# Strings

## Introduction to Strings

`string` objects, which we have met briefly before on this course are part of the C++ standard library. It also the better way in C++ of dealing with sequences of characters.

An instance of the `string` data type is actually an object that provides its own set of functions that allow us to perform a range of operations on the `string` object it's self. These functions allow us to perform tasks such as searching for a certain character, determining length and substituting characters. In addition to this `string` objects are defined so that they work intuitively with a few of the operators we have seen so far.

## So what is an Object?

Up to know we have been mainly concerned with storing and manipulating pieces of information in variables. These variables have been storing very basic, single pieces, of information.
The majority of information we want to represent in programming, certainly in games, such as the player, enemies, buildings, are better represented as objects. When we think about things such as a player in a FPS it combines and encapsulates certain qualities (amount of health, ammunition) and abilities (firing weapons, jumping). It makes very little sense to talk about the individual qualities and abilities in isolation from each other.

A large portion of modern programming languages have built in to them methods of working with objects, and defining our own. Languages such as Java aim to be entirely oriented around objects.
These objects combine both functions and data. The data that the object holds is called member data and the functions that are part of the object are called member functions. We can think of abilities and qualities in these terms. Abilities are generally member functions and, qualities member data.

Using the example of a player in an FPS he may be of a type called DeathMatchPlayer, defined by the game programmer, and would have member data for health, armour and functions such as `FireWeapon()` that described its ability to fire a weapon.

Every object of the same type has the same basic structure so each object will have the same set of member data and member functions. The difference is that each object will have its own name, every player in a death match game may be of the same type but they all have different names. Each object will also have its own values for the data – not everyone has the same amount of health and armour, especially a few minutes into the game. Essentially what this means is that each object, even if of the same type, is unique. The data values are not

linked between the objects, and calling the function from one object only effects the object the function belongs to.

The thing about objects that makes them good is that like functions in the C++ standard library you do not need to know the inner workings to use them. The same way to drive a car you do not need to be able to build one. You just need to know the objects member data and functions, and the interface for those functions.

A variable is a data type, which doesn't have to be part of the standard built in C++ data types, and an identifier (or name) for that variable. So in the same way we can have variables of standard types we can have variables of none standard types – including objects.

The classical definition of an object is:
"An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable" (Brooch '94)

Essentially an object has qualities and abilities that define its behaviour.

**Accessing member functions and data**
You access member data and functions by using the member selection operator or dot operator as it is sometimes known – `(.)`.
You place the member selection operator after the variable name of the object, followed by the name of the member data or function you want to access.
When using VSC++ or VS.NET if your declarations are correct it should list the available data or functions after you type the member selection operator in.

Example:
```
std::string    MyName = "Mark";

MyName.size();
```

The following example demonstrates an object of type string, stored in the variable identified by `MyName`. We then access the `size()` member function of the string type using the member selection operator.

**Using `string` Objects**
String objects have a variety of operations that can be performed on them via operators and member functions. The tutorial looks are using the functions and operators to manipulate strings.

## Arrays and C-Style strings

In the days of C and the beginnings of C++, before `string` objects were realised C++ programmers represented strings with arrays of characters terminated by a null character. These arrays of characters are now called C-style string because this is where the practice first began, and still continues as far as I am aware.

C-Style strings come in two types. The main and most commonly used type being this character array. You declare and initialise it like you would any other array.

```
char PlayerName[] = “teh pwnerer”;
```

C-Style strings terminate with a character called the *null character* that signifies their end. The null characters is represented by ‘\0’. In the code above It is automatically stored at the end of the string. These means that technically the array above - PlayerName[] – technically has 12 elements. However functions that work with C-Style strings will say that PlayerName has a length of 11 which makes sense and is inline with how `string` objects work.

As with arrays of any other type you can specify array size when you declare your array. So another way to declare and initialise a C-style string is:

```
char PlayerName[21] = “teh pwnerer”;
```

The code above creates a C-style string that can hold 20 printable characters and the terminating null value.

### C-style strings and functions

C-style strings, being arrays, and not objects do not have any member functions. There is however a file that is part of the standard C++ library – `cstring` – that contains a variety of functions for working with C-style strings.

You can check out the functions provided in the cstring header by looking at the following list: http://www.cplusplus.com/ref/cstring/

### C-strings and `string` objects

As well a string objects encapsulating string properties into an object in a way C-style strings fail to do, `string` objects are also designed to work seamlessly with C-style strings.

```
string word1 = "teh ";
char word2[] = "pwnerer";

string playername = word1 + word2;
```

```
if(word1 != word2)
{
     cout << "word1 and word2 are not equal\n";
}
```

The above are all valid uses of C-style strings with `string` objects.

You can concatenate C-style strings with `string` objects but the result is always a `string` object. The following is for example and error:

```
// error
char playername[] = word1 + word2;
```

To concatenate simply means to link, chain or join together

**C-style strings shortcomings**

C-style strings lack the functionality of `string` objects. They do not come, as default, with the inbuilt functions and operators that `string` objects do. It is this functionality that makes string manipulation easier.
They are also arrays and therefore have the same shortcomings as arrays, the biggest being their fixed length.
Ideally you should work with `string` objects, sometimes though especially when using certain libraries or functions, such as some that come as part of DirectX, you need to work with C-style strings. Luckily you can often work with `string` object and then convert them the C-style string when needed via the `string` object member function c_str().

## Advanced Arrays

### Parallel Arrays

Suppose we have a two-column table, in which the first column holds the names of students on a degree course and the second column holds their average mark from examinations and assessments. We want to write a program that will store the data in the table and allow the user to enter a student's name and be told that person's average mark.

| **Name** | Tamara | Evil_Banana | Teh_pwnerer | Geoffrey |
|---|---|---|---|---|
| **Marks** | 23 | 69 | 1 | 13 |

We could use two arrays, one for each column in the table.
The elements of the first array contain the students' names.
The elements of the second array containing the students' average marks. The names and marks will be stored in order in the two arrays, so that the name in the element of the first array, with a particular index value, will correspond to the mark in the element of the second array with the same index value.

For example, if we have the two arrays:

```
string Student_Names[50];
double Average_Mark[50];
```

If a particular student's name is in `Student_Names[5]` then that student's mark is in `Average_Mark[5]`.

Obviously this requires us to be careful, we need access both arrays with the same index otherwise we will be matching up the wrong name to the results.

### Multi-dimensional Arrays

Another ways of solving the above problem is through the use of multi-dimension arrays, in this case a 2D array.

This is going to be a brief introduction to multidimensional arrays, as they are more complicated than arrays and not necessary for the course.
This brief introduction is also going to mainly focus on 2D arrays.

Multi-dimension arrays are often used to describe game worlds and concepts such as screen surfaces when dealing with games.

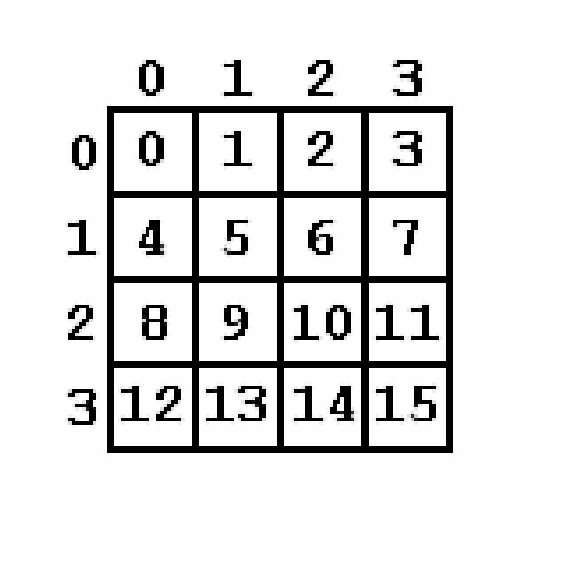
We sometimes need to index data in two or more dimensions. For example we might want to store the marks of a series of examinations for a class of students (or define a 3D world). Suppose there were 3 examinations, Design, Graphics

and Programming for a maximum of 50 students. Then we might define a 2-dimensional array as follows

```
int aiExamMarks[50][4];
```

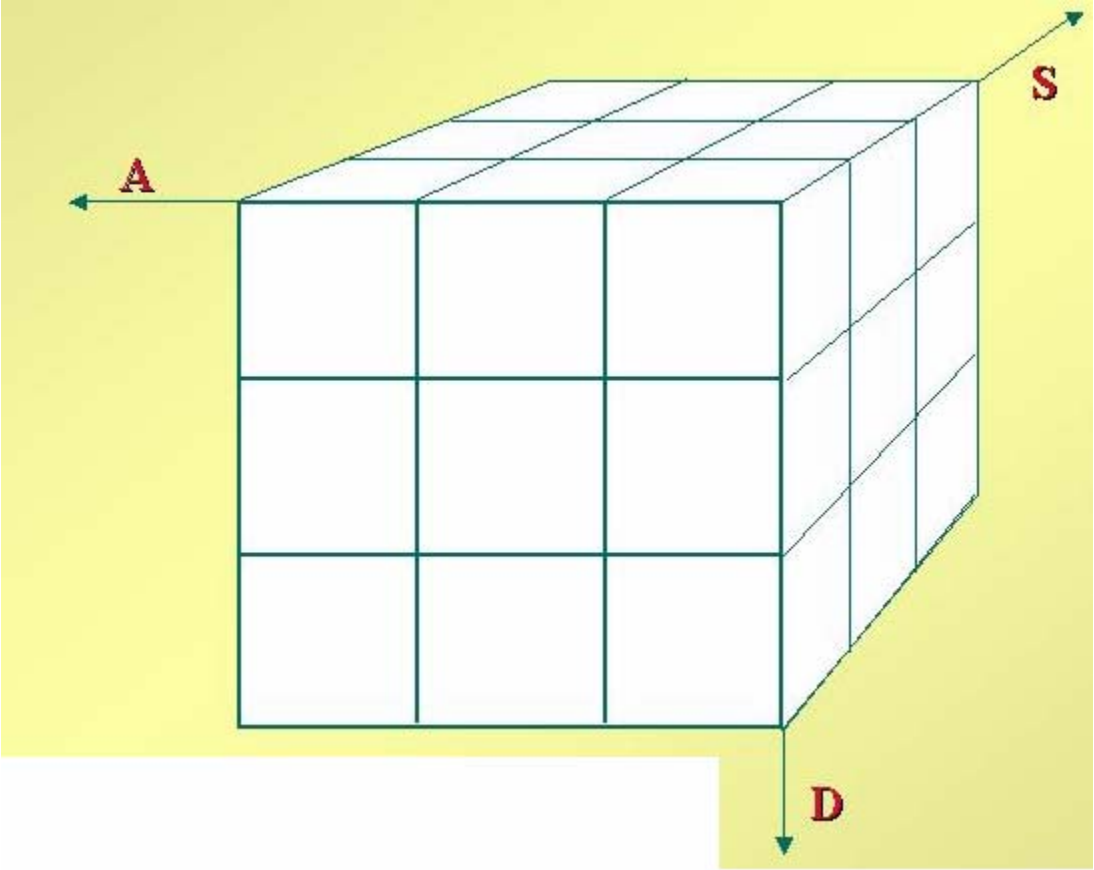
A column for name and one for each of the three subjects, a row for each of the students.

You can image a 2D array to look like a series of rows and columns:




Attempting to visual dimensions can be confusing when the dimensions of the array start to go above 3, a 3D array is essentially a cube, but how about a 4D array?

3D array visualisation:



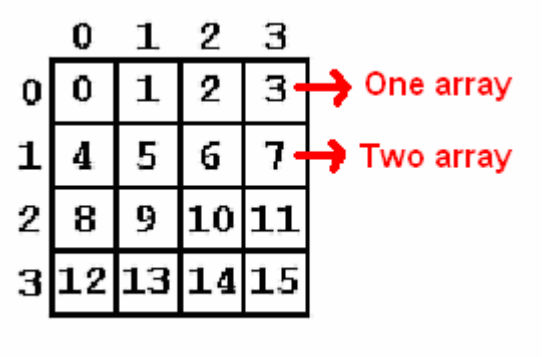

**Multi-Dimensionally Arrays and Memory**
Multi-dimensional arrays are also stored in a single contiguous block, with the values for the second index in sequence. Hence the above example would be stored as:

```
aiExamMarks[0][0]
aiExamMarks[0][1]
. . .
aiExamMarks[0][3]
aiExamMarks[1][0]
aiExamMarks[1][1]
. . .
aiExamMarks[1][3]
. . .
aiExamMarks[49][3]
```

**Visualising as Multiple 1D Arrays**
The 2-dimensional array, **aiExamMarks**, may be thought of as an array of 4 elements, each of which is an array of 50 elements whose type is **int**.
Thus:

- **aiExamMarks[0]** is the array of marks of student names

- **aiExamMarks[1]** is the array of marks for the Design examination
- **aiExamMarks[2]** is the array of marks for the Graphics examination
- **aiExamMarks[3]** is the array of marks for the Programming examination




### Initialisation of Multi-dimensional arrays

Multi-dimensional arrays may be initialised as a single block

```
int aiExamMarks[3][50] = {62, 33, 85, 41, 96, 66, . . .};
```

But it is clearer if we use nested braces for each of the three sub-arrays

```
int aiExamMarks[3][50] = {{62, 33, 85},
                          {41, 96, 66},
                             . . . };
```

As with 1-dimensional arrays, any un-initialised items will be set to zero. This applies also within each sub-array when, as in the second example, we use nested curly brackets **{}**.

Nested **for** statements are useful for iterating through all items in a multi-dimensional array, and can therefore be used to initialise the array.

```
const int ROWS = 15;
const int COLUMNS = 5;
int iValues[ROWS][COLUMNS];

for(int i = 0; i < ROWS; i++)
{
      for(int j = 0; j < COLUMNS; j++)
      {
            iValues[i][j] = 0;
      }
}
```

### Accessing Multi-dimensional arrays

Nested **for** statements are useful for iterating through all items in a multi-dimensional array, and therefore accessing each element in turn.

The following code prints out the array of exam marks:

```
int i, j;
for(i = 0; i < 3; i++)
{
    cout << “Examination “ << i + 1 << endl;
    for(j = 0; j < 50; j++)
      {
       cout << “Student “ << j + 1
            << “  Mark: “ << aiExamMarks[i][j]
            << endl;
      }
    cout << endl;
}
```

Of course, we can also display these values, grouped by student rather than by examination, by interchanging the order of the nested for loops, as follows:

```
for(j = 0; j < 50; j++)
{
      cout << “Student “ << j + 1 << “  Marks:“;
      for(i = 0; i < 3; i++)
      {
          cout << ‘\t’ << aiExamMarks[i][j];
      }
      cout << endl;
}
```

Normally not all the elements would be occupied in the array, so we would have to replace the limits, 3 and 50, in the for statements, with variables containing the actual number of examinations and students stored in the array, say **iExams** and **iStudents**.  Then the for statements in the first example would be altered to:

```
for(i = 0; i < iExams; i++)
{
    for(j = 0; j < iStudents; j++)
}
```

**Multi-dimensional arrays and Functions**

There is a technical problem about passing the name of a multi-dimensional array as a parameter to a function.

Arrays are stored in row major order, therefore we need to pass the number of columns to a function so that it can be determined where in memory the location of each rows begins.

A function to print out the values of the exam marks array might be coded as:

```
void DisplayMarks(int aiExamMarks[][4], int iExams, int iStudents)
{
```

```
   int i, j;

   for(i = 0; i < iExams; i++)
   {
      for(j = 0; j < iStudents; j++)
        {
        cout << aiExamMarks[i][j]) << '\t';
        }
      cout << endl;
   }
}
```

This function is fairly inflexible, since you can only use it when the second dimension of the array (number of columns) is 4. However, it does allow the first dimension of the array to vary.

A completely different, and more flexible approach, is to pass a two-dimensional array (and more generally a multi-dimensional array) as a one-dimensional array of its basic element type and provide each of its dimensions, other than the first dimension, as a separate parameter. The function can then calculate where each element in the multi-dimensional array is in the 1-diimensional array from its index values and the size of each dimension. For example, the previous example could be written as:

```
void DisplayMarks(int aiExamMarks[],
                  int iExams,         // occupied
                  int iStudents,      // elements
                  int iMaxStudents)   // physical size
{
   int i, j;

   for(i = 0; i < iExams; i++)
   {
      for(j = 0; j < iStudents; j++)
        {
         cout << array[i * iMaxStudents + j]) << '\t';
        }
      cout << endl;
   }
}
```

This assumes that all the students' marks for the first examination are stored first, followed by all the marks for the second examination and so on.