# Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

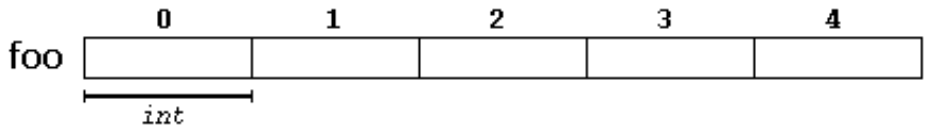
数组是放置在连续内存位置的一系列相同类型的元素，可以通过向唯一标识符添加索引来单独引用它们。

That means that, for example, five values of type `int` can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five `int` values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

这意味着，例如，5个int类型的值可以声明为数组，而不必声明5个不同的变量(每个变量都有自己的标识符)。相反，使用数组，这5个int值存储在连续的内存位置，并且可以使用相同的标识符和适当的索引访问这5个int值。

For example, an array containing 5 integer values of type `int` called `foo` could be represented as:

例如，一个包含5个int类型整型值的数组foo可以表示为:




where each blank panel represents an element of the array. In this case, these are values of type `int`. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

其中每个空白面板表示数组的一个元素。在本例中，这些是int类型的值。这些元素从0到4编号，0在第一个，4在最后一个;在c++中，数组的第一个元素总是用0(而不是1)编号，不管它的长度如何。

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

与常规变量一样，数组在使用前必须先声明。c++中典型的数组声明是:

`type name [elements];`
where `type` is a valid type (such as `int`, `float`...), `name` is a valid identifier and the `elements` field (which is always enclosed in square brackets `[]`), specifies the length of the array in terms of the number of elements.

其中type是一个有效的类型(如int, float...)，name是一个有效的标识符，elements字段(总是用方括号[]括起来)指定数组的元素数量的长度。

Therefore, the `foo` array, with five elements of type `int`, can be declared as:

因此，foo数组有五个int类型的元素，可以声明为:

```
int foo [5];
```

NOTE: The `elements` field within square brackets `[]`, representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

注意:方括号[]内的elements字段，表示数组中的元素数量，必须是常量表达式，因为数组是静态内存块，其大小必须在程序运行前的编译时确定。

## Initializing arrays

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

默认情况下，局部作用域的常规数组(例如，在函数中声明的数组)不初始化。这意味着它的所有元素都没有被设置为任何特定的值;它们的内容在声明数组时是不确定的。

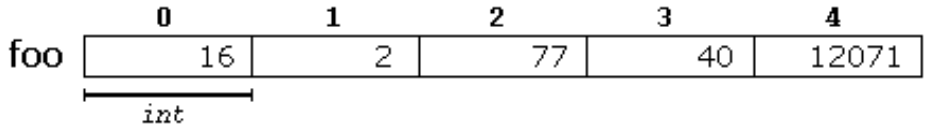
But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

但是数组中的元素可以在声明时显式地初始化为特定的值，方法是将这些初始值括在大括号{}中。例如:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

This statement declares an array that can be represented like this:

这个语句声明了一个可以这样表示的数组:

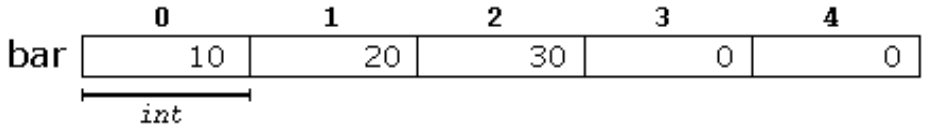



The number of values between braces `{}` shall not be greater than the number of elements in the array. For example, in the example above, `foo` was declared having 5 elements (as specified by the number enclosed in square brackets, `[]`), and the braces `{}` contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

大括号{}之间的值的数量不能大于数组中的元素数量。例如，在上面的例子中，foo声明有5个元素(由方括号中的数字[]指定)，大括号{}恰好包含5个值，每个元素对应一个值。如果用less声明，其余元素将被设置为它们的默认值(对于基本类型，这意味着它们用0填充)。例如:

```
int bar [5] = { 10, 20, 30 };
```

Will create an array like this:

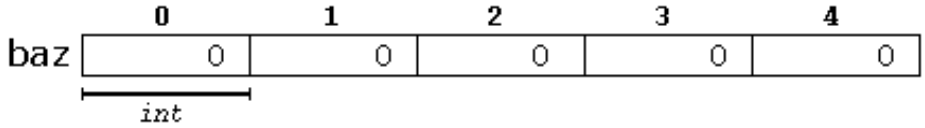



The initializer can even have no values, just the braces:

初始化式甚至可以没有值，只有花括号:

```
int baz [5] = { };
```

This creates an array of five `int` values, each initialized with a value of zero:




When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty `[]`. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces `{}`:

当为数组提供值的初始化时，c++允许保留方括号为空[]。在这种情况下，编译器会自动假定数组的大小与大括号{}中包含的值的数量相匹配:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array `foo` would be 5 `int` long, since we have provided 5 initialization values.

在这个声明之后，数组foo将是5个int long，因为我们提供了5个初始化值。

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

最后，c++的发展也导致了对数组的通用初始化。因此，在声明和初始化式之间不再需要等号。这两个表述是等价的:

```
int foo[] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };
```

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

静态数组，以及那些直接在命名空间(在任何函数之外)中声明的数组总是初始化的。如果没有明确指定初始化式，则所有元素都是默认初始化的(对于基本类型，使用0)。

## Accessing the values of an array

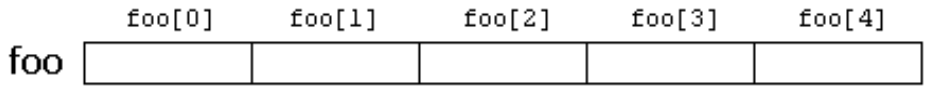
The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

数组中任何元素的值都可以被访问，就像访问相同类型的常规变量的值一样。的语法是:

`name[index]`

Following the previous examples in which `foo` had 5 elements and each of those elements was of type `int`, the name which can be used to refer to each element is the following:

在前面的例子中，foo有5个元素，每个元素的类型都是int，可以用来引用每个元素的名称如下:




For example, the following statement stores the value 75 in the third element of `foo`:

例如，下面的语句将值75存储在foo的第三个元素中

```
foo [2] = 75;
```

and, for example, the following copies the value of the third element of `foo` to a variable called `x`:

例如，下面的代码将foo的第三个元素的值复制到一个名为x的变量:

```
x = foo[2];
```

Therefore, the expression `foo[2]` is itself a variable of type `int`.

因此，表达式foo[2]本身就是一个int类型的变量。

Notice that the third element of `foo` is specified `foo[2]`, since the first one is `foo[0]`, the second one is `foo[1]`, and therefore, the third one is `foo[2]`. By this same reason, its last element is `foo[4]`. Therefore, if we write `foo[5]`, we would be accessing the sixth element of `foo`, and therefore actually exceeding the size of the array.

注意，foo的第三个元素被指定为foo[2]，因为第一个元素是foo[0]，第二个是foo[1]，因此，第三个元素是foo[2]。出于同样的原因，它的最后一个元素是foo[4]。因此，如果我们写foo[5]，我们将访问foo的第6个元素，因此实际上超过了数组的大小。

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced.

在c++中，超出数组的有效索引范围在语法上是正确的。这可能会产生问题，因为访问超出范围的元素不会在编译时导致错误，但会在运行时导致错误。允许这样做的原因将在后面的章节中介绍指针。

At this point, it is important to be able to clearly distinguish between the two uses that brackets `[]` have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets `[]` with arrays.

在这一点上，能够清楚区分括号[]与数组相关的两种用法是很重要的。它们执行两个不同的任务:一是在声明数组时指定数组的大小;第二个是在访问具体数组元素时为它们指定索引。不要将括号[]的这两种可能的用法与数组混淆。

```
int foo[5];         // declaration of a new array
foo[2] = 75;        // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

主要的区别在于，声明之前是元素的类型，而访问则不是。

Some other valid operations with arrays:

其他一些有效的数组操作:

```
foo[0] = a;
foo[a] = 75;
b = foo [a+2];
foo[foo[a]] = foo[2] + 5;
```

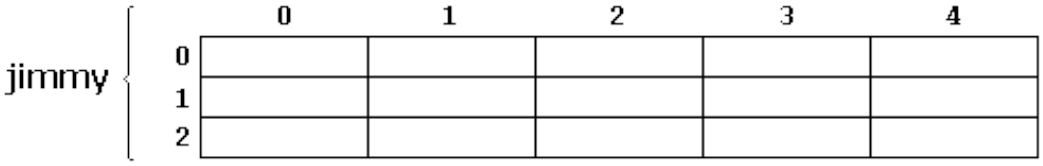
For example:

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

## Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

多维数组可以描述为“数组中的数组”。例如，可以将一个二维数组想象成一个由元素组成的二维表，所有元素都具有相同的统一数据类型。




`jimmy` represents a bidimensional array of 3 per 5 elements of type `int`. The C++ syntax for this is:

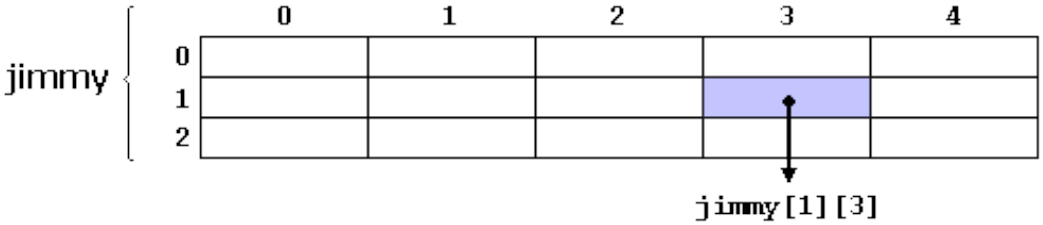
Jimmy表示一个二维数组，每5个元素中有3个int类型的元素。c++语法是:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

例如，在表达式中垂直引用第二个元素，水平引用第四个元素的方法是:

```
jimmy[1][3]
```




(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

多维数组不限于两个索引(即两个维度)。它们可以根据需要包含任意多的索引。但是要注意:数组所需的内存数量随着维度的增加呈指数增长。例如:

```
char century [100][365][24][60][60];
```

declares an array with an element of type `char` for each second in a century. This amounts to more than 3 billion `char`! So this declaration would consume more than 3 gigabytes of memory!

声明一个数组，该数组的元素类型为char，以表示一个世纪中的每秒钟。这相当于超过30亿char!因此，这个声明将消耗超过3g的内存!

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

最后，多维数组对程序员来说只是一种抽象，因为用一个简单的数组乘以它的索引也可以得到相同的结果:

```
int jimmy [3][5];   // is equivalent to
int jimmy [15];     // (3 * 5 = 15)
```

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bidimensional array while the other uses a simple array:

唯一的区别是，对于多维数组，编译器会自动记住每个想象维度的深度。下面两段代码产生了完全相同的结果，但其中一段使用了一个二维数组，而另一段使用了一个简单数组:

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
}
```

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

上面的两个代码片段都不会在屏幕上产生任何输出，但都以以下方式为名为jimmy的内存块赋值:

jimmy

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 4 | 6 | 8 | 10 |
| 2 | 3 | 6 | 9 | 12 | 15 |

Note that the code uses defined constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

请注意，代码使用已定义的常量作为宽度和高度，而不是直接使用它们的数值。这使代码具有更好的可读性，并允许在一个地方轻松地更改代码。

## Arrays as parameters

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

在某些情况下，我们可能需要将数组作为参数传递给函数。在c++中，不可能将数组表示的整个内存块直接作为参数传递给函数。但是可以传递的是它的地址。在实践中，这几乎有同样的效果，而且是一种更快、更有效的操作。

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

要接受数组作为函数的形参，可以将形参声明为数组类型，但使用空括号，忽略数组的实际大小。例如:

```
void procedure (int arg[])
```

This function accepts a parameter of type "array of `int`" called `arg`. In order to pass to this function an array declared as:

这个函数接受一个名为arg的类型为“array of int”的形参。为了传递给这个函数一个数组声明为:

```
int myarray [40];
```

it would be enough to write a call like this:

这样写一个调用就足够了:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

In the code above, the first parameter (`int arg[]`) accepts any array whose elements are of type `int`, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the array passed, without going out of range.

在上面的代码中，第一个参数(int arg[])接受任何元素为int类型的数组，无论其长度如何。因此，我们包含了第二个形参，它告诉函数作为第一个形参传递给它的每个数组的长度。这允许输出数组的for循环知道要在传递的数组中迭代的范围，而不会超出范围。

In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

在函数声明中，也可以包含多维数组。三维数组参数的格式为:

```
base_type[][depth][depth]
```

For example, a function with a multidimensional array as argument could be:

例如，以多维数组作为参数的函数可以是:

```
void procedure (int myarray[][3][4])
```

Notice that the first brackets `[]` are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

注意，第一个方括号[]为空，而下面的方括号为它们各自的维度指定大小。为了让编译器能够确定每个额外维度的深度，这是必要的。

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers. Although a clear understanding of pointers, explained in a coming chapter, helps a lot.

在某种程度上，传递数组作为参数总是会丢失一个维度。背后的原因是，由于历史原因，不能直接复制数组，因此真正传递的是指针。对于新手程序员来说，这是常见的错误来源。尽管对指针有清晰的理解(在下一章中解释)会有很大帮助。

## Library arrays

The arrays explained above are directly implemented as a language feature, inherited from the C language. They are a great feature, but by restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization.

上面解释的数组是直接作为语言特性实现的，继承自C语言。它们是一个很棒的特性，但是由于限制了它的复制，并且很容易退化为指针，它们可能会受到过度优化的影响。

To overcome some of these issues with language built-in arrays, C++ provides an alternative array type as a standard container. It is a type template (a class template, in fact) defined in header `<array>`.

为了克服语言内置数组的一些问题，c++提供了另一种数组类型作为标准容器。它是定义在头文件中的类型模板(实际上是类模板)。

Containers are a library feature that falls out of the scope of this tutorial, and thus the class will not be explained in detail here. Suffice it to say that they operate in a similar way to built-in arrays, except that they allow being copied (an actually expensive operation that copies the entire block of memory, and thus to use with care) and decay into pointers only when explicitly told to do so (by means of its member `data`).

容器是一个库特性，超出了本教程的范围，因此在这里不详细解释类。我只想说,他们以类似的方式操作内置数组,除非他们允许被复制(实际上一个昂贵的操作,复制整块的内存,因此小心使用)和衰变为指针只有当明确告诉这样做(通过其成员数据)。

Just as an example, these are two versions of the same example using the language built-in array described in this chapter, and the container in the library:

作为一个例子，这是同一个例子的两个版本，使用了本章描述的语言内置数组和标准库中的容器:

```
#include <iostream>

using namespace std;

int main()
{
  int myarray[3] = {10,20,30};
```

```
  for (int i=0; i<3; ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

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

int main()
{
  array<int,3> myarray {10,20,30};

  for (int i=0; i<myarray.size(); ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

As you can see, both kinds of arrays use the same syntax to access its elements: `myarray[i]`. Other than that, the main differences lay on the declaration of the array, and the inclusion of an additional header for the *library array*. Notice also how it is easy to access the size of the *library array*.

可以看到，这两种数组使用相同的语法来访问其元素:myarray[i]。除此之外，主要的区别在于数组的声明，以及库数组的附加头文件。还要注意访问库数组的大小是多么容易。