

Chapter 25 – Introduction to Arrays

1. What is an Array?

Whether stored in memory or on some other media, data in a computing system must occupy a space with an address. Without addresses, data would be lost.

Text files, such as used in earlier chapters, are character files. To use a file, a program must be able to find the starting location of that file. The address of each character in the file is computed in turn by the adding its relative distance to the beginning address of the file. Fortunately, operating systems supply the location information to a running program when requested and the programmer generally does not have to know the physical location of a file in order to use the file. Thanks to the operating system, the programmer just has to know the name of a file. Also fortunately, programs written in modern programming languages keep track of the address of the next character to read without additional work by the programmer!

In mathematics, an array is a subscripted variable that can represent multiple values. Each subscript is related to a different value assigned to that variable. For example:

$$A_0, A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9$$

The mathematical definition of an array fits both computer files and computer arrays. Each location capable of storing data has its own address. However, by convention, a collection of data stored on permanent media, such as tape or disk, is usually referred to as a file. Also by convention, a collection of data stored on more volatile media such as computer memory is usually referred to as an array.

In C++, both *null terminated strings* and *string class objects* can be treated as arrays of characters (type char).

In most computer languages, arrays are the most frequently used type of data structure and the basic algorithms that allow access to arrays are most fundamental to success as a programmer.

2. Single Dimension Static Arrays

The simplest form of an array is one of fixed length that is made up of *elements* (sometimes called *nodes*) of an atomic (simple or single value) type or class. For example, think of an array of integers that is 10 elements long. This is called a *single dimension array* because it has one *index*. It is also called a *static* array because it does not grow or shrink. In this integer array, each element acts as a potential storage place for an integer variable.

Array objects can be declared using the following syntax:

```
type-or-class array-name[size];
```

where *size* is the number of elements in the array.

Here is an example of declaring a 10-element array of integers called *a*:

```
int a[10];
```

Each *element* in an array has its own unique *subscript*, called an *index*, and functions as a variable of the *type* or *class* of the array. In C and C++, *indices* start from 0 and go to one less than the size of the array. To use an *element* of an array, the array name is written followed by the *index* of the desired *element* enclosed in square brackets.

array-name[index]

The 10 *elements* of the example array *a* can be accessed by writing the array name followed by the *index*. In array *a*, the lowest index is 0 and the highest is 9. Here are several examples of using elements of array *a* in C++. Note that integer expressions can be used for the index.

```
cin >> a[5];  
  
a[3] = z + 5;  
  
a[0]++;  
  
a[i] = a[z] * 4 / c;  
  
cout << a[x-1];
```

The elements of a static array can be initialized when the array is created. This has the form:

type-or-class array-name[] = {value, value, ... , value};

For example, the array *pay* could be declared and initialized in either of the following ways:

```
double pay[5];  
pay[0] = 5.15;  
pay[1] = 5.25;  
pay[2] = 5.50;  
pay[3] = 5.90;  
pay[4] = 6.20;
```

or

```
double pay[ ] = {5.15, 5.25, 5.50, 5.90, 6.20};
```

Array elements can be accessed in any order. Because of this, arrays are referred to as *random access data structures*. Here is an example of randomly accessing array elements to lookup the correct rate of pay for an employee. Notice that the program contains code to protect itself from using an index greater or less than the actual range of indices in the array.

```
#include <iostream>
using namespace std;

void main() {
    int const max = 5;
    double pay[ ] = {5.15, 5.25, 5.50, 5.90, 6.20};
    double hours;
    int emptytype;

    cout << "Enter the hours worked: ";
    cin >> hours;
    cout << "Enter employee pay rate type (0-" << max-1 << "): ";
    cin >> emptytype;

    if (emptytype >= max || emptytype < 0) cout << "Not known type!";
    else {
        if (hours <= 40) cout << "Weeks pay is $" << hours * pay[emptytype];
        else cout << "Weeks pay is $"
            << hours * pay[emptytype] + (hours - 40) * pay[emptytype] * 0.5;
    }
    cout << endl;
}
```

In the previous example program, if the user enters an *hours* worked amount of 20 and an employee *type* of 3, the run will look like this:

```
Enter the hours worked: 21
Enter the employee pay rate type (0-4): 3
Weeks pay is $123.9
```

Exercises 25.1

1. What is an array in C++?
2. What is an element of an array?
3. What is the purpose of an index?
4. What is the starting index of any array in C++?

5. What is the ending index of a C++ array of size 45?
6. Declare an array of 53 float elements.
7. Given the following array declaration

```
long z[8] = {1, 2, 4, 5, 7, 8, 10, 13};
```

What is the result of evaluating the following expression?

```
z[0] + z[3] + z[4] + z[7]
```

Programming Exercise 25.1

Create the previous example array program. Compile, debug, and link it, then run it several times to test it. Test each employee type and use types that are out of range.

Programming Exercise 25.2

In the country of Lotataxes, income tax rates are determined by comparing income amounts against a table of tax rates. This table is as follows:

less than one floriden	- no taxes
1 floriden	- 5%
2 floriden	- 7%
3 floriden	- 12%
4 floriden	- 19%
5 floriden	- 23%
6 floriden	- 31%
7 floriden	- 37%
8 floriden	- 43%
9 floriden	- 51%
more than 9 floriden	- 66%

Create a program that accepts amount of income and returns the tax rate and the amount of taxes due. Taxes must be looked up via a table in an array.

Programming Exercise 25.3

In the country of Lotataxes, inflation has run rampant and the tax table has had to be adjusted. It is now:

less than 1,000 floridens	- no taxes
up to 2,000 floridens	- 9%
up to 3,000 floridens	- 17%
up to 4,000 floridens	- 25%
up to 5,000 floridens	- 32%
up to 6,000 floridens	- 41%
up to 7,000 floridens	- 52%
up to 8,000 floridens	- 59%
up to 9,000 floridens	- 63%
up to 10,000 floridens	- 70%
more than 10,000 floridens	- 80%

Revise the program of 25.2 so that the original tax table array can still be used, but with the new values.

3. Traversing an Array

One of the most fundamental algorithms used on an array is *traversing* an array. Traversing an array is the process of accessing (often called visiting) all of the elements (which are often called nodes) of a data structure in a predetermined order without accessing (or visiting) any element (or node) more than once. (Traversing is usually the first algorithm used when learning any data structure.)

Traversing an array can be accomplished in a simple manner with the aide of a *for-loop* or counter in a *while-loop* to generate all of the indices in the array in a sequential order. Here is a fragment example of using the idea of traversal to assign and output all of the values in all of the elements of an array. The output will be in the reverse order the values were inserted into the array. Values will be integers from between 0 and 32,767.

```
#include <time.h>
#include <stdlib.h>
:
:
srand( (unsigned)time(NULL) );
for (int y=0; y<max; y++)
    a[y] = rand();
for (int x=max-1; x>0; x--)
    cout << "a[" << x << "] has the value " << a[x] << endl;
:
:
```

Programming Exercise 25.4

Write a program with an array of 100 integer elements. Assign the value of each element with the `rand()` function. Use traversal to find the following information about the array:

- a) the sum of all the elements
- b) the average of all the elements
- c) the sum of the elements with odd indices
- d) the sum of the elements with even indices
- e) the average of the first 10 numbers, the second 10 numbers, etc.

Programming Exercise 25.5

Single dimension arrays can be used as pass by reference parameters in a function header. The two possible forms of this are:

return-type function-name(parameter-list)

where if the *parameter-list* contains an array, it will have one of these forms:

*class *array-name*
class array-name[]
class array-name[array-size]

Alter the program of 25.4 so that each operation of the program is performed in a function.

Programming Exercise 25.6

Finding the smallest and largest elements in an array require the array to be traversed, checking the value of each element against the currently known minimum or maximum. If the value of the element being checked is less than the current minimum, the value of the element is copied into the variable holding the current minimum. If the value of the element being checked is less than the current maximum, the value of the element is copied into the variable holding the current maximum. (*Note: To make this work, the minimum variable must be assigned an impossibly large number before the loop starts and the maximum variable must be assigned an impossibly small variable before the loop starts.*)

Write a program with an array of 32,767 integer elements. Assign the value of each element with the `rand()` function. Find the minimum and maximum value stored in the array.

Programming Exercise 25.7

Reading from a file of unknown length and storing the data into an array requires a program to be able to respond to several additional possibilities. If the number of items in the file is greater than the size of the array, the program must stop reading data before the end-of-file condition. If the number of items in the file is less than the size of the array, the program must not use the entire array in any of the operations on arrays that we have used so far. In either event, the program must remember and respond to the actual number of data items in the array rather than just blindly using actual number of elements in the array.

- a) Create a program that uses a for loop and rand() to output random numbers to a file. Run the program twice to create two files. One file should contain 35,000 elements, the other should have its number of elements determined by using the rand() function as the upper limit of the for loop.
- b) Alter the program of 25.6 so that it loads its data from a file of unknown length. Run the program twice, once reading from the file of 35,000 elements and once with the file of an unknown number of elements.

Programming Exercise 25.8

Programming exercise 20.15 used a *for* statement and a switch or if statements to count the number of times two die rolled the integers 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12. By using the elements of an array of integers as counters, the program can be greatly simplified. Initializing the counters becomes two statements rather than 11. Choosing the right counter no longer requires ifs or a switch and can be reduced to one statement. Output of the data requires only two statements.

Modify 20.15 so that the elements of an integer array are used as the counters.

4. Dynamic Arrays

Static arrays are handy data structures for many applications. However, the number of elements an array will need is not always known in advance. If the maximum number of elements is known and that maximum will not create problems for the program or other programs running on the system, then an array large enough for all situations can be created. Of course, a programmer must still manage the list of data is being stored in the array, i.e. the programmer must include a variable indicating the current number of elements being used and continually check it against the maximum number of elements possible.

Arrays do not have to be static in size. They can be *dynamic*. That is, they may grow and shrink in size in order to accommodate a changing length list of values. The programmer must still keep track of the current number of elements being used and the maximum possible. The difference occurs when all the elements in the array contain values. Instead of simply not adding new values to the array, a dynamic array can expand to accommodate the new data.

To make an array dynamic, the declaration of the array must change. First, rather than declaring an array of a given type or class, a type of variable called a *pointer* to a type or class must be created instead. *Pointers* are variables that do not contain data. Instead they contain the address in computer memory of the data or, as in the case of an array, the first item of data in a list of data. Declaring a pointer to a type or class of data for the creation of a dynamic array has the following form:

```
class * array-name;
```

where * in the declaration determines that the array named in the declaration will be dynamic, i.e. *array-name* is a pointer to an array

Since the array named in the declaration is just a pointer to an array, the array itself must be created separately. This is done with the *new* function. The *new* function asks the operating system to assign memory for the array and returns the beginning memory address. To declare a dynamic single dimension array has the form:

```
array-name = new type-or-class[size];
```

Here is an example of declaring a dynamic array:

```
int max=10;  
int *a;  
a = new int[max];
```

On the second line of the example, *a* is declared as a pointer to an integer data memory location. On the third line, *a* is assigned the value of the starting address of an integer array 10 elements long.

If the program uses the integer variable *len* to keep track of the number of elements of the array *a* that have been used to store data from a list of unknown length, the time to expand the array can be determined. When the value in *len* matches the value in *max*, it is time to expand the array. This is done by:

- a) creating a second, larger array
- b) copying the contents of the first array to the second array
- c) adjusting the size of *max* to that of the new array
- d) deleting the first array
- e) assigning the variable *a* to point to the new array

Deleting a dynamic array is done with the *delete* statement. This has the form:

```
delete [ ] array-name;
```

where the memory pointed to by the array named is returned to the operating system for future use

Here is an example of expanding the dynamic array *a* by 5 elements:

```
int * temp;  
temp = new int[max+5];  
for (int x=0; x<max; x++) temp[x] = a[x];  
max += 5;  
delete [ ] a;  
a = temp;
```

Exercises 25.2

1. What is a pointer?
2. What is a dynamic array?
3. Why does expanding a dynamic array require a second array?
4. What is the purpose of *new*?
5. What is the purpose of *delete*?

Programming Exercise 25.9

Alter the programming assignment of 25.7 so that it uses a dynamic array starting at 100 elements. The array should expand by 100 elements each time it is necessary.

Programming Exercise 25.10

Using the programming assignment of 25.9, change the program so that the last half of the items read in are discarded from the array and the array size is reduced to the new length. Do this before any of the data is processed but after the array has been loaded.