CHAPTER VII

Pointers and Memory Allocation

Pointers are the double-edged swords of C programming. Using them, you can cut through to the heart of a problem. Your code can be efficient, terse, and elegant. Pointers can also slice your program to shreds.

With a pointer, you can write data anywhere. That's the point. If you have a "wild" pointer that points in the wrong place, none of your data is safe. The data you put on the heap can get damaged. The data structures used to manage the heap can be corrupted. Even operating-system information can be modified. Maybe all three.

What happens next? That depends on how badly mangled everything has gotten, and how much more the damaged parts of memory are used. At some point, maybe right away, maybe later, some function runs into real trouble. It could be one of the allocation functions, or one of your functions, or a library function.

The program might die with an error message. It might hang. It might go into an infinite loop. It might produce bad results. Or maybe, this time, nothing essential gets damaged, and the program seems to be just fine.

The exciting part is that the program might not fail visibly until long after the root problem has happened. It might not fail at all when you test it, only when users run it.

In C programs, any wild pointer or out-of-bounds array subscript can bring the house down this way. So can "double deallocation" (see FAQ VII.22). Did you ever wonder why some C programmers earn big bucks? Now you know part of the answer.

There are memory allocation tools that can help find leaks (see FAQ VII.21), double deallocations, some wild pointers and subscripts, and other problems. Such tools are not portable; they work only with specific operating systems, or even specific brands of compilers. If you can find such a tool, get it and use it; it can save you a lot of time and improve the quality of your software.

Pointer arithmetic is unique to C (and its derivatives, such as C++). Assembler language enables you to perform arithmetic on addresses, but all notion of data typing is lost. Most high-level languages don't enable you to do anything with pointers except see what they point to. C is different.

C does arithmetic on pointers the way a person might do arithmetic on street addresses. Say you live in a town where, on every block, all the street addresses are used. One side of the street uses consecutive even addresses; the other, consecutive odd addresses. If you wanted to know the address of the house five doors north of 158 River Rd., you wouldn't add 5 and look for number 163. You would multiply 5 (the number of houses you want to advance) by 2 (the "distance" between houses), add that number to 158, and head for 168 River Rd. Similarly, if you had a pointer to a two-byte short at address 158 (decimal) and added 5, the result would be a pointer to a short at address 168 (decimal). See FAQs VII.7 and VII.8 for details on adding and subtracting from pointers.

Street addresses work only within a given block. Pointer arithmetic works only within an array. In practice, that's not a limitation; an array is the only place pointer arithmetic makes sense. An array, in this case, doesn't need to be the contents of an array variable. `malloc()` and `calloc()` return a pointer to an array allocated off the heap. (What's the difference? See FAQ VII.16.)

Pointer declarations are hard to read. A declaration such as

```
char *p;
```

means that `*p` is a `char`. (The "star," or asterisk, is known as the indirection operator; when a program "goes indirect on a pointer," it refers to the pointed-to data.)

For most kinds of computers, a pointer is a pointer. Some have different pointers to data and to functions, or to bytes (`char*`s and `void*`s) and to words (everything else). If you use `sizeof`, you're unlikely to have a problem. Some C programs and programmers assume that any pointer can be stored in an `int`, or at least a `long`. That's not guaranteed. This isn't a big deal—unless your programs have to run on IBM-compatible PCs.

NOTE

Macintosh and UNIX programmers are excused from the following discussion.

The original IBM PC used a processor that couldn't efficiently use pointers that were more than 16 bits long. (This point can be argued, preferably over a few beers. The 16-bit "pointers" are offsets; see the discussion of base/offset pointers in FAQ IX.3.) With some contortions, the original IBM PC could use pointers that were effectively 20 bits long. Ever since, all sorts of software for IBM compatibles have been fighting that limit.

To get 20-bit pointers to data, you need to tell your compiler to use the right memory model, perhaps compact. Twenty-bit function pointers come with the medium memory model. The large and huge memory models use 20-bit pointers for both data and functions. Either way, you might need to specify `far` pointers (see FAQs VII.18 and VII.19).

The 286-based systems could break through the 20-bit ceiling, but not easily. Starting with the 386, PC compatibles have been able to use true 32-bit addresses. MS-DOS hasn't. Operating systems such as MS-Windows and OS/2 are catching up.

If you run out of conventional memory in an MS-DOS program, you might need to allocate from expanded or extended memory. Various C compilers and libraries enable you to do this in different ways.

All this is grossly unportable. Some of it works for most or all MS-DOS and MS-Windows C compilers. Some is specific to particular compilers. Some works only with a given add-on library. If you already have such a product, check its documentation for details. If you don't, sleep easy tonight, and dream of the fun that awaits you.

VII.1: What is indirection?

Answer:

If you declare a variable, its name is a direct reference to its value. If you have a pointer to a variable, or any other object in memory, you have an indirect reference to its value. If `p` is a pointer, the value of `p` is the address of the object. `*p` means "apply the indirection operator to `p`"; its value is the value of the object that `p` points to. (Some people would read it as "Go indirect on `p`.")

`*p` is an lvalue; like a variable, it can go on the left side of an assignment operator, to change the value. If `p` is a pointer to a constant, `*p` is not a modifiable lvalue; it can't go on the left side of an assignment. (See FAQ II.4 and the discussion at the beginning of this chapter.) Consider the following program. It shows that when `p` points to `i`, `*p` can appear wherever `i` can.

Listing VII.1. An example of indirection.

```
#include <stdio.h>
int
main()
{
        int i;
        int *p;
        i = 5;
        p = & i;    /* now *p == i */
        /* %P is described in FAQ VII.28 */
        printf("i=%d, p=%P, *p=%d\n", i, p, *p);
        *p = 6;     /* same as i = 6 */
        printf("i=%d, p=%P, *p=%d\n", i, p, *p);
        return 0; /* see FAQ XVI.4 */
}
```

After `p` points to `i` (`p = &i`), you can print `i` or `*p` and get the same thing. You can even assign to `*p`, and the result is the same as if you had assigned to `i`.

Cross Reference:

II.4: What is a `const` pointer?

VII.2: How many levels of pointers can you have?

Answer:

The answer depends on what you mean by "levels of pointers." If you mean "How many levels of indirection can you have in a single declaration?" the answer is "At least 12."

```
int     i = 0;
int     *ip01 = & i;
int     **ip02 = & ip01;
int     ***ip03 = & ip02;
int     ****ip04 = & ip03;
int     *****ip05 = & ip04;
int     ******ip06 = & ip05;
int     *******ip07 = & ip06;
int     ********ip08 = & ip07;
int     *********ip09 = & ip08;
int     **********ip10 = & ip09;
int     ***********ip11 = & ip10;
int     ************ip12 = & ip11;
************ip12 = 1;   /* i = 1 */
```

NOTE

The ANSI C standard says all compilers must handle at least 12 levels. Your compiler might support more.

If you mean "How many levels of pointer can you use before the program gets hard to read," that's a matter of taste, but there is a limit. Having two levels of indirection (a pointer to a pointer to something) is common. Any more than that gets a bit harder to think about easily; don't do it unless the alternative would be worse.

If you mean "How many levels of pointer indirection can you have at runtime," there's no limit. This point is particularly important for circular lists, in which each node points to the next. Your program can follow the pointers forever. Consider the following (rather dumb) example in Listing VII.2.

Listing VII.2. A circular list that uses infinite indirection.

```
/* Would run forever if you didn't limit it to MAX */
#include <stdio.h>
struct circ_list
{
        char    value[ 3 ];     /* e.g., "st" (incl '\0') */
        struct circ_list        *next;
};
struct circ_list    suffixes[] = {
        "th", & suffixes[ 1 ], /* 0th */
        "st", & suffixes[ 2 ], /* 1st */
        "nd", & suffixes[ 3 ], /* 2nd */
        "rd", & suffixes[ 4 ], /* 3rd */
        "th", & suffixes[ 5 ], /* 4th */
        "th", & suffixes[ 6 ], /* 5th */
        "th", & suffixes[ 7 ], /* 6th */
        "th", & suffixes[ 8 ], /* 7th */
```

```
        "th", & suffixes[ 9 ], /* 8th */
        "th", & suffixes[ 0 ], /* 9th */
};
#define MAX 20
main()
{
     int i = 0;
     struct circ_list    *p = suffixes;
     while (i <= MAX) {
             printf( "%d%s\n", i, p->value );
             ++i;
             p = p->next;
     }
```

Each element in `suffixes` has one suffix (two characters plus the terminating NUL character) and a pointer to the next element. `next` is a pointer to something that has a pointer, to something that has a pointer, ad infinitum.

The example is dumb because the number of elements in `suffixes` is fixed. It would be simpler to have an array of `suffixes` and to use the `i%10`'th element. In general, circular lists can grow and shrink; they're much more interesting than `suffixes` in Listing VII.2.

Cross Reference:

VII.1: What is indirection?

VII.3: What is a null pointer?

Answer:

There are times (see FAQ VII.4) when it's necessary to have a pointer that doesn't point to anything. The macro `NULL`, defined in `<stddef.h>`, has a value that's guaranteed to be different from any valid pointer. `NULL` is a literal zero, possibly cast to `void*` or `char*`. Some people, notably C++ programmers, prefer to use `0` rather than `NULL`.

You can't use an integer when a pointer is required. The exception is that a literal zero value can be used as the null pointer. (It doesn't have to be a literal zero, but that's the only useful case. Any expression that can be evaluated at compile time, and that is zero, will do. It's not good enough to have an integer variable that might be zero at runtime.)

NOTE

The null pointer might not be stored as a zero; see FAQ VII.10.

WARNING

You should never go indirect on a null pointer. If you do, your program might get garbage, get a value that's all zeros, or halt gracelessly.

Cross Reference:

VII.4: When is a null pointer used?

VII.10: Is NULL always equal to 0?

VII.24: What is a "null pointer assignment" error? What are bus errors, memory faults, and core dumps?

VII.4: When is a null pointer used?

Answer:

The null pointer is used in three ways:

To stop indirection in a recursive data structure
As an error value
As a sentinel value

Using a Null Pointer to Stop Indirection or Recursion

Recursion is when one thing is defined in terms of itself. A recursive function calls itself. The following factorial function calls itself and therefore is considered recursive:

```
/* Dumb implementation; should use a loop */
unsigned factorial( unsigned i )
{
     if ( i == 0 || i == 1 )
     {
          return 1;
     }
     else
     {
          return i * factorial( i - 1 );
     }
}
```

A recursive data structure is defined in terms of itself. The simplest and most common case is a (singularly) linked list. Each element of the list has some value, and a pointer to the next element in the list:

```
struct string_list
{
     char    *str;   /* string (in this case) */
     struct string_list      *next;
};
```

There are also doubly linked lists (which also have a pointer to the preceding element) and trees and hash tables and lots of other neat stuff. You'll find them described in any good book on data structures.

You refer to a linked list with a pointer to its first element. That's where the list starts; where does it stop? This is where the null pointer comes in. In the last element in the list, the next field is set to NULL when there is no following element. To visit all the elements in a list, start at the beginning and go indirect on the next pointer as long as it's not null:

```
while ( p != NULL )
{
     /* do something with p->str */
     p = p->next;
}
```

Notice that this technique works even if `p` starts as the null pointer.

Using a Null Pointer As an Error Value

The second way the null pointer can be used is as an error value. Many C functions return a pointer to some object. If so, the common convention is to return a null pointer as an error code:

```
if ( setlocale( cat, loc_p ) == NULL )
{
     /* setlocale() failed; do something */
     /* ... */
}
```

This can be a little confusing. Functions that return pointers almost always return a valid pointer (one that doesn't compare equal to zero) on success, and a null pointer (one that compares equal to zero) pointer on failure. Other functions return an `int` to show success or failure; typically, zero is success and nonzero is failure. That way, a "true" return value means "do some error handling":

```
if ( raise( sig ) != 0 ) {
        /* raise() failed; do something */
        /* ... */
}
```

The success and failure return values make sense one way for functions that return `int`s, and another for functions that return pointers. Other functions might return a count on success, and either zero or some negative value on failure. As with taking medicine, you should read the instructions first.

Using a Null Pointer As a Sentinel Value

The third way a null pointer can be used is as a "sentinel" value. A sentinel value is a special value that marks the end of something. For example, in `main()`, `argv` is an array of pointers. The last element in the array (`argv[argc]`) is always a null pointer. That's a good way to run quickly through all the elements:

```
/*
A simple program that prints all its arguments.
It doesn't use argc ("argument count"); instead,
it takes advantage of the fact that the last
value in argv ("argument vector") is a null pointer.
*/
#include <stdio.h>
#include <assert.h>
int
main( int argc, char **argv)
{
        int i;
        printf("program name = \"%s\"\n", argv[0]);
        for (i=1; argv[i] != NULL; ++i)
                printf("argv[%d] = \"%s\"\n",
                  i, argv[i]);
```

```
        assert(i == argc);     /* see FAQ XI.5 */
        return 0; /* see FAQ XVI.4 */
}
```

Cross Reference:

VII.3: What is a null pointer?

VII.10: Is NULL always equal to 0?

XX.2: Should programs always assume that command-line parameters can be used?

VII.5: What is a *void* pointer?

Answer:

A `void` pointer is a C convention for "a raw address." The compiler has no idea what type of object a `void` pointer "really points to." If you write

```
int     *ip;
```

`ip` points to an `int`. If you write

```
void    *p;
```

`p` doesn't point to a `void`!

In C and C++, any time you need a `void` pointer, you can use another pointer type. For example, if you have a `char*`, you can pass it to a function that expects a `void*`. You don't even need to cast it. In C (but not in C++), you can use a `void*` any time you need any kind of pointer, without casting. (In C++, you need to cast it.)

Cross Reference:

VII.6: When is a `void` pointer used?

VII.27: Can math operations be performed on a `void` pointer?

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

VII.6: When is a *void* pointer used?

Answer:

A `void` pointer is used for working with raw memory or for passing a pointer to an unspecified type.

Some C code operates on raw memory. When C was first invented, character pointers (`char *`) were used for that. Then people started getting confused about when a character pointer was a string, when it was a character array, and when it was raw memory.

For example, `strcpy()` is used to copy data from one string to another, and `strncpy()` is used to copy at most a certain length string to another:

```
char  *strcpy( char *str1, const char *str2 );
char  *strncpy( char *str1, const char *str2, size_t n );
```

`memcpy()` is used to move data from one location to another:

```
void  *memcpy( void *addr1, void *addr2, size_t n );
```

`void` pointers are used to mean that this is raw memory being copied. NUL characters (zero bytes) aren't significant, and just about anything can be copied. Consider the following code:

```
#include "thingie.h"    /* defines struct thingie */
struct thingie  *p_src, *p_dest;
/* ... */
memcpy( p_dest, p_src, sizeof( struct thingie) * numThingies );
```

This program is manipulating some sort of object stored in a `struct thingie`. `p1` and `p2` point to arrays, or parts of arrays, of `struct thingies`. The program wants to copy `numThingies` of these, starting at the one pointed to by `p_src`, to the part of the array beginning at the element pointed to by `p_dest`. `memcpy()` treats `p_src` and `p_dest` as pointers to raw memory; `sizeof( struct thingie) * numThingies` is the number of bytes to be copied.

The keyword `void` had been invented to mean "no value," so `void*` was adopted to mean "a pointer to some thing, I don't know what exactly." `void` pointers are often used with function pointers.

Cross Reference:

VII.5: What is a `void` pointer?

VII.14: When would you use a pointer to a function?

VII.7: Can you subtract pointers from each other? Why would you?

Answer:

If you have two pointers into the same array, you can subtract them. The answer is the number of elements between the two elements.

Consider the street address analogy presented in the introduction of this chapter. Say that I live at 118 Fifth Avenue and that my neighbor lives at 124 Fifth Avenue. The "size of a house" is two (on my side of the street, sequential even numbers are used), so my neighbor is (124–118)/2 (or 3) houses up from me. (There are two houses between us, 120 and 122; my neighbor is the third.) You might do this subtraction if you're going back and forth between indices and pointers.

You might also do it if you're doing a binary search. If `p` points to an element that's before what you're looking for, and `q` points to an element that's after it, then `(q-p)/2+p` points to an element between `p` and `q`. If that element is before what you want, look between it and `q`. If it's after what you want, look between `p` and it.

(If it's what you're looking for, stop looking.)

You can't subtract arbitrary pointers and get meaningful answers. Someone might live at 110 Main Street, but I can't subtract 110 Main from 118 Fifth (and divide by 2) and say that he or she is four houses away! If each block starts a new hundred, I can't even subtract 120 Fifth Avenue from 204 Fifth Avenue. They're on the same street, but in different blocks of houses (different arrays).

C won't stop you from subtracting pointers inappropriately. It won't cut you any slack, though, if you use the meaningless answer in a way that might get you into trouble.

When you subtract pointers, you get a value of some integer type. The ANSI C standard defines a `typedef`, `ptrdiff_t`, for this type. (It's in `<stddef.h>`.) Different compilers might use different types (`int` or `long` or whatever), but they all define `ptrdiff_t` appropriately.

Listing VII.7 is a simple program that demonstrates this point. The program has an array of structures, each 16 bytes long. The difference between `array[0]` and `array[8]` is 8 when you subtract `struct stuff` pointers, but 128 (hex 0x80) when you cast the pointers to raw addresses and then subtract.

NOTE

Pointers are usually cast to "raw addresses" by casting to `void*`. The example casts to `char*`, because `void*`s can't be subtracted; see FAQ VII.27.

If you subtract 8 from a pointer to `array[8]`, you don't get something 8 bytes earlier; you get something 8 *elements* earlier.

Listing VII.7. Pointer arithmetic.

```
#include <stdio.h>
#include <stddef.h>
struct stuff {
        char    name[16];
        /* other stuff could go here, too */
};
struct stuff array[] = {
        { "The" },
        { "quick" },
        { "brown" },
        { "fox" },
        { "jumped" },
        { "over" },
        { "the" },
        { "lazy" },
        { "dog." },
        /*
        an empty string signifies the end;
        not used in this program,
        but without it, there'd be no way
        to find the end (see FAQ IX.4)
        */
        { "" }
};
main()
{
```

```
        struct stuff    *p0 = & array[0];
        struct stuff    *p8 = & array[8];
        ptrdiff_t       diff = p8 - p0;
        ptrdiff_t       addr_diff = (char*) p8 - (char*) p0;
        /*
        cast the struct stuff pointers to void*
        (which we know printf() can handle; see FAQ VII.28)
        */
        printf("& array[0] = p0 = %P\n", (void*) p0);
        printf("& array[8] = p8 = %P\n", (void*) p8);
        /*
        cast the ptrdiff_t's to long's
        (which we know printf() can handle)
        */
        printf("The difference of pointers is %ld\n",
          (long) diff);
        printf("The difference of addresses is %ld\n",
          (long) addr_diff);
        printf("p8 - 8 = %P\n", (void*) (p8 - 8));
        /* example for FAQ VII.8 */
        printf("p0 + 8 = %P (same as p8)\n", (void*) (p0 + 8));
        return 0;   /* see FAQ XVI.4 */
}
```

Cross Reference:

VII.8: When you add a value to a pointer, what is really added?

VII.12: Can you add pointers together? Why would you?

VII.27: Can math operations be performed on a `void` pointer?

VII.8: When you add a value to a pointer, what is really added?

Answer:

If you think only in terms of raw addresses, what's "really" added is the value times the size of the thing being pointed to...and you're missing the point of how C pointers work. When you add an integer and a pointer, the sum points that many elements away, not just that many bytes away.

Look at the end of Listing VII.7. When you add 8 to `& array[0]`, you don't get something eight bytes away. You get `& array[8]`, which is eight *elements* away.

Think about the street-address analogy presented in this chapter's introduction. You live on the even-numbered side of Oak Street, at number 744. There are no gaps in the even numbers. The "size of a house" is 2. If someone wants to know the address of the place three doors up from you, he multiplies the size (2) times 3, and thus adds 6; the address is 750. The house one door down is 744 + (–1)*2, or 742.

Street-address arithmetic works only within a given block; pointer arithmetic works only within a given array. If you try to calculate the address 400 blocks south of you, you'll get –56 Oak Street; fine, but that doesn't mean anything. If your program uses a meaningless address, it'll probably blow up.

Cross Reference:

VII.7: Can you subtract pointers from each other? Why would you?

VII.12: Can you add pointers together? Why would you?

VII.27: Can math operations be performed on a `void` pointer?

VII.9: Is *NULL* always defined as 0?

Answer:

`NULL` is defined as either `0` or `(void*)0`. These values are almost identical; either a literal zero or a `void` pointer is converted automatically to any kind of pointer, as necessary, whenever a pointer is needed (although the compiler can't always tell when a pointer is needed).

Cross Reference:

VII.10: Is `NULL` always equal to 0?

VII.10: Is *NULL* always equal to 0?

Answer:

The answer depends on what you mean by "equal to." If you mean "compares equal to," such as

```
if ( /* ... */ )
{
     p = NULL;
}
else
{
     p = /* something else */;
}
/* ... */
if ( p == 0 )
```

then yes, `NULL` is always equal to 0. That's the whole point of the definition of a null pointer.

If you mean "is stored the same way as an integer zero," the answer is no, not necessarily. That's the most common way to store a null pointer. On some machines, a different representation is used.

The only way you're likely to tell that a null pointer isn't stored the same way as zero is by displaying a pointer in a debugger, or printing it. (If you cast a null pointer to an integer type, that might also show a nonzero value.)

Cross Reference:

VII.9: Is `NULL` always defined as 0?

VII.28: How do you print an address?

VII.11: What does it mean when a pointer is used in an *if* statement?

Answer:

Any time a pointer is used as a condition, it means "Is this a non-null pointer?" A pointer can be used in an `if`, `while`, `for`, or `do/while` statement, or in a conditional expression. It sounds a little complicated, but it's not.

Take this simple case:

```
if ( p )
{
     /* do something */
}
else
{
     /* do something else */
}
```

An `if` statement does the "then" (first) part when its expression compares unequal to zero. That is,

```
if ( /* something */ )
```

is always exactly the same as this:

```
if ( /* something */ != 0 )
```

That means the previous simple example is the same thing as this:

```
if ( p != 0 )
{
     /* do something (not a null pointer) */
}
else
{
     /* do something else (a null pointer) */
}
```

This style of coding is a little obscure. It's very common in existing C code; you don't have to write code that way, but you need to recognize such code when you see it.

Cross Reference:

VII.3: What is a null pointer?

VII.12: Can you add pointers together? Why would you?

Answer:

No, you can't add pointers together. If you live at 1332 Lakeview Drive, and your neighbor lives at 1364 Lakeview, what's 1332+1364? It's a number, but it doesn't mean anything. If you try to perform this type of calculation with pointers in a C program, your compiler will complain.

The only time the addition of pointers might come up is if you try to add a pointer and the difference of two pointers:

```
p = p + p2 - p1;
```

which is the same thing as this:

```
p = (p + p2) - p1.
```

Here's a correct way of saying this:

```
p = p + ( p2 - p1 );
```

Or even better in this case would be this example:

```
p += p2 - p1;
```

Cross Reference

VII.7: Can you subtract pointers from each other? Why would you?

VII.13: How do you use a pointer to a function?

Answer:

The hardest part about using a pointer-to-function is declaring it. Consider an example. You want to create a pointer, `pf`, that points to the `strcmp()` function. The `strcmp()` function is declared in this way:

```
int strcmp( const char *, const char * )
```

To set up `pf` to point to the `strcmp()` function, you want a declaration that looks just like the `strcmp()` function's declaration, but that has `*pf` rather than `strcmp`:

```
int     (*pf)( const char *, const char * );
```

Notice that you need to put parentheses around `*pf`. If you don't include parentheses, as in

```
int *pf( const char *, const char * );  /* wrong */
```

you'll get the same thing as this:

```
(int *) pf( const char *, const char * );  /* wrong */
```

That is, you'll have a declaration of a function that returns `int*`.

NOTE

For what it's worth, even experienced C programmers sometimes get this wrong. The simplest thing to do is remember where you can find an example declaration and copy it when you need to.

After you've gotten the declaration of `pf`, you can `#include <string.h>` and assign the address of `strcmp()` to `pf`:

```
pf = strcmp;
```

or

```
pf = & strcmp;  /* redundant & */
```

You don't need to go indirect on `pf` to call it:

```
if ( pf( str1, str2 ) > 0 )     /* ... */
```

Cross Reference:

VII.14: When would you use a pointer to a function?

VII.14: When would you use a pointer to a function?

Answer:

Pointers to functions are interesting when you pass them to other functions. A function that takes function pointers says, in effect, "Part of what I do can be customized. Give me a pointer to a function, and I'll call it when that part of the job needs to be done. That function can do its part for me." This is known as a "callback." It's used a lot in graphical user interface libraries, in which the style of a display is built into the library but the contents of the display are part of the application.

As a simpler example, say you have an array of character pointers (`char*`s), and you want to sort it by the value of the strings the character pointers point to. The standard `qsort()` function uses function pointers to perform that task. (For more on sorting, see Chapter III, "Sorting and Searching Data.") `qsort()` takes four arguments,

- a pointer to the beginning of the array,
- the number of elements in the array,
- the size of each array element, and
- a comparison function,

and returns an `int`.

The comparison function takes two arguments, each a pointer to an element. The function returns 0 if the pointed-to elements compare equal, some negative value if the first element is less than the second, and some positive value if the first element is greater than the second. A comparison function for integers might look like this:

```
int icmp( const int *p1, const int *p2 )
{
     return *p1 - *p2;
}
```

The sorting algorithm is part of `qsort()`. So is the exchange algorithm; it just copies bytes, possibly by calling `memcpy()` or `memmove()`. `qsort()` doesn't know what it's sorting, so it can't know how to compare them. That part is provided by the function pointer.

You can't use `strcmp()` as the comparison function for this example, for two reasons. The first reason is that `strcmp()`'s type is wrong; more on that a little later. The second reason is that it won't work. `strcmp()` takes two pointers to `char` and treats them as the first characters of two strings. The example deals with an array of character pointers (`char*`s), so the comparison function must take two pointers to character pointers (`char*`s). In this case, the following code might be an example of a good comparison function:

```
int strpcmp( const void *p1, const void *p2 )
{
     char * const *sp1 = (char * const *) p1;
     char * const *sp2 = (char * const *) p2;
     return strcmp( *sp1, *sp2 );
}
```

The call to `qsort()` might look something like this:

```
qsort( array, numElements, sizeof( char * ), pf2 );
```

`qsort()` will call `strpcmp()` every time it needs to compare two character pointers (`char*`s).

Why can't `strcmp()` be passed to `qsort()`, and why were the arguments of `strpcmp()` what they were? A function pointer's type depends on the return type of the pointed-to function, as well as the number and types of all its arguments. `qsort()` expects a function that takes two constant `void` pointers:

```
void qsort( void *base,
            size_t numElements,
            size_t sizeOfElement,
            int (*compFunct)( const void *, const void *) );
```

Because `qsort()` doesn't really know what it's sorting, it uses a `void` pointer in its argument (`base`) and in the arguments to the comparison function. `qsort()`'s `void*` argument is easy; any pointer can be converted to a `void*` without even needing a cast. The function pointer is harder.

For an array of character arrays, `strcmp()` would have the right algorithm but the wrong argument types. The simplest, safest way to handle this situation is to pass a function that takes the right argument types for `qsort()` and then casts them to the right argument types. That's what `strpcmp()` does.

If you have a function that takes a `char*`, and you know that a `char*` and a `void*` are the same in every environment your program might ever work in, you might cast the function pointer, rather than the pointed-to function's arguments, in this way:

```
char     table[ NUM_ELEMENTS ][ ELEMENT_SIZE ];
/* ... */
/* passing strcmp() to qsort for array of array of char */
qsort( table, NUM_ELEMENTS, ELEMENT_SIZE,
  ( int (*)( const void *, const void * ) ) strcmp );
```

Casting the arguments and casting the function pointer both can be error prone. In practice, casting the function pointer is more dangerous.

The basic problem here is using `void*` when you have a pointer to an unknown type. C++ programs sometime solve this problem with templates.

Cross Reference:

VII.5: What is a `void` pointer?

VII.6: When is a `void` pointer used?

VII.13: How do you use a pointer to a function?

VII.15: Can the size of an array be declared at runtime?

Answer:

No. In an array declaration, the size must be known at compile time. You can't specify a size that's known only at runtime. For example, if `i` is a variable, you can't write code like this:

```
char    array[i];    /* not valid C */
```

Some languages provide this latitude. C doesn't. If it did, the stack (see FAQ VII.20) would be more complicated, function calls would be more expensive, and programs would run a lot slower.

If you know that you have an array but you won't know until runtime how big it will be, declare a pointer to it and use `malloc()` or `calloc()` to allocate the array from the heap.

If you know at compile time how big an array is, you can declare its size at compile time. Even if the size is some complicated expression, as long as it can be evaluated at compile time, it can be used.

Listing VII.15 shows an example. It's a program that copies the `argv` array passed to `main()`.

Listing VII.15. Arrays with runtime size, using pointers and `malloc()`.

```
/*
A silly program that copies the argv array and all the pointed-to
strings. Just for fun, it also deallocates all the copies.
*/
#include <stdlib.h>
#include <string.h>
int
main(int argc, char** argv)
{
    char** new_argv;
    int i;
    /*
    Since argv[0] through argv[argc] are all valid, the
    program needs to allocate room for argc+1 pointers.
    */
    new_argv = (char**) calloc(argc+1, sizeof (char*));
    /* or malloc((argc+1) * sizeof (char*)) */
    printf("allocated room for %d pointers starting at %P\n",
      argc+1, new_argv);
    /*
    now copy all the strings themselves
    (argv[0] through argv[argc-1])
    */
    for (i = 0; i < argc; ++i) {
        /* make room for '\0' at end, too */
```

continues

Listing VII.15. continued

```
        new_argv[i] = (char*) malloc(strlen(argv[i]) + 1);
        strcpy(new_argv[i], argv[i]);
        printf("allocated %d bytes for new_argv[%d] at %P, "
        "copied \"%s\"\n",
          strlen(argv[i]) + 1, i, new_argv[i], new_argv[i]);
    }
    new_argv[argc] = NULL;
    /*
    To deallocate everything, get rid of the strings (in any
    order), then the array of pointers. If you free the array
    of pointers first, you lose all reference to the copied
    strings.
    */
    for (i = 0; i < argc; ++i) {
        free(new_argv[i]);
        printf("freed new_argv[%d] at %P\n", i, new_argv[i]);
        argv[i] = NULL;     /* paranoia; see note */
    }
    free(new_argv);
    printf("freed new_argv itself at %P\n", new_argv);
    return 0; /* see FAQ XVI.4 */
}
```

NOTE

Why does the program in Listing VII.15 assign `NULL` to the elements in `new_argv` after freeing them? This is paranoia based on long experience. After a pointer has been freed, you can no longer use the pointed-to data. The pointer is said to "dangle"; it doesn't point at anything useful. If you "`NULL` out" or "zero out" a pointer immediately after freeing it, your program can no longer get in trouble by using that pointer. True, you might go indirect on the null pointer instead, but that's something your debugger might be able to help you with immediately. Also, there still might be copies of the pointer that refer to the memory that has been deallocated; that's the nature of C. Zeroing out pointers after freeing them won't solve all problems; it can solve some. See FAQ VII.22 for a related discussion.

Cross Reference:

VII.16: Is it better to use `malloc()` or `calloc()`?

VII.20: What is the stack?

VII.21: What is the heap?

VII.22: What happens if you free a pointer twice?

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

VII.16: Is it better to use *malloc()* or *calloc()*?

Answer:

Both the `malloc()` and the `calloc()` functions are used to allocate dynamic memory. Each operates slightly different from the other. `malloc()` takes a size and returns a pointer to a chunk of memory at least that big:

```
void    *malloc( size_t size );
```

`calloc()` takes a number of elements, and the size of each, and returns a pointer to a chunk of memory at least big enough to hold them all:

```
void    *calloc( size_t numElements, size_t sizeOfElement );
```

There's one major difference and one minor difference between the two functions. The major difference is that `malloc()` doesn't initialize the allocated memory. The first time `malloc()` gives you a particular chunk of memory, the memory might be full of zeros. If memory has been allocated, freed, and reallocated, it probably has whatever junk was left in it. That means, unfortunately, that a program might run in simple cases (when memory is never reallocated) but break when used harder (and when memory is reused).

`calloc()` fills the allocated memory with all zero bits. That means that anything there you're going to use as a `char` or an `int` of any length, `signed` or `unsigned`, is guaranteed to be zero. Anything you're going to use as a pointer is set to all zero bits. That's usually a null pointer, but it's not guaranteed. (See FAQ VII.10.) Anything you're going to use as a `float` or `double` is set to all zero bits; that's a floating-point zero on some types of machines, but not on all.

The minor difference between the two is that `calloc()` returns an array of objects; `malloc()` returns one object. Some people use `calloc()` to make clear that they want an array. Other than initialization, most C programmers don't distinguish between

```
calloc( numElements, sizeOfElement)
```

and

```
malloc( numElements * sizeOfElement)
```

There's a nit, though. `malloc()` doesn't give you a pointer to an array. In theory (according to the ANSI C standard), pointer arithmetic works only within a single array. In practice, if any C compiler or interpreter were to enforce that theory, lots of existing C code would break. (There wouldn't be much use for `realloc()`, either, which also doesn't guarantee a pointer to an array.)

Don't worry about the array-ness of `calloc()`. If you want initialization to zeros, use `calloc()`; if not, use `malloc()`.

Cross Reference:

VII.7: Can you subtract pointers from each other? Why would you?

VII.8: When you add a value to a pointer, what is really added?

VII.10: Is `NULL` always equal to 0?

VII.17: How do you declare an array that will hold more than 64KB of data?

Answer:

The coward's answer is, you can't, portably. The ANSI/ISO C standard requires compilers to handle only single objects as large as (32KB – 1) bytes long.

Why is 64KB magic? It's the biggest number that needs more than 16 bits to represent it.

For some environments, to get an array that big, you just declare it. It works, no trouble. For others, you can't declare such an array, but you can allocate one off the heap, just by calling `malloc()` or `calloc()`.

On a PC compatible, the same limitations apply, and more. You need to use at least a large data model. (See the discussion at the beginning of the chapter.) You might also need to call "far" variants of `malloc()` or `calloc()`. For example, with Borland C and C++ compilers, you could write

```
far char *buffer = farmalloc(70000L);
```

Or with Microsoft C and C++ compilers, you could write

```
far char *buffer = fmalloc(70000L);
```

to allocate 70,000 bytes of memory into a buffer. (The `L` in `70000L` forces a `long` constant. An `int` constant might be only 15 bits long plus a sign bit, not big enough to store the value 70,000.)

Cross Reference:

VII.18: What is the difference between `far` and `near`?

VII.21: What is the heap?

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

VII.18: What is the difference between *far* and *near*?

Answer:

As described at the beginning of this chapter, some compilers for PC compatibles use two types of pointers. `near` pointers are 16 bits long and can address a 64KB range. `far` pointers are 32 bits long and can address a 1MB range.

`near` pointers operate within a 64KB segment. There's one segment for function addresses and one segment for data.

`far` pointers have a 16-bit base (the segment address) and a 16-bit offset. The base is multiplied by 16, so a `far` pointer is effectively 20 bits long. For example, if a `far` pointer had a segment of `0x7000` and an offset of `0x1224`, the pointer would refer to address `0x71224`. A `far` pointer with a segment of `0x7122` and an offset of `0x0004` would refer to the same address.

Before you compile your code, you must tell the compiler which memory model to use. If you use a small-code memory model, `near` pointers are used by default for function addresses. That means that all the functions need to fit in one 64KB segment. With a large-code model, the default is to use `far` function addresses. You'll get near pointers with a small data model, and far pointers with a large data model. These are just the defaults; you can declare variables and functions as explicitly `near` or `far`.

`far` pointers are a little slower. Whenever one is used, the code or data segment register needs to be swapped out. `far` pointers also have odd semantics for arithmetic and comparison. For example, the two `far` pointers in the preceding example point to the same address, but they would compare as different! If your program fits in a small-data, small-code memory model, your life will be easier. If it doesn't, there's not much you can do.

If it sounds confusing, it is. There are some additional, compiler-specific wrinkles. Check your compiler manuals for details.

Cross Reference:

VII.19: When should a `far` pointer be used?

VII.19: When should a *far* pointer be used?

Answer:

Sometimes you can get away with using a small memory model in most of a given program. (See FAQ VII.18.) There might be just a few things that don't fit in your small data and code segments.

When that happens, you can use explicit `far` pointers and function declarations to get at the rest of memory. A `far` function can be outside the 64KB segment most functions are shoehorned into for a small-code model. (Often, libraries are declared explicitly `far`, so they'll work no matter what code model the program uses.) A `far` pointer can refer to information outside the 64KB data segment. Typically, such pointers are used with `farmalloc()` and such, to manage a heap separate from where all the rest of the data lives.

If you use a small-data, large-code model, you should explicitly make your function pointers `far`.

Cross Reference:

VII.18: What is the difference between `far` and `near`?

VII.21: What is the heap?

VII.20: What is the stack?

Answer:

The stack is where all the functions' local (`auto`) variables are created. The stack also contains some information used to call and return from functions.

A "stack trace" is a list of which functions have been called, based on this information. When you start using a debugger, one of the first things you should learn is how to get a stack trace.

The stack is very inflexible about allocating memory; everything must be deallocated in exactly the reverse order it was allocated in. For implementing function calls, that is all that's needed. Allocating memory off the stack is extremely efficient. One of the reasons C compilers generate such good code is their heavy use of a simple stack.

There used to be a C function that any programmer could use for allocating memory off the stack. The memory was automatically deallocated when the calling function returned. This was a dangerous function to call; it's not available anymore.

Cross Reference:

VII.15: Can the size of an array be declared at runtime?

VII.21: What is the heap?

VII.21: What is the heap?

Answer:

The heap is where `malloc()`, `calloc()`, and `realloc()` get memory.

Getting memory from the heap is much slower than getting it from the stack. On the other hand, the heap is much more flexible than the stack. Memory can be allocated at any time and deallocated in any order. Such memory isn't deallocated automatically; you have to call `free()`.

Recursive data structures are almost always implemented with memory from the heap. Strings often come from there too, especially strings that could be very long at runtime.

If you can keep data in a local variable (and allocate it from the stack), your code will run faster than if you put the data on the heap. Sometimes you can use a better algorithm if you use the heap—faster, or more robust, or more flexible. It's a tradeoff.

If memory is allocated from the heap, it's available until the program ends. That's great if you remember to deallocate it when you're done. If you forget, it's a problem. A "memory leak" is some allocated memory that's no longer needed but isn't deallocated. If you have a memory leak inside a loop, you can use up all the memory on the heap and not be able to get any more. (When that happens, the allocation functions return a null pointer.) In some environments, if a program doesn't deallocate everything it allocated, memory stays unavailable even after the program ends.

NOTE

Memory leaks are hard to debug. Memory allocation tools can help find them.

Some programming languages don't make you deallocate memory from the heap. Instead, such memory is "garbage collected" automatically. This maneuver leads to some very serious performance issues. It's also a lot harder to implement. That's an issue for the people who develop compilers, not the people who buy them.

(Except that software that's harder to implement often costs more.) There are some garbage collection libraries for C, but they're at the bleeding edge of the state of the art.

Cross Reference:

VII.4: When is a null pointer used?

VII.20: What is the stack?

VII.22: What happens if you free a pointer twice?

Answer:

If you free a pointer, use it to allocate memory again, and free it again, of course it's safe.

NOTE

To be precise and accurate, the pointed-to memory, not the pointer itself, has been freed. Nothing about the pointer has changed. However, C programmers in a hurry (that's all of us, right?) will talk about "a freed pointer" to mean a pointer to freed memory.

If you free a pointer, the memory you freed might be reallocated. If that happens, you might get that pointer back. In this case, freeing the pointer twice is OK, but only because you've been lucky. The following example is silly, but safe:

```
#include <stdlib.h>

int
main(int argc, char** argv)
{
        char** new_argv1;
        char** new_argv2;
        new_argv1 = calloc(argc+1, sizeof(char*));
        free(new_argv1);    /* freed once */
        new_argv2 = (char**) calloc(argc+1, sizeof(char*));
        if (new_argv1 == new_argv2) {
                /*
                new_argv1 accidentally points to freeable memory
                */
                free(new_argv1);    /* freed twice */
        } else {
                free(new_argv2);
        }
        new_argv1 = calloc(argc+1, sizeof(char*));
        free(new_argv1);    /* freed once again */
        return 0;
}
```

In the preceding program, `new_argv1` is pointed to a chunk of memory big enough to copy the `argv` array, which is immediately freed. Then a chunk the same size is allocated, and its address is assigned to `new_argv2`. Because the first chunk was available again, `calloc` might have returned it again; in that case, `new_argv1` and

`new_argv2` have the same value, and it doesn't matter which variable you use. (Remember, it's the pointed-to memory that's freed, not the pointer variable.) Just for fun, `new_argv1` is pointed to allocated memory again, which is again freed. You can free a pointer as many times as you want; it's the memory you have to be careful about.

What if you free allocated memory, don't get it allocated back to you, and then free it again? Something like this:

```
void caller( ... )
{
        void *p;
        /* ... */
        callee( p );
        free( p );
}
void callee( void* p )
{
        /* ... */
        free( p );
        return;
}
```

In this example, the `caller()` function is passing `p` to the `callee()` function and then freeing `p`. Unfortunately, `callee()` is also freeing `p`. Thus, the memory that `p` points to is being freed twice. The ANSI/ISO C standard says this is undefined. Anything can happen. Usually, something very bad happens.

The memory allocation and deallocation functions could be written to keep track of what has been used and what has been freed. Typically, they aren't. If you `free()` a pointer, the pointed-to memory is assumed to have been allocated by `malloc()` or `calloc()` but not deallocated since then. `free()` calculates how big that chunk of memory was (see FAQ VII.26) and updates the data structures in the memory "arena." Even if the memory has been freed already, `free()` will assume that it wasn't, and it will blindly update the arena. This action is much faster than it would have been if `free()` had checked to see whether the pointer was OK to deallocate.

If something doesn't work right, your program is now in trouble. When `free()` updates the arena, it will probably write some information in a wrong place. You now have the fun of dealing with a wild pointer; see the description at the beginning of the chapter.

How can you avoid double deallocation? Write your code carefully, use memory allocation tools, or (preferably) do both.

Cross Reference:

VII.21: What is the heap?

VII.24: What is a "null pointer assignment" error? What are bus errors, memory faults, and core dumps?

VII.26: How does `free()` know how much memory to release?

VII.23: What is the difference between *NULL* and *NUL*?

Answer:

`NULL` is a macro defined in `<stddef.h>` for the null pointer.

`NUL` is the name of the first character in the ASCII character set. It corresponds to a zero value. There's no standard macro `NUL` in C, but some people like to define it.

NOTE

The digit 0 corresponds to a value of 80, decimal. Don't confuse the digit 0 with the value of `'\0'` (`NUL`)!

`NULL` can be defined as `((void*)0)`, `NUL` as `'\0'`. Both can also be defined simply as 0. If they're defined that way, they can be used interchangeably. That's a bad way to write C code. One is meant to be used as a pointer; the other, as a character. If you write your code so that the difference is obvious, the next person who has to read and change your code will have an easier job. If you write obscurely, the next person might have problems. Hint: Typically, the "next person" is the person who originally wrote the code. The time you save might be your own.

Cross Reference:

VII.3: What is a null pointer?

VII.24: What is a "null pointer assignment" error? What are bus errors, memory faults, and core dumps?

Answer:

These are all serious errors, symptoms of a wild pointer or subscript.

`Null pointer assignment` is a message you might get when an MS-DOS program finishes executing. Some such programs can arrange for a small amount of memory to be available "where the `NULL` pointer points to" (so to speak). If the program tries to write to that area, it will overwrite the data put there by the compiler. When the program is done, code generated by the compiler examines that area. If that data has been changed, the compiler-generated code complains with `null pointer assignment`.

This message carries only enough information to get you worried. There's no way to tell, just from a `null pointer assignment` message, what part of your program is responsible for the error. Some debuggers, and some compilers, can give you more help in finding the problem.

`Bus error: core dumped` and `Memory fault: core dumped` are messages you might see from a program running under UNIX. They're more programmer friendly. Both mean that a pointer or an array subscript was wildly out of bounds. You can get these messages on a read or on a write. They aren't restricted to null pointer problems.

The `core dumped` part of the message is telling you about a file, called `core`, that has just been written in your current directory. This is a dump of everything on the stack and in the heap at the time the program was running. With the help of a debugger, you can use the core dump to find where the bad pointer was used. That might not tell you why the pointer was bad, but it's a step in the right direction. If you don't have write permission in the current directory, you won't get a core file, or the `core dumped` message.

NOTE

Why "core"? The first UNIX systems ran on hardware that used magnetic cores, not silicon chips, for random access memory.

The same tools that help find memory allocation bugs can help find some wild pointers and subscripts, sometimes. The best such tools can find almost all occurrences of this kind of problem.

Cross Reference:

VII.3: What is a null pointer?

VII.25: How can you determine the size of an allocated portion of memory?

Answer:

You can't, really. `free()` can (see FAQ VII.26), but there's no way for your program to know the trick `free()` uses. Even if you disassemble the library and discover the trick, there's no guarantee the trick won't change with the next release of the compiler. Trying to second guess the compiler this way isn't just tricky, it's crazy.

Cross Reference:

VII.26: How does `free()` know how much memory to release?

VII.26: How does *free()* know how much memory to release?

Answer:

I could tell you, but then I'd have to kill you.

Seriously? There's no standard way. It can vary from compiler to compiler, even from version to version of the same compiler. `free()`, `malloc()`, `calloc()`, and `realloc()` are functions; as long as they all work the same way, they can work any way that works.

Most implementations take advantage of the same trick, though. When `malloc()` (or one of the other allocation functions) allocates a block of memory, it grabs a little more than it was asked to grab. `malloc()` doesn't return the address of the beginning of this block. Instead, it returns a pointer a little bit after that. At the very beginning of the block, before the address returned, `malloc()` stores some information, such as how big the block is. (If this information gets overwritten, you'll have wild pointer problems when you free the memory.)

There's no guarantee `free()` works this way. It could use a table of allocated addresses and their lengths. It could store the data at the end of the block (beyond the length requested by the call to `malloc()`). It could store a pointer rather than a count.

If you're desperate to hack a memory allocation library, write your own.

Cross Reference:

VII.25: How can you determine the size of an allocated portion of memory?

VII.27: Can math operations be performed on a *void* pointer?

Answer:

No. Pointer addition and subtraction are based on advancing the pointer by a number of elements. By definition, if you have a `void` pointer, you don't know what it's pointing to, so you don't know the size of what it's pointing to.

If you want pointer arithmetic to work on raw addresses, use character pointers.

NOTE

You can cast your `void*` to a `char*`, do the arithmetic, and cast it back to a `void*`.

Cross Reference:

VII.7: Can you subtract pointers from each other? Why would you?

VII.8: When you add a value to a pointer, what is really added?

VII.28: How do you print an address?

Answer:

The safest way is to use `printf()` (or `fprintf()` or `sprintf()`) with the `%P` specification. That prints a void pointer (`void*`). Different compilers might print a pointer with different formats. Your compiler will pick a format that's right for your environment.

If you have some other kind of pointer (not a `void*`) and you want to be very safe, cast the pointer to a `void*`:

```
printf( "%P\n", (void*) buffer );
```

There's no guarantee any integer type is big enough to store a pointer. With most compilers, an `unsigned long` is big enough. The second safest way to print an address (the value of a pointer) is to cast it to an `unsigned long`, then print that.

Cross Reference:

None.