //sets the capacity to 32.

v.reserve(v.size() + 10); //sets the capacity to 10 greater than its size.

The reserve member function allows you to change the capacity of the vector as above. You can also manipulate the size of the vector using the resize member function.

v.resize(24);

This function changes the size of the vector to 24. If the vector’s size was previously less than 24, further elements are initialised using the vector’s constructor function. If the vectors size was greater than 24 then the vector removes any elements after the first 24.

4.0 Strings

There are 2 ways of dealing with strings in C++, the first is called a C String, which is an array of characters, the second is to use a class, called string. We will look at both.

**4.1 C Strings**

A C-String is just an array of characters, now we are familiar with arrays, we can use an array of characters to create a c-string.

char s[10] = “Hello”; // This declares an array of characters with Hello in it..

Note that with the above declaration, some (but not all) of the values in the array have been initialised. s[0] is now ‘H’, s[1] is now ‘e’ and so on. There is a key difference with this type of array though, a ‘***null character***‘ is automatically placed immediately after the last character in the string. The ***null character***looks like ‘**\0**‘, so in the declaration above s[5] is equal to ‘\0′. The null character instructs the computer (or the programmer) where the end of the string is.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| s[0] | s[1] | s[2] | s[3] | s[4] | s[5] | s[6] | s[7] | s[8] | s[9] |
| H | e | l | l | o | \0 | ? | ? | ? | ? |

This enables us to run code such as;

int index=0;

while (s[index] != ‘\0′)

{  
 s[index] = ‘X';  
 index++;  
}

What would this do? For this code (and some other features we will come on to) to work, the ‘\0′ character needs to be the last character of the string. Suppose we inadvertently included this line of code;

s[5]=’!';

What would happen if the previous code was then run?

Because of the ‘\0′ character, the size of the array of characters should be one more than the intended string, to allow for the character – so the hello string would need to be at least 6 characters long. Note that the following two initialisations are identical.

s[4] = “abc”;  
s[]=”abc”;

However, in this case, the ‘\0′ character would be omitted;

s[] = {‘a’, ‘b’, ‘c’};

Sounds good? Unfortunately c-strings don’t always work as you would like;

s = “Hello”; // THIS DOES NOT WORK!  
if (s = = “Hello”)… // THIS DOES NOT WORK EITHER!

There are some functions however, which can help us manipulate c-strings. These functions are all stored in the <cstring> library, which should be included in your project. (You don’t need to include the using namespace std directive though, but for reasons beyond this course!). The following are included in the cstring library.

|  |  |  |
| --- | --- | --- |
| Function | Description | Caution |
| strcpy(*Target, Source*) | Copies the *source*string into the *target* string. | Doesn’t check to see if there is space in the*target* string. |
| strncpy(*Target, Source, Limit*) | Copies up to *Limit* characters of the *source*string into the *target* string. |  |
| strcat(*Target, Source*) | Concatenates the *Source*to the end of the*Target.* | Doesn’t check to see if there is space in the*target* string |
| strncat(*Target, Source, Limit*) | Same as strcat, except only *Limit* characters are copied. |  |
| strlen(*Source)* | Returns an integer for the length of the string (not counting the null character!) |  |
| strcmp(*String1, String2)* | Returns 0 if *String 1* and *String 2* are the same, a positive value if *String1*is greater than *String2*, and a negative number if *String2* is greater than *String1*. | 0 is normally ‘false’ which is the opposite from what you would expect if the two strings are the same!  Comparison is based on Lexicographic values. |
| strncmp(*String1, String2, Limit*) | Same as strcmp, except only *Limit* characters are compared. |  |

**Input & Output**

Strings can be output simply using the ‘<<‘ as with any other variable.  
However, when it comes to input, ‘>>’ can be used, BUT, care should  
be taken. All white-spaces are skipped when reading in c-strings in this way,  
and each reading stops at the next space or line break.

char a[50], b[50];

cout << “Enter some input:”;

cin >> a >> b;

cout << “first input = ” << a << endl <<  
“second input = ” << b;

In the above code, if the user inputs “Hello everyone how are you?”,  
the following is returned.

first input = hello

second input = everyone

So, the ‘>>’ operator can collect one word at a time. Alternatively,  
there is a member function of all input streams (cin, etc.) which is similar  
to the ‘.get’ member function called ‘getline’. Getline takes 2 parameters,  
first the c-string in which to store the value, and second the maximum number  
of characters to read in.

char a[50];

cout << “Enter some input:”;

cin.getline(a,50);

cout << a;

Note that while this example will work, only 49 characters will be read in,  
leaving space for the ‘\0′ character.

A few further string manipulation functions which may be useful when dealing with c-strings are included in the <cstdlib> library.

atoi(*string*);

The atoi function converts and alphabetic string to an integer, so atoi(“1234″) returns an integer 1234.

That said – what would these two functions do?

atol(*string*);

atof(*string*);

**4.2 String CLASS**

As well as dealing with strings as an array of characters, there is also a way we can use strings as a basic data type – a predefined class with member functions included to make programming easier. The string class is in the <string> library, with definitions in the std namespace. So to include these functions you need to include these lines in your code;

#include <string>  
using namespace std;

This data type has overloaded operators such as ‘=’ and ‘+’ to enable us to manipulated strings more conveniently than with c-strings. So, the following are allowed;

s1 = “Hello World!”;  
s3 = s1 + s2;

We now don’t need to worry if the string s3 is large enough to store the values from s1 and s2, as it is automatically resized if necessary. There is a constructor function to allow an empty string if no initialisation is  
given;

string s4;

We can still use the ‘<<‘ operator to output the contents of a string, and ‘>>’ works the same as before, i.e. will read into a string all characters until a space or linebreak. To read in an entire line we can use the following construct;

string line;  
cout << “Enter a line of input:\n”;  
getline(cin, line);  
cout << line << “End of Output\n”;

This will perform as you would expect it to.

The string data type is still an array, so you can still use s3[i] to access the ith element in the string s3, but using square brackets doesn’t check for illegal references (so you may search for a character outside of the size of the array). The string data type is built in with a member function to combat this and only return a value if it is a legal reference, the ‘.at’ member function;

s3.at(i); //a better option than s3[i];

This at function works by using another member function ‘length’ which returns the length of the string;

s3.length();

You can also use the at function to change individual characters from the string;

s3.at(9) = ‘k';

The other member functions of the string class are;

string str; // Constructor which creates an empty string object.  
string str(“sample”); // Constructor which creates an object with ‘sample’ data.  
string str(a\_string); // Constructor which creates a copy of a string object called a\_string.  
str[i]; // Returns (or writes to) a single character at index ‘i’, without checking for illegal index.  
str.at(i); // Checks for illegal index.  
str.substr(position, length); // Returns a substring of the calling object beginning from position with length characters.  
str1 = str2; //Initialises str1 to str2’s data.  
str += str2; //Concatanates str2 to the end of str1.  
str.empty(); //Returns true if str is an empty string, otherwise false.  
str1 + str2 //Returns a string that concatenates str2 to the end of str1.  
str.insert(pos, str2); //Inserts str2 into str beginning at point pos.  
str.remove(pos, length); // Removes a substring from str, with length characters starting from pos.  
str1 == str2, str1 != str2, str1 < str2, str1 > str2, str1 <= str2, str1>=str2; // Comparison operators.  
str.find(str1); //Returns the index of the first occurance of str1 in str.  
str.find(str1, pos); //Returns the index of the first occurance of str1 in str after pos.  
str.find\_first\_of(str1, pos); //Returns the index of the first instance of any character from str1, starting from pos.  
str.find\_first\_not\_of(str1, pos); //Returns the index of the first instance in str of any character not in str1, starting at pos.