Chapter 5 – Part I

Array and Strings

5.1. Introduction

Variables in a program have values associated with them. During program execution these values are accessed by using the identifier associated with the variable in expressions etc. In none of the programs written so far have very many variables been used to represent the values that were required. Thus even though programs have been written that could handle large lists of numbers it has not been necessary to use a separate identifier for each number in the list. This is because in all these programs it has never been necessary to keep a note of each number individually for later processing. For example in summing the numbers in a list only one variable was used to hold the current entered number which was added to the accumulated sum and was then overwritten by the next number entered. If that value were required again later in the program there would be no way of accessing it because the value has now been overwritten by the later input.

If only a few values were involved a different identifier could be declared for each variable, but now a loop could not be used to enter the values. Using a loop and assuming that after a value has been entered and used no further use will be made of it allows the following code to be written. This code enters six numbers and outputs their sum:

```
sum = 0.0;
for (i = 0; i < 6; i++)
  {
    cin >> x;
    sum += x;
  }
```

This of course is easily extended to n values where n can be as large as required. However if it was required to access the values later the above would not be suitable. It would be possible to do it as follows by setting up six individual variables:

```
 float a, b, c, d, e, f;
```

and then handling each value individually as follows:

```
sum = 0.0;
cin >> a;  sum += a;
```

```
    cin >> b;  sum += b;
    cin >> c;  sum += c;
    cin >> d;  sum += d;
    cin >> e;  sum += e;
    cin >> f;  sum += f;
```

which is obviously a very tedious way to program. To extend this solution so that it would work with more than six values then more declarations would have to be added, extra assignment statements added and the program re-compiled. If there were 10000 values imagine the tedium of typing the program (and making up variable names and remembering which is which)!

To get round this difficulty all high-level programming languages use the concept of a data structure called an **Array.**

5.2. Array

An **array** is a data structure which allows a collective name to be given to a group of elements which ***all have the same type***. An individual element of an array is identified by its own unique ***index*** (or **subscript**).

An array can be thought of as a collection of numbered boxes each containing one data item. The number associated with the box is the index of the item. To access a particular item the index of the box associated with the item is used to access the appropriate box. The index **must** be an integer and indicates the position of the element in the array. Thus the elements of an array are **ordered** by the index.

5.3. One Dimensional Array

5.3.1. Declaration of Arrays

Declaring the name and type of an array and setting the number of elements in an array is called dimensioning the array. The array must be declared before one uses in like other variables. In the array declaration one must define:

1. The type of the array (i.e. integer, floating point, char etc.)
2. Name of the array,
3. The total number of memory locations to be allocated or the maximum value of each subscript. i.e. the number of elements in the array (array size).

So the general syntax for the declaration is: ***DataTypename arrayname [array size];***

***Note**: array size cannot be a variable whose value is set while the program is running.*

For example data on the average temperature over the year in Ethiopia for each of the last 100 years could be stored in an array declared as follows:

```
float annual_temp[100];
```

This declaration will cause the compiler to allocate space for 100 consecutive float variables in memory. The number of elements in an array must be fixed at compile time. It is best to make the array size a constant and then, if required, the program can be changed to handle a different size of array by changing the value of the constant,

```
const int NE = 100;
float annual_temp[NE];
```

then if more records come to light it is easy to amend the program to cope with more values by changing the value of NE. This works because the compiler knows the value of the constant NE at compile time and can allocate an appropriate amount of space for the array. It would not work if an ordinary variable was used for the size in the array declaration since at compile time the compiler would not know a value for it.

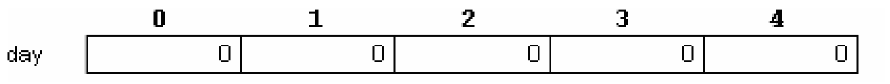
5.3.2. Initializing Arrays

When declaring an array of local scope (within a function), if we do not specify the array variable the array will not be initialized, so its content is undetermined until we store some values in it.

if we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare:

int day [5];

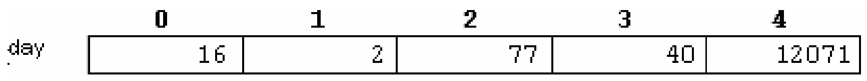
every element of day *will be set initially to* ***0:***



But additionally, when we declare an Array, we have the possibility to assign initial values to each one of its elements using curly brackets { }. For example:

int day [5] = { 16, 2, 77, 40, 12071 };

The above declaration would have created an array like the following one:



The number of elements in the array that we initialized within curly brackets { } must be equal or less than the length in elements that we declared for the array enclosed within square brackets [ ]. If we have less number of items for the initialization, the rest will be filled with zero.

For example, in the example of the day array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element. If we ignore the last initial value (12071) in the above initialization, 0 will be taken automatically for the last array element.

Because this can be considered as useless repetition, C++ allows the possibility of leaving empty the brackets [ ], where the number of items in the initialization bracket will be counted to set the size of the array.

```
int day [] = { 1, 2, 7, 4, 12,9 };
```

The compiler will count the number of initialization items which is 6 and set the size of the array day to 6 (i.e.: day[6])

You can use the initialization form only when defining the array. You cannot use it later, and cannot assign one array to another once. I.e.

```
int arr [] = {16, 2, 77, 40, 12071};

int ar [4];

ar[]={1,2,3,4};//not allowed

arr=ar;//not allowed
```

Note: when initializing an array, we can provide fewer values than the array elements. E.g. int a [10] = {10, 2, 3}; in this case the compiler sets the remaining elements to zero.

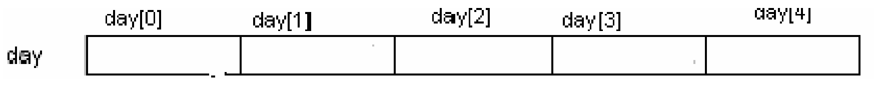
5.3.3. Accessing and processing array elements

- ✓ In any point of the program in which the array is visible we can access individually anyone of its elements for reading or modifying it as if it was a normal variable. To access individual elements, index or subscript is used. The format is the following:

  name [*index*]

- ✓ In c++ the first element has an index of 0 and the last element has an index, which is one less the size of the array (i.e. arraysize-1). Thus, from the above declaration, day[0] is the first element and day[4] is the last element.

- ✓ Following the previous examples where ***day*** had 5 elements and each element is of type int, the name, which we can use to refer to each element, is the following one:



- ✓ For example, to store the value 75 in the third element of the array variable ***day*** a suitable sentence would be:

  day[2] = 75; //as the third element is found at index 2

- ✓ And, for example, to pass the value of the third element of the array variable ***day*** to the variable a , we could write:

  a = day[2];

- ✓ Therefore, for all the effects, the expression ***day[2]*** is like any variable of type int with the same properties. Thus an array declaration enables us to create a lot of variables of the same type with a single declaration and we can use an index to identify individual elements.
- ✓ Notice that the third element of day is specified ***day[2]*** , since first is ***day[0]*** , second ***day[1]*** , and therefore, third is ***day[2]*** . By this same reason, its last element is ***day [4]***. Since if we wrote day [5], we would be acceding to the sixth element of day and therefore exceeding the size of the array. This might give you either error or unexpected value depending on the compiler.
- ✓ In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can cause certain detectable problems, since they do not cause compilation errors but they can cause unexpected results or serious errors during execution. The reason why this is allowed will be seen ahead when we begin to use pointers.
- ✓ At this point it is important to be able to clearly distinguish between the two uses the square brackets [ ] have for arrays.
  - One is to set the size of arrays during declaration
  - The other is to specify indices for a specific array element when accessing the elements of the array
- ✓ We must take care of not confusing these two possible uses of brackets [ ] with arrays:

Eg: int day[5]; *// declaration of a new Array (begins with a type name)*

day[2] = 75; *// access to an element of the Array.*

Other valid operations with arrays in accessing and assigning:

int a=1;

day [0] = a;

```
day[a] = 5;
 b = day [a+2];
day [day[a]] = day [2] + 5;
day [day[a]] = day[2] + 5;
```

Eg: A*rrays example, display the sum of the numbers in the array*

```
#include <iostream.h>
int day [ ] = {16, 2, 77, 40, 12071};
int n, result=0;
void main () {
        for ( n=0 ; n<5 ; n++ ) {
                result += day[n]; }
        cout << result;
        getch();
}
```

5.3.4. Copying Arrays

- ✓ The assignment operator *cannot* be applied to array variables:

```
const int SIZE=10
int x [SIZE] ;
int y [SIZE] ;
x = y ;              // Error - Illegal
```

- ✓ Only individual elements can be assigned to using the index operator, e.g., **`x[1] = y[2];.`** To make all elements in 'x' the same as those in 'y' (equivalent to assignment), a loop has to be used.

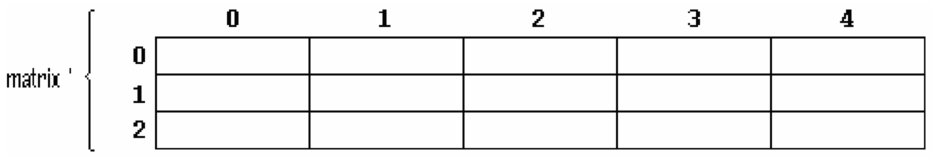
```
// Loop to do copying, one element at a time
for (int i = 0 ; i < SIZE; i++)
      x[i] = y[i];
```

- ✓ This code will copy the elements of array y into x, overwriting the original contents of x. A loop like this has to be written whenever an array assignment is needed.
- ✓ Notice the use of a constant to store the array size. This avoids the literal constant '10' appearing a number of times in the code. If the code needs to be edited to use different

sized arrays, only the constant needs to be changed. If the constant is not used, all the '10's would have to be changed individually - it is easy to miss one out.

5.4. Multidimensional Arrays

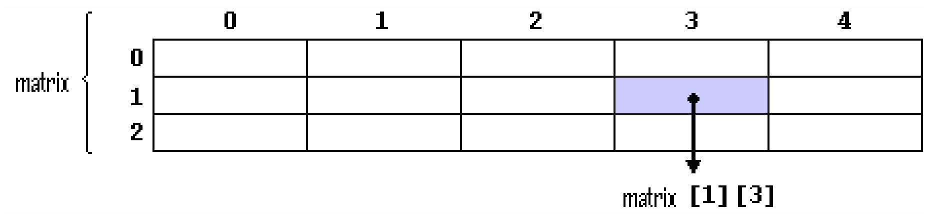
- ✓ An array may have more than one dimension. Each dimension is represented as a subscript in the array. Therefore a two dimensional array has two subscripts, a three dimensional array has three subscripts, and so on.
- ✓ Arrays can have any number of dimensions, although most of the arrays that you create will likely be of one or two dimensions.
- ✓ A chess board is a good example of a two-dimensional array. One dimension represents the eight rows, the other dimension represents the eight columns.
- ✓ Multidimensional arrays can be described as arrays of arrays. For example, a bi-dimensional array can be imagined as a bi-dimensional table of a uniform concrete data *type.*



- ✓ Matrix represents a bi-dimensional array of 3 per 5 values of type int . The way to declare this array would be:

```
int matrix[3][5];
```

- ✓ For example, the way to reference the second element vertically and fourth horizontally in an expression would be: **matrix[1][3]**



(remember that array indices always begin by **0**)

- ✓ ***Multidimensional arrays*** are not limited to two indices (two dimensions). They can contain so many indices as needed, although it is rare to have to represent more than 3 dimensions. Just consider the amount of memory that an array with many indices may need. For example: char century [100][365][24][60][60];

- ✓ Assigns a **char** for each second contained in a century, that is more than 3 billion **chars** ! What would consume about 3000 *megabytes* of RAM memory if we could declare it?
- ✓ Multidimensional arrays are nothing else than an abstraction, since we can simply obtain the same results with a simple array by putting a factor between its indices:
  - o **int matrix [3][5];** is equivalent to **int matrix [15];** (3 * 5 = 15)

Initializing Multidimensional Arrays

- ✓ To initialize a multidimensional arrays , you must assign the list of values to array elements in order, with last array subscript changing while the first subscript while the first subscript holds steady. Therefore, if the program has an array **`int theArray[5][3],`** the first three elements go **`int theArray[0];`** the next three into **`theArray[1];`** and so forth.

The program initializes this array by writing

```
int theArray[5][3] ={ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
              11, 12, 13, 14, 15};
```

for the sake of clarity, the program could group the initializations with braces, as shown below.

```
int theArray[5][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8,
              9}, {10, 11, 12}, {13, 14,15} };
```

- ✓ The compiler ignores the inner braces, which clarify how the numbers are distributed. Each value should be separated by comma, regardless of whither inner braces are include. The entire initialization must set must appear within braces, and it must end with a semicolon.

Omitting the Array Size

- ✓ If a one-dimensional array is initialized, the size can be omitted as it can be found from the number of initializing elements:

```
int x[] = { 1, 2, 3, 4} ;
```

- ✓ This initialization creates an array of four elements.

Note however:

```
int x[][] = { {1,2}, {3,4} } ; // error is not
                   allowed.
```

```
        and must be written
        int x[2][2] = { {1,2}, {3,4} } ;
```

✓ Example of multidimensional array

```
  #include<iostream.h>
  void main(){
   int SomeArray[5][2] =  {{0,0},{1,2}, {2,4},{3,6}, {4,8}}
   for ( int i=0; i<5; i++)
    for (int j = 0; j<2;j++)
    {
        cout<<"SomeArray["<<i<<"]["<<j<<'']: '';
        cout<<endl<<SomeArray[i][ j];
    }
}
```