

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.

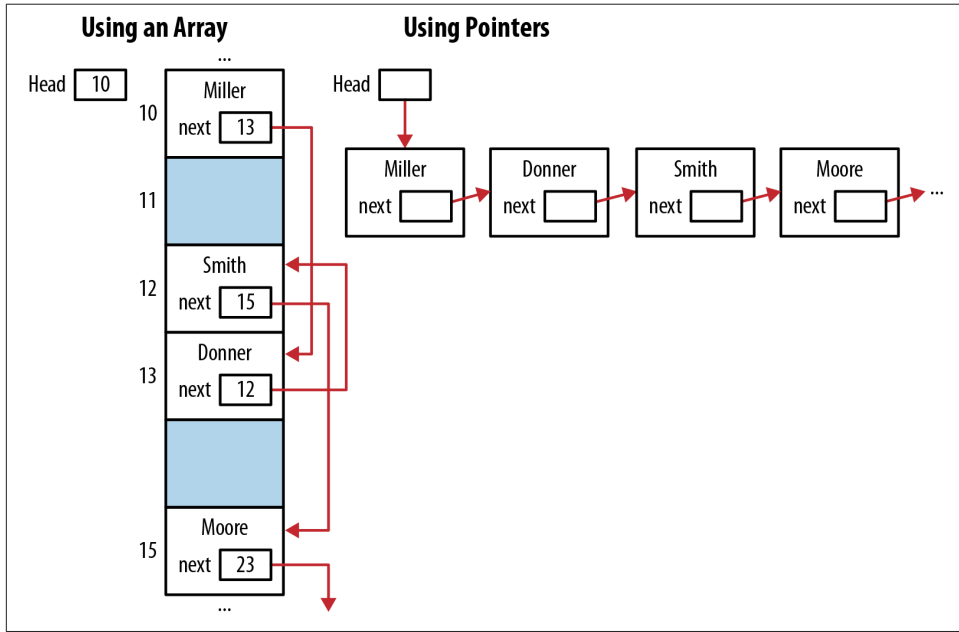


Figure 1-1. Array versus pointers representation of a linked list

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.

```
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**.

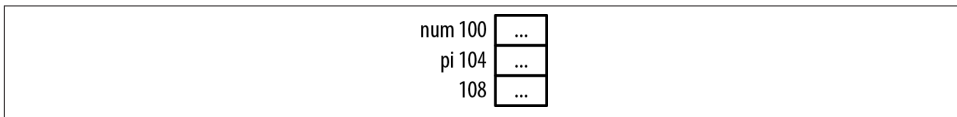


Figure 1-2. Memory diagram

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 `pci` 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:

```
const int *pci;
```

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

1. <code>pci</code> is a variable	<code>const int *pci;</code>
2. <code>pci</code> is a pointer variable	<code>const int *pci;</code>
3. <code>pci</code> is a pointer variable to an integer	<code>const int *pci;</code>
4. <code>pci</code> is a pointer variable to a constant integer	<code>const int *pci;</code>

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**.

num	100	0
pi	104	100
	108	...

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
```

num 100	200
pi 104	100
108	...

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 discomfoting 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.

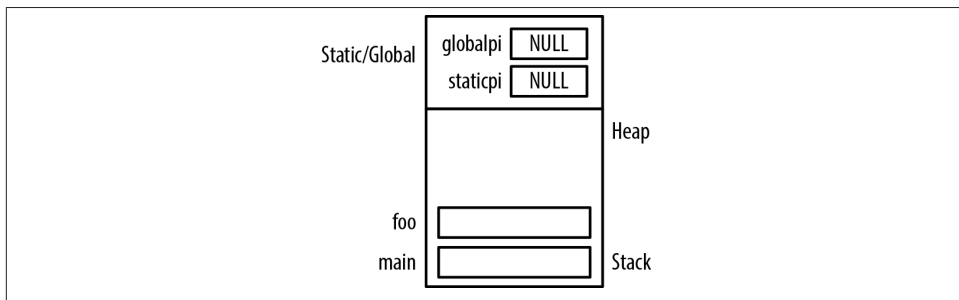


Figure 1-6. Memory allocation for global and static pointers

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 1-3¹](#) 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 `uintptr_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:

```
size_t = 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 Compiler (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:

vector[0]	100	28
vector[1]	104	41
vector[2]	108	7
pi	112	100

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.

s	120	...
ps	124	120
c	128	...
pc	132	128

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.

vector[0]	100	28
vector[1]	104	41
vector[2]	108	7
p0	112	100
p0	116	104
p0	120	108

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.

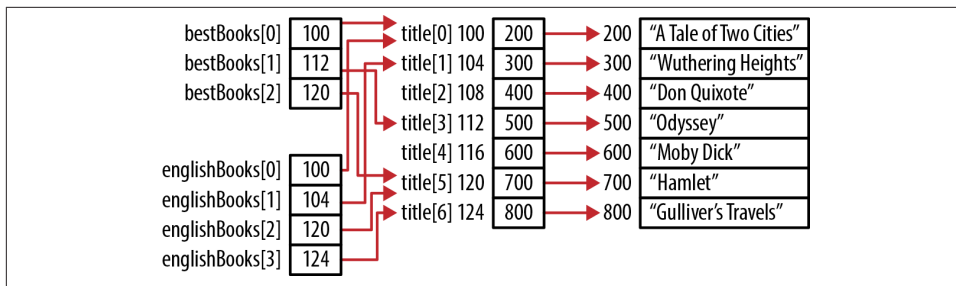


Figure 1-10. Pointers to pointers

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 titles 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.

num 100	5
limit 104	500
pi 108	100
pci 112	104

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`.

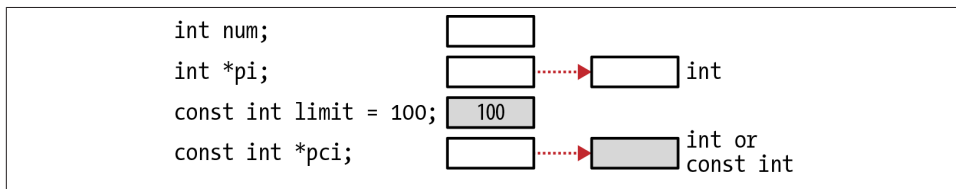


Figure 1-12. Pointer to a constant

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](#).

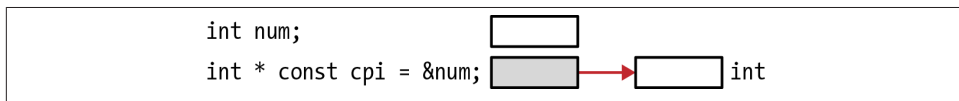


Figure 1-13. Constant pointers to nonconstants

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](#).

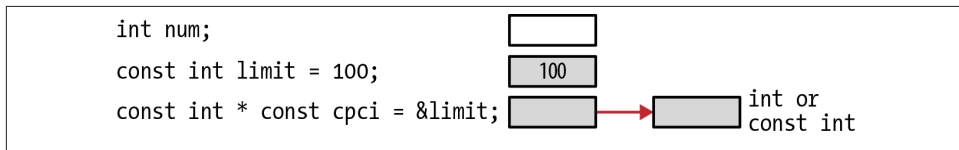


Figure 1-14. Constant pointers to constants

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](#).

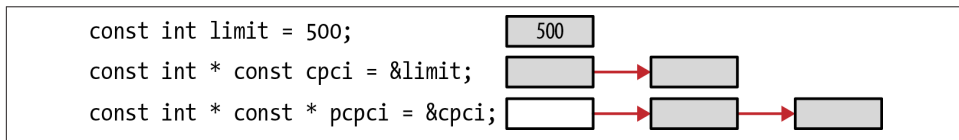


Figure 1-15. Pointer to (constant pointer to constant)

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.

Dynamic Memory Management in C

Much of the power of pointers stems from their ability to track dynamically allocated memory. The management of this memory through pointers forms the basis for many operations, including those used to manipulate complex data structures. To be able to fully exploit these capabilities, we need to understand how dynamic memory management occurs in C.

A C program executes within a *runtime system*. This is typically the environment provided by an operating system. The runtime system supports the stack and heap along with other program behavior.

Memory management is central to all programs. Sometimes memory is managed by the runtime system implicitly, such as when memory is allocated for automatic variables. In this case, variables are allocated to the enclosing function's stack frame. In the case of static and global variables, memory is placed in the application's data segment, where it is zeroed out. This is a separate area from executable code and other data managed by the runtime system.

The ability to allocate and then deallocate memory allows an application to manage its memory more efficiently and with greater flexibility. Instead of having to allocate memory to accommodate the largest possible size for a data structure, only the actual amount required needs to be allocated.

For example, arrays are fixed size in versions of C prior to C99. If we need to hold a variable number of elements, such as employee records, we would be forced to declare an array large enough to hold the maximum number of employees we believe would be needed. If we underestimate the size, we are forced to either recompile the application with a larger size or to take other approaches. If we overestimate the size, then we will waste space. The ability to dynamically allocate memory also helps when dealing with data structures using a variable number of elements, such as a linked list or a queue.



C99 introduced Variable Length Arrays (VLAs). The array's size is determined at runtime and not at compile time. However, once created, arrays still do not change size.

Languages such as C also support dynamic memory management where objects are allocated memory from the heap. This is done manually using functions to allocate and deallocate memory. The process is referred to as *dynamic memory management*.

We start this chapter with a quick overview of how memory is allocated and freed. Next, we present basic allocation functions such as `malloc` and `realloc`. The `free` function is discussed, including the use of `NULL` along with such problems as double free.

Dangling pointers are a common problem. We will present examples to illustrate when dangling pointers occur and techniques to handle the problem. The last section presents alternate techniques for managing memory. Improper use of pointers can result in unpredictable behavior. By this we mean the program can produce invalid results, corrupt data, or possibly terminate the program.

Dynamic Memory Allocation

The basic steps used for dynamic memory allocation in C are:

1. Use a `malloc` type function to allocate memory
2. Use this memory to support the application
3. Deallocate the memory using the `free` function

While there are some minor variations to this approach, this is the most common technique. In the following example, we allocate memory for an integer using the `malloc` function. The pointer assigns five to the allocated memory, and then the memory is released using the `free` function:

```
int *pi = (int*) malloc(sizeof(int));
*pi = 5;
printf("pi: %d\n", *pi);
free(pi);
```

When this sequence is executed, it will display the number 5. [Figure 2-1](#) illustrates how memory is allocated right before the `free` function is executed. For the purposes of this chapter, we will assume that the example code is found in the `main` function unless otherwise noted.

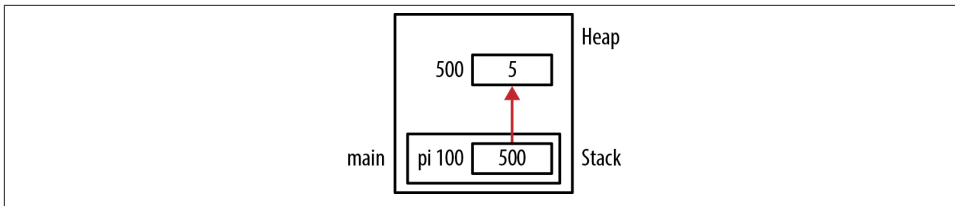


Figure 2-1. Allocating memory for an integer

The `malloc` function single argument specifies the number of bytes to allocate. If successful, it returns a pointer to memory allocated from the heap. If it fails, it returns a null pointer. Testing the validity of an allocated pointer is discussed in “[Using the malloc Function](#)”. The `sizeof` operator makes the application more portable and determines the correct number of bytes to allocate for the host system.

In this example, we are trying to allocate enough memory for an integer. If we assume its size is 4, we can use:

```
int *pi = (int*) malloc(4));
```

However, the size of an integer can vary, depending on the memory model used. A portable approach is to use the `sizeof` operator. This will return the correct size regardless of where the program is executing.



A common error involving the dereference operator is demonstrated below:

```
int *pi;
*pi = (int*) malloc(sizeof(int));
```

The problem is with the lefthand side of the assignment operation. We are dereferencing the pointer. This will assign the address returned by `malloc` to the address stored in `pi`. If this is the first time an assignment is made to the pointer, then the address contained in the pointer is probably invalid. The correct approach is shown below:

```
pi = (int*) malloc(sizeof(int));
```

The dereference operator should not be used in this situation.

The `free` function, also discussed in more detail later, works in conjunction with `malloc` to deallocate the memory when it is no longer needed.



Each time the `malloc` function (or similar function) is called, a corresponding call to the `free` function must be made when the application is done with the memory to avoid memory leaks.

Once memory has been freed, it should not be accessed again. Normally, you would not intentionally access it after it had been deallocated. However, this can occur accidentally, as illustrated in the section “[Dangling Pointers](#)” on page 51. The system behaves in an implementation-dependent manner when this happens. A common practice is to always assign NULL to a freed pointer, as discussed in “[Assigning NULL to a Freed Pointer](#)” on page 48.

When memory is allocated, additional information is stored as part of a data structure maintained by the heap manager. This information includes, among other things, the block's size, and is typically placed immediately adjacent to the allocated block. If the application writes outside of this block of memory, then the data structure can be corrupted. This can lead to strange program behavior or corruption of the heap, as we will see in [Chapter 7](#).

Consider the following code sequence. Memory is allocated for a string, allowing it to hold up to five characters plus the byte for the NUL termination character. The for loop writes zeros to each location but does not stop after writing six bytes. The for statement's terminal condition requires that it write eight bytes. The zeros being written are binary zeros and not the ASCII value for the character zero:

```
char *pc = (char*) malloc(6);  
for(int i=0; i<8; i++) {  
    *pc[i] = 0;  
}
```

In [Figure 2-2](#), extra memory has been allocated at the end of the six-byte string. This represents the extra memory used by the heap manager to keep track of the memory allocation. If we write past the end of the string, this extra memory will be corrupted. The extra memory is shown following the string in this example. However, its actual placement and its original content depend on the compiler.

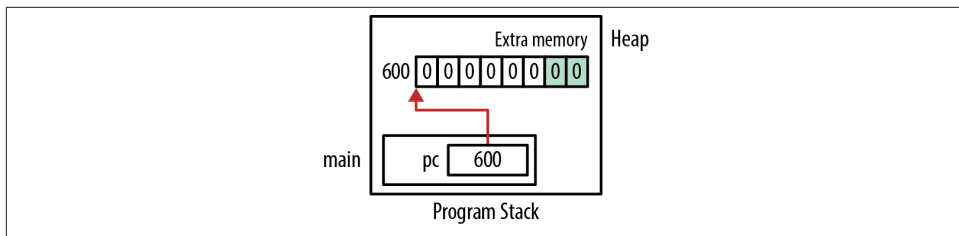


Figure 2-2. Extra memory used by heap manager

Memory Leaks

A memory leak occurs when allocated memory is never used again but is not freed. This can happen when:

- The memory's address is lost
- The free function is never invoked though it should be (sometimes called a hidden leak)

A problem with memory leaks is that the memory cannot be reclaimed and used later. The amount of memory available to the heap manager is decreased. If memory is repeatedly allocated and then lost, then the program may terminate when more memory is needed but `malloc` cannot allocate it because it ran out of memory. In extreme cases, the operating system may crash.

This is illustrated in the following simple example:

```
char *chunk;
while (1) {
    chunk = (char*) malloc(1000000);
    printf("Allocating\n");
}
```

The variable `chunk` is assigned memory from the heap. However, this memory is not freed before another block of memory is assigned to it. Eventually, the application will run out of memory and terminate abnormally. At minimum, memory is not being used efficiently.

Losing the address

An example of losing the address of memory is illustrated in the following code sequence where `pi` is reassigned a new address. The address of the first allocation of memory is lost when `pi` is allocated memory a second time.

```
int *pi = (int*) malloc(sizeof(int));
*pi = 5;
...
pi = (int*) malloc(sizeof(int));
```

This is illustrated in [Figure 2-3](#) where the before and after images refer to the program's state before and after the second `malloc`'s execution. The memory at address 500 has not been released, and the program no longer holds this address anywhere.

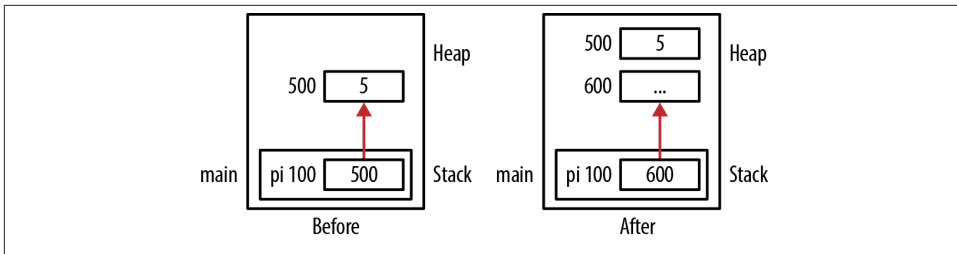


Figure 2-3. Losing an address

Another example allocates memory for a string, initializes it, and then displays the string character by character:

```
char *name = (char*)malloc(strlen("Susan")+1);
strcpy(name, "Susan");
while(*name != 0) {
    printf("%c", *name);
    name++;
}
```

However, it increments `name` by one with each loop iteration. At the end, `name` is left pointing to the string's NUL termination character, as illustrated in Figure 2-4. The allocated memory's starting address has been lost.

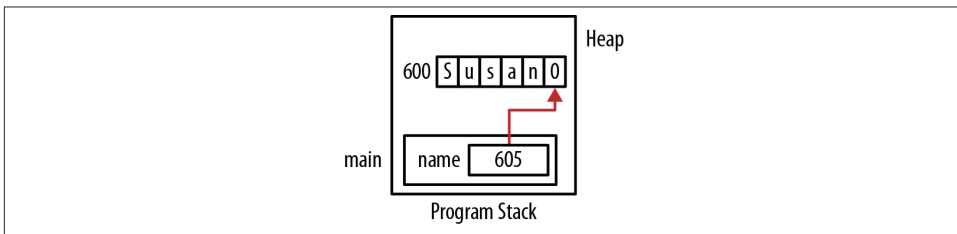


Figure 2-4. Losing address of dynamically allocated memory

Hidden memory leaks

Memory leaks can also occur when the program should release memory but does not. A hidden memory leak occurs when an object is kept in the heap even though the object is no longer needed. This is frequently the result of programmer oversight. The primary problem with this type of leak is that the object is using memory that is no longer needed and should be returned to the heap. In the worst case, the heap manager may not be able to allocate memory when requested, possibly forcing the program to terminate. At best, we are holding unneeded memory.

Memory leaks can also occur when freeing structures created using the `struct` keyword. If the structure contains pointers to dynamically allocated memory, then these pointers

may need to be freed before the structure is freed. An example of this is found in [Chapter 6](#).

Dynamic Memory Allocation Functions

Several memory allocation functions are available to manage dynamic memory. While what is available may be system dependent, the following functions are found on most systems in the *stdlib.h* header file:

- `malloc`
- `realloc`
- `calloc`
- `free`

The functions are summarized in [Table 2-1](#).

Table 2-1. Dynamic memory allocation functions

Function	Description
<code>malloc</code>	Allocates memory from the heap
<code>realloc</code>	Reallocates memory to a larger or smaller amount based on a previously allocated block of memory
<code>calloc</code>	Allocates and zeros out memory from the heap
<code>free</code>	Returns a block of memory to the heap

Dynamic memory is allocated from the heap. With successive memory allocation calls, there is no guarantee regarding the order of the memory or the continuity of memory allocated. However, the memory allocated will be aligned according to the pointer's data type. For example, a four-byte integer would be allocated on an address boundary evenly divisible by four. The address returned by the heap manager will contain the lowest byte's address.

In [Figure 2-3](#), the `malloc` function allocates four bytes at address 500. The second use of the `malloc` function allocates memory at address 600. They both are on four-byte address boundaries, and they did not allocate memory from consecutive memory locations.

Using the malloc Function

The function `malloc` allocates a block of memory from the heap. The number of bytes allocated is specified by its single argument. Its return type is a pointer to void. If memory is not available, `NULL` is returned. The function does not clear or otherwise modify the memory, thus the contents of memory should be treated as if it contained garbage. The function's prototype follows:

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

The function possesses a single argument of type `size_t`. This type is discussed in [Chapter 1](#). You need to be careful when passing variables to this function, as problems can arise if the argument is a negative number. On some systems, a `NULL` value is returned if the argument is negative.

When `malloc` is used with an argument of zero, its behavior is implementation-specific. It may return a pointer to `NULL` or it may return a pointer to a region with zero bytes allocated. If the `malloc` function is used with a `NULL` argument, then it will normally generate a warning and execute returning zero bytes.

The following shows a typical use of the `malloc` function:

```
int *pi = (int*) malloc(sizeof(int));
```

The following steps are performed when the `malloc` function is executed:

1. Memory is allocated from the heap
2. The memory is *not* modified or otherwise cleared
3. The first byte's address is returned



Since the `malloc` function may return a `NULL` value if it is unable to allocate memory, it is a good practice to check for a `NULL` value before using the pointer as follows:

```
int *pi = (int*) malloc(sizeof(int));
if(pi != NULL) {
    // Pointer should be good
} else {
    // Bad pointer
}
```

To cast or not to cast

Before the pointer to void was introduced to C, explicit casts were required with `malloc` to stop the generation of warnings when assignments were made between incompatible pointer types. Since a pointer to void can be assigned to any other pointer type, explicit casting is no longer required. Some developers consider explicit casts to be a good practice because:

- They document the intention of the `malloc` function
- They make the code compatible with C++ (or earlier C compiler), which require explicit casts

Using casts will be a problem if you fail to include the header file for `malloc`. The compiler may generate warnings. By default, C assumes functions return an integer. If you fail to include a prototype for `malloc`, it will complain when you try to assign an integer to a pointer.

Failing to allocate memory

If you declare a pointer but fail to allocate memory to the address it points to before using it, that memory will usually contain garbage, resulting typically in an invalid memory reference. Consider the following code sequence:

```
int *pi;  
...  
printf("%d\n", *pi);
```

The allocation of memory is shown in [Figure 2-5](#). This issue is covered in more detail in [Chapter 7](#).

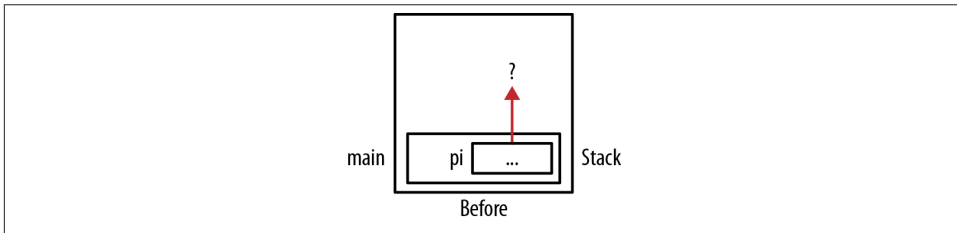


Figure 2-5. Failure to allocate memory

When executed, this can result in a runtime exception. This type of problem is common with strings, as shown below:

```
char *name;  
printf("Enter a name: ");  
scanf("%s", name);
```

While it may seem like this would execute correctly, we are using memory referenced by `name`. However, this memory has not been allocated. This problem can be illustrated graphically by changing the variable, `pi`, in [Figure 2-5](#) to `name`.

Not using the right size for the malloc function

The `malloc` function allocates the number of bytes specified by its argument. You need to be careful when using the function to allocate the correct number of bytes. For example, if we want to allocate space for 10 doubles, then we need to allocate 80 bytes. This is achieved as shown below:

```
double *pd = (double*)malloc(NUMBER_OF_DOUBLES * sizeof(double));
```



Use the `sizeof` operator when specifying the number of bytes to allocate for data types whenever possible.

In the following example, an attempt is made to allocate memory for 10 doubles:

```
const int NUMBER_OF_DOUBLES = 10;
double *pd = (double*)malloc(NUMBER_OF_DOUBLES);
```

However, the code only allocated 10 bytes.

Determining the amount of memory allocated

There is no standard way to determine the total amount of memory allocated by the heap. However, some compilers provide extensions for this purpose. In addition, there is no standard way of determining the size of a memory block allocated by the heap manager.

For example, if we allocate 64 bytes for a string, the heap manager will allocate additional memory to manage this block. The total size allocated, and the amount used by the heap manager, is the sum of these two quantities. This was illustrated in [Figure 2-2](#).

The maximum size that can be allocated with `malloc` is system dependent. It would seem like this size should be limited by `size_t`. However, limitations can be imposed by the amount of physical memory present and other operating system constraints.

When `malloc` executes, it is supposed to allocate the amount of memory requested and then return the memory's address. What happens if the underlying operating system uses "lazy initialization" where it does not actually allocate the memory until it is accessed? A problem can arise at this point if there is not enough memory available to allocate. The answer depends on the runtime and operating systems. A typical developer normally would not need to deal with this question because such initialization schemes are quite rare.

Using malloc with static and global pointers

You cannot use a function call when initializing a static or global variable. In the following code sequence, we declare a static variable and then attempt to initialize it using `malloc`:

```
static int *pi = malloc(sizeof(int));
```

This will generate a compile-time error message. The same thing happens with global variables but can be avoided for static variables by using a separate statement to allocate memory to the variable as follows. We cannot use a separate assignment statement with global variables because global variables are declared outside of a function and executable code, such as the assignment statement, must be inside of a function:

```
static int *pi;  
pi = malloc(sizeof(int));
```



From the compiler standpoint, there is a difference between using the initialization operator, `=`, and using the assignment operator, `=`.

Using the calloc Function

The `calloc` function will allocate and clear memory at the same time. Its prototype follows:

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



To clear memory means its contents are set to all binary zeros.

The function will allocate memory determined by the product of the `numElements` and `elementSize` parameters. A pointer is returned to the first byte of memory. If the function is unable to allocate memory, `NULL` is returned. Originally, this function was used to aid in the allocation of memory for arrays.

If either `numElements` or `elementSize` is zero, then a null pointer may be returned. If `calloc` is unable to allocate memory, a null pointer is returned and the global variable, `errno`, is set to `ENOMEM` (out of memory). This is a POSIX error code and may not be available on all systems.

Consider the following example where `pi` is allocated a total of 20 bytes, all containing zeros:

```
int *pi = calloc(5, sizeof(int));
```

Instead of using `calloc`, the `malloc` function along with the `memset` function can be used to achieve the same results, as shown below:

```
int *pi = malloc(5 * sizeof(int));  
memset(pi, 0, 5 * sizeof(int));
```



The `memset` function will fill a block with a value. The first argument is a pointer to the buffer to fill. The second is the value used to fill the buffer, and the last argument is the number of bytes to be set.

Use `calloc` when memory needs to be zeroed out. However, the execution of `calloc` may take longer than using `malloc`.



The function `cfree` is no longer needed. In the early days of C it was used to free memory allocated by `calloc`.

Using the `realloc` Function

Periodically, it may be necessary to increase or decrease the amount of memory allocated to a pointer. This is particularly useful when a variable size array is needed, as will be demonstrated in [Chapter 4](#). The `realloc` function will reallocate memory. Its prototype follows:

```
void *realloc(void *ptr, size_t size);
```

The function `realloc` returns a pointer to a block of memory. The function takes two arguments. The first is a pointer to the original block, and the second is the requested size. The reallocated block's size will be different from the size of the block referenced by the first argument. The return value is a pointer to the reallocated memory.

The requested size may be smaller or larger than the currently allocated amount. If the size is less than what is currently allocated, then the excess memory is returned to the heap. There is no guarantee that the excess memory will be cleared. If the size is greater than what is currently allocated, then if possible, the memory will be allocated from the region immediately following the current allocation. Otherwise, memory is allocated from a different region of the heap and the old memory is copied to the new region.

If the size is zero and the pointer is not null, then the pointer will be freed. If space cannot be allocated, then the original block of memory is retained and is not changed. However, the pointer returned is a null pointer and the `errno` is set to `ENOMEM`.

The function's behavior is summarized in [Table 2-2](#).

Table 2-2. Behavior of `realloc` function

First Parameter	Second Parameter	Behavior
null	NA	Same as <code>malloc</code>
Not null	0	Original block is freed
Not null	Less than the original block's size	A smaller block is allocated using the current block
Not null	Larger than the original block's size	A larger block is allocated either from the current location or another region of the heap

In the following example, we use two variables to allocate memory for a string. Initially, we allocate 16 bytes but only use the first 13 bytes (12 hexadecimal digits and the null termination character (0)):

```
char *string1;  
char *string2;  
string1 = (char*) malloc(16);  
strcpy(string1, "0123456789AB");
```

Next, we use the `realloc` function to specify a smaller region of memory. The address and contents of these two variables are then displayed:

```
string2 = realloc(string1, 8);  
printf("string1 Value: %p [%s]\n", string1, string1);  
printf("string2 Value: %p [%s]\n", string2, string2);
```

The output follows:

```
string1 Value: 0x500 [0123456789AB]  
string2 Value: 0x500 [0123456789AB]
```

The allocation of memory is illustrated in [Figure 2-6](#).

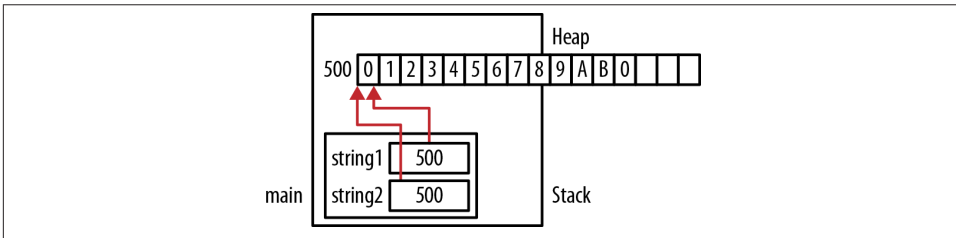


Figure 2-6. *realloc* example

The heap manager was able to reuse the original block, and it did not modify its contents. However, the program continued to use more than the eight bytes requested. That is, we did not change the string to fit into the eight-byte block. In this example, we should have adjusted the length of the string so that it fits into the eight reallocated bytes. The simplest way of doing this is to assign a NUL character to address 507. Using more space than allocated is not a good practice and should be avoided, as detailed in [Chapter 7](#).

In this next example, we will reallocate additional memory:

```
string1 = (char*) malloc(16);  
strcpy(string1, "0123456789AB");  
string2 = realloc(string1, 64);  
printf("string1 Value: %p [%s]\n", string1, string1);  
printf("string2 Value: %p [%s]\n", string2, string2);
```

When executed, you may get results similar to the following:

```
string1 Value: 0x500 [0123456789AB]
string2 Value: 0x600 [0123456789AB]
```

In this example, `realloc` had to allocate a new block of memory. [Figure 2-7](#) illustrates the allocation of memory.

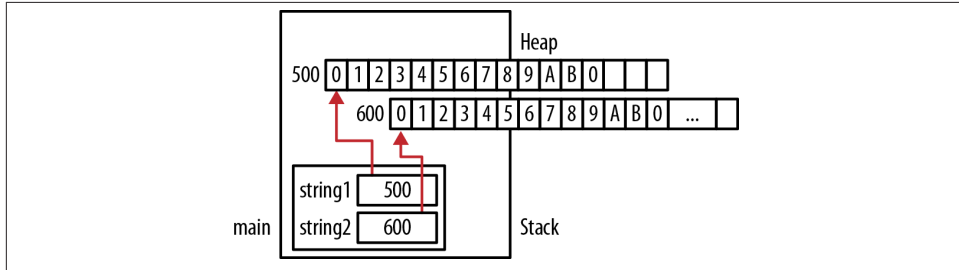


Figure 2-7. Allocating additional memory

The `alloca` Function and Variable Length Arrays

The `alloca` function (Microsoft's `malloc`) allocates memory by placing it in the stack frame for the function. When the function returns, the memory is automatically freed. This function can be difficult to implement if the underlying runtime system is not stack-based. As a result, this function is nonstandard and should be avoided if the application needs to be portable.

In C99, Variable Length Arrays (VLAs) were introduced, allowing the declaration and creation of an array within a function whose size is based on a variable. In the following example, an array of `char` is allocated for use in a function:

```
void compute(int size) {
    char* buffer[size];
    ...
}
```

This means the allocation of memory is done at runtime and memory is allocated as part of the stack frame. Also, when the `sizeof` operator is used with the array, it will be executed at runtime rather than compile time.

A small runtime penalty will be imposed. Also, when the function exits, the memory is effectively deallocated. Since we did not use a `malloc` type function to create it, we should not use the `free` function to deallocate it. The function should not return a pointer to this memory either. This issue is addressed in [Chapter 5](#).



VLA's do not change size. Their size is fixed once they are allocated. If you need an array whose size actually changes, then an approach such as using the `realloc` function, as discussed in the section “Using the `realloc` Function” on page 44, is needed.

Deallocating Memory Using the `free` Function

With dynamic memory allocation, the programmer is able to return memory when it is no longer being used, thus freeing it up for other uses. This is normally performed using the `free` function, whose prototype is shown below:

```
void free(void *ptr);
```

The pointer argument should contain the address of memory allocated by a `malloc` type function. This memory is returned to the heap. While the pointer may still point to the region, always assume it points to garbage. This region may be reallocated later and populated with different data.

In the simple example below, `pi` is allocated memory and is eventually freed:

```
int *pi = (int*) malloc(sizeof(int));  
...  
free(pi);
```

Figure 2-8 illustrates the allocation of memory immediately before and right after the `free` function executes. The dashed box at address 500 that indicates the memory has been freed but still may contain its value. The variable `pi` still contains the address 500. This is called a **dangling pointer** and is discussed in detail in the section “Dangling Pointers” on page 51.

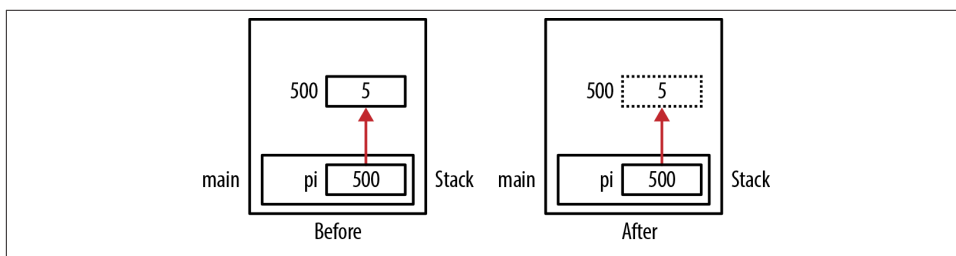


Figure 2-8. Release of memory using `free`

If the `free` function is passed a null pointer, then it normally does nothing. If the pointer passed has been allocated by other than a `malloc` type function, then the function's behavior is undefined. In the following example, `pi` is allocated the address of `num`. However, this is not a valid heap address:

```
int num;
int *pi = &num;
free(pi); // Undefined behavior
```



Manage memory allocation/deallocation at the same level. For example, if a pointer is allocated within a function, deallocate it in the same function.

Assigning NULL to a Freed Pointer

Pointers can cause problems even after they have been freed. If we try to dereference a freed pointer, its behavior is undefined. As a result, some programmers will explicitly assign NULL to a pointer to designate the pointer as invalid. Subsequent use of such a pointer will result in a runtime exception.

An example of this approach follows:

```
int *pi = (int*) malloc(sizeof(int));
...
free(pi);
pi = NULL;
```

The allocation of memory is illustrated in [Figure 2-9](#).

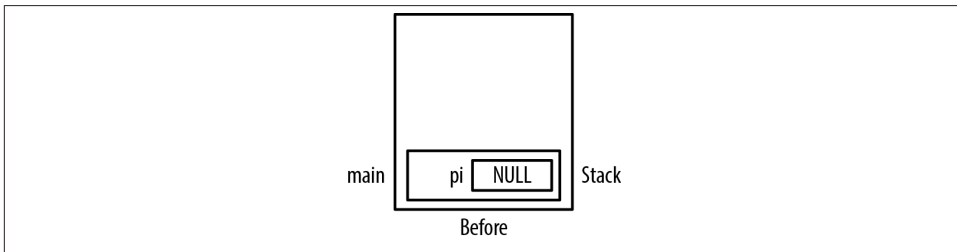


Figure 2-9. Assigning NULL after using free

This technique attempts to address problems like dangling pointers. However, it is better to spend time addressing the conditions that caused the problems rather than crudely catching them with a null pointer. In addition, you cannot assign NULL to a constant pointer except when it is initialized.

Double Free

The term *double free* refers to an attempt to free a block of memory twice. A simple example follows:


```

int *pi = (int*) malloc(sizeof(int));
*pi = 5;
free(pi);
...
free(pi);

```

The execution of the second `free` function will result in a runtime exception. A less obvious example involves the use of two pointers, both pointing to the same block of memory. As shown below, the same runtime exception will result when we accidentally try to free the same memory a second time:

```

p1 = (int*) malloc(sizeof(int));
int *p2 = p1;
free(p1);
...
free(p2);

```

This allocation of memory is illustrated in [Figure 2-10](#).



When two pointers reference the same location, it is referred to as *aliasing*. This concept is discussed in [Chapter 8](#).

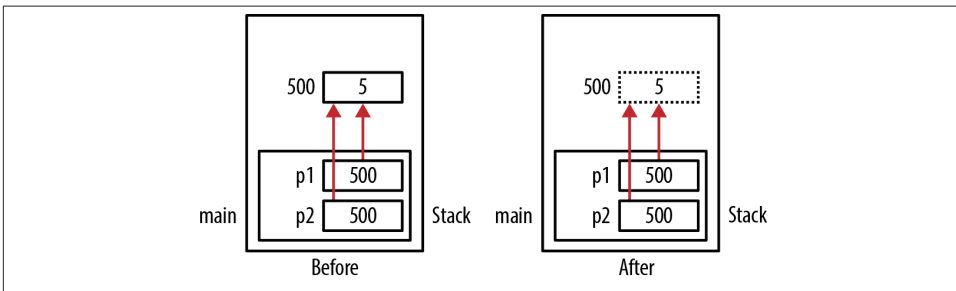


Figure 2-10. Double free

Unfortunately, heap managers have a difficult time determining whether a block has already been deallocated. Thus, they don't attempt to detect the same memory being freed twice. This normally results in a corrupt heap and program termination. Even if the program does not terminate, it represents questionable problem logic. There is no reason to free the same memory twice.

It has been suggested that the `free` function should assign a `NULL` or some other special value to its argument when it returns. However, since pointers are passed by value, the `free` function is unable to explicitly assign `NULL` to the pointer. This is explained in more detail in the section [“Passing a Pointer to a Pointer”](#) on page 68.

The Heap and System Memory

The heap typically uses operating system functions to manage its memory. The heap's size may be fixed when the program is created, or it may be allowed to grow. However, the heap manager does not necessarily return memory to the operating system when the `free` function is called. The deallocated memory is simply made available for subsequent use by the application. Thus, when a program allocates and then frees up memory, the deallocation of memory is not normally reflected in the application's memory usage as seen from the operating system perspective.

Freeing Memory upon Program Termination

The operating system is responsible for maintaining the resources of an application, including its memory. When an application terminates, it is the operating system's responsibility to reallocate this memory for other applications. The state of the terminated application's memory, corrupted or uncorrupted, is not an issue. In fact, one of the reasons an application may terminate is because its memory is corrupted. With an abnormal program termination, cleanup may not be possible. Thus, there is no reason to free allocated memory before the application terminates.

With this said, there may be other reasons why this memory should be freed. The conscientious programmer may want to free memory as a quality issue. It is always a good habit to free memory after it is no longer needed, even if the application is terminating. If you use a tool to detect memory leaks or similar problems, then deallocating memory will clean up the output of such tools. In some less complex operating systems, the operating system may not reclaim memory automatically, and it may be the program's responsibility to reclaim memory before terminating. Also, a later version of the application could add code toward the end of the program. If the previous memory has not been freed, problems could arise.

Thus, ensuring that all memory is free before program termination:

- May be more trouble than it's worth
- Can be time consuming and complicated for the deallocation of complex structures
- Can add to the application's size
- Results in longer running time
- Introduces the opportunity for more programming errors

Whether memory should be deallocated prior to program termination is application-specific.

Dangling Pointers

If a pointer still references the original memory after it has been freed, it is called a dangling pointer. The pointer does not point to a valid object. This is sometimes referred to as a premature free.

The use of dangling pointers can result in a number of different types of problems, including:

- Unpredictable behavior if the memory is accessed
- *Segmentation faults* when the memory is no longer accessible
- Potential security risks

These types of problems can result when:

- Memory is accessed after it has been freed
- A pointer is returned to an automatic variable in a previous function call (discussed in the section “[Pointers to Local Data](#)” on page 66)

Dangling Pointer Examples

Below is a simple example where we allocate memory for an integer using the `malloc` function. Next, the memory is released using the `free` function:

```
int *pi = (int*) malloc(sizeof(int));
*pi = 5;
printf("pi: %d\n", *pi);
free(pi);
```

The variable `pi` will still hold the integer’s address. However, this memory may be reused by the heap manager and may hold data other than an integer. [Figure 2-11](#) illustrates the program’s state immediately before and after the `free` function is executed. The `pi` variable is assumed to be part of the `main` function and is located at address 100. The memory allocated using `malloc` is found at address 500.

When the `free` function is executed, the memory at address 500 has been deallocated and should not be used. However, most runtime systems will not prevent subsequent access or modification. We may still attempt to write to the location as shown below. The result of this action is unpredictable.

```
free(pi);
*pi = 10;
```

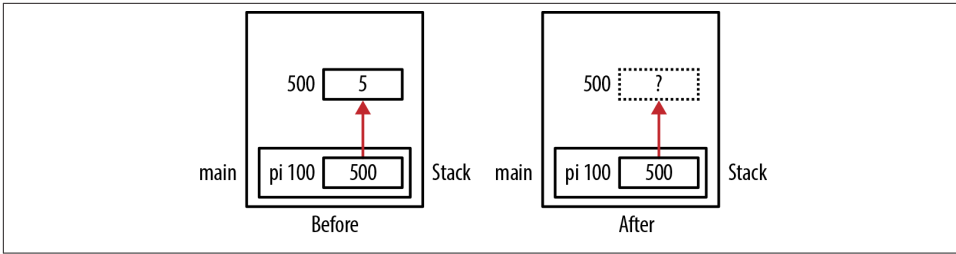


Figure 2-11. *Dangling pointer*

A more insidious example occurs when more than one pointer references the same area of memory and one of them is freed. As shown below, `p1` and `p2` both refer to the same area of memory, which is called pointer aliasing. However, `p1` is freed:

```
int *p1 = (int*) malloc(sizeof(int));
*p1 = 5;
...
int *p2;
p2 = p1;
...
free(p1);
...
*p2 = 10;    // Dangling pointer
```

Figure 2-12 illustrates the allocation of memory where the dotted box represents freed memory.

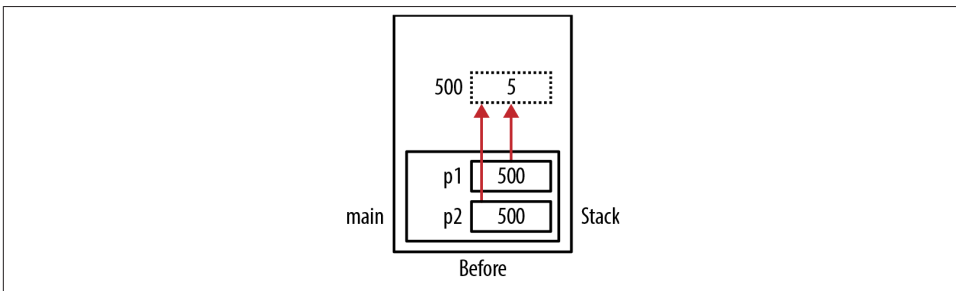


Figure 2-12. *Dangling pointer with aliased pointers*

A subtle problem can occur when using block statements, as shown below. Here `pi` is assigned the address of `tmp`. The variable `pi` may be a global variable or a local variable. However, when `tmp`'s enclosing block is popped off of the program stack, the address is no longer valid:

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

```

{
    int tmp = 5;
    pi = &tmp;
}
// pi is now a dangling pointer
foo();

```

Most compilers will treat a block statement as a stack frame. The variable `tmp` was allocated on the stack frame and subsequently popped off the stack when the block statement was exited. The pointer `pi` is now left pointing to a region of memory that may eventually be overridden by a different activation record, such as the function `foo`. This condition is illustrated in [Figure 2-13](#).

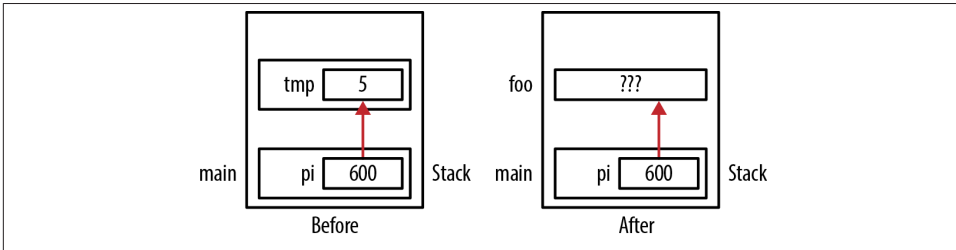


Figure 2-13. Block statement problem

Dealing with Dangling Pointers

Debugging pointer-induced problems can be difficult to resolve at times. Several approaches exist for dealing with dangling pointers, including:

- Setting a pointer to NULL after freeing it. Its subsequent use will terminate the application. However, problems can still persist if multiple copies of the pointer exist. This is because the assignment will only affect one of the copies, as illustrated in the section [“Double Free” on page 48](#).
- Writing special functions to replace the `free` function (see [“Writing your own free function” on page 70](#)).
- Some systems (runtime/debugger) will overwrite data when it is freed (e.g., 0xDEADBEEF - Visual Studio will use 0xCC, 0xCD, or 0xDD, depending on what is freed). While no exceptions are thrown, when the programmer sees memory containing these values where they are not expected, he knows that the program may be accessing freed memory.
- Use third-party tools to detect dangling pointers and other problems.

Displaying pointer values can be helpful in debugging dangling pointers, but you need to be careful how they are displayed. We have already discussed how to display pointer

values in [“Displaying Pointer Values” on page 9](#). Make sure you display them consistently to avoid confusion when comparing pointer values. The `assert` macro can also be useful, as demonstrated in [“Dealing with Uninitialized Pointers” on page 162](#).

Debug Version Support for Detecting Memory Leaks

Microsoft provides techniques for addressing overwriting of dynamically allocated memory and memory leaks. This approach uses special memory management techniques in debug versions of a program to:

- Check the heap’s integrity
- Check for memory leaks
- Simulate low heap memory situations

Microsoft does this by using a special data structure to manage memory allocation. This structure maintains debug information, such as the filename and line number where `malloc` is called. In addition, buffers are allocated before and after the actual memory allocation to detect overwriting of the actual memory. More information about this technique can be found at [Microsoft Developer Network](#).

The [Mudflap Libraries](#) provide a similar capability for the GCC compiler. Its runtime library supports the detection of memory leaks, among other things. This detection is accomplished by instrumenting the pointer dereferencing operations.

Dynamic Memory Allocation Technologies

So far, we have talked about the heap manager’s allocating and deallocating memory. However, the implementation of this technology can vary by compiler. Most heap managers use a heap or data segment as the source for memory. However, this approach is subject to fragmentation and may collide with the program stack. Nevertheless, it is the most common way of implementing the heap.

Heap managers need to address many issues, such as whether heaps are allocated on a per process and/or per thread basis and how to protect the heap from security breaches.

There are a number of heap managers, including OpenBSD’s `malloc`, Hoard’s `malloc`, and TCMalloc developed by Google. The GNU C library allocator is based on the general-purpose allocator `dlmalloc`. It provides facilities for debugging and can help in tracking memory leaks. The `dlmalloc`’s logging feature tracks memory usage and memory transaction, among other actions.

A manual technique for managing the memory used for structures is presented in [“Avoiding malloc/free Overhead” on page 139](#).

Garbage Collection in C

The `malloc` and `free` functions provide a way of manually allocating and deallocating memory. However, there are numerous issues regarding the use of manual memory management in C, such as performance, achieving good locality of reference, threading problems, and cleaning up memory gracefully.

Several nonstandard techniques can be used to address some of these issues, and this section explores some of them. A key feature of these techniques is the automatic deallocation of memory. When memory is no longer needed, it is collected and made available for use later in the program. The deallocated memory is referred to as garbage. Hence, the term *garbage collection* denotes the processing of this memory.

Garbage collection is useful for a number of reasons, including:

- Freeing the programmer from having to decide when to deallocate memory
- Allowing the programmer to focus on the application's problem

One alternative to manual memory management is the **Boehm-Weiser Collector**. However, this is not part of the language.

Resource Acquisition Is Initialization

Resource Acquisition Is Initialization (RAII) is a technique invented by Bjarne Stroustrup. It addresses the allocation and deallocation of resources in C++. The technique is useful for guaranteeing the allocation and subsequent deallocation of a resource in the presence of exceptions. Allocated resources will eventually be released.

There have been several approaches for using RAII in C. The GNU compiler provides a nonstandard extension to support this. We will illustrate this extension by showing how memory can be allocated and then freed within a function. When the variable goes out of scope, the deallocation process occurs automatically.

The GNU extension uses a macro called `RAII_VARIABLE`. It declares a variable and associates with the variable:

- A type
- A function to execute when the variable is created
- A function to execute when the variable goes out of scope

The macro is shown below:

```
#define RAII_VARIABLE(vartype,varname,initval,dtor) \
    void _dtor_ ## varname (vartype * v) { dtor(*v); } \
    vartype varname __attribute__((cleanup(_dtor_ ## varname))) = (initval)
```

In the following example, we declare a variable called `name` as a pointer to `char`. When it is created, the `malloc` function is executed, allocating 32 bytes to it. When the function is terminated, `name` goes out of scope and the `free` function is executed:

```
void raiiExample() {
    RAII_VARIABLE(char*, name, (char*)malloc(32), free);
    strcpy(name, "RAII Example");
    printf("%s\n", name);
}
```

When this function is executed, the string “RAII_Example” will be displayed.

Similar results can be achieved without using the GNU extension.

Using Exception Handlers

Another approach to deal with the deallocation of memory is to use **exception handling**. While exception handling is not a standard part of C, it can be useful if available and possible portability issues are not a concern. The following illustrates the approach using the Microsoft Visual Studio version of the C language.

Here the `try` block encloses any statements that might cause an exception to be thrown at runtime. The `finally` block will be executed regardless of whether an exception is thrown. The `free` function is guaranteed to be executed.

```
void exceptionExample() {
    int *pi = NULL;
    __try {
        pi = (int*)malloc(sizeof(int));
        *pi = 5;
        printf("%d\n", *pi);
    }
    __finally {
        free(pi);
    }
}
```

You can implement exception handling in C using several other approaches.

Summary

Dynamic memory allocation is a significant C language feature. In this chapter, we focused on the manual allocation of memory using the `malloc` and `free` functions. We addressed a number of common problems involving these functions, including the failure to allocate memory and dangling pointers.

There are other nonstandard techniques for managing dynamic memory in C. We touched on a few of these garbage collection techniques, including RAII and exception handling.