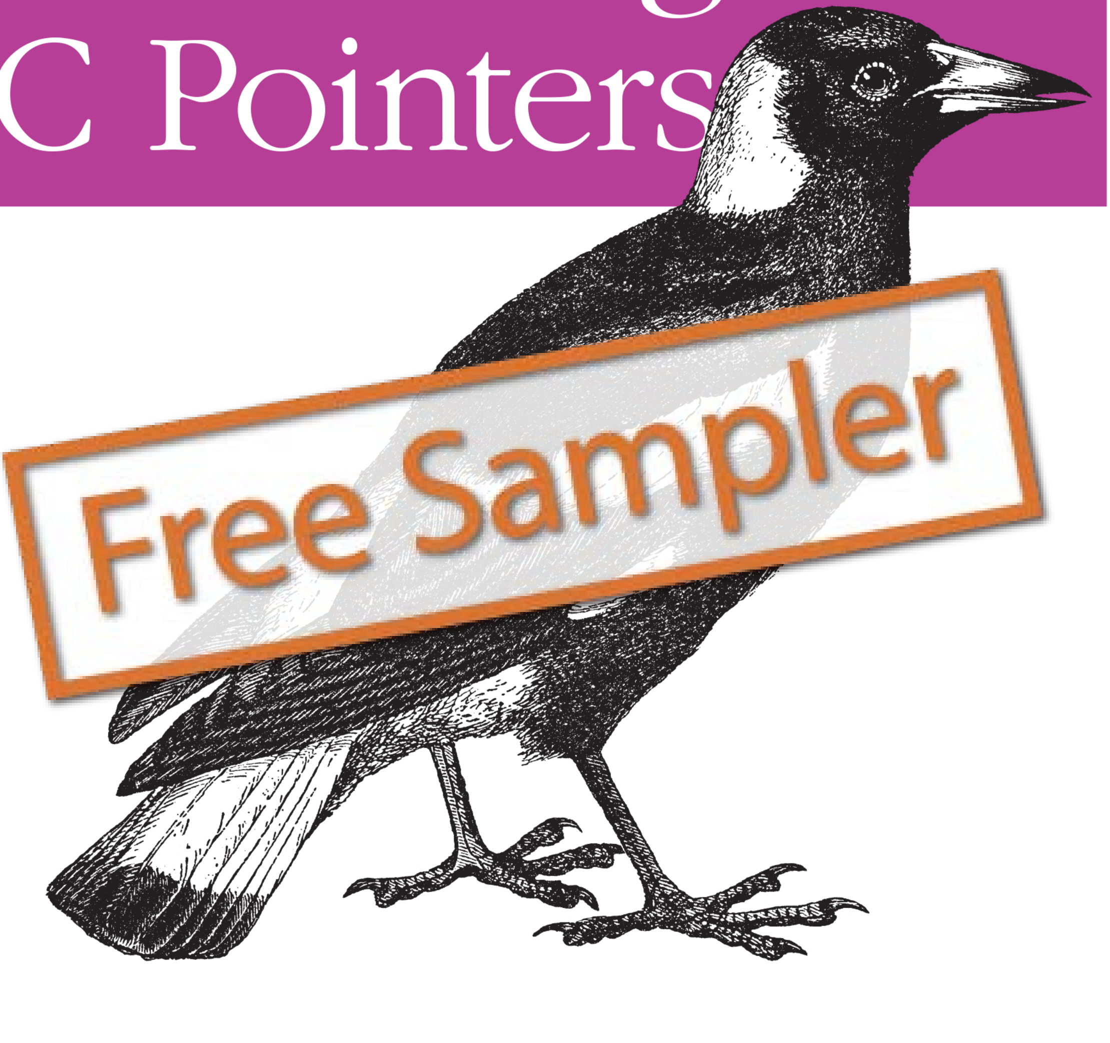
Core Techniques for Memory Management

Understanding and Using C Pointers



O'REILLY®

Richard Reese

Understanding and Using C Pointers

by Richard Reese

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Nathan Jepson
Production Editor: Rachel Steely
Copyeditor: Andre Barnett
Proofreader: Rachel Leach
Indexer: Potomac Indexing, LLC, Angela Howard
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Kara Ebrahim

May 2013: First Edition

Revision History for the First Edition:

2013-04-30: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449344184* for release details.

ISBN: 978-1-449-34418-4

[LSI]

Table of Contents

CHAPTER 1
Introduction

A solid understanding of pointers and the ability to effectively use them separates a novice C programmer from a more experienced one. Pointers pervade the language and provide much of its flexibility. They provide important support for dynamic memory allocation, are closely tied to array notation, and, when used to point to functions, add another dimension to flow control in a program.

Pointers have long been a stumbling block in learning C. The basic concept of a pointer is simple: it is a variable that stores the address of a memory location. The concept, however, quickly becomes complicated when we start applying pointer operators and try to discern their often cryptic notations. But this does not have to be the case. If we start simple and establish a firm foundation, then the advanced uses of pointers are not hard to follow and apply.

The key to comprehending pointers is understanding how memory is managed in a C program. After all, pointers contain addresses in memory. If we don't understand how memory is organized and managed, it is difficult to understand how pointers work. To address this concern, the organization of memory is illustrated whenever it is useful to explain a pointer concept. Once you have a firm grasp of memory and the ways it can be organized, understanding pointers becomes a lot easier.

This chapter presents an introduction to pointers, their operators, and how they interact with memory. The first section examines how they are declared, the basic pointer operators, and the concept of null. There are various types of "nulls" supported by C so a careful examination of them can be enlightening.

The second section looks more closely at the various memory models you will undoubtedly encounter when working with C. The model used with a given compiler and operating system environment affects how pointers are used. In addition, we closely examine various predefined types related to pointers and the memory models.

Pointer operators are covered in more depth in the next section, including pointer arithmetic and pointer comparisons. The last section examines constants and pointers. The numerous declaration combinations offer many interesting and often very useful possibilities.

Whether you are a novice C programmer or an experienced programmer, this book will provide you with a solid understanding of pointers and fill the gaps in your education. The experienced programmer will want to pick and choose the topics of interest. The beginning programmer should probably take a more deliberate approach.

Pointers and Memory

When a C program is compiled, it works with three types of memory:

Static/Global
: Statically declared variables are allocated to this type of memory. Global variables also use this region of memory. They are allocated when the program starts and remain in existence until the program terminates. While all functions have access to global variables, the scope of static variables is restricted to their defining function.

Automatic
: These variables are declared within a function and are created when a function is called. Their scope is restricted to the function, and their lifetime is limited to the time the function is executing.

Dynamic
: Memory is allocated from the heap and can be released as necessary. A pointer references the allocated memory. The scope is limited to the pointer or pointers that reference the memory. It exists until it is released. This is the focus of Chapter 2.

Table 1-1 summarizes the scope of and lifetime of variables used in these memory regions.

Table 1-1. Scope and lifetime

	Scope	Lifetime
Global	The entire file	The lifetime of the application
Static	The function it is declared within	The lifetime of the application
Automatic (local)	The function it is declared within	While the function is executing
Dynamic	Determined by the pointers that reference this memory	Until the memory is freed

Understanding these types of memory will enable you to better understand how pointers work. Most pointers are used to manipulate data in memory. Understanding how memory is partitioned and organized will clarify how pointers manipulate memory.

A pointer variable contains the address in memory of another variable, object, or function. An object is considered to be memory allocated using one of the memory allocation functions, such as the `malloc` function. A pointer is normally declared to be of a specific type depending on what it points to, such as a pointer to a `char`. The object may be any C data type such as integer, character, string, or structure. However, nothing inherent in a pointer indicates what type of data the pointer is referencing. A pointer only contains an address.

Why You Should Become Proficient with Pointers

Pointers have several uses, including:

- Creating fast and efficient code
- Providing a convenient means for addressing many types of problems
- Supporting dynamic memory allocation
- Making expressions compact and succinct
- Providing the ability to pass data structures by pointer without incurring a large overhead
- Protecting data passed as a parameter to a function

Faster and more efficient code can be written because pointers are closer to the hardware. That is, the compiler can more easily translate the operation into machine code. There is not as much overhead associated with pointers as might be present with other operators.

Many data structures are more easily implemented using pointers. For example, a linked list could be supported using either arrays or pointers. However, pointers are easier to use and map directly to a next or previous link. An array implementation requires array indexes that are not as intuitive or as flexible as pointers.

Figure 1-1 illustrates how this can be visualized using arrays and pointers for a linked list of employees. The lefthand side of the figure uses an array. The head variable indicates that the linked list's first element is at index 10 of the array. Each array's element contains a structure that represents an employee. The structure's `next` field holds the index in the array of the next employee. The shaded elements represent unused array elements.

The righthand side shows the equivalent representation using pointers. The head variable holds a pointer to the first employee's node. Each node holds employee data as well as a pointer to the next node in the linked list.

The pointer representation is not only clearer but also more flexible. The size of an array typically needs to be known when it is created. This will impose a restriction on the number of elements it can hold. The pointer representation does not suffer from this limitation as a new node can be dynamically allocated as needed.

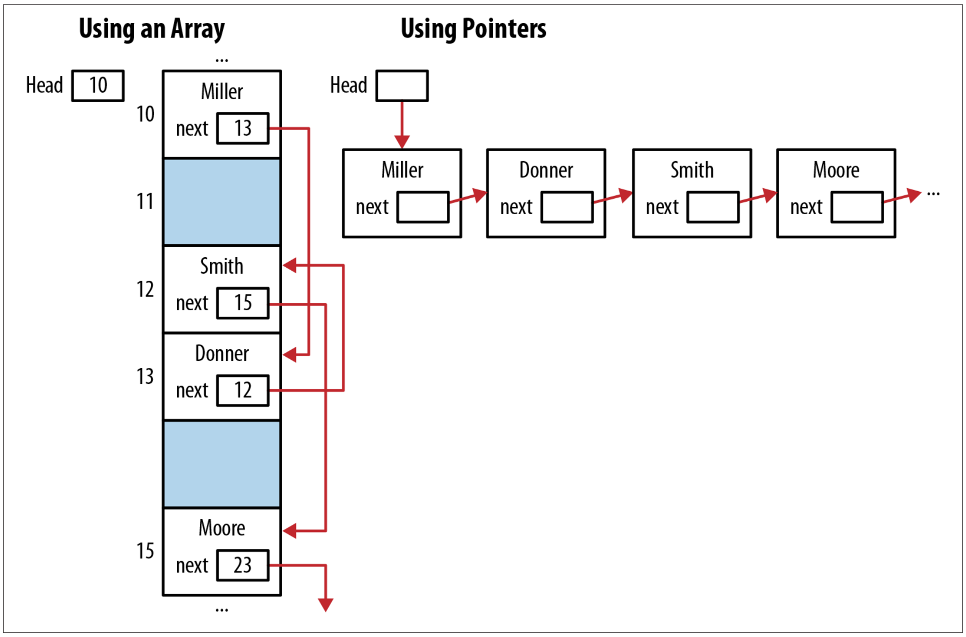


Figure 1-1. Array versus pointers representation of a linked list

Dynamic memory allocation is effected in C through the use of pointers. The `malloc` and `free` functions are used to allocate and release dynamic memory, respectively. Dynamic memory allocation enables variable-sized arrays and data structures, such as linked lists and queues. However, in the new C standard, C11, variable size arrays are supported.

Compact expressions can be very descriptive but can also be cryptic, as pointer notation is not always fully understood by many programmers. Compact expressions should address a specific need and not be cryptic just to be cryptic. For example, in the following sequence, the third character of the `names`' second element is displayed with two different `printf` functions. If this usage of pointers is confusing, don't worry—we will explain how dereferencing works in more detail in the section "Dereferencing a Pointer Using the Indirection Operator" on page 11. While the two approaches are equivalent and will display the character `n`, the simpler approach is to use array notation.

```c
char *names[] = {"Miller","Jones","Anderson"};
printf("%c\n",*(*(names+1)+2));
printf("%c\n",names[1][2]);
```

Pointers represent a powerful tool to create and enhance applications. On the downside, many problems can occur when using pointers, such as:

- Accessing arrays and other data structures beyond their bounds
- Referencing automatic variables after they have gone out of existence
- Referencing heap allocated memory after it has been released
- Dereferencing a pointer before memory has been allocated to it

These types of problems will be examined in more detail in Chapter 7.

The syntax and semantics of pointer usage are fairly well defined in the C specification. However, there are situations where the specification does not explicitly define pointer behavior. In these cases the behavior is defined to be either:

Implementation-defined
: Some specific, documented implementation is provided. An example of implementation-defined behavior is how the high-order bit is propagated in an integer shift right operation.

Unspecified
: Some implementation is provided but is not documented. An example of an unspecified behavior is the amount of memory allocated by the `malloc` function with an argument of zero. A list of unspecified behavior can be found at CERT Secure Coding Appendix DD.

Undefined
: There are no requirements imposed and anything can happen. An example of this is the value of a pointer deallocated by the `free` functions. A list of unspecified behavior can be found at CERT Secure Coding Appendix CC.

Sometimes there are locale-specific behaviors. These are usually documented by the compiler vendor. Providing locale-specific behavior allows the compiler-writer latitude in generating more efficient code.

Declaring Pointers

Pointer variables are declared using a data type followed by an asterisk and then the pointer variable's name. In the following example, an integer and a pointer to an integer are declared:

```
int num;
int *pi;
```

The use of white spaces around the asterisk is irrelevant. The following declarations are all equivalent:

```
int* pi;
int * pi;
int *pi;
int*pi;
```

The use of white space is a matter of user preference.

The asterisk declares the variable as a pointer. It is an overloaded symbol as it is also used for multiplication and dereferencing a pointer.

Figure 1-2 illustrates how memory would typically be allocated for the above declaration. Three memory locations are depicted by the three rectangles. The number to the left of each rectangle is its address. The name next to the address is the variable assigned to this location. The address 100 is used here for illustrative purposes. The actual address of a pointer, or any variable for that matter, is not normally known, nor is its value of interest in most applications. The three dots represent uninitialized memory.

Pointers to uninitialized memory can be a problem. If such a pointer is dereferenced, the pointer's content probably does not represent a valid address, and if it does, it may not contain valid data. An invalid address is one that the program is not authorized to access. This will result in the program terminating on most platforms, which is significant and can lead to a number of problems, as discussed in Chapter 7.

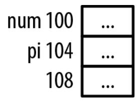


Figure 1-2. Memory diagram

The variables `num` and `pi` are located at addresses 100 and 104, respectively. Both are assumed to occupy four bytes. Both of these sizes will differ, depending on the system configuration as addressed in the section "Pointer Size and Types" on page 15. Unless otherwise noted, we will use four-byte integers for all of our examples.

In this book, we will use an address such as 100 to explain how pointers work. This will simplify the examples. When you execute the examples you will get different addresses, and these addresses can even change between repeated executions of the program.

There are several points to remember:

- The content of `pi` should eventually be assigned the address of an integer variable.
- These variables have not been initialized and thus contain garbage.
- There is nothing inherent to a pointer's implementation that suggests what type of data it is referencing or whether its contents are valid.
- However, the pointer type has been specified and the compiler will frequently complain when the pointer is not used correctly.

By garbage, we mean the memory allocation could contain any value. When memory is allocated it is not cleared. The previous contents could be anything. If the previous contents held a floating point number, interpreting it as an integer would likely not be useful. Even if it contained an integer, it would not likely be the right integer. Thus, its contents are said to hold garbage.

While a pointer may be used without being initialized, it may not always work properly until it has been initialized.

How to Read a Declaration

Now is a good time to suggest a way to read pointer declarations, which can make them easier to understand. The trick is to read them backward. While we haven't discussed pointers to constants yet, let's examine the following declaration:

```c
const int *pci;
```

Reading the declaration backward allows us to progressively understand the declaration (Figure 1-3).

1. `pci` is a variable	`const int *`**`pci;`**
2. `pci` is a pointer variable	`const int `**`*pci;`**
3. `pci` is a pointer variable to an integer	`const `**`int *pci;`**
4. `pci` is a pointer variable to a constant integer	**`const int *pci;`**

Figure 1-3. The backward declaration

Many programmers find that reading the declaration backward is less complex.

When working with complex pointer expressions, draw a picture of them, as we will do in many of our examples.

Address of Operator

The address of operator, `&`, will return its operand's address. We can initialize the `pi` pointer with the address of `num` using this operator as follows:

```
num = 0;
pi = &num;
```

The variable `num` is set to zero, and `pi` is set to point to the address of `num` as illustrated in Figure 1-4.

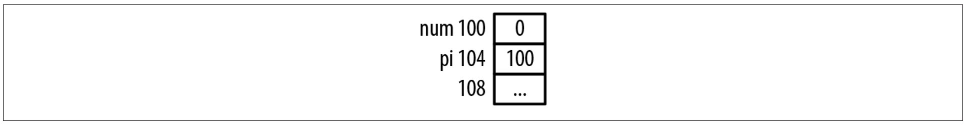


Figure 1-4. Memory assignments

We could have initialized `pi` to point to the address of `num` when the variables were declared as illustrated below:

```
int num;
int *pi = &num;
```

Using these declarations, the following statement will result in a syntax error on most compilers:

```
num = 0;
pi = num;
```

The error would appear as follows:

```
error: invalid conversion from 'int' to 'int*'
```

The variable `pi` is of type pointer to an integer and `num` is of type integer. The error message is saying we cannot convert an integer to a pointer to the data type integer.

Assignment of integers to a pointer will generally cause a warning or error.

Pointers and integers are not the same. They may both be stored using the same number of bytes on most machines, but they are not the same. However, it is possible to cast an integer to a pointer to an integer:

```
pi = (int *)num;
```

This will not generate a syntax error. When executed, though, the program may terminate abnormally when the program attempts to dereference the value at address zero. An address of zero is not always valid for use in a program on most operating systems. We will discuss this in more detail in the section "The Concept of Null" on page 11.

It is a good practice to initialize a pointer as soon as possible, as illustrated below:

```
int num;
int *pi;
pi = &num;
```

Displaying Pointer Values

Rarely will the variables we use actually have an address such as 100 and 104. However, the variable's address can be determined by printing it out as follows:

```
int num = 0;
int *pi = &num;

printf("Address of num: %d  Value: %d\n",&num, num);
printf("Address of pi: %d  Value: %d\n",&pi, pi);
```

When executed, you may get output as follows. We used real addresses in this example. Your addresses will probably be different:

```
Address of num: 4520836  Value: 0
Address of pi: 4520824  Value: 4520836
```

The `printf` function has a couple of other field specifiers useful when displaying pointer values, as summarized in Table 1-2.

Table 1-2. Field specifiers

Specifier	Meaning
%x	Displays the value as a hexadecimal number.
%o	Displays the value as an octal number.
%p	Displays the value in an implementation-specific manner; typically as a hexadecimal number.

Their use is demonstrated below:

```
printf("Address of pi: %d  Value: %d\n",&pi, pi);
printf("Address of pi: %x  Value: %x\n",&pi, pi);
```

```
    printf("Address of pi: %o  Value: %o\n",&pi, pi);
    printf("Address of pi: %p  Value: %p\n",&pi, pi);
```

This will display the address and contents of `pi`, as shown below. In this case, `pi` holds the address of `num`:

```
Address of pi: 4520824  Value: 4520836
Address of pi: 44fb78  Value: 44fb84
Address of pi: 21175570  Value: 21175604
Address of pi: 0044FB78  Value: 0044FB84
```

The `%p` specifier differs from `%x` as it typically displays the hexadecimal number in uppercase. We will use the `%p` specifier for addresses unless otherwise indicated.

Displaying pointer values consistently on different platforms can be challenging. One approach is to cast the pointer as a pointer to void and then display it using the `%p` format specifier as follows:

```
    printf("Value of pi: %p\n", (void*)pi);
```

Pointers to void is explained in "Pointer to void" on page 14. To keep our examples simple, we will use the `%p` specifier and not cast the address to a pointer to void.

Virtual memory and pointers

To further complicate displaying addresses, the pointer addresses displayed on a *virtual operating system* are not likely to be the real physical memory addresses. A virtual operating system allows a program to be split across the machine's physical address space. An application is split into pages/frames. These pages represent areas of main memory. The pages of the application are allocated to different, potentially noncontiguous areas of memory and may not all be in memory at the same time. If the operating system needs memory currently held by a page, the memory may be swapped out to secondary storage and then reloaded at a later time, frequently at a different memory location. These capabilities provide a virtual operating system with considerable flexibility in how it manages memory.

Each program assumes it has access to the machine's entire physical memory space. In reality, it does not. The address used by a program is a virtual address. The operating system maps the virtual address to a real physical memory address when needed.

This means code and data in a page may be in different physical locations as the program executes. The application's virtual addresses do not change; they are the addresses we see when we examine the contents of a pointer. The virtual addresses are transparently mapped to real addresses by the operating system.

The operating system handles all of this, and it is not something that the programmer has control over or needs to worry about. Understanding these issues explains the addresses returned by a program running in a virtual operating system.

Dereferencing a Pointer Using the Indirection Operator

The indirection operator, *, returns the value pointed to by a pointer variable. This is frequently referred to as dereferencing a pointer. In the following example, `num` and `pi` are declared and initialized:

```
    int num = 5;
    int *pi = &num;
```

The indirection operator is then used in the following statement to display 5, the value of `num`:

```
   printf("%p\n",*pi);   // Displays 5
```

We can also use the result of the dereference operator as an *lvalue*. The term lvalue refers to the operand found on the left side of the assignment operator. All `lvalues` must be modifiable since they are being assigned a value.

The following will assign 200 to the integer pointed to by `pi`. Since it is pointing to the variable `num`, 200 will be assigned to `num`. Figure 1-5 illustrates how memory is affected:

```
    *pi = 200;
    printf("%d\n",num);   // Displays 200
```

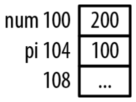


Figure 1-5. Memory assigned using dereference operator

Pointers to Functions

A pointer can be declared to point to a function. The declaration notation is a bit cryptic. The following illustrates how to declare a pointer to a function. The function is passed void and returns void. The pointer's name is `foo`:

```
   void (*foo)();
```

A *pointer* to a function is a rich topic area and will be covered in more detail in Chapter 3.

The Concept of Null

The concept of null is interesting and sometimes misunderstood. Confusion can occur because we often deal with several similar, yet distinct concepts, including:

- The null concept
- The null pointer constant
- The `NULL` macro

- The ASCII `NUL`
- A null string
- The null statement

When `NULL` is assigned to a pointer, it means the pointer does not point to anything. The null concept refers to the idea that a pointer can hold a special value that is not equal to another pointer. It does not point to any area of memory. Two null pointers will always be equal to each other. There can be a null pointer type for each pointer type, such as a pointer to a character or a pointer to an integer, although this is uncommon.

The null concept is an abstraction supported by the null pointer constant. This constant may or may not be a constant zero. A C programmer need not be concerned with their actual internal representation.

The NULL macro is a constant integer zero cast to a pointer to void. In many libraries, it is defined as follows:

```
#define NULL    ((void *)0)
```

This is what we typically think of as a null pointer. Its definition frequently can be found within several different header files, including *stddef.h*, *stdlib.h*, and *stdio.h*.

If a nonzero bit pattern is used by the compiler to represent null, then it is the compiler's responsibility to ensure all uses of `NULL` or 0 in a pointer context are treated as null pointers. The actual internal representation of null is implementation-defined. The use of `NULL` and 0 are language-level symbols that represent a null pointer.

The ASCII `NUL` is defined as a byte containing all zeros. However, this is not the same as a null pointer. A string in C is represented as a sequence of characters terminated by a zero value. The null string is an empty string and does not contain any characters. Finally, the null statement consists of a statement with a single semicolon.

As we will see, a null pointer is a very useful feature for many data structure implementations, such as a linked list where it is often used to mark the end of the list.

If the intent was to assign the null value to `pi`, we use the NULL type as follows:

```
    pi = NULL;
```

A null pointer and an uninitialized pointer are different. An uninitialized pointer can contain any value, whereas a pointer containing NULL does not reference any location in memory.

Interestingly, we can assign a zero to a pointer, but we cannot assign any other integer value. Consider the following assignment operations:

```
pi = 0;
pi = NULL;
pi = 100;   // Syntax error
pi = num;   // Syntax error
```

A pointer can be used as the sole operand of a logical expression. For example, we can test to see whether the pointer is set to NULL using the following sequence:

```
if(pi) {
   // Not NULL
} else {
   // Is NULL
}
```

Either of the two following expressions are valid but are redundant. It may be clearer, but explicit comparison to NULL is not necessary.

If pi has been assigned a NULL value in this context, then it will be interpreted as the binary zero. Since this represents false in C, the else clause will be executed if pi contains NULL.

```
if(pi == NULL) ...
if(pi != NULL) ...
```

A null pointer should never be dereferenced because it does not contain a valid address. When executed it will result in the program terminating.

To NULL or not to NULL

Which is better form: using NULL or using 0 when working with pointers? Either is perfectly acceptable; the choice is one of preference. Some developers prefer to use NULL because it is a reminder that we are working with pointers. Others feel this is unnecessary because the zero is simply hidden.

However, NULL should not be used in contexts other than pointers. It might work some of the time, but it is not intended to be used this way. It can definitely be a problem when used in place of the ASCII NUL character. This character is not defined in any standard C header file. It is equivalent to the character literal, '\0', which evaluates to the decimal value zero.

The meaning of zero changes depending on its context. It might mean the integer zero in some contexts, and it might mean a null pointer in a different context. Consider the following example:

```
int num;
int *pi = 0;   // Zero refers to the null pointer,NULL
pi = &num;
*pi = 0;        // Zero refers to the integer zero
```

We are accustomed to overloaded operators, such as the asterisk used to declare a pointer, to dereference a pointer, or to multiply. The zero is also overloaded. We may find this discomforting because we are not used to overloading operands.

Pointer to void

A pointer to void is a general-purpose pointer used to hold references to any data type. An example of a pointer to void is shown below:

```
void *pv;
```

It has two interesting properties:

- A pointer to void will have the same representation and memory alignment as a pointer to `char`.

- A pointer to void will never be equal to another pointer. However, two void pointers assigned a `NULL` value will be equal.

Any pointer can be assigned to a pointer to void. It can then be cast back to its original pointer type. When this happens the value will be equal to the original pointer value. This is illustrated in the following sequence, where a pointer to `int` is assigned to a pointer to void and then back to a pointer to `int`:

```
int num;
int *pi = &num;
printf("Value of pi: %p\n", pi);
void* pv = pi;
pi = (int*) pv;
printf("Value of pi: %p\n", pi);
```

When this sequence is executed as shown below, the pointer address is the same:

```
Value of pi: 100
Value of pi: 100
```

Pointers to void are used for data pointers, not function pointers. In "Polymorphism in C" on page 194, we will reexamine the use of pointers to void to address polymorphic behavior.

> Be careful when using pointers to void. If you cast an arbitrary pointer to a pointer to void, there is nothing preventing you from casting it to a different pointer type.

The sizeof operator can be used with a pointer to void. However, we cannot use the operator with void as shown below:

```
size_t size = sizeof(void*);   // Legal
size_t size = sizeof(void);    // Illegal
```

The size_t is a data type used for sizes and is discussed in the section "Predefined Pointer-Related Types" on page 16.

Global and static pointers

If a pointer is declared as global or static, it is initialized to NULL when the program starts. An example of a global and static pointer follows:

```
int *globalpi;

void foo() {
   static int *staticpi;
   ...
}

int main() {
   ...
}
```

Figure 1-6 illustrates this memory layout. Stack frames are pushed onto the stack, and the heap is used for dynamic memory allocation. The region above the heap is used for static/global variables. This is a conceptual diagram only. Static and global variables are frequently placed in a data segment separate from the data segment used by the stack and heap. The stack and heap are discussed in "Program Stack and Heap" on page 58.

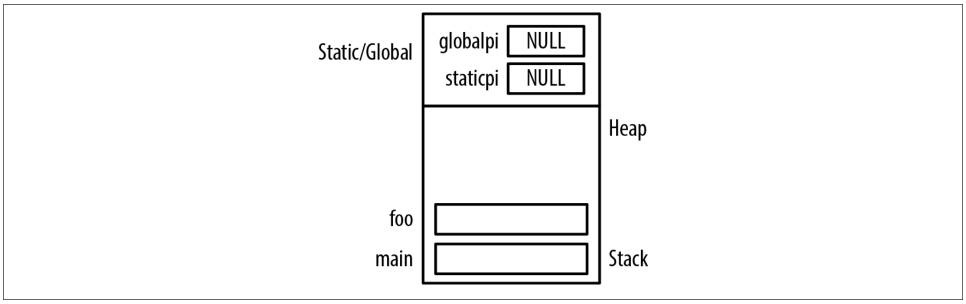


Figure 1-6. Memory allocation for global and static pointers

Pointer Size and Types

Pointer size is an issue when we become concerned about application compatibility and portability. On most modern platforms, the size of a pointer to data is normally the same regardless of the pointer type. A pointer to a `char` has the same size as a pointer to a structure. While the C standard does not dictate that size be the same for all data types, this is usually the case. However, the size of a pointer to a function may be different from the size of a pointer to data.

The size of a pointer depends on the machine in use and the compiler. For example, on modern versions of Windows the pointer is 32 or 64 bits in length. For DOS and Windows 3.1 operating systems, pointers were 16 or 32 bits in length.

Memory Models

The introduction of 64-bit machines has made more apparent the differences in the size of memory allocated for data types. With different machines and compilers come different options for allocating space to C primitive data types. A common notation used to describe different data models is summarized below:

```
I In L Ln LL LLn P Pn
```

Each capital letter corresponds to an integer, long, or pointer. The lowercase letters represent the number of bits allocated for the data type. Table 1-3[1] summarizes these models, where the number is the size in bits:

Table 1-3. Machine memory models

C Data Type	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
_int32		32			
int	32	64	32	32	16
long	64	64	32	32	32
long long			64		
pointer	64	64	64	32	32

The model depends on the operating system and compiler. More than one model may be supported on the same operating system; this is often controlled through compiler options.

Predefined Pointer-Related Types

Four predefined types are frequently used when working with pointers. They include:

1. Adapted from *http://en.wikipedia.org/wiki/64-bit*.

`size_t`
: Created to provide a safe type for sizes

`ptrdiff_t`
: Created to handle pointer arithmetic

`intptr_t` *and* `uintprt_t`
: Used for storing pointer addresses

In the following sections, we will illustrate the use of each type with the exception of `ptrdiff_t`, which will be discussed in the section "Subtracting two pointers" on page 24.

Understanding size_t

The type `size_t` represents the maximum size any object can be in C. It is an unsigned integer since negative numbers do not make sense in this context. Its purpose is to provide a portable means of declaring a size consistent with the addressable area of memory available on a system. The `size_t` type is used as the return type for the `sizeof` operator and as the argument to many functions, including `malloc` and `strlen`, among others.

It is good practice to use `size_t` when declaring variables for sizes such as the number of characters and array indexes. It should be used for loop counters, indexing into arrays, and sometimes for pointer arithmetic.

The declaration of `size_t` is implementation-specific. It is found in one or more standard headers, such as *stdio.h* and *stdlib.h*, and it is typically defined as follows:

```
#ifndef __SIZE_T
#define __SIZE_T
typedef unsigned int size_t;
#endif
```

The define directives ensure it is only defined once. The actual size will depend on the implementation. Typically, on a 32-bit system, it will be 32 bits in length, while on a 64-bit system it will be 64 bits in length. Normally, the maximum possible value for `size_t` is `SIZE_MAX`.

Usually `size_t` can be used to store a pointer, but it is not a good idea to assume `size_t` is the same size as a pointer. As we will see in "Using the sizeof operator with pointers" on page 18, `intptr_t` is a better choice.

Be careful when printing values defined as `size_t`. These are unsigned values, and if you choose the wrong format specifier, you'll get unreliable results. The recommended format specifier is `%zu`. However, this is not always available. As an alternative, consider using `%u` or `%lu`.

Consider the following example, where we define a variable as a `size_t` and then display it using two different format specifiers:

```
    size_t sizet = -5;
    printf("%d\n",sizet);
    printf("%zu\n",sizet);
```

Since a variable of type `size_t` is intended for use with positive integers, using a negative value can present problems. When we assign it a negative number and use the `%d` and then the `%zu` format specifiers, we get the following output:

```
-5
4294967291
```

The `%d` field interprets `size_t` as a signed integer. It displays a –5 because it holds a –5. The `%zu` field formats `size_t` as an unsigned integer. When –5 is interpreted as a signed integer, its high-order bit is set to one, indicating that the integer is negative. When interpreted as an unsigned number, the high-order bit is interpreted as a large power of 2. This is why we saw the large integer when we used the `%zu` field specifier.

A positive number will be displayed properly as shown below:

```
    sizet = 5;
    printf("%d\n",sizet);     // Displays 5
    printf("%zu\n",sizet);    // Displays 5
```

Since `size_t` is unsigned, always assign a positive number to a variable of that type.

Using the sizeof operator with pointers

The `sizeof` operator can be used to determine the size of a pointer. The following displays the size of a pointer to `char`:

```
    printf("Size of *char: %d\n",sizeof(char*));
```

The output follows:

```
Size of *char: 4
```

Always use the `sizeof` operator when the size of a pointer is needed.

The size of a function pointer can vary. Usually, it is consistent for a given operating system and compiler combination. Many compilers support the creation of a 32-bit or 64-bit application. It is possible that the same program, compiled with different options, will use different pointer sizes.

On a Harvard architecture, the code and data are stored in different physical memory. For example, the Intel MCS-51 (8051) microcontroller is a Harvard machine. Though Intel no longer manufactures the chip, there are many binary compatible derivatives available and in use today. The Small Device C Complier (SDCC) supports this type of processor. Pointers on this machine can range from 1 to 4 bytes in length. Thus, the size of a pointer should be determined when needed, as its size is not consistent in this type of environment.

Using intptr_t and uintptr_t

The types `intptr_t` and `uintptr_t` are used for storing pointer addresses. They provide a portable and safe way of declaring pointers, and will be the same size as the underlying pointer used on the system. They are useful for converting pointers to their integer representation.

The type `uintptr_t` is the unsigned version of `intptr_t`. For most operations `intptr_t` is preferred. The type `uintptr_t` is not as flexible as `intptr_t`. The following illustrates how to use `intptr_t`:

```
int num;
intptr_t *pi = &num;
```

If we try to assign the address of an integer to a pointer of type `uintptr_t` as follows, we will get a syntax error:

```
uintptr_t *pu = &num;
```

The error follows:

```
error: invalid conversion from 'int*' to
       'uintptr_t* {aka unsigned int*}' [-fpermissive]
```

However, performing the assignment using a cast will work:

```
intptr_t *pi = &num;
uintptr_t *pu = (uintptr_t*)&num;
```

We cannot use `uintptr_t` with other data types without casting:

```
char c;
uintptr_t *pc = (uintptr_t*)&c;
```

These types should be used when portability and safety are an issue. However, we will not use them in our examples to simplify their explanations.

Avoid casting a pointer to an integer. In the case of 64-bit pointers, information will be lost if the integer was only four bytes.

Early Intel processors used a 16-bit segmented architecture where near and far pointers were relevant. In today's virtual memory architecture, they are no longer a factor. The far and near pointers were extensions to the C standard to support segmented architecture on early Intel processors. Near pointers were only able to address about 64KB of memory at a time. Far pointers could address up to 1MB of memory but were slower than near pointers. Huge pointers were far pointers normalized so they used the highest possible segment for the address.

Pointer Operators

There are several operators available for use with pointers. So far we have examined the dereference and address-of operators. In this section, we will look closely into pointer arithmetic and comparisons. Table 1-4 summarizes the pointer operators.

Table 1-4. Pointer operators

Operator	Name	Meaning
*		Used to declare a pointer
*	Dereference	Used to dereference a pointer
->	Point-to	Used to access fields of a structure referenced by a pointer
+	Addition	Used to increment a pointer
-	Subtraction	Used to decrement a pointer
== !=	Equality, inequality	Compares two pointers
> >= < <=	Greater than, greater than or equal, less than, less than or equal	Compares two pointers
(data type)	Cast	To change the type of pointer

Pointer Arithmetic

Several arithmetic operations are performed on pointers to data. These include:

- Adding an integer to a pointer
- Subtracting an integer from a pointer
- Subtracting two pointers from each other
- Comparing pointers

These operations are not always permitted on pointers to functions.

Adding an integer to a pointer

This operation is very common and useful. When we add an integer to a pointer, the amount added is the product of the integer times the number of bytes of the underlying data type.

The size of primitive data types can vary from system to system, as discussed in "Memory Models" on page 16. However, Table 1-5 shows the common sizes found in most systems. Unless otherwise noted, these values will be used for the examples in this book.

Table 1-5. Data type sizes

Data Type	Size in Bytes
byte	1
char	1
short	2
int	4
long	8
float	4
double	8

To illustrate the effects of adding an integer to a pointer, we will use an array of integers, as shown below. Each time one is added to `pi`, four is added to the address. The memory allocated for these variables is illustrated in Figure 1-7. Pointers are declared with data types so that these sorts of arithmetic operations are possible. Knowledge of the data type size allows the automatic adjustment of the pointer values in a portable fashion:

```
int vector[] = {28, 41, 7};
int *pi = vector;       // pi: 100

printf("%d\n",*pi);     // Displays 28
pi += 1;                // pi: 104
printf("%d\n",*pi);     // Displays 41
pi += 1;                // pi: 108
printf("%d\n",*pi);     // Displays 7
```

When an array name is used by itself, it returns the address of an array, which is also the address of the first element of the array:

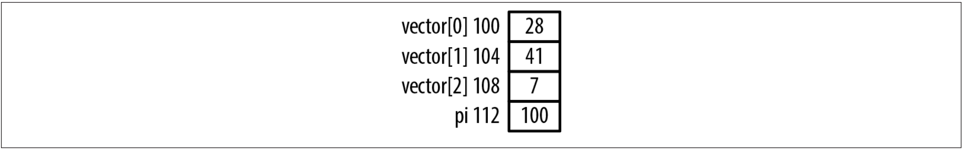


Figure 1-7. Memory allocation for vector

In the following sequence, we add three to the pointer. The variable `pi` will contain the address 112, the address of `pi`:

```
 pi = vector;
 pi += 3;
```

The pointer is pointing to itself. This is not very useful but illustrates the need to be careful when performing pointer arithmetic. Accessing memory past the end of an array is a dangerous thing to do and should be avoided. There is no guarantee that the memory access will be a valid variable. It is very easy to compute an invalid or useless address.

The following declarations will be used to illustrate the addition operation performed with a `short` and then a `char` data type:

```
short s;
short *ps = &s;
char c;
char *pc = &c;
```

Let's assume memory is allocated as shown in Figure 1-8. The addresses used here are all on a four-byte word boundary. Real addresses may be aligned on different boundaries and in a different order.

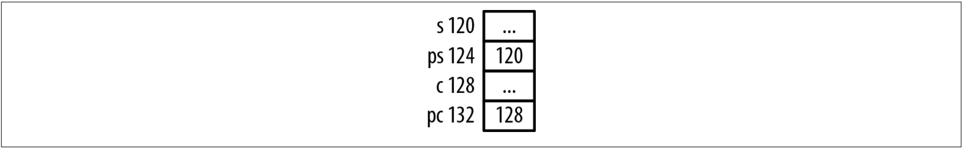


Figure 1-8. Pointers to short and char

The following sequence adds one to each pointer and then displays their contents:

```
printf("Content of ps before: %d\n",ps);
ps = ps + 1;
printf("Content of ps after:  %d\n",ps);

printf("Content of pc before: %d\n",pc);
pc = pc + 1;
printf("Content of pc after:  %d\n",pc);
```

When executed, you should get output similar to the following:

```
Content of ps before: 120
Content of ps after:  122
Content of pc before: 128
Address of pc after:  129
```

The `ps` pointer is incremented by two because the size of a `short` is two bytes. The `pc` pointer is incremented by one because its size is one byte. Again, these addresses may not contain useful information.

Pointers to void and addition

Most compilers allow arithmetic to be performed on a pointer to void as an extension. Here we will assume the size of a pointer to void is four. However, trying to add one to a pointer to void may result in a syntax error. In the following code snippet, we declare and attempt to add one to the pointer:

```
int num = 5;
void *pv = &num;
printf("%p\n",pv);
pv = pv+1;         //Syntax warning
```

The resulting warning follows:

```
warning: pointer of type 'void *' used in arithmetic [-Wpointerarith]
```

Since this is not standard C, the compiler issued a warning. However, the resulting address contained in `pv` will be incremented by four bytes.

Subtracting an integer from a pointer

Integer values can be subtracted from a pointer in the same way they are added. The size of the data type times the integer increment value is subtracted from the address. To illustrate the effects of subtracting an integer from a pointer, we will use an array of integers as shown below. The memory created for these variables is illustrated in Figure 1-7.

```
int vector[] = {28, 41, 7};
int *pi = vector + 2;  // pi: 108

printf("%d\n",*pi);    // Displays 7
pi--;                  // pi: 104
printf("%d\n",*pi);    // Displays 41
pi--;                  // pi: 100
printf("%d\n",*pi);    // Displays 28
```

Each time one is subtracted from `pi`, four is subtracted from the address.

Subtracting two pointers

When one pointer is subtracted from another, we get the difference between their addresses. This difference is not normally very useful except for determining the order of elements in an array.

The difference between the pointers is the number of "units" by which they differ. The difference's sign depends on the order of the operands. This is consistent with pointer addition where the number added is the pointer's data type size. We use "unit" as the operand. In the following example, we declare an array and pointers to the array's elements. We then take their difference:

```
int vector[] = {28, 41, 7};
int *p0 = vector;
int *p1 = vector+1;
int *p2 = vector+2;

printf("p2-p0:  %d\n",p2-p0);    // p2-p0:  2
printf("p2-p1:  %d\n",p2-p1);    // p2-p1:  1
printf("p0-p1:  %d\n",p0-p1);    // p0-p1:  -1
```

In the first `printf` statement, we find the difference between the positions of the array's last element and its first element is 2. That is, their indexes differ by 2. In the last `printf` statement, the difference is a –1, indicating that `p0` immediately precedes the element pointed to by `p1`. Figure 1-9 illustrates how memory is allocated for this example.

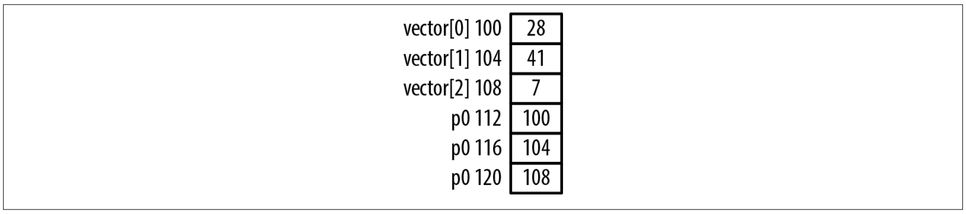


Figure 1-9. Subtracting two pointers

The type `ptrdiff_t` is a portable way to express the difference between two pointers. In the previous example, the result of subtracting two pointers is returned as a `ptrdiff_t` type. Since pointer sizes can differ, this type simplifies the task of working with their differences.

Don't confuse this technique with using the dereference operator to subtract two numbers. In the following example, we use pointers to determine the difference between the value stored in the array's first and second elements:

```
printf("*p0-*p1:  %d\n",*p0-*p1);  //  *p0-*p1:  -13
```

Comparing Pointers

Pointers can be compared using the standard comparison operators. Normally, comparing pointers is not very useful. However, when comparing pointers to elements of an array, the comparison's results can be used to determine the relative ordering of the array's elements.

We will use the vector example developed in the section "Subtracting two pointers" on page 24 to illustrate the comparison of pointers. Several comparison operators are applied to the pointers, and their results are displayed as 1 for true and 0 for false:

```
int vector[] = {28, 41, 7};
int *p0 = vector;
int *p1 = vector+1;
int *p2 = vector+2;

printf("p2>p0:  %d\n",p2>p0);    // p2>p0:  1
printf("p2<p0:  %d\n",p2<p0);    // p2<p0:  0
printf("p0>p1:  %d\n",p0>p1);    // p0>p1:  0
```

Common Uses of Pointers

Pointers can be used in a variety of ways. In this section, we will examine different ways of using pointers, including:

- Multiple levels of indirection
- Constant pointers

Multiple Levels of Indirection

Pointers can use different levels of indirection. It is not uncommon to see a variable declared as a pointer to a pointer, sometimes called a *double pointer*. A good example of this is when program arguments are passed to the `main` function using the traditionally named `argc` and `argv` parameters. This is discussed in more detail in Chapter 5.

The example below uses three arrays. The first array is an array of strings used to hold a list of book titles:

```
char *titles[] = {"A Tale of Two Cities",
       "Wuthering Heights","Don Quixote",
       "Odyssey","Moby-Dick","Hamlet",
       "Gulliver's Travels"};
```

Two additional arrays are provided whose purpose is to maintain a list of the "best books" and English books. Instead of holding copies of the titles, they will hold the address of a title in the `titles` array. Both arrays will need to be declared as a pointer to a pointer to a `char`. The array's elements will hold the addresses of the `titles` array's elements. This will avoid having to duplicate memory for each title and results in a single location for titles. If a title needs to be changed, then the change will only have to be performed in one location.

The two arrays are declared below. Each array element contains a pointer that points to a second pointer to `char`:

```
char **bestBooks[3];
char **englishBooks[4];
```

The two arrays are initialized and one of their elements is displayed, as shown below. In the assignment statements, the value of the righthand side is calculated by applying the subscripts first, followed by the address-of operator. For example, the second statement assigns the address of the fourth element of `titles` to the second element of `bestBooks`:

```
bestBooks[0] = &titles[0];
bestBooks[1] = &titles[3];
bestBooks[2] = &titles[5];

englishBooks[0] = &titles[0];
englishBooks[1] = &titles[1];
englishBooks[2] = &titles[5];
englishBooks[3] = &titles[6];

printf("%s\n",*englishBooks[1]);    // Wuthering Heights
```

Memory is allocated for this example as shown in Figure 1-10.

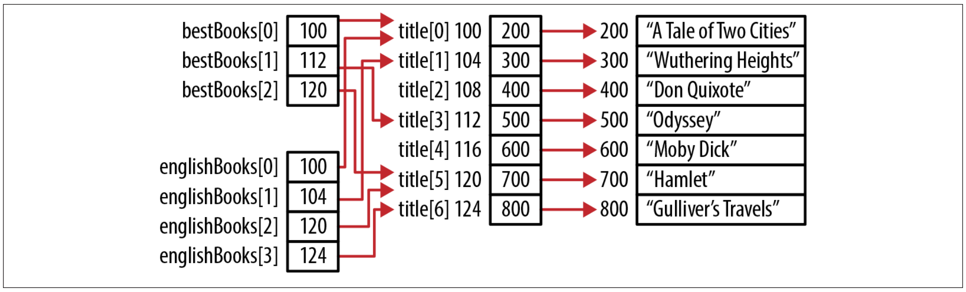


Figure 1-10. Pointers to pointers

Using multiple levels of indirection provides additional flexibility in how code can be written and used. Certain types of operations would otherwise be more difficult. In this example, if the address of a title changes, it will only require modification to the `title` array. We would not have to modify the other arrays.

There is not an inherent limit on the number of levels of indirection possible. Of course, using too many levels of indirection can be confusing and hard to maintain.

Constants and Pointers

Using the `const` keyword with pointers is a rich and powerful aspect of C. It provides different types of protections for different problem sets. Of particular power and usefulness is a pointer to a constant. In Chapters 3 and 5, we will see how this can protect users of a function from modification of a parameter by the function.

Pointers to a constant

A pointer can be defined to point to a constant. This means the pointer cannot be used to modify the value it is referencing. In the following example, an integer and an integer constant are declared. Next, a pointer to an integer and a pointer to an integer constant are declared and then initialized to the respective integers:

```
int num = 5;
const int limit = 500;
int *pi;                // Pointer to an integer
const int *pci;         // Pointer to a constant integer

pi = &num;
pci = &limit;
```

This is illustrated in Figure 1-11.

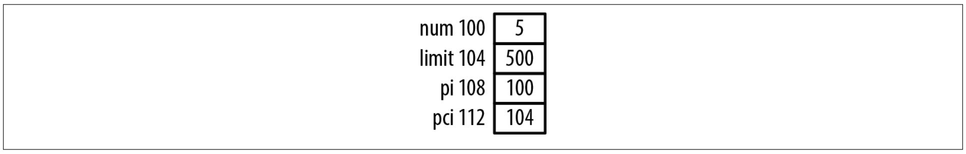


Figure 1-11. Pointer to a constant integer

The following sequence will display the address and value of these variables:

```
printf("  num - Address: %p  value: %d\n",&num, num);
printf("limit - Address: %p  value: %d\n",&limit, limit);
printf("   pi - Address: %p  value: %p\n",&pi, pi);
printf("  pci - Address: %p  value: %p\n",&pci, pci);
```

When executed, this sequence will produce values similar to the following:

```
  num - Address: 100  value: 5
limit - Address: 104  value: 500
   pi - Address: 108  value: 100
  pci - Address: 112  value: 104
```

Dereferencing a constant pointer is fine if we are simply reading the integer's value. Reading is a perfectly legitimate and necessary capability, as shown below:

```
printf("%d\n", *pci);
```

We cannot dereference a constant pointer to change what the pointer references, but we can change the pointer. The pointer value is not constant. The pointer can be changed to reference another constant integer or a simple integer. Doing so will not be a problem. The declaration simply limits our ability to modify the referenced variable through the pointer.

This means the following assignment is legal:

```
    pci = &num;
```

We can dereference pci to read it; however, we cannot dereference it to modify it.

Consider the following assignment:

```
    *pci = 200;
```

This will result in the following syntax error:

```
'pci' : you cannot assign to a variable that is const
```

The pointer thinks it is pointing to a constant integer; therefore, it does allow the modification of the integer using the pointer. We can still modify num using its name. We just can't use pci to modify it.

Conceptually, a constant pointer can also be visualized as shown in Figure 1-12. The clear boxes represent variables that can be changed. The shaded boxes represent variables that cannot be changed. The shaded box pointed to by pci cannot be changed using pci. The dashed line indicates that the pointer can reference that data type. In the previous example, pci pointed to limit.

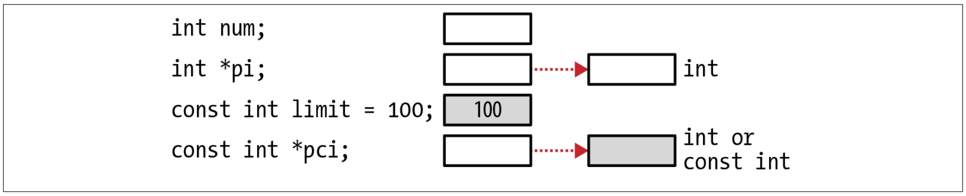


Figure 1-12. Pointer to a constant

The declaration of pci as a pointer to a constant integer means:

- pci can be assigned to point to different constant integers
- pci can be assigned to point to different nonconstant integers
- pci can be dereferenced for reading purposes
- pci cannot be dereferenced to change what it points to

The order of the type and the const keyword is not important. The following are equivalent:

```
const int *pci;
int const *pci;
```

Constant pointers to nonconstants

We can also declare a constant pointer to a nonconstant. When we do this, it means that while the pointer cannot be changed, the data pointed to can be modified. An example of such a pointer follows:

```
int num;
int *const cpi = &num;
```

With this declaration:

- cpi must be initialized to a nonconstant variable
- cpi cannot be modified
- The data pointed to by cpi can be modified

Conceptually, this type of pointer can be visualized as shown in Figure 1-13.

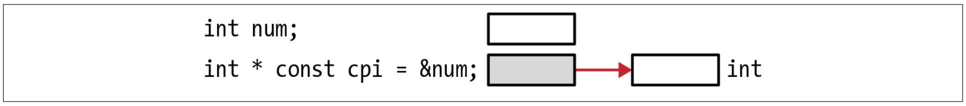


Figure 1-13. Constant pointers to nonconstants

It is possible to dereference cpi and assign a new value to whatever cpi is referencing. The following are two valid assignments:

```
*cpi = limit;
*cpi = 25;
```

However, if we attempt to initialize cpi to the constant limit as shown below, we will get a warning:

```
const int limit = 500;
int *const cpi = &limit;
```

The warning will appear as follows:

```
warning: initialization discards qualifiers from pointer target type
```

If cpi referenced the constant limit, the constant could be modified. This is not desirable. We generally prefer constants to remain constant.

Once an address has been assigned to `cpi`, we cannot assign a new value to `cpi` as shown below:

```
int num;
int age;
int *const cpi = &num;
cpi = &age;
```

The error message generated is shown below:

```
'cpi' : you cannot assign to a variable that is const
```

Constant pointers to constants

A constant pointer to a constant is an infrequently used pointer type. The pointer cannot be changed, and the data it points to cannot be changed through the pointer. An example of a constant pointer to a constant integer follows:

```
const int * const cpci = &limit;
```

A constant pointer to a constant can be visualized as shown in Figure 1-14.

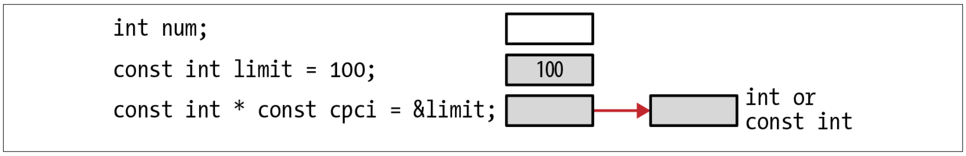


Figure 1-14. Constant pointers to constants

As with pointers to constants, it is not necessary to assign the address of a constant to `cpci`. Instead, we could have used `num` as shown below:

```
int num;
const int * const cpci = &num;
```

When the pointer is declared, we must initialize it. If we do not initialize it as shown below, we will get a syntax error:

```
const int * const cpci;
```

The syntax error will be similar to the following:

```
'cpci' : const object must be initialized if not extern
```

Given a constant pointer to a constant we cannot:

- Modify the pointer
- Modify the data pointed to by the pointer

Trying to assign a new address to `cpci` will result in a syntax error:

```
    cpci = &num;
```

The syntax error follows:

```
'cpci' : you cannot assign to a variable that is const
```

If we try to dereference the pointer and assign a value as shown below, we will also get a syntax error:

```
    *cpci = 25;
```

The error generated will be similar to the following:

```
'cpci' : you cannot assign to a variable that is const
expression must be a modifiable lvalue
```

Constant pointers to constants are rare.

Pointer to (constant pointer to constant)

Pointers to constants can also have multiple levels of indirection. In the following example, we declare a pointer to the `cpci` pointer explained in the previous section. Reading complex declarations from right to left helps clarify these types of declarations:

```
    const int * const cpci = &limit;
    const int * const * pcpci;
```

A pointer to a constant pointer to a constant can be visualized as shown in Figure 1-15.

```
const int limit = 500;
const int * const cpci = &limit;
const int * const * pcpci = &cpci;
```

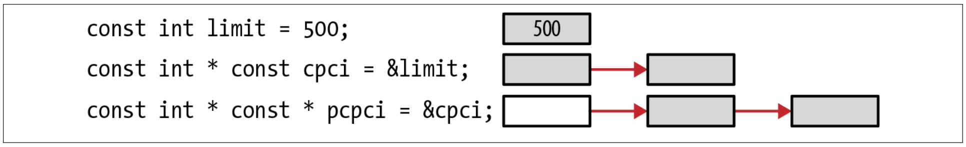


Figure 1-15. Pointer to (constant pointer to constant)

The following illustrates their use. The output of this sequence should display 500 twice:

```
    printf("%d\n",*cpci);
    pcpci = &cpci;
    printf("%d\n",**pcpci);
```

The following table summarizes the first four types of pointers discussed in the previous sections:

Pointer Type	Pointer Modifiable	Data Pointed to Modifiable
Pointer to a nonconstant	✓	✓
Pointer to a constant	✓	X
Constant pointer to a nonconstant	X	✓
Constant pointer to a constant	X	X

Summary

In this chapter, we covered the essential aspects of pointers, including how to declare and use pointers in common situations. The interesting concept of null and its variations was covered, along with a number of pointer operators.

We found that the size of a pointer can vary, depending on the memory model supported by the target system and compiler. We also explored the use of the `const` keyword with pointers.

With this foundation, we are prepared to explore the other areas where pointers have proved to be quite useful. This includes their use as parameters to functions, in support of data structures, and in dynamically allocating memory. In addition, we will see the effect of their use in making applications more secure.