IX CHAPTER

Arrays

A big part of C's popularity is due to the way it handles arrays. C handles arrays very efficiently for three reasons.

First, except for some interpreters that are helpfully paranoid, array subscripting is done at a very low level. There's not enough information at runtime to tell how long an array is, or whether a subscript is valid. The language of the ANSI/ISO C standard says that if you use an invalid subscript, the behavior is undefined. That means that your program can (a) work correctly, maybe, (b) halt or crash dramatically, (c) continue running but get the wrong answer, or (d) none of the above. You don't know what your program will do. This is a Bad Thing. Some people use this weakness as justification to criticize C as merely a high-level assembler language. Certainly, when C programs fail, they can fail spectacularly. But when they're written and tested well, they run fast.

Second, arrays and pointers work very well together. When used in an expression, the value of an array is the same as a pointer to its first element. That makes pointers and arrays almost interchangeable. Using pointers can be twice as fast as using array subscripts. (See FAQ IX.5 for an example.)

Third, when an array is passed as a parameter to a function, it's exactly as if a pointer to the first element was passed. There's no feature built into the C language for copying the contents of arrays ("call by value"). (Structures that contain arrays are copied, which might seem inconsistent.) Just the address ("call by reference") is much faster than call by value. C++ and ANSI C have the `const` keyword, which allows call by reference to be as safe as call by value. For details, see FAQ II.4 and the beginning of Chapter VII, "Pointers and Memory Allocation."

The equivalence of array and pointer parameters causes some confusion. A function defined as

```
void  f( char a[ MAX ] )
{
       /* ... */
}
```

(in which MAX is a #defined "manifest constant" or some other value known at compile time) is *exactly* the same as this:

```
void  f( char *a )
{
       /* ... */
}
```

This equivalence is the third advantage described previously. Most C programmers learn it early. It's confusing because it's the *only* case in which pointers and arrays mean exactly the same thing. If you write (anywhere but in the declaration of a function parameter)

```
char    a[ MAX ];
```

then space is made for MAX characters. If you write

```
char    *a;
```

instead, then space is made for a char pointer, which is probably only as big as two or four chars. This can be a real disaster if you define

```
char    a[ MAX ];
```

in a source file but declare

```
extern  char    *a;
```

in a header file. The best way to check this is to always have the declaration visible (by #including the appropriate header file) when making a definition.

If you define

```
char    a[ MAX ];
```

in a source file, you can declare

```
extern  char    a[];
```

in the appropriate header file. This tells any #including files that a is an array, not a pointer. It doesn't say how long a is. This is called an "incomplete" type. Using incomplete types this way is a common practice, and a good one.

IX.1: Do array subscripts always start with zero?

Answer:

Yes. If you have an array a[MAX] (in which MAX is some value known at compile time), the first element is a[0], and the last element is a[MAX-1]. This arrangement is different from what you would find in some other

languages. In some languages, such as some versions of BASIC, the elements would be `a[1]` through `a[MAX]`, and in other languages, such as Pascal, you can have it either way.

WARNING

`a[MAX]` is a valid address, but the value there is not an element of array `a` (see FAQ IX.2).

This variance can lead to some confusion. The "first element" in non-technical terms is the "zero'th" element according to its array index. If you're using spoken words, use "first" as the opposite of "last." If that's not precise enough, use pseudo-C. You might say, "The elements a sub one through a sub eight," or, "The second through ninth elements of `a`."

There's something you can do to try to fake array subscripts that start with one. Don't do it. The technique is described here only so that you'll know why not to use it.

Because pointers and arrays are almost identical, you might consider creating a pointer that would refer to the same elements as an array but would use indices that start with one. For example:

```
/* don't do this!!! */
int     a0[ MAX ];
int     *a1 = a0 - 1;   /* & a[ -1 ] */
```

Thus, the first element of `a0` (if this worked, which it might not) would be the same as `a1[1]`. The last element of `a0`, `a0[MAX-1]`, would be the same as `a1[MAX]`. There are two reasons why you shouldn't do this.

The first reason is that it might not work. According to the ANSI/ISO standard, it's undefined (which is a Bad Thing). The problem is that `&a[-1]` might not be a valid address; see FAQ IX.3. Your program might work all the time with some compilers, and some of the time with all compilers. Is that good enough?

The second reason not to do this is that it's not C-like. Part of learning C is to learn how array indices work. Part of reading (and maintaining) someone else's C code is being able to recognize common C idioms. If you do weird stuff like this, it'll be harder for people to understand your code. (It'll be harder for you to understand your own code, six months later.)

Cross Reference:

IX.2: Is it valid to address one element beyond the end of an array?

IX.3: Why worry about the addresses of the elements beyond the end of an array?

IX.2: Is it valid to address one element beyond the end of an array?

Answer:

It's valid to address it, but not to see what's there. (The really short answer is, "Yes, so don't worry about it.")

With most compilers, if you say

```
int     i, a[MAX], j;
```

then either `i` or `j` is at the part of memory just after the last element of the array. The way to see whether `i` or `j` follows the array is to compare their addresses with that of the element following the array. The way to say this in C is that either

```
& i == & a[ MAX ]
```

is true or

```
& a[ MAX ] == & j
```

is true. This isn't guaranteed; it's just the way it usually works.

The point is, if you store something in `a[MAX]`, you'll usually clobber something outside the `a` array. Even looking at the value of `a[MAX]` is technically against the rules, although it's not usually a problem.

Why would you ever want to say `&a[MAX]`? There's a common idiom of going through every member of a loop using a pointer (see FAQ IX.5). Instead of

```
for ( i = 0; i < MAX; ++i )
{
        /* do something */;
}
```

C programmers often write this:

```
for ( p = a; p < & a[ MAX ]; ++p )
{
        /* do something */;
}
```

The kind of loop shown here is so common in existing C code that the C standard says it must work.

Cross Reference:

IX.3: Why worry about the addresses of the elements beyond the end of an array?

IX.5: Is it better to use a pointer to navigate an array of values, or is it better to use a subscripted array name?

IX.3: Why worry about the addresses of the elements beyond the end of an array?

Answer:

If your programs ran only on nice machines on which the addresses were always between `0x00000000` and `0xFFFFFFFF` (or something similar), you wouldn't need to worry. But life isn't always that simple.

Sometimes addresses are composed of two parts. The first part (often called the "base") is a pointer to the beginning of some chunk of memory; the second part is an offset from the beginning of that chunk. The most notorious example of this is the Intel 8086, which is the basis for all MS-DOS programs. (Your shiny new

Pentium chip runs most MS-DOS applications in 8086 compatibility mode.) This is called a "segmented architecture." Even nice RISC chips with linear address spaces have register indexing, in which one register points to the beginning of a chunk, and the second is an offset. Subroutine calls are usually implemented with an offset from a stack pointer.

What if your program was using base/offset addresses, and some array `a0` was the first thing in the chunk of memory being pointed to? (More formally, what if the base pointer was the same as `& a0[ 0 ]`?) The point is, because the base can't be changed (efficiently) and the offset can't be negative, there might not be a valid way of saying "the element before `a0[0]`." The ANSI C standard specifically says attempts to get at this element are undefined. That's why the idea discussed in FAQ IX.1 might not work.

The only other time there could be a problem with the address of the element beyond the end of an array is if the array is the last thing that fits in memory (or in the current memory segment). If the last element of `a` (that is, `a[MAX-1]`) is at the last address in memory, what's the address of the element after it? There isn't one. The compiler must complain that there's not enough room for the array, if that's what it takes to ensure that `&a[MAX]` is valid.

You can say you'll only ever write programs for Windows or UNIX or Macintoshes. The people who defined the C programming language don't have that luxury. They had to define C so that it would work in weird environments, such as microprocessor-controlled toasters and anti-lock braking systems and MS-DOS. They defined it so that programs written strictly by the rules can be compiled and run for almost anything. Whether you want to break the strict rules sometimes is between you, your compiler, and your customers.

Cross Reference:

IX.1: Do array subscripts always start with zero?

IX.2: Is it valid to address one element beyond the end of an array?

IX.4: Can the *sizeof* operator be used to tell the size of an array passed to a function?

Answer:

No. There's no way to tell, at runtime, how many elements are in an array parameter just by looking at the array parameter itself. Remember, passing an array to a function is exactly the same as passing a pointer to the first element. This is a Good Thing. It means that passing pointers and arrays to C functions is very efficient.

It also means that the programmer must use some mechanism to tell how big such an array is. There are two common ways to do that. The first method is to pass a count along with the array. This is what `memcpy()` does, for example:

```
char    source[ MAX ], dest[ MAX ];
/* ... */
memcpy( dest, source, MAX );
```

The second method is to have some convention about when the array ends. For example, a C "string" is just a pointer to the first character; the string is terminated by an ASCII NUL (`'\0'`) character. This is also commonly done when you have an array of pointers; the last is the null pointer. Consider the following function, which takes an array of `char*`s. The last `char*` in the array is `NULL`; that's how the function knows when to stop.

```
void printMany( char *strings[] )
{
        int     i;
        i = 0;
        while ( strings[ i ] != NULL )
        {
             puts( strings[ i ] );
             ++i;
        }
}
```

Most C programmers would write this code a little more cryptically:

```
void  printMany( char *strings[] )
{
        while ( *strings )
        {
                puts( *strings++ );
        }
}
```

As discussed in FAQ IX.5, C programmers often use pointers rather than indices. You can't change the value of an array tag, but because `strings` is an array parameter, it's really the same as a pointer (see FAQ IX.6). That's why you can increment `strings`. Also,

```
while ( *strings )
```

means the same thing as

```
while ( *strings != NULL )
```

and the increment can be moved up into the call to `puts()`.

If you document a function (if you write comments at the beginning, or if you write a "manual page" or a design document), it's important to describe how the function "knows" the size of the arrays passed to it. This description can be something simple, such as "null terminated," or "`elephants` has `numElephants` elements." (Or "`arr` should have 13 elements," if your code is written that way. Using hard coded numbers such as 13 or 64 or 1024 is not a great way to write C code, though.)

Cross Reference:

IX.5: Is it better to use a pointer to navigate an array of values, or is it better to use a subscripted array name?

IX.6: Can you assign a different address to an array tag?

IX.5: Is it better to use a pointer to navigate an array of values, or is it better to use a subscripted array name?

Answer:

It's easier for a C compiler to generate good code for pointers than for subscripts.

Say that you have this:

```
/* X is some type */
X       a[ MAX ];       /* array */
X       *p;     /* pointer */
X       x;      /* element */
int     i;      /* index */
```

Here's one way to loop through all elements:

```
/* version (a) */
for ( i = 0; i < MAX; ++i )
{
        x = a[ i ];
        /* do something with x */
}
```

On the other hand, you could write the loop this way:

```
/* version (b) */
for ( p = a; p < & a[ MAX ]; ++p )
{
        x = *p;
        /* do something with x */
}
```

What's different between these two versions? The initialization and increment in the loop are the same. The comparison is about the same; more on that in a moment. The difference is between `x=a[i]` and `x=*p`. The first has to find the address of `a[i]`; to do that, it needs to multiply `i` by the size of an `X` and add it to the address of the first element of `a`. The second just has to go indirect on the `p` pointer. Indirection is fast; multiplication is relatively slow.

This is "micro efficiency." It might matter, it might not. If you're adding the elements of an array, or simply moving information from one place to another, much of the time in the loop will be spent just using the array index. If you do any I/O, or even call a function, each time through the loop, the relative cost of indexing will be insignificant.

Some multiplications are less expensive than others. If the size of an `X` is 1, the multiplication can be optimized away (1 times anything is the original anything). If the size of an `X` is a power of 2 (and it usually is if `X` is any of the built-in types), the multiplication can be optimized into a left shift. (It's like multiplying by 10 in base 10.)

What about computing `&a[MAX]` every time though the loop? That's part of the comparison in the pointer version. Isn't it as expensive computing `a[i]` each time? It's not, because `&a[MAX]` doesn't change during the loop. Any decent compiler will compute that, once, at the beginning of the loop, and use the same value each time. It's as if you had written this:

```
/* how the compiler implements version (b) */
X       *temp = & a[ MAX ];     /* optimization */
for ( p = a; p < temp; ++p )
{
        x = *p;
        /* do something with x */
}
```

This works only if the compiler can tell that `a` and `MAX` can't change in the middle of the loop.

There are two other versions; both count down rather than up. That's no help for a task such as printing the elements of an array in order. It's fine for adding the values or something similar. The index version presumes that it's cheaper to compare a value with zero than to compare it with some arbitrary value:

```
/* version (c) */
for ( i = MAX - 1; i >= 0; --i )
{
        x = a[ i ];
        /* do something with x */
}
```

The pointer version makes the comparison simpler:

```
/* version (d) */
for ( p = & a[ MAX - 1 ]; p >= a; --p )
{
        x = *p;
        /* do something with x */
}
```

Code similar to that in version (d) is common, but not necessarily right. The loop ends only when `p` is less than `a`. That might not be possible, as described in FAQ IX.3.

The common wisdom would finish by saying, "Any decent optimizing compiler would generate the same code for all four versions." Unfortunately, there seems to be a lack of decent optimizing compilers in the world. A test program (in which the size of an `X` was not a power of 2 and in which the "do something" was trivial) was built with four very different compilers. Version (b) always ran much faster than version (a), sometimes twice as fast. Using pointers rather than indices made a big difference. (Clearly, all four compilers optimize `&a[MAX]` out of the loop.)

How about counting down rather than counting up? With two compilers, versions (c) and (d) were about the same as version (a); version (b) was the clear winner. (Maybe the comparison is cheaper, but decrementing is slower than incrementing?) With the other two compilers, version (c) was about the same as version (a) (indices are slow), but version (d) was slightly faster than version (b).

So if you want to write portable efficient code to navigate an array of values, using a pointer is faster than using subscripts. Use version (b); version (d) might not work, and even if it does, it might be compiled into slower code.

Most of the time, though, this is micro-optimizing. The "do something" in the loop is where most of the time is spent, usually. Too many C programmers are like half-sloppy carpenters; they sweep up the sawdust but leave a bunch of two-by-fours lying around.

Cross Reference:

IX.2: Is it valid to address one element beyond the end of an array?

IX.3: Why worry about the addresses of the elements beyond the end of an array?

IX.6: Can you assign a different address to an array tag?

Answer:

No, although in one common special case, it looks as if you can.

An array tag is not something you can put on the left side of an assignment operator. (It's not an "lvalue," let alone a "modifiable lvalue.") An array is an object; the array tag is a pointer to the first element in that object.

For an external or static array, the array tag is a constant value known at link time. You can no more change the value of such an array tag than you can change the value of 7.

Assigning to an array tag would be missing the point. An array tag is not a pointer. A pointer says, "Here's one element; there might be others before or after it." An array tag says, "Here's the first element of an array; there's nothing before it, and you should use an index to find anything after it." If you want a pointer, use a pointer.

In one special case, it looks as if you can change an array tag:

```
void  f( char a[ 12 ] )
{
        ++a;    /* legal! */
}
```

The trick here is that array parameters aren't really arrays. They're really pointers. The preceding example is equivalent to this:

```
void  f( char *a )
{
        ++a;    /* certainly legal */
}
```

You can write this function so that the array tag can't be modified. Oddly enough, you need to use pointer syntax:

```
void  f( char * const a )
{
        ++a;    /* illegal */
}
```

Here, the parameter is an lvalue, but the `const` keyword means it's not modifiable.

Cross Reference:

IX.4: Can the `sizeof` operator be used to tell the size of an array passed to a function?

IX.7: What is the difference between *array_name* and *&array_name*?

Answer:

One is a pointer to the first element in the array; the other is a pointer to the array as a whole.

NOTE

It's strongly suggested that you put this book down for a minute and write the declaration of a variable that points to an array of `MAX` characters. Hint: Use parentheses. If you botch this assignment, what do you get instead? Playing around like this is the only way to learn the arcane syntax C uses for pointers to complicated things. The solution is at the end of this answer.

An array is a type. It has a base type (what it's an array of), a size (unless it's an "incomplete" array), and a value (the value of the whole array). You can get a pointer to this value:

```
char    a[ MAX ];          /* array of MAX characters */
char    *p;                /* pointer to one character */
/* pa is declared below */
pa = & a;
p = a;  /* = & a[ 0 ] */
```

After running that code fragment, you might find that `p` and `pa` would be printed as the same value; they both point to the same address. They point to different types of `MAX` characters.

The wrong answer is

```
char *( ap[ MAX ] );
```

which is the same as this:

```
char *ap[ MAX ];
```

This code reads, "`ap` is an array of `MAX` pointers to characters."

Cross Reference:

None.

IX.8: Why can't constant values be used to define an array's initial size?

Answer:

There are times when constant values can be used and there are times when they can't. A C program can use what C considers to be constant expressions, but not everything C++ would accept.

When defining the size of an array, you need to use a constant expression. A constant expression will always have the same value, no matter what happens at runtime, and it's easy for the compiler to figure out what that value is. It might be a simple numeric literal:

```
char    a[ 512 ];
```

Or it might be a "manifest constant" defined by the preprocessor:

```
#define MAX     512
/* ... */
char    a[ MAX ];
```

Or it might be a `sizeof`:

```
char    a[ sizeof( struct cacheObject ) ];
```

Or it might be an expression built up of constant expressions:

```
char    buf[ sizeof( struct cacheObject ) * MAX ];
```

Enumerations are allowed too.

An initialized `const int` variable is not a constant expression in C:

```
int     max = 512;      /* not a constant expression in C */
char    buffer[ max ];  /* not valid C */
```

Using `const ints` as array sizes is perfectly legal in C++; it's even recommended. That puts a burden on C++ compilers (to keep track of the values of `const int` variables) that C compilers don't need to worry about. On the other hand, it frees C++ programs from using the C preprocessor quite so much.

Cross Reference:

XV.1: Should C++ additions to a compiler be used in a C program?

XV.2: What is the difference between C++ and C?

IX.9: What is the difference between a string and an array?

Answer:

An array is an array of anything. A string is a specific kind of an array with a well-known convention to determine its length.

There are two kinds of programming languages: those in which a string is just an array of characters, and those in which it's a special type. In C, a string is just an array of characters (type `char`), with one wrinkle: a C string always ends with a NUL character. The "value" of an array is the same as the address of (or a pointer to) the first element; so, frequently, a C string and a pointer to `char` are used to mean the same thing.

An array can be any length. If it's passed to a function, there's no way the function can tell how long the array is supposed to be, unless some convention is used. The convention for strings is NUL termination; the last character is an ASCII NUL (`'\0'`) character.

In C, you can have a literal for an integer, such as the value of `42`; for a character, such as the value of `'*'`; or for a floating-point number, such as the value of `4.2e1` for a `float` or `double`.

NOTE

Actually, what looks like a type `char` literal is just a type `int` literal with a funny syntax. `42` and `'*'` are exactly the same value. This isn't the case for C++, which has true `char` literals and function parameters, and which generally distinguishes more carefully between a `char` and an `int`.

There's no such thing as a literal for an array of integers, or an arbitrary array of characters. It would be very hard to write a program without string literals, though, so C provides them. Remember, C strings conventionally end with a NUL character, so C string literals do as well. "`six times nine`" is 15 characters long (including the NUL terminator), not just the 14 characters you can see.

There's a little-known, but very useful, rule about string literals. If you have two or more string literals, one after the other, the compiler treats them as if they were one big string literal. There's only one terminating NUL character. That means that `"Hello, " "world"` is the same as `"Hello, world"`, and that

```
char    message[] =
  "This is an extremely long prompt\n"
  "How long is it?\n"
  "It's so long,\n"
  "It wouldn't fit on one line\n";
```

is exactly the same as some code that wouldn't fit on this page of the book.

When defining a string variable, you need to have either an array that's long enough or a pointer to some area that's long enough. Make sure that you leave room for the NUL terminator. The following example code has a problem:

```
char greeting[ 12 ];
strcpy( greeting, "Hello, world" );    /* trouble */
```

There's a problem because `greeting` has room for only 12 characters, and `"Hello, world"` is 13 characters long (including the terminating NUL character). The NUL character will be copied to someplace beyond the `greeting` array, probably trashing something else nearby in memory. On the other hand,

```
char   greeting[ 12 ] = "Hello, world";  /* not a string */
```

is OK if you treat `greeting` as a char array, not a string. Because there wasn't room for the NUL terminator, the NUL is not part of `greeting`. A better way to do this is to write

```
char    greeting[] = "Hello, world";
```

to make the compiler figure out how much room is needed for everything, including the terminating NUL character.

String literals are arrays of characters (type `char`), not arrays of constant characters (type `const char`). The ANSI C committee could have redefined them to be arrays of `const char`, but millions of lines of code would have screamed in terror and suddenly not compiled. The compiler won't stop you from trying to modify the contents of a string literal. You shouldn't do it, though. A compiler can choose to put string literals in some part of memory that can't be modified—in ROM, or somewhere the memory mapping registers will forbid writes. Even if string literals are someplace where they could be modified, the compiler can make them shared. For example, if you write

```
char    *p = "message";
char    *q = "message";
p[ 4 ] = '\0';  /* p now points to "mess" */
```

(and the literals are modifiable), the compiler can take one of two actions. It can create two separate string constants, or it can create just one (that both `p` and `q` point to). Depending on what the compiler did, `q` might still be a message, or it might just be a mess.

NOTE

This is "C humor." Now you know why so few programmers quit their day jobs for stand-up comedy.

Cross Reference:

IX.1: Do array subscripts always start with zero?