C++ Lecture 5
Arrays

CSci 588: Data Structures, Algorithms and Software Design

http://www.cplusplus.com/doc/tutorial/arrays/

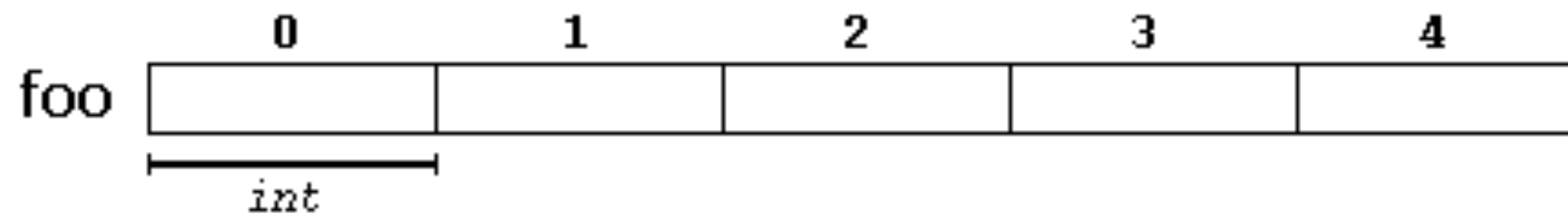
All material not from online sources copyright © Travis Desell, 2013

Overview

1. Initializing Arrays
2. Accessing Values in an Array
3. Multidimensional Arrays
4. Arrays as Parameters

Arrays: Initializing Arrays

Arrays

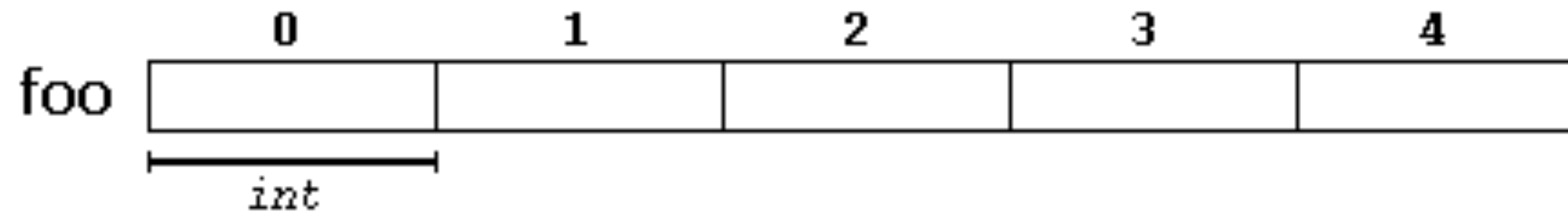


Arrays are very similar to vectors, except significantly more primitive (and therefore can be faster depending on how you're using them).

An array is a series of elements of the same type placed in contiguous memory locations.

These elements can be individual referenced by adding an index to a unique identifier.

Arrays

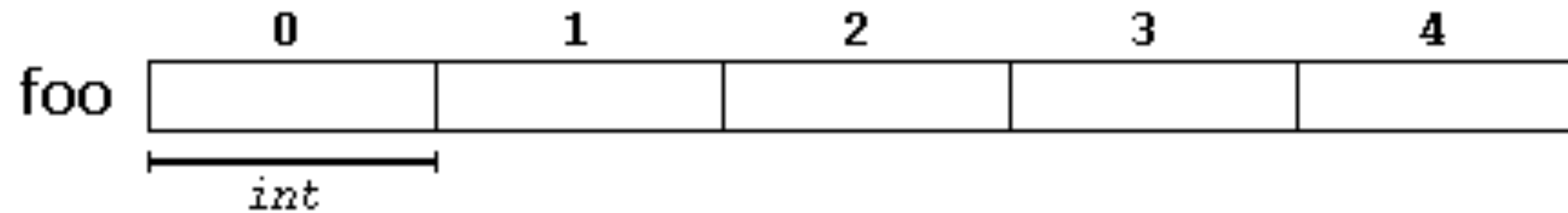



```
int foo [5];
```

For example, that means you can declare an array with 5 int elements without having to declare five different variables.

Instead, using an array the five int elements are stored in contiguous memory and accessed with the same identifier and an index (similar to a vector).

Arrays




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

This is represented above. Each blank panel represents an element in the array. These elements are numbered 0 to 4, with 0 the first and 4 the last. In C++ the indices of an always begin with 0 (not 1, as in other programming languages like fortran).

Arrays

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

Like other variables, an array must be declared before it is used. The typical declaration for an array is:

type name [number_elements];

The above code crates an array of 5 ints.

Arrays

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

Note, that the elements specifying how many elements are in an array must be a constant expression so the compiler can determine the size of the array at compile time.

If you want to have arrays whose size is determined at runtime, or dynamically-sized arrays, then you need to use pointers (more on that later).

Arrays: Accessing Values in an Array

Initializing Values in an Array

By default, regular arrays of *local scope* (i.e., those declared within a function) are left uninitialized.

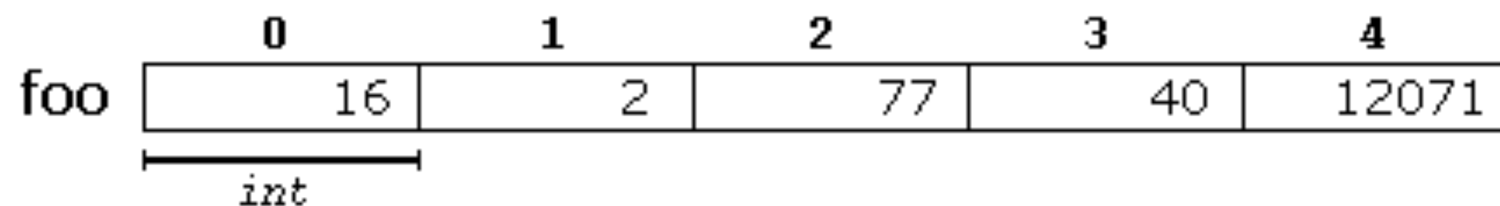
This means that none of their elements are set to a particular value; their contents are undetermined at the point the array is declared.

However, 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).

Initializing Values in an Array

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

However, the values within an array can be initialized to specific values using {}s (see above). This will declare an array that looks like this:



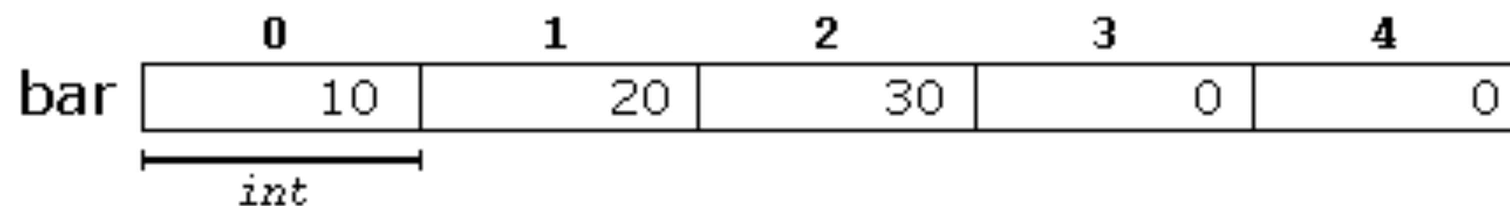
Initializing Values in an Array

The number of values within the {}s cannot be greater than the size of the array declared, however they can be declared with less.

If they're declared with less, the other values will be filled in with the default values for that particular type (which for primitive numbers is 0).

The following code will create the following array:

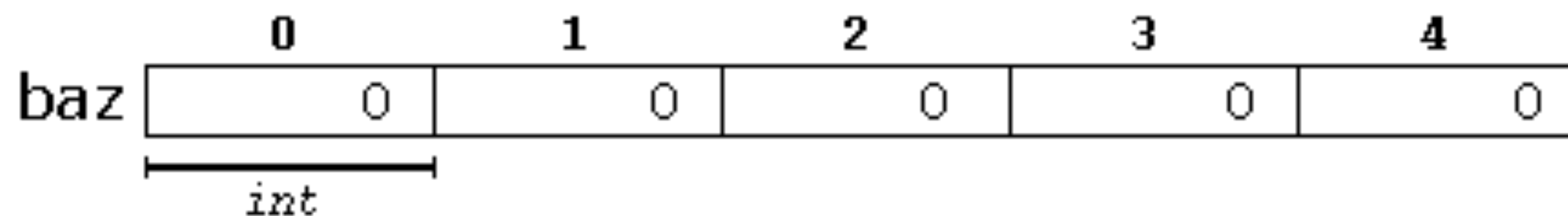
```
int bar [5] = { 10, 20, 30 };
```



Initializing Values in an Array

The initializer can also have no values, which will initialize the contents of the array all to the default value:

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



Initializing Values in an Array

C++ also allows for the possibility of letting the compiler figure out how many values are in the array dependent on what's within the {}s:

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

Initializing Values in an Array

C++ has since evolved to allow *universal initialization* for arrays. Therefore, it is also possible to drop the equal sign. The two following statements are identical:

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

Arrays: Multidimensional Arrays

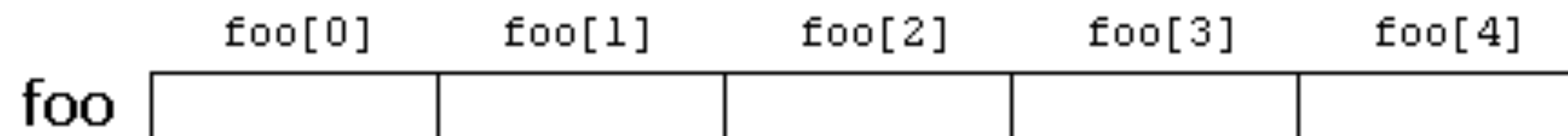
Accessing Values in an Array

Values in an array can be accessed just like the value of a regular variable of the same type, by using the [] operator:

name[index]

For example, we can create an array of 5 ints, and set the element at index 2 to 75 with the following code (note that the 2nd index is actually the 3rd element in the array):

```
int foo [5];
foo[2] = 75;
```



Accessing Values in an Array

It is possible to copy from the value at the 2nd index in the array with similar code:

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

Accessing Values in an Array

Again, the element at index 2 is actually the 3rd element in the foo array (which has 5 elements). This means the last element is at index 4. If we tried to access the 5th element:

foo[5]

Which exceeds the size of the array and can cause problems. Unlike using vectors and the *at* function, C++ does not check to ensure that you're accessing something you should be, and this can result in segmentation faults or even worse errors at runtime.

Accessing Values in an Array

You should be careful with the two different uses for the [] operator. Remember that they can be used *both* for declaring the size of an array, and for accessing the element in an array. Be careful not to confuse the two:

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

Accessing Values in an Array

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

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

There are also some other valid operations with arrays (you can use variables to calculate array indices):

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

Arrays: Multidimensional Arrays

Multidimensional Arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bi-dimensional array can be seen as a table made of elements with each having a uniform data type.

The C++ syntax for a 2-dimensional array is:

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

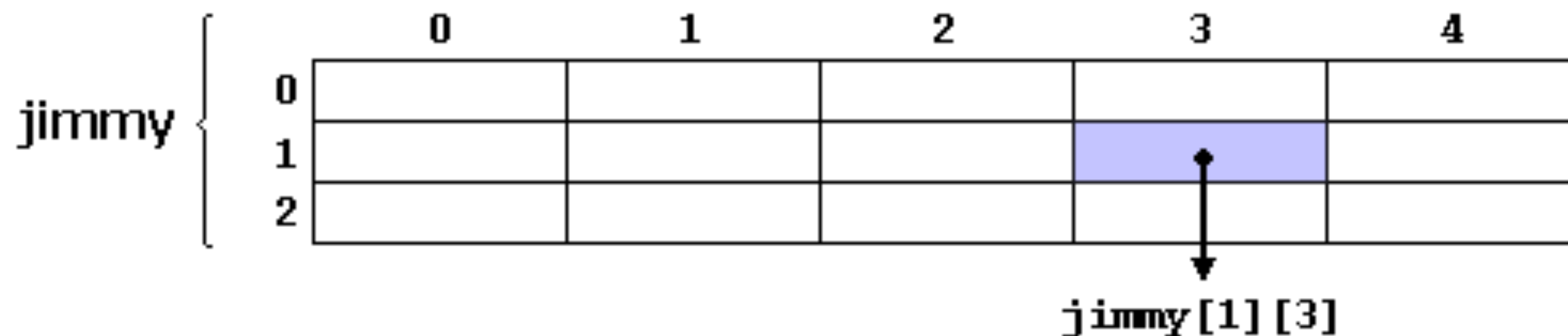
For more dimensions, simply add more []s. This can be seen as:

jimmy	0	1	2	3	4
0					
1					
2					

Multidimensional Arrays

For example, to access the second element vertically and fourth element horizontally in an expression would be (remember indices start at 0):

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



Multidimensional Arrays

Multidimensional arrays are not limited to two indices (two dimensions). They can contain as many dimensions as needed, but note that the amount of memory increases exponentially with each dimension:

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

The above array would take up 100 * 365 * 24 * 60 * 60 = 3,153,600,000 characters of memory (over 3GB memory).

Multidimensional Arrays

In general, multidimensional arrays are just an abstraction to make life easier for programmers, as the same results can be achieved with a single array:

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

With the difference being that the compiler automatically remembers the depth of each imaginary dimension.

Multidimensional Arrays

The following code produces the same result, with one using a bi-dimensional array and the other using a single dimensional array:

multidimensional array	pseudo-multidimensional array

`#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);`
	`    }`
	`}`

Multidimensional Arrays

multidimensional array	pseudo-multidimensional array

`#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);`
	`    }`
	`}`

These both produce the same array:

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

Arrays: Arrays as Parameters

Arrays as Parameters

At some point you'll need to pass an array as a parameter to a function. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument.

Instead, what is passed is the address of the array (i.e., a pointer). This is much faster than copying the entire array, and is similar to using the & for pass-by-reference.

Arrays are ALWAYS passed by reference, never by copy.

Arrays as Parameters

To accept an array as a parameter type to a function, the parameters can be declared as the array type but with empty brackets omitting the size of the array:

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

This function called 'procedure' will accept any array of ints. For example:

```
int myarray[40];
procedure (myarray);

int myotherarray[20];
procedure (myotherarray);
```

Both are acceptable.

Arrays as Parameters

For example, to print an array:

```
// 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);
}
```

This will print out:

```
5 10 15
2 4 6 8 10
```

Arrays as Parameters

In the previous code, the function will accept any array whose elements are of type int, whatever its length. For that reason it's required to include a second parameter telling the function the length of each array that we pass to it (otherwise we could potentially go out of range of the array and would not know when to stop the loop).

Arrays as Parameters

It is also possible to pass multidimensional arrays to functions. The format for a tri-dimensional array is:

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

Which can be called with:

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

Note that the first brackets are empty, however the second and third brackets need the numbers for the size of their respective dimensions as the compiler needs this to calculate where in the array to access.

Arrays as Parameters

In a way, passing an array to a function always loses a dimension. The reason for this is for historical reasons, arrays cannot be copied and are always passed as a pointer (more on that later).

This is a common source of errors for novice programmers. Properly understanding pointers (coming up next) will help.