Arrays

In order to store a collection of data, C++ offers a variety of constructs. Arrays are used to store items of the same type in a sequential fashion. Arrays are divided into elements, and each element can be accessed using an index.

For example: int a[4]; declares an array of four elements, each one able to store

an integer value.

The index of the array starts at 0. Therefore the above array a has elements a[0], a[1] ... a[3].

Integers can be assigned to these elements as you can to regular variables:

a[0] = 25; will store 25 into the first element of the array.

One must be very careful not to try to store a value beyond the end of the array (i.e. a[4] = 12). Since array a only has 4 elements, a[4] would be a fifth element that does not exist. The storage used would not belong to the array at all, and might cause trouble to the rest of the program. This is a very common source of errors that are difficult to trace, since they usually do not become apparent immediately.

Arrays can be loaded one element at a time (usually by means of a loop) or can be initialized:

int b [] = { 3, 4, 56};

Array b is initialized to the values in braces, and C++ reserves three elements for it. In this case, the dimension of the array is not necessary, since C++ can infer its length by the number of values used to initialize it. Let's assume we want to fill array int c[20] ; with the twenty integers from 1 to 20. We could do this in a loop:

```
int i = 0; // used for the array

index int c [20]; while ( i < 20) {

c [i] = i + 1; i++;

}
```

The variable i would be incremented at the end of the loop. At any given time through the loop, let's say when we are loading the seventh element, i would be seven, and the line c[i] = i+1 would be c[7] = 7 +1, which would put the value 8 into the eighth element (remember the fist is 0). When i reaches 19, 20 is stores into c[19] (twentieth element) , i is incremented by the i++ operation, and the condition fails, ending the loop.

Of course you could write the same loop in a more compressed way by judiciously combining the assignment and the increment, but in this instance it might be a little more obscure.

A version using a for loop: int

c[20];

for (int i =0; i < 20; i++) c[i] =

i + 1;

Again, this could still be made shorter.

Arrays can be passed as arguments to functions like any other type of data. We will see how in the next section.

Arrays and functions

Arrays can be passed as arguments to functions by merely using their name. For example:

```
 void fill_with_ones(int a [], int

size)

{ for (int j = 0; j < size;

j++)      a [j] = 1;

}
```

Function fill_with_ones will accept any array of int as a parameter, and an int representing its size, and it will fill all of its elements with ones, a pretty useless exercise.

Function main could use fill_with_ones in the following way:

```
const int DIMENSION = 12 int z[DIMENSION];

fill_with_ones(z,DIMENSION);
```

As you can see, passing the array to the function only requires that you pass the name of the array. In addition, in this case, the size is required by the function definition, so that it knows when it has reached the end of the array.

The main difference between passing arrays as parameters and passing other variables lies in the fact hat arrays are not copied when passed to the function.

When passing variables of type integer, for example, as in the case of DIMENSION, a copy of the value is used to initialize the parameter in the called function. In our case, DIMENSION's value is copied into parameter *size* in the called function. Even if the called function changes the value of its parameter in the course of its execution, the value of DIMENSION would be unaltered, since *size* contained only a copy.

In the case of array parameters, a copy does not take place, since that would entail making a copy of all the elements of the array, a very expensive operation. In order to avoid the expense of creating a copy of a potentially huge set of data, only the address in memory of the original array is passed as a parameter (and not its contents).

That means that parameter "a" in our function becomes another name for the array *z*, it refers to the same position in memory. Any changes to any of the elements of array parameter *a* will result in a actual change of the original array *z*.

If you want to prevent this, you should declare the array parameter as *constant* in the called function: as in *void fill_with_ones (const int a [], int size)*. In this case, of course, our original array would not be filled with ones; in fact, because of the keyword *const*, assignments to elements of array *a* would not be allowed in the body of the function, and you would get a compiler error.

You can use the word *const* any time you want to make sure no changes are allowed. For example, we defined DIMENSION as an integer that cannot be changed. Any attempt at writing DIMENSION = 23 later in the program would generate a compiler error.

To summarize, arguments--actual paramters-- sent to a function are normally copied into the function parameters, and any change to the latter will not be transmitted to the former. The first exception to this mechanism occurs when arrays are used as arguments/parameters: in that case, the function parameter will become an alias for the array itself and, unless declared as *const*, any changes to it will propagate to the original, since they are really sharing the same memory.

Lastly, you might have noticed that no dimension was specified for the array parameter in fill_with_ones. That is not a mistake, it means that the function allows you to pass an array of integers to it, no matter what its size is. In fact, the dimension is never specified, which means that array of integers of any dimension can be used to call the function.

Two-dimensional arrays

Two-dimensional arrays can be thought of as arrays that, instead of containing values of type int, float, string, etc.) contain arrays.

When defining a two-dimensional array, one has to specify the size of the main array first followed by the size of the arrays that are the elements of the main array.

For example: *int*

table[3][4];

defines a two-dimensional array called *table* of size 3 (three elements) where each

one of them is an array of int of 4 elements. We can initialize such an array as

follows: int table [3][4] = { {1, 2, 3, 5}, {9, 8, 7, 3}, {10, 23, 17, 14} };

It can be seen that table is an array of 3 elements and that each one of them is in turn an array of 4 elements of type int.

In order to access the values, one has to specify first the index of the element of the main array where the array that contains the value is followed by the index of the value in that inner array.

For example: the value 9 corresponds to *table[1][0],* because 9 is in the second element (index 1) of the main array (where the second element is the array {9, 8, 7, 3}) followed by the index into that inner array where 9 is. The index into the inner array is 0, because 9 is the first element in that array.

The value 14 is found in the third element of the first array. But that third element is itself an array, and 14 is the fourth element of that inner array.

Therefore, 14 can be accessed as *table[2][3].*

Loops can be used to access in sequence the elements of a two-dimensional array.

For example, if we wanted to ad 1 to every element of the two-dimensional array *table* we could do it with two nested for loops:

```
 for (int i = 0;  i < 3; i++)
for (int j = 0 ;  j < 4; j++)
       table [i][j] = table [i][j] + 1; //or table[i][j]++
```

The first for loop goes through the elements of the outer array one at a time. When i is 0, we are processing the first element of the outer array, But that first element is itself an array of four elements. So, for i = 0, the inner for loop goes through all the elements (4) of the inner array which is the first component of the outer one.

In other words, *table[0]* is the first array {1, 2, 3, 5} that we traverse using the second loop using j as the index into that four element array.

Of course, two-dimensional arrays can contain arrays of other data types.

```
double example[2][3]  = { {2.3, 4.56, 6,2345} , {0.57, 4.234, 5.5} };
```

Array *example* is an array of two elements where each one of them is an array of three values of type double.

Another example: replace all even values in a two-dimensional array of 2 arrays of 3 integers each (dimensions 2 by 3) with 0.

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

void change( int [][3], int);//prototype


int main(){


int testing[2][3] = { {5, 7, 8}, {2, 4, 13}};
change(testing, 2);//change even numbers to 0

//display the new contents of the array

for (int i=0; i < 2 ; i++) {

        for (int j =0; j < 3; j++)
                cout << testing[i][j] << "  ";

cout << endl;//new line at the end of each inner array
  }
}
```