Chapter 20 Working with Memory

20-1. Type Conversions

All of C's data objects have a specific type. A numeric variable can be an int or a float, a pointer can be a pointer to a double or char, and so on. Programs often require that different types be combined in expressions and statements. What happens in such cases? Sometimes C automatically handles the different types, so you don't need to be concerned. Other times, you must explicitly convert one data type to another to avoid erroneous results. You've seen this in earlier chapters when you had to convert or cast a type void pointer to a specific type before using it. In this and other situations, you need a clear understanding of when explicit type conversions are necessary and what types of errors can result when the proper conversion isn't applied. The following sections cover C's automatic and explicit type conversions.

20-1-1. Automatic Type Conversions

As the name implies, automatic type conversions are performed automatically by the C compiler without your needing to do anything. However, you should be aware of what's going on so that you can understand how C evaluates expressions.

Type Promotion in Expressions

When a C expression is evaluated, the resulting value has a particular data type. If all the components in the expression have the same type, the resulting type is that type as well. For example, if x and y are both type int, the following expression is type int also:

```
x + y
```

What if the components of an expression have different types? In that case, the expression has the same type as its most comprehensive component. From least-comprehensive to most-comprehensive, the numerical data types are

char

int

long

float

double

Thus, an expression containing an int and a char evaluates to type int, an expression containing a long and a float evaluates to type float, and so on.

Within expressions, individual operands are promoted as necessary to match the associated operands in the expression. Operands are promoted, in pairs, for each binary operator in the expression. Of course, promotion isn't needed if both operands are the same type. If they aren't, promotion follows these rules:

- If either operand is a double, the other operand is promoted to type double.
- If either operand is a float, the other operand is promoted to type float.
- If either operand is a long, the other operand is converted to type long.

For example, if x is an int and y is a float, evaluating the expression x/y causes x to be promoted to type float before the expression is evaluated. This doesn't mean that the type of variable x is changed. It means that a type float

copy of x is created and used in the expression evaluation. The value of the expression is, as you just learned, type float. Likewise, if x is a type double and y is a type float, y will be promoted to double.

Conversion by Assignment

Promotions also occur with the assignment operator. The expression on the right side of an assignment statement is always promoted to the type of the data object on the left side of the assignment operator. Note that this might cause a "demotion" rather than a promotion. If f is a type float and i is a type int, i is promoted to type float in this assignment statement:

```
f = i;
```

In contrast, the assignment statement

```
i = f;
```

causes f to be demoted to type int. Its fractional part is lost on assignment to i. Remember that f itself isn't changed at all; promotion affects only a copy of the value. Thus, after the following statements are executed:

```
float f = 1.23;
int i;
i = f;
```

the variable i has the value 1, and f still has the value 1.23. As this example illustrates, the fractional part is lost when a floating-point number is converted to an integer type.

You should be aware that when an integer type is converted to a floating-point type, the resulting floating-point value might not exactly match the integer value. This is because the floating-point format used internally by the computer can't accurately represent every possible integer number. For example, the following code could result in display of 2.999995 instead of 3:

```
float f;
int i = 3;
f = i;
printf("%f", f);
```

In most cases, any loss of accuracy caused by this would be insignificant. To be sure, however, keep integer values in type int or type long variables.

20-1-2. Explicit Conversions Using Typecasts

A *typecast* uses the cast operator to explicitly control type conversions in your program. A typecast consists of a type name, in parentheses, before an expression. Casts can be performed on arithmetic expressions and pointers. The result is that the expression is converted to the type specified by the cast. In this manner, you can control the type of expressions in your program rather than relying on C's automatic conversions.

Casting Arithmetic Expressions

Casting an arithmetic expression tells the compiler to represent the value of the expression in a certain way. In effect, a cast is similar to a promotion, which was discussed earlier. However, a cast is under your control, not the compiler's. For example, if i is a type int, the expression

```
(float)i
```

casts i to type float. In other words, the program makes an internal copy of the value of i in floating-point format.

When would you use a typecast with an arithmetic expression? The most common use is to avoid losing the fractional part of the answer in an integer division. Listing 20.1 illustrates this. You should compile and run this program.

Listing 20.1. When one integer is divided by another, any fractional part of the answer is lost.

```
1: #include <stdio.h>
2:
3: main()
4: {
5:     int i1 = 100, i2 = 40;
6:     float f1;
7:
8:     f1 = i1/i2;
9:     printf("%lf\n", f1);
10:    return(0);
11: }
2.000000
```

ANALYSIS: The answer displayed by the program is 2.000000, but 100/40 evaluates to 2.5. What happened? The expression i1/i2 on line 8 contains two type int variables. Following the rules explained earlier in this chapter, the value of the expression is type int itself. As such, it can represent only whole numbers, so the fractional part of the answer is lost.

You might think that assigning the result of i1/i2 to a type float variable promotes it to type float. This is correct, but now it's too late; the fractional part of the answer is already gone.

To avoid this sort of inaccuracy, you must cast one of the type int variables to type float. If one of the variables is cast to type float, the previous rules tell you that the other variable is promoted automatically to type float, and the value of the expression is also type float. The fractional part of the answer is thus preserved. To demonstrate this, change line 8 in the source code so that the assignment statement reads as follows:

```
f1 = (float)i1/i2;
```

The program will then display the correct answer.

Casting Pointers

You have already been introduced to the casting of pointers. As you saw on Day 18, "Getting More from Functions," a type void pointer is a generic pointer; it can point to anything. Before you can use a void pointer, you must cast it to the proper type. Note that you don't need to cast a pointer in order to assign a value to it or to compare it with NULL. However, you must cast it before dereferencing it or performing pointer arithmetic with it. For more details on casting void pointers, review Day 18.

DO use a cast to promote or demote variable values when necessary.

DON'T use a cast just to prevent a compiler warning. You might find that using a cast gets rid of a warning, but before removing the warning this way, be sure you understand why you're getting the warning.

20-2. Allocating Memory Storage Space

The C library contains functions for allocating memory storage space at runtime, a process called *dynamic memory allocation*. This technique can have significant advantages over explicitly allocating memory in the program source code by declaring variables, structures, and arrays. This latter method, called *static memory allocation,* requires you to know when you're writing the program exactly how much memory you need. Dynamic memory allocation allows the program to react, while it's executing, to demands for memory, such as user input. All the functions for handling dynamic memory allocation require the header file STDLIB.H; with some compilers, MALLOC.H is required as well. Note that all allocation functions return a type void pointer. As you learned on Day 18, a type void pointer must be cast to the appropriate type before being used.

Before we move on to the details, a few words are in order about memory allocation. What exactly does it mean? Each computer has a certain amount of memory (random access memory, or RAM) installed. This amount varies from system to system. When you run a program, whether a word processor, a graphics program, or a C program you wrote yourself, the program is loaded from disk into the computer's memory. The memory space the program occupies includes the program code as well as space for all the program's static data--that is, data items that are declared in the source code. The memory left over is what's available for allocation using the functions in this section.

How much memory is available for allocation? It all depends. If you're running a large program on a system with only a modest amount of memory installed, the amount of free memory will be small. Conversely, when a small program is running on a multimegabyte system, plenty of memory will be available. This means that your programs can't make any assumptions about memory availability. When a memory allocation function is called, you must check its return value to ensure that the memory was allocated successfully. In addition, your programs must be able to gracefully handle the situation when a memory allocation request fails. Later in this chapter, you'll learn a technique for determining exactly how much memory is available.

Also note that your operating system might have an effect on memory availability. Some operating systems make only a portion of physical RAM available. DOS 6.x and earlier falls into this category. Even if your system has multiple megabytes of RAM, a DOS program will have direct access to only the first 640 KB. (Special techniques can be used to access the other memory, but these are beyond the scope of this book.) In contrast, UNIX usually will make all physical RAM available to a program. To complicate matters further, some operating systems, such as Windows and OS/2, provide virtual memory that permits storage space on the hard disk to be allocated as if it were RAM. In this situation, the amount of memory available to a program includes not only the RAM installed, but also the virtual-memory space on the hard disk.

For the most part, these operating system differences in memory allocation should be transparent to you. If you use one of the C functions to allocate memory, the call either succeeds or fails, and you don't need to worry about the details of what's happening.

20-2-1. The malloc() Function

In earlier chapters, you learned how to use the malloc() library function to allocate storage space for strings. The malloc() function isn't limited to allocating memory for strings, of course; it can allocate space for any storage need. This function allocates memory by the byte. Recall that malloc()'s prototype is

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

The argument size_t is defined in STDLIB.H as unsigned. The malloc() function allocates *num* bytes of storage space and returns a pointer to the first byte. This function returns NULL if the requested storage space couldn't be allocated or if *num* == 0. Review the section called "The malloc() Function" on Day 10, "Characters and Strings," if you're still a bit unclear on its operation.

Listing 20.2 shows you how to use malloc() to determine the amount of free memory available in your system. This program works fine under DOS, or in a DOS box under Windows. Be warned, however, that you might get strange results on systems such as OS/2 and UNIX, which use hard disk space to provide "virtual" memory. The program might take a very long time to exhaust available memory.

Listing 20.2. Using malloc() to determine how much memory is free.

```
1: /* Using malloc() to determine free memory.*/
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5:
6: /* Definition of a structure that is
7:    1024 bytes (1 kilobyte) in size.) */
8:
9: struct kilo {
10:    struct kilo *next;
11:    char dummy[1022];
12: };
13:
14: int FreeMem(void);
15:
16: main()
17: {
18:
19:    printf("You have %d kilobytes free.\n", FreeMem());
20:    return(0);
21: }
22:
23: int FreeMem(void)
24: {
25:    /*Returns the number of kilobytes (1024 bytes)
26:    of free memory. */
27:
28:    int counter;
29:    struct kilo *head, *current, *nextone;
30:
31:    current = head = (struct kilo*) malloc(sizeof(struct kilo));
32:
33:    if (head == NULL)
34:       return 0;      /*No memory available.*/
35:
36:    counter = 0;
37:    do
38:    {
39:       counter++;
40:       current->next = (struct kilo*) malloc(sizeof(struct kilo));
41:       current = current->next;
42:    } while (current != NULL);
43:
44:    /* Now counter holds the number of type kilo
45:       structures we were able to allocate. We
46:       must free them all before returning. */
47:
48:    current = head;
49:
```

```
50:    do
51:    {
52:       nextone = current->next;
53:       free(current);
54:       current = nextone;
55:    } while (nextone != NULL);
56:
57:    return counter;
58: }
You have 60 kilobytes free.
```

ANALYSIS: Listing 20.2 operates in a brute-force manner. It simply loops, allocating blocks of memory, until the malloc() function returns NULL, indicating that no more memory is available. The amount of available memory is then equal to the number of blocks allocated multiplied by the block size. The function then frees all the allocated blocks and returns the number of blocks allocated to the calling program. By making each block one kilobyte, the returned value indicates directly the number of kilobytes of free memory. As you may know, a kilobyte is not exactly 1000 bytes, but rather 1024 bytes (2 to the 10th power). We obtain a 1024-byte item by defining a structure, which we cleverly named kilo, that contains a 1022-byte array plus a 2-byte pointer.

The function FreeMem() uses the technique of linked lists, which was covered in more detail on Day 15, "Pointers: Beyond the Basics." In brief, a linked list consists of structures that contain a pointer to their own type (in addition to other data members). There is also a head pointer that points to the first item in the list (the variable head, a pointer to type kilo). The first item in the list points to the second, the second points to the third, and so on. The last item in the list is identified by a NULL pointer member. See Day 15 for more information.

20-2-2. The calloc() Function

The calloc() function also allocates memory. Rather than allocating a group of bytes as malloc() does, calloc() allocates a group of objects. The function prototype is

```
void *calloc(size_t num, size_t size);
```

Remember that size_t is a synonym for unsigned on most compilers. The argument *num* is the number of objects to allocate, and *size* is the size (in bytes) of each object. If allocation is successful, all the allocated memory is cleared (set to 0), and the function returns a pointer to the first byte. If allocation fails or if either num or size is 0, the function returns NULL.

Listing 20.3 illustrates the use of calloc().

Listing 20.3. Using the calloc() function to allocate memory storage space dynamically.

```
1: /* Demonstrates calloc(). */
2:
3: #include <stdlib.h>
4: #include <stdio.h>
5:
6: main()
7: {
8:     unsigned num;
9:     int *ptr;
10:
11:     printf("Enter the number of type int to allocate: ");
12:     scanf("%d", &num);
13:
```

```
14:     ptr = (int*)calloc(num, sizeof(int));
15:
16:     if (ptr != NULL)
17:         puts("Memory allocation was successful.");
18:     else
19:         puts("Memory allocation failed.");
20:     return(0);
21: }
Enter the number of type int to allocate: 100
Memory allocation was successful.
Enter the number of type int to allocate: 99999999
Memory allocation failed.
```

This program prompts for a value on lines 11 and 12. This number determines how much space the program will attempt to allocate. The program attempts to allocate enough memory (line 14) to hold the specified number of int variables. If the allocation fails, the return value from calloc() is NULL; otherwise, it's a pointer to the allocated memory. In the case of this program, the return value from calloc() is placed in the int pointer, ptr. An if statement on lines 16 through 19 checks the status of the allocation based on ptr's value and prints an appropriate message.

Enter different values and see how much memory can be successfully allocated. The maximum amount depends, to some extent, on your system configuration. Some systems can allocate space for 25,000 occurrences of type int, whereas 30,000 fails. Remember that the size of an int depends on your system.

20-2-3. The realloc() Function

The realloc() function changes the size of a block of memory that was previously allocated with malloc() or calloc(). The function prototype is

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

The ptr argument is a pointer to the original block of memory. The new size, in bytes, is specified by *size*. There are several possible outcomes with realloc():

- If sufficient space exists to expand the memory block pointed to by ptr, the additional memory is allocated and the function returns ptr.
- If sufficient space does not exist to expand the current block in its current location, a new block of the size for size is allocated, and existing data is copied from the old block to the beginning of the new block. The old block is freed, and the function returns a pointer to the new block.
- If the ptr argument is NULL, the function acts like malloc(), allocating a block of size bytes and returning a pointer to it.
- If the argument size is 0, the memory that ptr points to is freed, and the function returns NULL.
- If memory is insufficient for the reallocation (either expanding the old block or allocating a new one), the function returns NULL, and the original block is unchanged.

Listing 20.4 demonstrates the use of realloc().

Listing 20.4. Using realloc() to increase the size of a block of dynamically allocated memory.

```
1: /* Using realloc() to change memory allocation. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
```

```
6:
7: main()
8: {
9:     char buf[80], *message;
10:
11:     /* Input a string. */
12:
13:     puts("Enter a line of text.");
14:     gets(buf);
15:
16:     /* Allocate the initial block and copy the string to it. */
17:
18:     message = realloc(NULL, strlen(buf)+1);
19:     strcpy(message, buf);
20:
21:     /* Display the message. */
22:
23:     puts(message);
24:
25:     /* Get another string from the user. */
26:
27:     puts("Enter another line of text.");
28:     gets(buf);
29:
30:     /* Increase the allocation, then concatenate the string to it.
*/
31:
32:     message = realloc(message,(strlen(message) + strlen(buf)+1));
33:     strcat(message, buf);
34:
35:     /* Display the new message. */
36:     puts(message);
37:     return(0);
38: }
Enter a line of text.
This is the first line of text.
This is the first line of text.
Enter another line of text.
This is the second line of text.
This is the first line of text.This is the second line of text.
```

ANALYSSIS: This program gets an input string on line 14, reading it into an array of characters called buf. The string is then copied into a memory location pointed to by message (line 19). message was allocated using realloc() on line 18. realloc() was called even though there was no previous allocation. By passing NULL as the first parameter, realloc() knows that this is a first allocation.

Line 28 gets a second string in the buf buffer. This string is concatenated to the string already held in message. Because message is just big enough to hold the first string, it needs to be reallocated to make room to hold both the first and second strings. This is exactly what line 32 does. The program concludes by printing the final concatenated string.

20-2-4. The free() Function

When you allocate memory with either malloc() or calloc(), it is taken from the dynamic memory pool that is available to your program. This pool is sometimes called the *heap,* and it is finite. When your program finishes using a particular block of dynamically allocated memory, you should deallocate, or free, the memory to make it available for future use. To free memory that was allocated dynamically, use free(). Its prototype is

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

The free() function releases the memory pointed to by ptr. This memory must have been allocated with malloc(), calloc(), or realloc(). If ptr is NULL, free() does nothing. Listing 20.5 demonstrates the free() function. (It was also used in Listing 20.2.)

Listing 20.5. Using free() to release previously allocated dynamic memory.

```
1: /* Using free() to release allocated dynamic memory. */
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <string.h>
6:
7: #define BLOCKSIZE 30000
8:
9: main()
10: {
11:     void *ptr1, *ptr2;
12:
13:     /* Allocate one block. */
14:
15:     ptr1 = malloc(BLOCKSIZE);
16:
17:     if (ptr1 != NULL)
18:         printf("\nFirst allocation of %d bytes
successful.",BLOCKSIZE);
19:     else
20:     {
21:         printf("\nAttempt to allocate %d bytes
failed.\n",BLOCKSIZE);
22:         exit(1);
23:     }
24:
25:     /* Try to allocate another block. */
26:
27:     ptr2 = malloc(BLOCKSIZE);
28:
29:     if (ptr2 != NULL)
30:     {
31:         /* If allocation successful, print message and exit. */
32:
33:         printf("\nSecond allocation of %d bytes successful.\n",
34:                BLOCKSIZE);
35:         exit(0);
36:     }
37:
```

```
38:     /* If not successful, free the first block and try again.*/
39:
40:     printf("\nSecond attempt to allocate %d bytes
failed.",BLOCKSIZE);
41:     free(ptr1);
42:     printf("\nFreeing first block.");
43:
44:     ptr2 = malloc(BLOCKSIZE);
45:
46:     if (ptr2 != NULL)
47:         printf("\nAfter free(), allocation of %d bytes
successful.\n",
48:                 BLOCKSIZE);
49:     return(0);
50: }
First allocation of 30000 bytes successful.
Second allocation of 30000 bytes successful.
```

ANALYSIS: This program tries to dynamically allocate two blocks of memory. It uses the defined constant BLOCKSIZE to determine how much to allocate. Line 15 does the first allocation using malloc(). Lines 17 through 23 check the status of the allocation by checking to see whether the return value was equal to NULL. A message is displayed, stating the status of the allocation. If the allocation failed, the program exits. Line 27 tries to allocate a second block of memory, again checking to see whether the allocation was successful (lines 29 through 36). If the second allocation was successful, a call to exit() ends the program. If it was not successful, a message states that the attempt to allocate memory failed. The first block is then freed with free() (line 41), and a new attempt is made to allocate the second block.

You might need to modify the value of the symbolic constant BLOCKSIZE. On some systems, the value of 30000 produces the following program output:

```
First allocation of 30000 bytes successful.
Second attempt to allocate 30000 bytes failed.
Freeing first block.
After free(), allocation of 30000 bytes successful.
```

On systems with virtual memory, of course, allocation will always succeed.

DO free allocated memory when you're done with it.

DON'T assume that a call to malloc(), calloc(), or realloc() was successful. In other words, always check to see that the memory was indeed allocated.

20-3. Manipulating Memory Blocks

So far today, you've seen how to allocate and free blocks of memory. The C library also contains functions that can be used to manipulate blocks of memory--setting all bytes in a block to a specified value, and copying and moving information from one location to another.

20-3-1. The memset() Function

To set all the bytes in a block of memory to a particular value, use memset(). The function prototype is

```
void * memset(void *dest, int c, size_t count);
```

The argument dest points to the block of memory. c is the value to set, and *count* is the number of bytes, starting at dest, to be set. Note that while c is a type int, it is treated as a type char. In other words, only the low-order byte is used, and you can specify values of c only in the range 0 through 255.

Use memset() to initialize a block of memory to a specified value. Because this function can use only a type char as the initialization value, it is not useful for working with blocks of data types other than type char, except when you want to initialize to 0. In other words, it wouldn't be efficient to use memset() to initialize an array of type int to the value 99, but you could initialize all array elements to the value 0. memset() will be demonstrated in Listing 20.6.

20-3-2. The memcpy() Function

memcpy() copies bytes of data between memory blocks, sometimes called *buffers*. This function doesn't care about the type of data being copied--it simply makes an exact byte-for-byte copy. The function prototype is

```
void *memcpy(void *dest, void *src, size_t count);
```

The arguments dest and src point to the destination and source memory blocks, respectively. *count* specifies the number of bytes to be copied. The return value is dest. If the two blocks of memory overlap, the function might not operate properly--some of the data in src might be overwritten before being copied. Use the memmove() function, discussed next, to handle overlapping memory blocks. memcpy() will be demonstrated in Listing 20.6.

20-3-3. The memmove() Function

memmove() is very much like memcpy(), copying a specified number of bytes from one memory block to another. It's more flexible, however, because it can handle overlapping memory blocks properly. Because memmove() can do everything memcpy() can do with the added flexibility of dealing with overlapping blocks, you rarely, if ever, should have a reason to use memcpy(). The prototype is

```
void *memmove(void *dest, void *src, size_t count);
```

dest and src point to the destination and source memory blocks, and *count* specifies the number of bytes to be copied. The return value is dest. If the blocks overlap, this function ensures that the source data in the overlapped region is copied before being overwritten. Listing 20.6 demonstrates memset(), memcpy(), and memmove().

Listing 20.6. A demonstration of memset(), memcpy(), and memmove().

```
1: /* Demonstrating memset(), memcpy(), and memmove(). */
2:
3: #include <stdio.h>
4: #include <string.h>
4:
5: char message1[60] = "Four score and seven years ago ...";
6: char message2[60] = "abcdefghijklmnopqrstuvwxyz";
7: char temp[60];
8:
9: main()
10: {
11:    printf("\nmessage1[] before memset():\t%s", message1);
12:    memset(message1 + 5, `@', 10);
13:    printf("\nmessage1[] after memset():\t%s", message1);
```

```
14:
15:    strcpy(temp, message2);
16:    printf("\n\nOriginal message: %s", temp);
17:    memcpy(temp + 4, temp + 16, 10);
18:    printf("\nAfter memcpy() without overlap:\t%s", temp);
19:    strcpy(temp, message2);
20:    memcpy(temp + 6, temp + 4, 10);
21:    printf("\nAfter memcpy() with overlap:\t%s", temp);
22:
23:    strcpy(temp, message2);
24:    printf("\n\nOriginal message: %s", temp);
25:    memmove(temp + 4, temp + 16, 10);
26:    printf("\nAfter memmove() without overlap:\t%s", temp);
27:    strcpy(temp, message2);
28:    memmove(temp + 6, temp + 4, 10);
29:    printf("\nAfter memmove() with overlap:\t%s\n", temp);
30:
31: }
message1[] before memset():     Four score and seven years ago ...
message1[] after memset():      Four @@@@@@@@@@seven years ago ...
Original message: abcdefghijklmnopqrstuvwxyz
After memcpy() without overlap: abcdqrstuvwxyzopqrstuvwxyz
After memcpy() with overlap:    abcdefefefefefefqrstuvwxyz
Original message: abcdefghijklmnopqrstuvwxyz
After memmove() without overlap:        abcdqrstuvwxyzopqrstuvwxyz
After memmove() with overlap:   abcdefefghijklmnqrstuvwxyz
```

ANALYSIS: The operation of memset() is straightforward. Note how the pointer notation message1 + 5 is used to specify that memset() is to start setting characters at the sixth character in message1[] (remember, arrays are zero-based). As a result, the 6th through 15th characters in message1[] have been changed to @.

When source and destination do not overlap, memcpy() works fine. The 10 characters of temp[] starting at position 17 (the letters q through z) have been copied to positions 5 though 14, where the letters e though n were originally located. If, however, the source and destination overlap, things are different. When the function tries to copy 10 characters starting at position 4 to position 6, an overlap of 8 positions occurs. You might expect the letters e through n to be copied over the letters g through p. Instead, the letters e and f are repeated five times.

If there's no overlap, memmove() works just like memcpy(). With overlap, however, memmove() copies the original source characters to the destination.

DO use memmove() instead of memcpy() in case you're dealing with overlapping memory regions.

DON'T try to use memset() to initialize type int, float, or double arrays to any value other than 0.

20-4. Working with Bits

As you may know, the most basic unit of computer data storage is the bit. There are times when being able to manipulate individual bits in your C program's data is very useful. C has several tools that let you do this.

The C bitwise operators let you manipulate the individual bits of integer variables. Remember, a *bit* is the smallest possible unit of data storage, and it can have only one of two values: 0 or 1. The bitwise operators can be used only with integer types: char, int, and long. Before continuing with this section, you should be familiar with binary notation--the way the computer internally stores integers. If you need to review binary notation, refer to Appendix C, "Working with Binary and Hexadecimal Numbers."

The bitwise operators are most frequently used when your C program interacts directly with your system's hardware--a topic that is beyond the scope of this book. They do have other uses, however, which this chapter introduces.

20-4-1. The Shift Operators

Two shift operators shift the bits in an integer variable by a specified number of positions. The << operator shifts bits to the left, and the >> operator shifts bits to the right. The syntax for these binary operators is

```
x << n
```

and

```
x >> n
```

Each operator shifts the bits in x by n positions in the specified direction. For a right shift, zeros are placed in the n high-order bits of the variable; for a left shift, zeros are placed in the n low-order bits of the variable. Here are a few examples:

Binary 00001100 (decimal 12) right-shifted by 2 evaluates to binary 00000011 (decimal 3).

Binary 00001100 (decimal 12) left-shifted by 3 evaluates to binary 01100000 (decimal 96).

Binary 00001100 (decimal 12) right-shifted by 3 evaluates to binary 00000001 (decimal 1).

Binary 00110000 (decimal 48) left-shifted by 3 evaluates to binary 10000000 (decimal 128).

Under certain circumstances, the shift operators can be used to multiply and divide an integer variable by a power of 2. Left-shifting an integer by *n* places has the same effect as multiplying it by 2*n*, and right-shifting an integer has the same effect as dividing it by 2*n*. The results of a left-shift multiplication are accurate only if there is no overflow--that is, if no bits are "lost" by being shifted out of the high-order positions. A right-shift division is an integer division, in which any fractional part of the result is lost. For example, if you right-shift the value 5 (binary 00000101) by one place, intending to divide by 2, the result is 2 (binary 00000010) instead of the correct 2.5, because the fractional part (the .5) is lost. Listing 20.7 demonstrates the shift operators.

Listing 20.7. Using the shift operators.

```
1:  /* Demonstrating the shift operators. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:       unsigned int y, x = 255;
8:       int count;
9:
10:      printf("Decimal\t\tshift left by\tresult\n");
11:
12:      for (count = 1; count < 8; count++)
```

```
13:      {
14:          y = x << count;
15:          printf("%d\t\t%d\t\t%d\n", x, count, y);
16:      }
17:      printf("\n\nDecimal\t\tshift right by\tresult\n");
18:
19:      for (count = 1; count < 8; count++)
20:      {
21:           y = x >> count;
22:           printf("%d\t\t%d\t\t%d\n", x, count, y);
23:      }
24:      return(0);
25: }
Decimal         shift left by   result
255             1               254
255             2               252
255             3               248
255             4               240
255             5               224
255             6               192
255             7               128
Decimal         shift right by  result
255             1               127
255             2               63
255             3               31
255             4               15
255             5               7
255             6               3
255             7               1
```

20-4-2. The Bitwise Logical Operators

Three bitwise logical operators are used to manipulate individual bits in an integer data type, as shown in Table 20.1. These operators have names similar to the TRUE/FALSE logical operators you learned about in earlier chapters, but their operations differ.

Table 20.1. The bitwise logical operators.

Operator	Description
&	AND
|	Inclusive OR
^	Exclusive OR

These are all binary operators, setting bits in the result to 1 or 0 depending on the bits in the operands. They operate as follows:

- Bitwise AND sets a bit in the result to 1 only if the corresponding bits in both operands are 1; otherwise, the bit is set to 0. The AND operator is used to turn off, or clear, one or more bits in a value.
- Bitwise inclusive OR sets a bit in the result to 0 only if the corresponding bits in both operands are 0; otherwise, the bit is set to 1. The OR operator is used to turn on, or set, one or more bits in a value.

- Bitwise exclusive OR sets a bit in the result to 1 if the corresponding bits in the operands are different (if one is 1 and the other is 0); otherwise, the bit is set to 0.

The following are examples of how these operators work:

Operation	*Example*
AND	11110000
	& 01010101

	01010000
Inclusive OR	11110000
	| 01010101

	11110101
Exclusive OR	11110000
	^ 01010101

	10100101

You just read that bitwise AND and bitwise inclusive OR can be used to clear or set, respectively, specified bits in an integer value. Here's what that means. Suppose you have a type char variable, and you want to ensure that the bits in positions 0 and 4 are cleared (that is, equal to 0) and that the other bits stay at their original values. If you AND the variable with a second value that has the binary value 11101110, you'll obtain the desired result. Here's how this works:

In each position where the second value has a 1, the result will have the same value, 0 or 1, as was present in that position in the original variable:

```
0 & 1 == 0
1 & 1 == 1
```

In each position where the second value has a 0, the result will have a 0 regardless of the value that was present in that position in the original variable:

```
0 & 0 == 0
1 & 0 == 0
```

Setting bits with OR works in a similar way. In each position where the second value has a 1, the result will have a 1, and in each position where the second value has a 0, the result will be unchanged:

```
0 | 1 == 1
1 | 1 == 1
0 | 0 == 0
1 | 0 == 1
```

20-4-3. The Complement Operator

The final bitwise operator is the complement operator, ~. This is a unary operator. Its action is to reverse every bit in its operand, changing all 0s to 1s, and vice versa. For example, ~254 (binary 11111110) evaluates to 1 (binary 00000001).

All the examples in this section have used type char variables containing 8 bits. For larger variables, such as type int and type long, things work exactly the same.

20-4-4. Bit Fields in Structures

The final bit-related topic is the use of bit fields in structures. On Day 11, "Structures," you learned how to define your own data structures, customizing them to fit your program's data needs. By using bit fields, you can accomplish even greater customization and save memory space as well.

A *bit field* is a structure member that contains a specified number of bits. You can declare a bit field to contain one bit, two bits, or whatever number of bits are required to hold the data stored in the field. What advantage does this provide?

Suppose that you're programming an employee database program that keeps records on your company's employees. Many of the items of information that the database stores are of the yes/no variety, such as "Is the employee enrolled in the dental plan?" or "Did the employee graduate from college?" Each piece of yes/no information can be stored in a single bit, with 1 representing yes and 0 representing no.

Using C's standard data types, the smallest type you could use in a structure is a type char. You could indeed use a type char structure member to hold yes/no data, but seven of the char's eight bits would be wasted space. By using bit fields, you could store eight yes/no values in a single char.

Bit fields aren't limited to yes/no values. Continuing with this database example, imagine that your firm has three different health insurance plans. Your database needs to store data about the plan in which each employee is enrolled (if any). You could use 0 to represent no health insurance and use the values 1, 2, and 3 to represent the three plans. A bit field containing two bits is sufficient, because two binary bits can represent values of 0 through 3. Likewise, a bit field containing three bits could hold values in the range 0 through 7, four bits could hold values in the range 0 through 15, and so on.

Bit fields are named and accessed like regular structure members. All bit fields have type unsigned int, and you specify the size of the field (in bits) by following the member name with a colon and the number of bits. To define a structure with a one-bit member named dental, another one-bit member named college, and a two-bit member named health, you would write the following:

```
struct emp_data {
unsigned dental      : 1;
unsigned college     : 1;
unsigned health      : 2;
...
};
```

The ellipsis (...) indicates space for other structure members. The members can be bit fields or fields made up of regular data types. Note that bit fields must be placed first in the structure definition, before any nonbit field structure members. To access the bit fields, use the structure member operator just as you do with any structure member. For the example, you can expand the structure definition to something more useful:

```
struct emp_data {
unsigned dental      : 1;
unsigned college     : 1;
unsigned health      : 2;
char fname[20];
char lname[20];
```

```
char ssnumber[10];
};
```

You can then declare an array of structures:

```
struct emp_data workers[100];
```

To assign values to the first array element, write something like this:

```
workers[0].dental = 1;
workers[0].college = 0;
workers[0].health = 2;
strcpy(workers[0].fname, "Mildred");
```

Your code would be clearer, of course, if you used symbolic constants YES and NO with values of 1 and 0 when working with one-bit fields. In any case, you treat each bit field as a small, unsigned integer with the given number of bits. The range of values that can be assigned to a bit field with *n* bits is from 0 to 2^{n-1}. If you try to assign an out-of-range value to a bit field, the compiler won't report an error, but you will get unpredictable results.

DO use defined constants YES and NO or TRUE and FALSE when working with bits. These are much easier to read and understand than 1 and 0.

DON'T define a bit field that takes 8 or 16 bits. These are the same as other available variables such as type char or int.

20-5. Summary

This chapter covered a variety of C programming topics. You learned how to allocate, reallocate, and free memory at runtime, and you saw commands that give you flexibility in allocating storage space for program data. You also saw how and when to use typecasts with variables and pointers. Forgetting about typecasts, or using them improperly, is a common cause of hard-to-find program bugs, so this is a topic worth reviewing! You also learned how to use the memset(), memcpy(), and memmove() functions to manipulate blocks of memory. Finally, you saw the ways in which you can manipulate and use individual bits in your programs.

Q&A

Q What's the advantage of dynamic memory allocation? Why can't I just declare the storage space I need in my source code?

A If you declare all your data storage in your source code, the amount of memory available to your program is fixed. You have to know ahead of time, when you write the program, how much memory will be needed. Dynamic memory allocation lets your program control the amount of memory used to suit the current conditions and user input. The program can use as much memory as it needs, up to the limit of what's available in the computer.

Q Why would I ever need to free memory?

A When you're first learning to use C, your programs aren't very big. As your programs grow, their use of memory also grows. You should try to write your programs to use memory as efficiently as possible. When you're done with memory, you should release it. If you write programs that work in a multitasking environment, other applications might need memory that you aren't using.

Q What happens if I reuse a string without calling realloc()?

A You don't need to call realloc() if the string you're using was allocated enough room. Call realloc() when your current string isn't big enough. Remember, the C compiler lets you do almost anything, even things you shouldn't! You can overwrite one string with a bigger string as long as the new string's length is equal to or smaller than the original string's allocated space. However, if the new string is bigger, you will also overwrite whatever came after the string in memory. This could be nothing, or it could be vital data. If you need a bigger allocated section of memory, call realloc().

Q What's the advantage of the memset(), memcpy(), and memmove() functions? Why can't I just use a loop with an assignment statement to initialize or copy memory?

A You can use a loop with an assignment statement to initialize memory in some cases. In fact, sometimes this is the only way to do it--for example, setting all elements of a type float array to the value 1.23. In other situations, however, the memory will not have been assigned to an array or list, and the mem...() functions are your only choice. There are also times when a loop and assignment statement would work, but the mem...() functions are simpler and faster.

Q When would I use the shift operators and the bitwise logical operators?

A The most common use for these operators is when a program is interacting directly with the computer hardware--a task that often requires specific bit patterns to be generated and interpreted. This topic is beyond the scope of this book. Even if you never need to manipulate hardware directly, you can use the shift operators, in certain circumstances, to divide or multiply integer values by powers of 2.

Q Do I really gain that much by using bit fields?

A Yes, you can gain quite a bit with bit fields. (Pun intended!) Consider a circumstance similar to the example in this chapter in which a file contains information from a survey. People are asked to answer TRUE or FALSE to the questions asked. If you ask 100 questions of 10,000 people and store each answer as a type char as T or F, you will need 10,000 * 100 bytes of storage (because a character is 1 byte). This is 1 million bytes of storage. If you use bit fields instead and allocate one bit for each answer, you will need 10,000 * 100 bits. Because 1 byte holds 8 bits, this amounts to 130,000 bytes of data, which is significantly less than 1 million bytes.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the difference between the malloc() and calloc() memory-allocation functions?

2. What is the most common reason for using a typecast with a numeric variable?

3. What variable type do the following expressions evaluate to? Assume that c is a type char variable, i is a type int variable, l is a type long variable, and f is a type float variable.

a. (c + i + l)
b. (i + 32)
c. (c + `A')
d. (i + 32.0)
e. (100 + 1.0)

4. What is meant by dynamically allocating memory?

5. What is the difference between the memcpy() function and the memmove() function?

6. Imagine that your program uses a structure that must (as one of its members) store the day of the week as a value between 1 and 7. What's the most memory-efficient way to do so?

7. What is the smallest amount of memory in which the current date can be stored? (Hint: month/day/year--think of year as an offset from 1900.)

8. What does 10010010 << 4 evaluate to?

9. What does 10010010 >> 4 evaluate to?

10. Describe the difference between the results of the following two expressions:

```
(01010101 ^ 11111111 )
( ~01010101 )
```

Exercises

1. Write a malloc() command that allocates memory for 1,000 longs.

2. Write a calloc() command that allocates memory for 1,000 longs.
3. Assume that you have declared an array as follows:

```
float data[1000];
```

Show two ways to initialize all elements of the array to 0. Use a loop and an assignment statement for one method, and the memset() function for the other.

4. BUG BUSTER: Is anything wrong with the following code?

```
void func()
{
int number1 = 100, number2 = 3;
float answer;
answer = number1 / number2;
printf("%d/%d = %lf", number1, number2, answer)
}
```

5. BUG BUSTER: What, if anything, is wrong with the following code?

```
void *p;
p = (float*) malloc(sizeof(float));
*p = 1.23;
```

6. BUG BUSTER: Is the following structure allowed?

```
struct quiz_answers {
char student_name[15];
unsigned answer1   : 1;
unsigned answer2   : 1;
unsigned answer3   : 1;
unsigned answer4   : 1;
unsigned answer5   : 1;
}
```

Answers are not provided for the following exercises.

7. Write a program that uses each of the bitwise logical operators. The program should apply the bitwise operator to a number and then reapply it to the result. You should observe the output to be sure you understand what's going on.
8. Write a program that displays the binary value of a number. For instance, if the user enters 3, the program should display 00000011. (Hint: You will need to use the bitwise operators.)