Chapter Four
Array, Pointers and Strings

Chapter Four

4. Arrays and Strings

4.1. 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.**

4.2.   What is an arrays

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.

4.3.   One Dimensional Array

4.3.1. Declaration of Arrays

An array declaration is very similar to a variable declaration. First a type is given for the elements of the array, then an identifier for the array and, within square brackets, the number of elements in the array. The number of elements **must be an integer**.

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.

4.3.2. Accessing Array Elements

Given the declaration above of a 100-element array the compiler reserves space for 100 consecutive floating point values and accesses these values using an index/subscript that takes values from 0 to 99. The *__first element__* in an array in C++ always has the *__index 0__*, and if the array has *n* elements the *__last__* element will have the *__index n-1__*.

An **array element** is accessed by writing the identifier of the array followed by the subscript in square brackets. Thus to set the 15th element of the array above to 1.5 the following assignment is used:

```
annual_temp[14] = 1.5;
```

Note that since the first element is at index 0, then the *__ith__* element is at *__index i-1__*. Hence in the above the 15th element has index 14.

An array element can be used anywhere an identifier may be used. Here are some examples assuming the following declarations:

```
const int NE = 100,
          N = 50;
int i, j, count[N];
float annual_temp[NE];
float sum, av1, av2;
```

A value can be read into an array element directly, using cin

```
cin >> count[i];
```

The element can be increased by 5,

```
        count[i] = count[i] + 5;
```
or, using the shorthand form of the assignment

```
        count[i] += 5;
```
Array elements can form part of the condition for an if statement, or indeed, for any other logical expression:

```
if (annual_temp[j] < 10.0)
   cout << "It was cold this year "
        << endl;
```
for statements are the usual means of accessing every element in an array. Here, the first NE elements of the array annual_temp are given values from the input stream cin.

```
for (i = 0; i < NE; i++)
  cin >> annual_temp[i];
```
The following code finds the average temperature recorded in the first ten elements of the array.

```
sum = 0.0;
for (i = 0; i <10; i++)
    sum += annual_temp[i];
av1 = sum / 10;
```
Notice that it is good practice to use named constants, rather than literal numbers such as 10. If the program is changed to take the average of the first 20 entries, then it all too easy to forget to change a 10 to 20. If a const is used consistently, then changing its value will be all that is necessary.

For example, the following example finds the average of the last k entries in the array. k could either be a variable, or a declared constant. Observe that a change in the value of k will still calculate the correct average (provided `k<=NE`).

```
sum = 0.0;
for (i = NE - k; i < NE; i++)
  sum += annual_temp[i];
av2 = sum / k;
```
Important - C++ does not check that the subscript that is used to reference an array element actually lies in the subscript range of the array. Thus C++ will allow the assignment of a value to `annual_temp[200]`, however the effect of this assignment is unpredictable. For example it could lead to the program attempting to assign a value to a memory element that is outside the program's allocated memory space. This would lead

to the program being terminated by the operating system. Alternatively it might actually access a memory location that is within the allocated memory space of the program and assign a value to that location, changing the value of the variable in your program which is actually associated with that memory location, or overwriting the machine code of your program. Similarly reading a value from `annual_temp[200]` might access a value that has not been set by the program or might be the value of another variable. It is the programmer's responsibility to ensure that if an array is declared with `n` elements then no attempt is made to reference any element with a subscript outside the range 0 to `n-1`. Using an index, or subscript, that is out of range is called Subscript Overflow. Subscript overflow is one of the commonest causes of erroneous results and can frequently cause very strange and hard to spot errors in programs.

4.3.3. Initialization of arrays

The initialization of simple variables in their declaration has already been covered. An array can be initialized in a similar manner. In this case the initial values are given as a list enclosed in curly brackets. For example initializing an array to hold the first few prime numbers could be written as follows:

```
int primes[] = {1, 2, 3, 5, 7, 11, 13};
```

Note that the array has not been given a size, the compiler will make it large enough to hold the number of elements in the list. In this case primes would be allocated space for seven elements. If the array is given a size then this size must be greater than or equal to the number of elements in the initialization list. For example:

```
int primes[10] = {1, 2, 3, 5, 7};
```

would reserve space for a ten element array but would only initialize the first five elements.

Example Program: Printing Outliers in Data

The requirement specification for a program is:

A set of positive data values (200) are available. It is required to find the average value of these values and to count the number of values that are more than 10% above the average value.

Since the data values are all positive a negative value can be used as a sentinel to signal the end of data entry. Obviously this is a problem in which an array must be used since the values must first be entered to find the average and then each value must be compared with this average. Hence the use of an array to store the entered values for later re-use.

An initial algorithmic description is:

```
initialize.
enter elements into array and sum elements.
evaluate average.
scan array and count number greater than
                    10% above average.
output results.
```

This can be expanded to the complete algorithmic description:

```
set sum to zero.
set count to zero.
set nogt10 to zero.
enter first value.
while value is positive
   {
    put value in array element with index count.
    add value to sum.
    increment count.
    enter a value.
   }
average = sum/count.
for index taking values 0 to count-1
    if array[index] greater than 1.1*average
        then increment nogt10.
output average, count and nogt10.
```

In the above the variable nogt10 is the number greater than 10% above the average value. It is easy to argue that after exiting the while loop, count is set to the number of positive numbers entered. Before entering the loop count is set to zero and the first number is entered, that is count is one less than the number of numbers entered. Each time round the loop another number is entered and count is incremented hence count remains one less than the number of numbers entered. But the number of numbers entered is one greater than the number of positive numbers so count is therefore equal to the number of positive numbers.

A **main()** program written from the above algorithmic description is given below:

```
void main()
{
  const int NE = 200;   // maximum no of elements in array
  float sum = 0.0;      // accumulates sum
  int count = 0;        // number of elements entered
  int nogt10 = 0;       // counts no greater than 10%
```

```
                        // above average
float x;                // holds each no as input
float indata[NE];       // array to hold input
float average;          // average value of input values
int i;                  // control variable

    // Data entry, accumulate sum and count
    // number of +ve numbers entered
cout << "Enter numbers, -ve no to terminate: " << endl;
cin >> x;
while (x >= 0.0)
  {
    sum = sum + x;
    indata[count] = x;
    count = count + 1;
    cin >> x;
  }

    // calculate average
average = sum/count;

    // Now compare input elements with average
for (i = 0; i < count; i++)
  {
    if (indata[i] > 1.1 * average)
      nogt10++;
  }

    // Output results
cout << "Number of values input is " << n;
cout << endl
     << "Number more than 10% above average is "
     << nogt10 << endl;
}
```

Since it was assumed in the specification that there would be less than 200 values the array size is set at 200. In running the program less than 200 elements may be entered, if n elements where n < 200 elements are entered then they will occupy the first n places in the array indata. It is common to set an array size to a value that is the maximum we think will occur in practice, though often not all this space will be used.

Example Program: Test of Random Numbers

The following program simulates the throwing of a dice by using a random number generator to generate integers in the range 0 to 5. The user is asked to enter the number of trials and the program outputs how many times each possible number occurred.

An array has been used to hold the six counts. This allows the program to increment the correct count using one statement inside the loop rather than using a switch statement

with six cases to choose between variables if separate variables had been used for each count. Also it is easy to change the number of sides on the dice by changing a constant. Because C++ arrays start at subscript 0 the count for an i occurring on a throw is held in the i-1th element of this count array. By changing the value of the constant die_sides the program could be used to simulate a die_sides-sided die without any further change.

```cpp
#include <iostream.h>
#include <stdlib.h>     // time.h and stdlib.h required for
#include <time.h>       // random number generation

void main()
{
  const int die_sides = 6;    // maxr-sided die
  int count[die_sides];        // holds count of each
                              // possible value
  int no_trials,          // number of trials
      roll,               // random integer
      i;                  // control variable
  float sample;           // random fraction 0 .. 1

      // initialize random number generation and count
      // array and input no of trials
  srand(time(0));
  for (i=0; i < die_sides; i++)
    count[i] = 0;
  cout << "How many trials? ";
  cin >> no_trials;

      // carry out trials
  for (i = 0; i < no_trials; i++)
    {
      sample = rand()/float(RAND_MAX);
      roll = int ( die_sides * sample);
          // returns a random integer in 0 to die_sides-1
      count[roll]++;          // increment count
    }

      // Now output results
  for (i = 0; i < die_sides; i++)
    {
      cout << endl << "Number of occurrences of "
          << (i+1) << " was " << count[i];
    }
  cout << endl;
}
```

4.3.4. Copying Arrays

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

```cpp
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 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.

4.4. 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.

Suppose the program contains a class named square. The declaration of array named board that represents would be

```
Square board[8][8];
```

The program could also represent the same data with a one dimensional, 64-square array. For example, it could include the statement

```
Square board[64];
```

Such a representation does not correspond as closely to the real-world object as the two dimensional array, however.

Suppose that when the game begins. The king id located in the fourth position in the first row. Counting from zero that position corresponds to board[0][3] in the two dimensional array, assuming that the first subscript corresponds to the row, and the second to the column.

4.4.1. 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.

4.4.2. 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];
  }
}
```

4.5. Strings representation and manipulation

String in C++ is nothing but a sequence of character in which the last character is the null character '\0'. The null character indicates the end of the string. Any array of character can be converted into string type in C++ by appending this special character at the end of the array sequence.

In C++ strings of characters are held as an **array of characters**, one character held in each array element. In addition a special **null character**, represented by `` `\0' ``, is appended to the end of the string to indicate the end of the string. Hence if a string has `n` characters then it requires an `n+1` element array (at least) to store it. Thus the character `` `a' `` is stored in a single byte, whereas the single-character string `"a"` is stored in two consecutive bytes holding the character `` `a' `` and the null character.

A string variable `s1` could be declared as follows:

```
char s1[10];
```

The string variable `s1` could hold strings of length up to nine characters since space is needed for the final null character. Strings can be initialized at the time of declaration just as other variables are initialized. For example:

```
char s1[] = "example";
char s2[20] = "another example"
would store the two strings as follows:
s1 |e|x|a|m|p|l|e|\0|
s2  |a|n|o|t|h|e|r| |e|x|a|m|p|l|e|\0|?|?|?|?|
```

In the first case the array would be allocated space for eight characters, that is space for the seven characters of the string and the null character. In the second case the string is set by the declaration to be twenty characters long but only sixteen of these characters are set, i.e. the fifteen characters of the string and the null character. Note that the length of a string does not include the terminating null character.

4.5.1. String Output

A string is output by sending it to an output stream, for example:

```
cout << "The string s1 is " << s1 << endl;
would print
The string s1 is example
```

The **setw(width)** I/O manipulator can be used before outputting a string, the string will then be output right-justified in the field width. If the field width is less than the length of the string then the field width will be expanded to fit the string exactly. If the string is to be left-justified in the field then the **setiosflags** manipulator with the argument **ios::left** can be used.

4.5.2. String Input

When the input stream **cin** is used space characters, **newline** etc. are used as separators and terminators. Thus when inputting numeric data **cin** skips over any leading spaces and terminates reading a value when it finds a white-space character (**space, tab, newline** etc.). This same system is used for the input of strings, hence a string to be input cannot start with leading spaces, also if it has a space character in the middle then input will be terminated on that space character. The null character will be appended to the end of the string in the character array by the stream functions. If the string s1 was initialized as in the previous section, then the statement

```
cin << s1;
```
would set the string s1 as follows when the string **"first"** is entered (without the double quotes)
```
|f|i|r|s|t|\0|e|\0|
```
Note that the last two elements are a relic of the initialization at declaration time. If the string that is entered is longer than the space available for it in the character array then C++ will just write over whatever space comes next in memory. This can cause some very strange errors when some of your other variables reside in that space!

To read a string with several words in it using cin we have to call cin once for each word. For example to read in a name in the form of a Christian name followed by a surname we might use code as follows:

```
char christian[12], surname[12];
cout << "Enter name ";
cin >> christian;
cin >> surname;
cout << "The name entered was "
     << christian << " "
     << surname;
```
The name would just be typed by the user as, for example, **Ian Aitchison** and the output would then be

```
      The name entered was Ian Aitchison
      Enter a string: Law is a bottomless pit.

      You entered:     Law
```

Where did the rest of the phrase go?

It turns the insertion operator >> consider a space to be a terminating character.

Thus it will read strings consisting of a single word, but anything typed after a space is thrown away.

To read text containing blanks we use another function, **cin::get()**.

```
      #include<iostream.h>
      void main()
      {
        const int max=80;
        char str[max];
      cout<<"\n Enter a string;";
      cin.get(str,max);  // max avoid buffer overflow
      cout<<"\n You entered :  "<<str;
      }
```

Reading multiple lines

We have solved the problem of reading strings with embedded blanks, but what about strings with multiple lines? It turns out that the **cin::get()** function can take a third argument to help out in this situation.

This argument specifies the character that tells the function to stop reading. The default value of this argument is the **newline('\n')** character, but if you call the function with some other character for this argument, the default will be overridden by the specified character.

In the next example, we call the function with a dollar sign (**'$'**) as the third argument

```
      //reads multiple lines, terminates on '$' character
      #include<iostream.h>
      void main(){
       const int max=80;
      char str[max];
      cout<<"\n Enter a string:\n";
      cin.get(str, max, '$');          //terminates with $
      cout<<\n You entered:\n"<<str; }
```

now you can type as many lines of input as you want. The function will continue to accept characters until you enter the terminated character **$** (or untill you exceed the size of the array. Remember, you must still press Enter key after typing the **'$'** character .

4.5.3. Avoiding buffer over flow

The strings in the program invites the user to type in a string. What happens if the user enters a string that is longer than the array used to hold it? There is no built-in mechanism in C++ to keep a program from inserting array elements outside an array.

However, it is possible to tell the >> operator to limit the number of characters it places in an array.

```
//avoids buffer overflow with cin.width
#include<iostream.h>
#include<iomanip.h>       //for setw
void main(){
const int MAX=20;
char str[MAX];
cout<<"\n Enter a string: ";
cin>>setw(MAX)>>str;
cout<<"\n You entered :"<<str;
}
```

4.5.4. String constants

You can initialize a string to a constant value when you define it. Here's an example'

```
#include<iostream.h>
void main(){
char str[] = "Welcome to C++ programming language";
cout<<str;
}
```

if you tried to the string program with strings that contain more than one word , you may have unpleasant surprise. Copying string the hard way

The best way to understand the true nature of strings is to deal with them character by character

```
#include<iostream.h>
#include<string.h>   //for strlen()
void main()
{
const int max=80;
char str1[]='' Oh, Captain, my Captain!"
        our  fearful trip is done";
char str2[max];
for(int i=0; i<strlen(str1);i++)
str2[i]=str1[1];
str2[i]='\0';
cout<<endl;
cout<<str2;
}
```

4.5.5. Copying string the easy way

Of course you don't need to use a **for** loop to copy a string. As you might have guesses, a library function will do it for you. You can copy strings using **strcpy** or **strncpy** function. We assign strings by using the string copy function **strcpy**. The prototype for this function is in **string.h.**

```
strcpy(destination, source);
```

strcpy copies characters from the location specified by source to the location specified by destination. It stops copying characters after it copies the terminating null character.

o   The return value is the value of the destination parameter.

You must make sure that the destination string is large enough to hold all of the characters in the source string (including the terminating null character).

Example:
```
#include <iostream.h>
#include <string.h>
    void main(){
     char me[20] = "David";
     cout << me << endl;
     strcpy(me, "YouAreNotMe");
     cout << me << endl ;
     return;
   }
```

There is also another function **strncpy,** is like **strcpy**, except that it copies only a zspecified number of characters.

```
strncpy(destination, source, int n);
```

It may not copy the terminating null character.

Example
```
#include <iostream.h>
#include <string.h>
void main() {
    char str1[] = "String test";
    char str2[] = "Hello";
    char one[10];
    strncpy(one, str1, 9);
    one[9] = '\0';
    cout << one << endl;
    strncpy(one, str2, 2);
    cout << one << endl;
    strcpy(one, str2);
    cout << one << endl;
```

```
    }
```

4.5.6. Concatenating strings

In C++ the + operator cannot normally be used to concatenate string, as it can in some languages such as BASIC; that is you can't say

```
    Str3 = str1 + str2;
```

You can use strcat() or strncat

The function `strcat` concatenates (appends) one string to the end of another string.

```
    strcat(destination, source);
```

o   The first character of the source string is copied to the location of the terminating null character of the destination string.

o   The destination string must have enough space to hold both strings and a terminating null character.

Example:

```
#include <iostream.h>
#include <string.h>

   void main() {
      char str1[30];
      strcpy(str1, "abc");
      cout << str1 << endl;
      strcat(str1, "def");
      cout << str1 << endl;

      char str2[] = "xyz";
      strcat(str1, str2);
      cout << str1 << endl;
      str1[4] = '\0';
      cout << str1 << endl;
   }
```

The function strncat is like strcat except that it copies only a specified number of characters.

```
    strncat(destination, source, int n);
```
 It may not copy the terminating null character.

Example:

```
#include <iostream.h>
   #include <string.h>
   void  main() {
      char str1[30];
      strcpy(str1, "abc");
      cout << str1 << endl;
      strncat(str1, "def", 2);
      str1[5] = '\0';
      cout << str1 << endl;
```

```
        char str2[] = "xyz";
        strcat(str1, str2);
        cout << str1 << endl;
        str1[4] = '\0';
        cout << str1 << endl;
    }
```

4.5.7. Comparing strings

Strings can be compared using strcmp or strncmp functions

The function strcmp compares two strings.

strcmp(str1, str2);

strcmp returns: **< 0**   **if str1 is less than str2**
                                 = 0   **if str1 is equal to str2**
                                 > 0   **if str1 is greater than str2**

Example:

```
    #include <iostream.h>
    #include <string.h>
    void main() {
        cout << strcmp("abc", "def") << endl;
        cout << strcmp("def", "abc") << endl;
        cout << strcmp("abc", "abc") << endl;
        cout << strcmp("abc", "abcdef") << endl;
        cout << strcmp("abc", "ABC") << endl;
    }
```

The function **strncmp** is like **strcmp** except that it compares only a specified number of characters.

strncmp(str1, str2, int n);

strncmp does not compare characters after a terminating null character has been found in one of the strings.

Example:

```
    #include <iostream.h>
    #include <string.h>
    void main()
     {
     cout << strncmp("abc", "def", 2) << endl;
     cout << strncmp("abc", "abcdef", 3) << endl;
     cout << strncmp("abc", "abcdef", 2) << endl;
     cout << strncmp("abc", "abcdef", 5) << endl;
     cout << strncmp("abc", "abcdef", 20) << endl;
     }
```