Arrays

Arrays are data structures consisting of related data items of the same type.

After discussing how arrays are declared, created and initialized, we present a series of practical examples that demonstrate several common array manipulations.

We then explain how character strings (represented until now by string objects) can also be represented by character arrays.

We present an example of searching arrays to find particular elements.

The chapter also introduces one of the most important computing applications sorting data (i.e., putting the data in some particular order).

An array is a consecutive group of memory locations that all have the same type. To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.
Figure shows an integer array called `c`. This array contains 12 elements.

A program refers to any one of these elements by giving the name of the array followed by the position number of the particular element in square brackets (`[]`). The position number is more formally called a subscript or index (this number specifies the number of elements from the beginning of the array).

The first element in every array has subscript 0 (zero) and is sometimes called the zero-th element. Thus, the elements of array `c` are `c[0]` (pronounced "c sub zero"), `c[1]`, `c[2]` and so on. The highest subscript in array `c` is 11, which is 1 less than 12 the number of elements in the array.

Array names follow the same conventions as other variable names, i.e., they must be identifiers.

A subscript must be an integer or integer expression (using any integral type). If a program uses an expression as a subscript, then the program evaluates the expression to determine the subscript. For example, if we assume that variable `a` is equal to 5 and that variable `b` is equal to 6, then the statement

\[ c[a+b] += 2; \]


To divide the value of `c[6]` by 2 and assign the result to the variable `x`, we would write

\[ x = c[6] / 2; \]
Declaring Arrays

Arrays occupy space in memory. The programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

and the compiler reserves the appropriate amount of memory.

The `arraySize` must be an integer constant greater than zero.

For example, to tell the compiler to reserve 12 elements for integer array `c`, use the declaration

```
int c[ 12 ]; // c is an array of 12 integers
```

Memory can be reserved for several arrays with a single declaration. The following declaration reserves 100 elements for the integer array `b` and 27 elements for the integer array `x`.

```
int b[ 100 ], // b is an array of 100 integers
    x[ 27 ]; // x is an array of 27 integers
```

Arrays can be declared to contain values of any non-reference data type. For example, an array of type `char` can be used to store a character string.
Initializing an Array in a Declaration with an Initializer List

The elements of an array also can be initialized in the array declaration by following the array name with an equals sign and a comma-separated list (enclosed in braces) of initializers.

```c
int n[10] = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};
```

If there are fewer initializers than elements in the array, the remaining array elements are initialized to zero. For example,

```c
int n[10] = {0};
```

If the array size is omitted from a declaration with an initializer list, the compiler determines the number of elements in the array by counting the number of elements in the initializer list. For example,

```c
int n[] = {1, 2, 3, 4, 5};
```

creates a five-element array.

If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size. The array declaration

```c
int n[5] = {32, 27, 64, 18, 95, 14};
```

causes a compilation error, because there are six initializers and only five array elements.
Example

Following program calculate The Euclidian norm of vector (1,2)

```c
#include "stdio.h"
#include "conio.h"
#include "math.h"
int main() {
    float a[2]={1,2};
    float n;
    n=sqrt(a[0]*a[0]+a[1]*a[1]);
    printf("%f",n);
    getch();
    return 0;
}
```

Array a can be declerated as follows

```c
#include "stdio.h"
#include "conio.h"
#include "math.h"
int main() {
    float a[]={1,2};
    float n;
    n=sqrt(a[0]*a[0]+a[1]*a[1]);
    printf("%f",n);
    getch();
    return 0;
}
```
Or we can get elements of vector from user.

```c
#include "stdio.h"
#include "conio.h"
#include "math.h"
int main() {
    float a[2];
    float n;
    for (int i=0; i<2; i++) {
        printf("%d. component of vector ",i+1);
        scanf("%f", &a[i]);
        n+=a[i]*a[i];
    }
    n=sqrt(n);
    printf("%f", n);
    getch();
    return 0;
}
```

By changing 2, we get general Euclidian norm program.

---

Using Character Arrays to Store and Manipulate Strings

To this point, we have discussed only integer arrays. However, arrays may be of any type. We now introduce storing character strings in character arrays. A string such as "hello" is actually an array of characters.

A character array can be initialized using a string literal. For example, the declaration

```c
char string1[] = "first";
```

initializes the elements of array string1 to the individual characters in the string literal "first".

The size of array string1 in the preceding declaration is determined by the compiler based on the length of the string.
It is important to note that the string "first" contains five characters plus a special string-termination character called the null character.

Thus, array string1 actually contains six elements. The character constant representation of the null character is '\0' (backslash followed by zero).

All strings represented by character arrays end with this character.

A character array representing a string should always be declared large enough to hold the number of characters in the string and the terminating null character.

Character arrays also can be initialized with individual character constants in an initializer list. The preceding declaration is equivalent to the more tedious form

```
char string1[ ] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Note the use of single quotes to delineate each character constant. Also, note that we explicitly provided the terminating null character as the last initializer value. Without it, this array would simply represent an array of characters, not a string.
Because a string is an array of characters, we can access individual characters in a string directly with array subscript notation. For example,

\[
\text{string1[0]} \text{ is the character 'f',}
\]
\[
\text{string1[3]} \text{ is the character 's' and string1[5] is the null character.}
\]

We also can input a string directly into a character array from the keyboard using scanf. For example, the declaration

```c
char string2[20];
```

creates a character array capable of storing a string of 19 characters and a terminating null character. The statement

```c
scanf("%s",string2);
```

reads a string from the keyboard into string2 and appends the null character to the end of the string input by the user.

Note that the preceding statement provides only the name of the array and no information about the size of the array.

It is the programmer's responsibility to ensure that the array into which the string is read is capable of holding any string the user types at the keyboard.

By default, scanf reads characters from the keyboard until the first white-space character is encountered regardless of the array size.
The `gets` statement collects a string of characters terminated by a newline from the keyboard. The characters are cached in a read buffer and put into a variable.

Syntax: `gets(variable);`

```c
#include "stdio.h"
#include "conio.h"
#include "math.h"

int main() {
    char ad[10];

    printf("what is your name ");
    gets(ad);

    printf("\nHello %s", ad);
    getch();
    return 0;
}
```