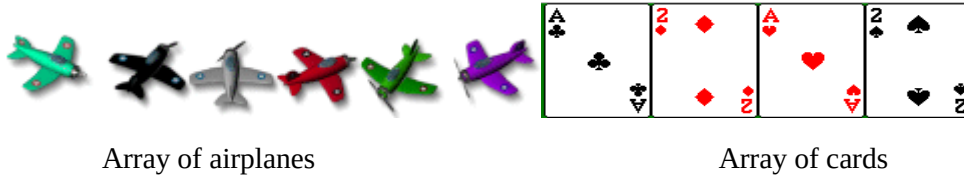


Introduction to Arrays

Arrays Fundamentals



An array is a group of items that can be identified as similar because they are of the same nature. Arrays come in two flavors: one dimensional and multi-dimensional arrays. Every one of the pictures above represents a single dimensional array.

Declaring an Array

Just like any variable you are already familiar with, an array has to be declared before being used. Yet the difference this time is that you need to tell the compiler what kind of array you are defining, an array of books? An array of students? An array of billiard balls? An arrays of clothes? This is because, once more, the compiler wants to know how much space your array is going to occupy in the computer memory. This is because when you declare an array of items, the compiler puts each one of the items in an appropriate location.

Like any other variable, the syntax of declaring an array is:

```
DataType ArrayName[dimension\order];
```

The array is first identified by its kind, which could be a char, an int, a float, etc.; followed by its name that follows the C++ naming rules. The name is then followed by square brackets that specify the dimension of the array or its size.

Here are examples of declaring arrays:

```
int age[12];
```

```
float grade[100];
```

```
double angle[360];
```

int Age[12]; declares a group or array of 12 values, each one being an integer.

`float Grade[100];` declares an array of 100 floating-point values.

`double Angle[360];` declares an array of double-precision numbers. There are 360 of these items in the group.

Initializing an Array

Just like any variable can be initialized, an array also can be initialized. To accomplish this, for a one-dimensional array, the syntax used is:

```
DataType ArrayName[dimension] = { element1, element2, ..., elementn};
```

Therefore, you can start with the data type to specify the kind of array you are declaring. This is followed by the array name, and the square brackets. After specifying the dimension or not, and after the closing square bracket, type the assignment operator. The elements, also called items that compose the array are included between an opening curly bracket '{' and a closing curly bracket '}'. Each item is separate from the next by a comma operator. As a normal C/C++ initialization, you end it with a semi-colon.

Here are examples of declaring an initializing arrays:

```
int number[12] = {18, 42, 25, 12, 34, 15, 63, 72, 92, 26, 26, 12};
```

```
double distance[5] = {44.14, 720.52, 96.08, 468.78, 6.28};
```

If you have decided to initialize the array while you are declaring it, you can omit the dimension. Therefore, these arrays can be declared as follows:

```
int number[] = {18, 42, 25, 12, 34, 15, 63, 72, 92, 26, 26, 12};
```

```
double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28};
```

Processing the Elements of an Array

After initializing an array, its elements are counted from left to right. Each element of the array, also called a member of the array, has a specific and constant position. The position of an item is also called its index. The first member of the array, the most left, has an index of 0. The second member of the array has an index of 1. Since each array has a number of items which can be specified as n , the last member of the array has an index of $n-1$.

Based on this system of indexing, to locate a member of an array, use its index in the group. Imagine you declare and initialize an array as follows:

```
double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28};
```

To locate the value of the 3rd member of the array, you would type distance[2]. In the same way, the 1st member of the array can be located with distance[0].

Once you can locate a member of the array, you can display its value using cout. Here is an example:

```
#include <iostream.h>

int main()
{
    double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28};
    cout << "2nd member = " << distance[1] << endl;
    cout << "5th member = " << distance[4] << endl;
    return 0; }
```

This would produce:

2nd member = 720.52

5th member = 6.28

Using this approach, each member of the array can have its value accessed. Here is an example:

```
#include <iostream.h>

int main()
{
    double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28};
    cout << "Distance 1: " << distance[0] << endl;
    cout << "Distance 2: " << distance[1] << endl;
    cout << "Distance 3: " << distance[2] << endl;
    cout << "Distance 4: " << distance[3] << endl;
    cout << "Distance 5: " << distance[4] << endl;
    return 0; }
```

This would produce:

Distance 1: 44.14

Distance 2: 720.52

Distance 3: 96.08

Distance 4: 468.78

Distance 5: 6.28

The Size of an Array

When declaring an array, we saw that you must specify the number of items that the array is made of. Here is an example:

```
float averagePrice[45];
```

Depending on how you want to deal with your array, you may sometimes need to increase or decrease its dimension. To do this, you would need to locate the declaration of the array and change its dimension. If the program is long and the array is declared in some unusual place, this could take some time. The alternative is to define a constant prior to declaring the array and use that constant to hold the dimension of the array.

Here is an example:

```
#include <iostream.h>

int main()
{
    const int numberOfItems = 5;
    double distance[numberOfItems] = {44.14, 720.52, 96.08, 468.78, 6.28};
    cout << "Distance 1: " << distance[0] << endl;
    cout << "Distance 2: " << distance[1] << endl;
    cout << "Distance 3: " << distance[2] << endl;
    cout << "Distance 4: " << distance[3] << endl;
    cout << "Distance 5: " << distance[4] << endl;
    return 0;
}
```

You can use such a constant in a for loop to scan the array and access each of its members.

Here is an example:

```
#include <iostream.h>

int main()
{
    const int numberOfItems = 5;
    double distance[numberOfItems] = {44.14, 720.52, 96.08, 468.78, 6.28};
    cout << "Members of the array\n";
    for(int i = 0; i < numberOfItems; ++i)
        cout << "Distance " << i + 1 << ": " << distance[i] << endl;
}
```

```
return 0; }
```

In both cases, this would produce:

Members of the array

Distance 1: 44.14

Distance 2: 720.52

Distance 3: 96.08

Distance 4: 468.78

Distance 5: 6.28

We knew the dimensions of the arrays we have used so far, because we could count the number of members of the array. Imagine you declare a large array, possibly made of 100 or 300 members, you wouldn't start counting the number of members. C/C++ provides the `sizeof` operator that can be used to get the dimension of an array. The syntax you would use is:

```
sizeof(ArrayName) / sizeof(DataType)
```

Imagine you declare an array as follows:

```
int number[] = {18, 42, 25, 12, 34, 15, 63, 72, 92, 26, 26, 12, 127, 4762, 823, 236, 84, 5};
```

Instead of counting the number of members of this array (it makes me dizzy when I try), you can use the `sizeof` operator as follows:

```
int NumberOfItemsOfTheArray = sizeof(Number)/sizeof(int);
```

One of the advantages of the `sizeof` operator used to get the number of members of the array is that it can be used on a for loop to scan an array, either to locate the members or to look for a value in the array. Here is an example of using this concept:

```
#include <iostream.h>

int main()
{
    double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28};

    // Using the sizeof operator to get the dimension of the array
    int index = sizeof(distance) / sizeof(double);

    cout << "Array members and their values\n";

    // Using a for loop to scan an array
    for(int i = 0; i < index; ++i)

        cout << "Distance : " << i + 1 << distance[i] << endl;
```

```
        return 0;
    }
```

This would produce:

Array members and their values

Distance : 144.14

Distance : 2720.52

Distance : 396.08

Distance : 4468.78

Distance : 56.28

Filling Up an Array

When you declare an array without initializing it, we have mentioned that the compiler reserves an amount of memory space for the members of the array. But that is only what the compiler does. Each part of such reserved space is filled with garbage. Therefore, you must make sure that you know the value held by a member of the array before making any attempt to process the value held by that member of the array. Consider the following example:

```
#include <iostream.h>

int main()
{
    const int numberOfItems = 5;
    double distance[numberOfItems];

    cout << "Distance 1: " << distance[0] << endl;
    cout << "Distance 2: " << distance[1] << endl;
    cout << "Distance 3: " << distance[2] << endl;
    cout << "Distance 4: " << distance[3] << endl;
    cout << "Distance 5: " << distance[4] << endl;

    return 0;
}
```

This would produce:

```
Distance 1: -9.25596e+061
Distance 2: -9.25596e+061
Distance 3: -9.25596e+061
Distance 4: -9.25596e+061
Distance 5: -9.25596e+061
```

As you can see, the members of the array in the beginning don't have any recognizable value. There are two solutions to this problem. You can either initialize the array or request the values of the members of the array from the user.

So far, when we used an array, we made sure to provide the exact number of members we needed for the array. We also saw that we could declare and initialize an array without specifying its dimension. The advantage of not specifying the dimension of the array is that we trust the compiler to find out the number of elements of the array. If you decide to specify the dimension of the array and initialize it, make sure you specify the elements less than or equal to the number you specified. Here is an example:

```
#include <iostream.h>
int main()
{
    const int numberOfItems = 5;
    double distance[numberOfItems] = {44.14, 720.52, 96.08};
    cout << "Distance 1: " << distance[0] << endl;
    cout << "Distance 2: " << distance[1] << endl;
    cout << "Distance 3: " << distance[2] << endl;
    cout << "Distance 4: " << distance[3] << endl;
    cout << "Distance 5: " << distance[4] << endl;
    return 0;
}
```

This would produce:

```
Distance 1: 44.14
Distance 2: 720.52
Distance 3: 96.08
```

Distance 4: 0

Distance 5: 0

If you provide more members than the number of elements you specified, the compiler would provide garbage values to the extra members. Here is an example:

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

```
int main()
```

```
{    const int NumberOfItems = 5;
    double distance[NumberOfItems] = {44.14, 720.52, 96.08, 468.78, 6.28};
    cout << "Distance 1: " << distance[0] << endl;
    cout << "Distance 2: " << distance[1] << endl;
    cout << "Distance 3: " << distance[2] << endl;
    cout << "Distance 4: " << distance[3] << endl;
    cout << "Distance 5: " << distance[4] << endl;
    cout << "Distance 6: " << distance[5] << endl;
    cout << "Distance 7: " << distance[6] << endl;
    cout << "Distance 8: " << distance[7] << endl;
    return 0;
}
```

This would produce:

Distance 1: 44.14

Distance 2: 720.52

Distance 3: 96.08

Distance 4: 468.78

Distance 5: 6.28

Distance 6: 2.64214e-308

Distance 7: 2.12414e-314

Distance 8: 1.00532e-307

Depending on the compiler you are using, you would also receive a (strong) warning.

Streaming Array Members

We have already seen how to define an array, how to locate the elements of an array, and how to display the values of these elements (displaying the value of a member of an array is one aspect of streaming). The arrays we have seen so far had their dimensions and their elements defined by the programmer. Many times you will have to get these elements from the user.

When you need to get an array from the user, first decide on what kind of array it is. Next, try to think of the maximum number of members you will need for the array. When you define an array and specify its dimension, the compiler will reserve the number of cells in memory that can accommodate your array. Here is an example:

```
int Page[5];
```

Each member of the array can be located using its index, as we have seen so far. In the same way, you can request the value of any member of the array using its index. In the following example, we declare an array of 5 integers and then we request the values of the 1st and the 4th members:

```
#include <iostream.h>
int main()
{
    const int counter = 5;
    int page[counter];
    cout << "Enter the number of pages of your books\n";
    cout << "Book 1: ";
    cin >> page[0];
    cout << "Book 4: ";
    cin >> page[3];
    cout << "\nSummary of books";
    cout << "\nBook 1: " << page[0] << " pages";
    cout << "\nBook 4: " << page[3] << " pages\n";
    return 0;
}
```

Here is an example of running the program:

Enter the number of pages of your books

Book 1: 842

Book 4: 1204

Summary of books

Book 1: 842 pages

Book 4: 1204 pages

Operations on Arrays

Each member of an array is a pseudo-variable and can be processed as such. This means that you can add the values of two members of the array($\text{Number}[2] + \text{Number}[0]$), you can subtract the value of one of the members from another member($\text{member}[1] - \text{Number}[4]$). In the same way, you can perform multiplication, division, or remainder operations on members of an array.

One of the regular operations performed on an array consists of adding the values of the members to produce a sum. Here is an example:

```
#include <iostream.h>

int main()
{
    const int max = 10; // We know that we need a constant number of elements
    int number[max];
    int sum = 0; // We will calculate their sum
    cout << "Please type 10 integers.\n";
    for( int i = 0; i < max; i++ )
    {
        cout << "Number " << i + 1 << ": ";
        cin >> number[i];
        sum += number[i];
    }
    cout << "\n\nThe sum of these numbers is " << sum << endl;
    return 0;
}
```

This would produce:

Please type 10 integers.

Number 1: 120

Number 2: 42

Number 3: 75

Number 4: 38

Number 5: 904

Number 6: 6

Number 7: 26

Number 8: 55

Number 9: 92

Number 10: 20

The sum of these numbers is 1378

Another type of operation regularly performed on an array consists of looking for a value held by one of its members. For example, you can try to know if one of the members holds a particular value you are looking for. Here is an example:

```
#include <iostream.h>
int main()
{
    int numbers[] = {8, 25, 36, 44, 52, 60, 75, 89}; // Declare the members of the array
    int find;
    int i, m = 8;
    cout << "Enter a number to search: ";
    cin >> find;
    for (i = 0; (i < m) && (Numbers[i] != Find); ++i)
        continue;
    if (i == m) // Find whether the number typed is a member of the array
        cout << find << " is not in the list" << endl;
    else
        cout << find << " is the " << i + 1
```

```
        << "th element in the list" << endl;

    return 0;
}
```

This would produce:

Enter a number to search: 44

44 is the 4th element in the list

One of the most regular operations performed consists of comparing the values of different members to get the lowest value of the members. Here is an example:

// Example of finding the minimum member of an array

```
#include <iostream.h>

int main()
{
    int numbers[] = {8, 25, 36, 44, 52, 60, 75, 89}; // The members of the array
    int minimum = numbers[0];

    int a = 8;

    // Compare the members
    for (int i = 1; i < a; ++i) {
        if (numbers[i] < minimum)
            minimum = numbers[i];
    }

    // Announce the result
    cout << "The lowest member value of the array is " << minimum << "." << endl;

    return 0;
}
```

This would produce:

The lowest member value of the array is 8.

You can use this same approach to get the maximum value of the members of an array. Here is an example:

// Example of finding the maximum member of an array

```
#include <iostream.h>

int main()
{
    // The members of the array
    int numbers[] = {8, 25, 36, 44, 52, 60, 75, 89};
    int maximum = numbers[0];

    int a = 8;

    // Compare the members
    for (int i = 1; i < a; ++i) {
        if (numbers[i] > maximum)
            maximum = numbers[i];
    }

    // Announce the result
    cout << "The highest member value of the array is "
        << maximum << "." << endl;

    return 0;
}
```

Arrays and Functions

An array can be passed to a function as argument. An array can also be returned by a function. To declare and define that a function takes an array as argument, declare the function as you would do for any regular function and, in its parentheses, specify that the argument is an array. Here is an example:

```
#include <iostream.h>

void DisplayTheArray(double member[5]);

int main()
{
    const int numberOfItems = 5;
    double distance[numberOfItems] = {44.14, 720.52, 96.08, 468.78, 6.28};
```

```

        return 0;
    }
    void DisplayTheArray(double member[5])
    {
        for(int i = 0; i < 5; ++i)
            cout << "\nDistance " << i + 1 << ": " << member[i];

        cout << endl;
    }

```

You don't have to specify the dimension of the array. This means that you can leave the square brackets empty:

```

#include <iostream.h>
void DisplayTheArray(double member[]);
int main()
{
    const int NumberOfItems = 5;
    double distance[NumberOfItems] = {44.14, 720.52, 96.08, 468.78, 6.28};
    return 0;
}
void DisplayTheArray(double member[])
{
    for(int i = 0; i < 5; ++i)
        cout << "\nDistance " << i + 1 << ": " << member[i];

    cout << endl;
}

```

We have already seen that when you declare an array, the compiler reserves an amount of memory space for the members of the array. To locate these members, the compiler aligns them in a consecutive manner. For example (hypothetically), if a member is located at 1804 Lockwood drive, the next member would be located at 1805 Lockwood Drive. This allows the compiler not only to know where the members of a particular array are stored, but also in what block (like the block houses of a city) the array starts. This means that, when you ask the compiler to locate a member of an array, the compiler starts where the array

starts and moves on subsequently until it finds the member you specified. If the compiler reaches the end of the block but doesn't find the member you specified, it stops, instead of looking for it all over the computer memory.

Based on this, when you call a function that has an array as argument, the compiler only needs the name of the array to process it. Therefore, the above function can be called as follows:

```
#include <iostream.h>

void DisplayTheArray(double member[])
{
    for(int i = 0; i < 5; ++i)
        cout << "\nDistance " << i + 1 << ": " << member[i];
    cout << endl;
}

int main()
{
    const int numberOfItems = 5;
    double distance[numberOfItems] = {44.14, 720.52, 96.08, 468.78, 6.28};
    cout << "Members of the array";
    DisplayTheArray(distance);
    return 0;
}
```

This would produce:

Members of the array

Distance 1: 44.14

Distance 2: 720.52

Distance 3: 96.08

Distance 4: 468.78

Distance 5: 6.28

The good scenario we have used so far is that we know the number of members of our array and we can directly use it in the function that is passed the argument. Imagine that we want the function to know how many members the array has and we want to let the function know while we are

calling it, after all, in some circumstances, we will not always know how many members we want the function to process. This should easily be done as follows:

```
#include <iostream.h>

void DisplayTheArray(double member[])
{
    for(int i = 0; i < 5; ++i)
        cout << "\nDistance " << i + 1 << ": " << member[i];
    cout << endl;
}

int main()
{
    const int NumberOfItems = 5;
    double distance[NumberOfItems] = {44.14, 720.52, 96.08, 468.78, 6.28};
    cout << "Members of the array";
    DisplayTheArray(distance[3]);
    return 0;
}
```

Unfortunately, this program will not compile. Remember, we saw that the compiler wants only the name of the array, because the name by itself represents the whole array. `distance[3]` is a specific value of a member of the array, it is not a group of values. In other words, `distance[3]` is the same as 468.78. It is as if we want to pass 468.78 as the value to be treated and not as a subsequent group of values, because that is what an array is. Therefore, the compiler complains that you are passing a value after you have specifically stated that the argument was a group of values and not a single value.

When you declare and define a function that takes an array as argument, if you plan to process the array, for example, if you want the calling function to control the number of elements to be processed, you should/must pass another argument that will allow the function to know how many members of the array would be considered. Such a function can be declared as follows:

```
#include <iostream.h>

void DisplayTheArray(double member[], int count);

int main()
```

```

{
    double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28, 68.04, 364.55, 6234.12};
    // Processing 5 members of the array
    cout << "Members of the array";
    DisplayTheArray(distance, 5);
    // Processing all members of the array
    int sizeOfArray = sizeof(distance)/sizeof(double);
    cout << "\nMembers of the array";
    DisplayTheArray(distance, sizeOfArray);
    return 0;
}

void DisplayTheArray(double member[], int counter)
{
    for(int i = 0; i < counter; ++i)
        cout << "\nDistance " << (i + 1) << ": " << member[i];
    cout << endl;
}

```

This would produce:

Members of the array

Distance 1: 44.14

Distance 2: 720.52

Distance 3: 96.08

Distance 4: 468.78

Distance 5: 6.28

Members of the array

Distance 1: 44.14

Distance 2: 720.52

Distance 3: 96.08

Distance 4: 468.78

Distance 5: 6.28
Distance 6: 68.04
Distance 7: 364.55
Distance 8: 6234.12

Using this same concept of passing accompanying arguments, you can control how the called function would process the array. For example, you can specify the starting and end point to the processing of the array. Here is an example:

```
#include <iostream.h>
// This function process an array but starts and ends at specific positions
void DisplayTheArray(double mbr[], int Start, int End);
int main()
{
    double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28, 68.04, 364.55, 6234.12};
    // Scan the array from the 3rd to the 7th member
    cout << "Members of the array";
    DisplayTheArray(distance, 2, 6);
    return 0;
}
void DisplayTheArray(double member[], int start, int ending)
{
    for(int i = start; i < ending; ++i)
        cout << "\nDistance " << i + 1 << ": " << member[i];
    cout << endl;
}
```

This would produce:

Members of the array
Distance 3: 96.08
Distance 4: 468.78
Distance 5: 6.28
Distance 6: 68.04

When declaring a function that takes an array argument, as we learned with other arguments, you don't have to provide a name to the argument. Simply typing the square brackets on the right side of the data type in the parentheses is enough. The name of the argument is only necessary when defining the function. Therefore, the above function can be declared as follows:

```
#include <iostream.h>

// This function process an array but starts and ends at specific positions
void DisplayTheArray(double[], int, int);

int main()
{
    double distance[] = {44.14, 720.52, 96.08, 468.78, 6.28, 68.04, 364.55, 6234.12};

    // Scan the array from the 3rd to the 7th member
    cout << "Members of the array";

    DisplayTheArray(distance, 2, 6);

    return 0;
}

void DisplayTheArray(double member[], int Start, int Ending)
{
    for(int i = Start; i < Ending; ++i)
        cout << "\nDistance " << i + 1 << ": " << member[i];

    cout << endl;
}
```

Two-Dimensional Arrays

A 2-dimensional array is an array of arrays. In other words, it is an array where each member of the array is also an array. Consider the following table

Declaring and Initializing a 2-Dimensional Array

This two-dimensional array is made of rows and columns. Each column represents one category of data that every one of the rows shares with the other rows. As different as each map looks, it still remains a map; each country on the table is known for its map, its flag, its area, and its

population, though remaining different from the others. To see another two-dimensional array, look at a calendar that displays a month with its week days.

Like the above table, a 2-dimensional array is made rows and columns. To declare it, use double pair of a opening and closing square brackets. Here is an example:

```
int numberOfStudentsPerClass[12][50];
```

This declaration creates a first group of 12 elements; it could be an array of 12 classes. Each element of the array contains 50 elements. In other words, each of the 12 members of the group is an array of 50 items. Simply stated, this declarations creates 12 classes and each class contains 50 students.

Before using the members of an arrays, you should/must make sure you know the values that its members hold. As done with one-dimensional arrays, there are two ways you can solve this problem: you can initialize the array or you can get its values by another means.

You can initialize an array the same way you would proceed a one-dimensional array: simply provide a list of values in the curly brackets. A multidimensional array is represented as an algebraic matrix as $M \times N$. This means that the array is made of M rows and N columns. For example, a 5×8 matrix is made of 5 rows and 8 columns. To know the actual number of members of a multidimensional array, you can multiply the number of rows by the number of columns. Therefore a 2×16 array contains $2 \times 16 = 32$ members.

Based on this, when initializing a 2-dimensional array, make sure you provide a number of values that is less than or equal to the number of members.

Here is an example:

```
double distance[2][4] = {44.14, 720.52, 96.08, 468.78, 6.28, 68.04, 364.55, 6234.12};
```

To locate a member of the array, this time, each must be identified by its double index. The first member is indexed at $[0][0]$. The second is at $[0][1]$. For a 2×4 array as this one, the 5th member is at $[1][0]$. You can use this same approach to display the values of the members of the array. Here is an example:

```

#include <iostream.h>

int main()
{
    // A 2-Dimensional array
    double distance[2][4] = {44.14, 720.52, 96.08, 468.78, 6.28, 68.04, 364.55, 6234.12};
    // Scan the array from the 3rd to the 7th member
    cout << "Members of the array";
    cout << "\nDistance [0][0]" << ": " << distance[0][0];
    cout << "\nDistance [0][1]" << ": " << distance[0][1];
    cout << "\nDistance [0][2]" << ": " << distance[0][2];
    cout << "\nDistance [0][3]" << ": " << distance[0][3];
    cout << "\nDistance [1][0]" << ": " << distance[1][0];
    cout << "\nDistance [1][1]" << ": " << distance[1][1];
    cout << "\nDistance [1][2]" << ": " << distance[1][2];
    cout << "\nDistance [1][3]" << ": " << distance[1][3];
    cout << endl;
    return 0;
}

```

This would produce:

Members of the array

Distance [0][0]: 44.14

Distance [0][1]: 720.52

Distance [0][2]: 96.08

Distance [0][3]: 468.78

Distance [1][0]: 6.28

Distance [1][1]: 68.04

Distance [1][2]: 364.55

Distance [1][3]: 6234.12

To make the above array a little easier to read when initializing it, you can type the values of each row on its own line. For example, the above array can be initialized as follows:

```
double distance[2][4] = { 44.14, 720.52, 96.08, 468.78,  
                          6.28, 68.04, 364.55, 6234.12 };
```

C++ also allows you to include each row in its own pair of curly brackets. You must separate each row from the next with a comma. Once again, this makes code easier to read. Here is an example:

```
double distance[2][4] = { { 44.14, 720.52, 96.08, 468.78 },  
                          { 6.28, 68.04, 364.55, 6234.12 } };
```

Processing a 2-Dimensional Array

To scan a 2-dimensional array, you should know how many columns the array contains. You can use two for loops to navigate the array. Here is an example:

```
#include <iostream>  
  
int main()  
{  
    // A 2-Dimensional array  
    double distance[][4] = {  
        { 44.14, 720.52, 96.08, 468.78 },  
        { 6.28, 68.04, 364.55, 6234.12 } };  
  
    // Scan the array from the 3rd to the 7th member  
    cout << "Members of the array";  
  
    for(int i = 0; i < 2; ++i)  
        for(int j = 0; j < 4; ++j)  
            cout << "\nDistance [" << i << "][" << j << "]: " << distance[i][j];  
  
    cout << endl;  
    return 0;  
}
```

This would produce the same result as previously.

Multidimensional Arrays

Multi-dimensional arrays are characterized by more than one line of representation. An array of an array known as multi-dimensional array. It includes two dimensional array and above.

Here are examples of a three-dimensional arrays.

Initialization of three dimensional array

```
int SW[2][3][4] = {3, 4, 2, 3, 0, -3, 9, 11, 23, 12, 23,  
2, 13, 4, 56, 3, 5, 9, 3, 5, 5, 1, 4, 9};
```

Better way to initialize this array with same elements as above.

```
int SW[2][3][4] = { { {3, 4, 2, 3}, {0, -3, 9, 11},  
{23, 12, 23, 2} }, { {13, 4, 56, 3},  
{5, 9, 3, 5}, {3, 1, 4, 9} } };
```

This array SW can hold a maximum of 24 elements. You can think this example as: Each of the 2 elements can hold 4 elements, which makes 8 elements and each of those 8 elements can hold 3 elements. Hence, total number of elements this array can hold is 24.

How to display the above three dimensional array elements? Write possible code fragment.