Data Structures: Arrays

Most data types, the built-in types (int, float, bool, etc) use a single variable to store a value.

Many programming problems require a group of the same types of information and it doesn't seem very efficient to make a different variable to hold each item.

For example, if you write a program to process 100 exam scores it would be easier to store all of the information in one place rather than generate 100 variable names to hold each of the test scores.

The Array Data Type

An **array** is a collection of data items associated with a particular name.

For example the 100 exam scores for a class of students could be associated with an array named *scores* rather than 100 different variables.

We can reference each individual item, called an **array element** in the array.

We designate individual elements by using the array name and the element's position, starting with 0 for the first array element.

If we use the *scores* example the first score would be referred to as scores[0], the second as scores[1], etc.

Array Declaration

Arrays in C++ are specified using **array declarations** that specify the type, name, and size of the array:

```
float x[8];
```

C++ associates eight memory cells with the name x. Each element of the array x may contain a single floating-point value.

Example

Consider the array with the elements given below:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

Then consider the following statements and what they do:

Statement	**Explanation**
`cout << x[0];`	Displays the value of x[0], or 16.0.
`x[3] = 25.0;`	Stores the value of 25.0 in x[3].
`sum = x[0] + x[1];`	Stores the sum of x[0] and x[1], or 28.0 in the variable sum.
`sum += x[2];`	Adds x[2] to sum. The new sum is 34.0.
`x[3] += 1.0;`	Adds 1.0 to x[3]. The new x[3] is 26.0
`x[2] = x[0] + x[1};`	Stores the sum of x[0] and x[1] in x[2]. The new x[2] is 28.0

After all these commands are executed the new array would look like:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	28.0	26.0	2.5	12.0	14.0	-54.5

Parallel Arrays

We can use several arrays as a way of organizing information about a particular situation without declaring an unwieldy number of variables.

If you were writing a program to track information about employees you

might declare the following arrays:

```
const int NUM_EMP = 10;        // number of employees
bool onVacation[NUM_EMP];
int vacationDays[NUM_EMP];
string dayOff[NUM_EMP];
```

If we consider that each one of these arrays follow the same sizing and each element in the array corresponds to the same employee (the i-th employee) these three arrays are called **parallel arrays**. It is a good way to organize multiple pieces of information about a particular subject.

onVacation	
[0]	true
[1]	false
[2]	true
[3]	false
[4]	true
[5]	false
[6]	true
[7]	false
[8]	true
[9]	false

vacationDays	
[0]	10
[1]	12
[2]	3
[3]	8
[4]	15
[5]	5
[6]	6
[7]	9
[8]	10
[9]	15

dayOff	
[0]	Monday
[1]	Wednesday
[2]	Tuesday
[3]	Friday
[4]	Friday
[5]	Monday
[6]	Thursday
[7]	Wednesday
[8]	Tuesday
[9]	Thursday

Array Initialization

The statements below declare and initialize three arrays. The list of initial values for each array is enclosed in braces and follows the assignment operator =. The first array element (subscript 0) stores the first value in each list, etc.

```
const int SIZE = 7;
int scores[SIZE] = {100, 73, 88, 84, 40, 97};
char grades[] = {'A', 'C', 'B', 'B', 'F', 'A'};
char myName[SIZE] = {'F', 'R', 'A', 'N', 'K'};
```

The length of the list of initial values can't exceed the size of the array.

If the list contains fewer elements than the size allows, the value of the

element is not initialized.

If the size of the array is not declared, indicated by the empty brackets [], the compiler sets the array size to the number of elements in the initial value list.

Sequential Access to Array Elements

One of the benefits of storing information in an array is that the information can be accessed quickly by using a loop to access the different elements of the array rapidly.

To enter data into an array, print its contents, or perform other sequential tasks, use a for loop whose loop control variable (i) is also the array subscript (x[i]).

Increase the value of the loop-control by 1 causes the next array element to proceed.

Example

The array **cube** declared below stores the cubes of the first 10 integers (for example `cube[1]` is 1, `cube[9]` is 729.

```
int cube[10];

for (int i = 0; i < 10; i++)
{
     cube[i] = i * i * i;
}
```