# Arrays

Unfortunately, C++ arrays lack many of the nice features of Java arrays:

- There is no length operation.
- There is no runtime check for an out-of-bounds index.
- It is not possible to assign from one array to another, and there is no arraycopy operation.
- It is not possible to change the size of an array dynamically.

In many cases it is better to use a vector class (either the one provided by the C++ standard template library -- more on this later -- or one you write yourself). A vector class can be defined to include Java-like features.

Here's an example C++ array declaration:

```
  int A[10];
```

This declares an array of 10 integers named A. Note that the brackets must follow the variable name (they cannot precede it as they can in Java). Note also that the array size must be part of the declaration (except for array parameters, more on this in a minute), and the size must be an integer expression that evaluates to a non-negative number.

Because in C++ an array declaration includes its size, there is no need to call new as is done in Java. Declaring an array causes storage to be allocated.

Multi-dimensional arrays are defined using one pair of brackets and one size for each dimension as in Java; for example, the following declares M to be a 10-by-20 array of ints:

```
  int M[10][20];
```

## Array Initialization

C++ arrays can be initialized using a brace notation in very much the same way as they are in Java:

| Java | C++ |
|---|---|
| `double[] x = new double[] { 1.1, 2.2, 3.3 };` | `double x[] = { 1.1, 2.2, 3.3 };` |

## Passing 1-D array to a function

As a concrete example, you know you can write a Java method to return the sum of the numbers in an array as follows:

```
double sum(double[ ] buf) // Java version
{
    double s = 0.0;
    for (int i=0 ; i<buf.length ; i++)
        s += buf[i];
    return s;
}
```

Moreover, any attempt to access an array location outside the bounds of the array will result in an immediate runtime exception being thrown.

Since the C++ runtime system *cannot* determine the length of an array:

1. There is no way to query the length inside `sum` as you can in Java (e.g., by writing `buf.length`).
2. All attempts to read or write array positions outside the bounds of the allocated array *will* be performed, even if the index is negative! Unless the index is *way* out of bounds (i.e., a *very* large negative or positive number), no exception will be thrown – or even detected – when the invald access is performed. Typically your program will *at best* produce unexpected results; more likely it will crash later when control has moved to some unrelated part of your code, thereby making it all the more difficult for you to determine *why* your program has crashed.

Therefore you must make sure that any code processing an array has a way of ensuring that it only uses valid indices. There are many conventions for that. One very common approach when passing arrays to methods is to pass not only the array itself, but also its length. For example, a typical way to translate the array summing code we saw above from Java to C++ would be as follows:

```
double sum(double buf[ ], int bufLength) // C++ version
{
    double s = 0.0;
    for (int i=0 ; i<bufLength ; i++)
        s += buf[i];
    return s;
}
```

Notes:

1. Obviously, the caller of such C++ functions and methods must pass not only the array, but its length as well.
2. There is a small but important syntactical difference: "`double[ ] buf`" (in Java) *versus* "`double buf[ ]`" (in C++). (We will see another way to declare C++ arrays on a later page in this web site.)
3. Since the C++ runtime cannot determine the length of arrays, there is no construct in C++ analogous to Java's "for each":

```
    double sum(double[ ] buf) // Another Java version using the "for
each" construct
    {
        double s = 0.0
        for (double v : buf) // "for each v in buf": supported in
Java, but not in C++
            s += v;
        return s;
       }
```

4. While conventional C++ arrays do not know their size, several array-like classes in the STL (standard template library) *do* know their size and *can* be used more like Java arrays. For example, `std::array`, `std::vector`, and `std::initializer_list` behave this way and can be used (since C++11) with this "for each" construct.

## Format for Declaring 2-Dimensional Arrays

As just explained, the subscripts for an array reference are enclosed in square brackets. Use these also when you want to declare the array.

e.g. General Format: *data_type variable_name [ row_size ][column_size]*

e.g. Declare an integer array of 2 rows, 3 columns: *int Nums[2][3];*

Often you will see a constant declared to hold the array size, and then this constant used in the array declaration.
i.e.
*const R_SIZE = 2;*
*const C_SIZE = 3;*
*int Nums[R_SIZE][C_SIZE];*

Remember that array elements can also be initialized when the array is declared. Examine the following code segment carefully to determine the precise syntax required.

```
int Nums[2][3] =
     {
     {1, 2, 3},
     {4, 5, 6}
     };
```

Format for Referencing 2-Dimensional Arrays

When you reference a particular array element, use a number, constant, variable or expression in the brackets.

- *Nums[0][1];*
- *Nums[A_SIZE][A_SIZE];*
- *Nums[i][j];*
- *Nums[i+1][j+3];*

Of course the values in the brackets must be within the defined limits of the array size. Refering to *Nums[788][0]* would result in an error if *Nums* was only declared as having 2 rows. The *0* is OK though, because the numbering starts at 0, not 1.

In most programs, 2-dimensional arrays go hand in hand with nested FOR loops because that is a quick easy way to reference all of the array elements in a 2-D array. For example, here's a little program segment that initializes a 3 x 5 integer array called Nums.

```
int Nums[3][5];
int i, j;

for ( i=0; i < 3; i++)      // march down the rows
{
     for ( j=0; j< 5; j++)   // march across the columns
     {
         Nums[i][j] = 0;     // set array element to zero
     }
}
```

This starts at the *zero'th* row, and accesses each column of that row from left to right.
i.e. *Nums[0][0] Nums[0][1] Nums[0][2] Nums[0][3] Nums[0][4]*

It then moves down to the next row and works accross the columns for that row.
i.e. *Nums[1][0] Nums[1][1] Nums[1][2] Nums[1][3] Nums[1][4]*

Here is a complete program summarizing everything discussed so far.

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

int main()
{
    int Nums[3][5];
    int i, j;

    for ( i=0; i < 3; i++)          // march down the rows
    {
        for ( j=0; j< 5; j++)       // march across the columns
        {
            Nums[i][j] = 0;        // set array element to zero
        }
    }
}
```

## Passing 2-D Arrays to Functions

Before we examine how to pass a 2-D array in detail here's a sample program that takes the initialization loop from the last program and turns it into a function:

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

const int R_SIZE=2;
const int C_SIZE=3;

void InitArray( int [][C_SIZE] );  // function prototype

int main()
{
   int Nums1[2][3];
   InitArray(Nums1);

}

// Function:  InitArray
// Purpose:   To initialize an array.
// Parameters: Base address of an array.
// Returns:   void
// -------------------------------------------------------
```

```
void InitArray(int p_Array[][C_SIZE])
{

   int i, j;

   for ( i=0; i < R_SIZE; i++)
   {
      for ( j=0; j< C_SIZE; j++)
      {
         p_Array[i][j] = 0;
      }
   }

} // end InitArray function
```

Now, let's take a closer look at how those arrays are passed. In C++, arrays are **not** passed by ***value*** to functions, they are passed by ***reference***. Because of this, you do **not** have to use the **&** reference character. You simply pass the ***base address*** of an array to a function. To do this, just supply the name of the array like this:

```
InitArray(Nums);
```

Notice that only the array name *Nums* appears in the parameter list; it is not followed by any subscripts at all.

Before we talk about declaring the function remember how we declared the array:

```
const R_SIZE=2
const C_SIZE=3
int Nums[R_SIZE][C_SIZE];
```

We used constants for the number of rows and columns.

You may have noticed in the review at the beginning of this section, that the ***column size*** is specified in both:

- the function prototype at the top of the program, and
  *void InitArray( int [][C_SIZE]);*
- the function declaration line.
  *void InitArray(int p_Array[][C_SIZE])*

This is because arrays are stored in memory in ***one row after another***. The function needs to now how many columns there are in a 2D array so it can find the next row. It's as if 2D arrays were stored as a big 1D array. In a 2D array cells referenced like this:

```
    int array[R_SIZE][C_SIZE];
    array[3][2];
```
are actually being referenced like this:

```
    int array[R_SIZE*C_SIZE];
    array[3*C_SIZE + 2];
```
but the compiler doesn't know what value to multiply by unless you supply it in the function definition.

It is also important to note that **arrays are not a legal return type**.