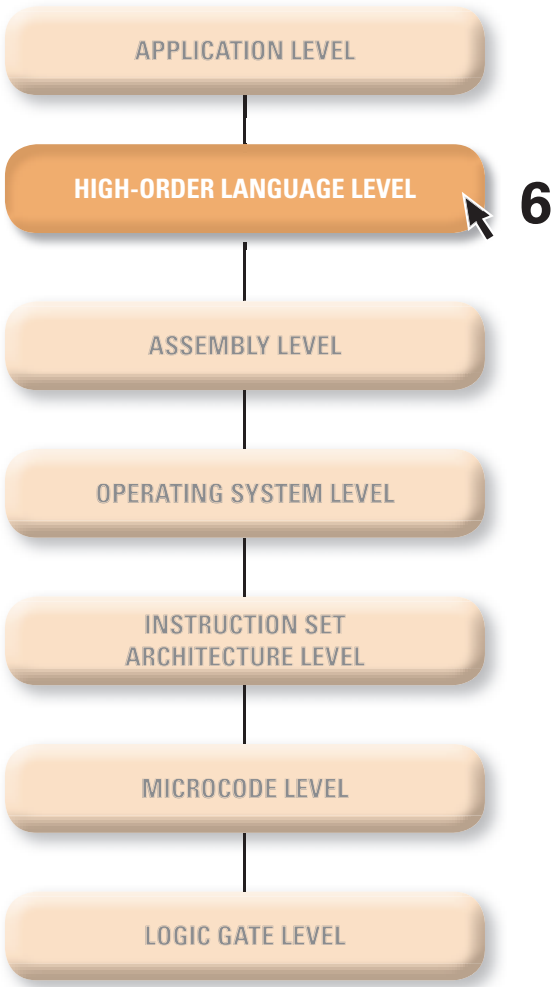
LEVEL

6

High-Order Language



CHAPTER 2

C

TABLE OF CONTENTS

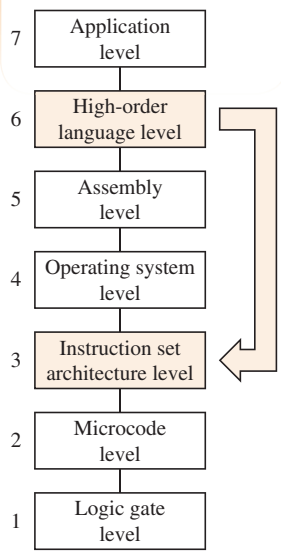


FIGURE 2.1
The function of a compiler, which translates a program in a Level 6 language to an equivalent program in a language at a lower level.

A program inputs information, processes it, and outputs the results. This chapter shows how a C program inputs, processes, and outputs values. It reviews programming at Level HOL6 and assumes that you have experience writing programs in some high-order language—not necessarily C—such as C++, Java, or Python. Because this text presents concepts that are common to all those languages, you should be able to follow the discussion despite any differences in the language with which you are familiar.

2.1 Variables

A computer can directly execute statements in machine language only at Level ISA3, the instruction set architecture level. So a Level HOL6 statement must first be translated to Level ISA3 before executing. FIGURE 2.1 shows the function of a compiler, which performs the translation from a Level HOL6 language to the Level ISA3 language. The figure shows translation to Level 3. Some compilers translate from Level 6 to Level 5, which then requires another translation from Level 5 to Level 3.

The C Compiler

To execute the programs in this text, you need access to a C compiler. Running a program is a three-step process:

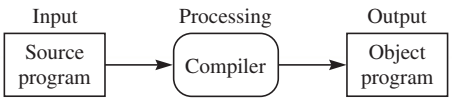
- Write the program in C using a text editor. This version is called the *source program*.
- Invoke the compiler to translate, or compile, the source program from C to machine language. The machine language version is called the *object program*.
- Execute the object program.

Some systems allow you to specify the last two of these steps with a single command, usually called the *run* command. Whether or not you specify the compilation and execution separately, some translation is required before a Level HOL6 program can be executed.

When you write the source program, it will be saved in a file on disk just as any other text document would be. The compiler will produce another file, called a *code file*, for the object program. Depending on your compiler, the object program may or may not be visible on your file directory after the compilation.

If you want to execute a program that was previously compiled, you do not need to translate it again. You can simply execute the object program

FIGURE 2.2
The compiler as a program.



directly. If you ever delete the object program from your disk, you can always get it back from the source program by compiling again. But the translation can go only from a high level to a low level. If you delete the source program, you cannot recover it from the object program.

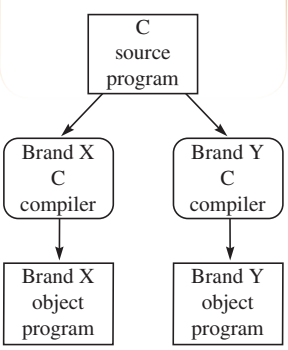
Your C compiler is software, not hardware. It is a program that is stored in a file on your disk. Like all programs, the compiler has input, does processing, and produces output. FIGURE 2.2 shows that the input to the compiler is the source program and the output is the object program.

Machine Independence

Level ISA3 languages are machine dependent. If you write a program in a Level ISA3 language for execution on a Brand X computer, it cannot run on a Brand Y computer. An important property of the languages at Level HOL6 is their machine independence. If you write a program in a Level HOL6 language for execution on a Brand X computer, it will run with only slight modification on a Brand Y computer.

FIGURE 2.3 shows how C achieves its machine independence. Suppose you write an applications program in C to do some statistical analysis. You want to sell it to people who own Brand X computers and to others who own Brand Y. The statistics program can be executed only if it is in machine language. Because machine language is machine dependent, you will need two machine-language versions, one for Brand X and one for Brand Y. Because C is a common high-order language, you will probably have access to a C compiler for the Brand X machine and a C compiler for the Brand Y machine. If so, you can simply invoke the Brand X C compiler on one machine to produce the Brand X machine-language version, and invoke the Brand Y C compiler on the other machine for the Brand Y version. You need to write only one C program.

FIGURE 2.3
The machine independence of a Level HOL6 language.



The C Memory Model

The C programming language has three different kinds of variables—global variables, local variables, and dynamically allocated variables. The value of a variable is stored in the main memory of a computer, but where in memory

it is stored depends on the kind of variable. There are three special sections of memory corresponding to the three kinds of variables:

The C memory model

- Global variables are stored at a fixed location in memory.
- Local variables and parameters are stored on the run-time stack.
- Dynamically allocated variables are stored on the heap.

Global variables are declared outside of any function and remain in place throughout the execution of the entire program. Local variables are declared within a function. They come into existence when the function is called and cease to exist when the function terminates. Dynamically allocated variables come into existence with the execution of the `malloc()` function and cease to exist with the execution of the `free()` function.

The push and pop operations

A stack is a container of values that stores values with the push operation and retrieves them with the pop operation. The policy for storage and retrieval is last in, first out. That is, when you pop a value from a stack, the value you get is the last one that was pushed. For this reason, a stack is sometimes called a *LIFO list*, where *LIFO* is an acronym for *last in*, *first out*.

Every C statement that executes is part of a function. A C function has a return type, a name, and a list of parameters. A program consists of a special function whose name is `main`, which is a function that is called by the operating system. A program executes by executing the statements in the main function. It is possible for a main statement to call another function. When a function executes, allocation on the run-time stack takes place in the following order:

Function call

- Push storage for the return value.
- Push the actual parameters.
- Push the return address.
- Push storage for the local variables.

Then, when the function terminates, deallocation from the run-time stack takes place in the opposite order:

Function return

- Pop the local variables.
- Pop the return address and use it to determine the next instruction to execute.
- Pop the parameters.
- Pop the return value and use it as specified in the calling statement.

These actions occur whether the function is the main function or is a function called by a statement in another function.

The programs in this chapter illustrate the memory model of the C programming language. Later chapters show the object code for the same programs after the compiler translates them to Level Asmb5.

Global Variables and Assignment Statements

Every C variable has three attributes:

The three attributes of a C variable

- Name
- Type
- Value

A variable's name is an identifier determined arbitrarily by the programmer. A variable's type specifies the kind of values it can have. FIGURE 2.4 shows a program that declares two global variables, inputs values for them, operates on the values, and outputs the result. This is a nonsense program whose sole purpose is to illustrate some features of the C language.

FIGURE 2.4
The assignment statement with global variables.

```
// Stan Warford
// A nonsense program to illustrate global variables

#include <stdio.h>

char ch;
int j;

int main() {
   scanf("%c %d", &ch, &j);
   j += 5;
   ch++;
   printf("%c\n%d\n", ch, j);
   return 0;
}
```

Input

```
M 419
```

Output

```
N
424
```

The first two lines in Figure 2.4 are comments, which are ignored by the compiler. Comments in a C source program begin with two slash characters, `//`, and continue until the end of the line. The next line in the program is

```
#include <stdio.h>
```

which is a compiler directive to make a library of functions available to the program. In this case, the library file `stdio.h`, which stands for *standard input/ouput*, contains the input function `scanf()` and the output function `printf()`, used later in the program. This directive, or one similar to it, is necessary for all programs that use `scanf()` and `printf()`.

The next two lines in the program

```
char ch;
int j;
```

declare two global variables. The name of the first variable is `ch`. Its type is character, as specified by the word `char`, which precedes its name. As with most variables, its value cannot be determined from the listing. Instead, it gets its value from an input statement. The name of the second variable is `j` with type integer, as specified by `int`. Every C program has a main function, which contains the executable statements of the program. In Figure 2.4, because the variables are declared outside the main program, they are global variables.

Global variables are declared outside of `main()`.

The next line in the program

```
int main() {
```

declares the main program to be a function that returns an integer. The C compiler must generate code that executes on a particular operating system. It is up to the operating system to interpret the value returned. The standard convention is that a return value of 0 indicates that no errors occurred during the program's execution. If an execution error does occur, the program is interrupted and returns some nonzero value without reaching the last executable statement of `main()`. What happens in such a case depends on the particular operating system and the nature of the error. All the C programs in this text use the common convention of returning 0 as the last executable statement in the main function.

The returned value for `main()`

The first executable statement in Figure 2.4 is

```
scanf("%c %d", &ch, &j);
```

The first parameter, `"%c %d"`, is the format string, which contains two conversion specifiers, `%c` and `%d`. The second and third parameters are `&ch` and `&j`, which receive the values that are input. The standard input device can be either the keyboard or a disk file. In a Unix environment, the default

input device is the keyboard. You can redirect the input to come from a disk file when you execute the program. This input statement gives the first value in the input stream to `ch` and the second value to `j`.

The conversion specifiers in the formatting string are placeholders and correspond in order to the remaining parameters in the parameter list. In Figure 2.4, placeholder `%c` corresponds to parameter `&ch`, and placeholder `%d` corresponds to parameter `&j`. Placeholder `%c` instructs the program to scan a character into variable `ch`, and specifier `%d` instructs the program to scan an optionally signed decimal integer into variable `j`. The ampersand character `&` is the C address operator and is required for variables in the `scanf()` function. Because the function changes the values of the variables, it requires the addresses of where the variables are stored in main memory instead of the values of the variables.

The address operator

The space character between `%c` and `%d` tells the input scanner to ignore any white-space characters like spaces and tabs before the integer. You can put any number of spaces before the number 419 in the input and the output will remain unchanged. However, if you put a space before the input character M, the program will not work correctly, because `ch` will get the space character. If you want to allow any number of spaces before the input character, put a space before the `%c` placeholder in the format string.

The second executable statement is

```
j += 5;
```

The assignment operator in C is `=`, which is pronounced "gets." The above statement is pronounced "j plus gets 5" and is equivalent to the assignment statement

The C assignment operator

```
j = j + 5;
```

which is pronounced "j gets j plus five."

Unlike some programming languages, C treats characters as if they were integers. You can perform arithmetic on them. The next executable statement

```
ch++;
```

adds 1 to `ch` with the increment operator. It is identical to the assignment statement

```
ch = ch + 1;
```

The next executable statement is

```
printf("%c\n%d\n", ch, j);
```

This output function uses the format string `"%c\n%d\n"` where `%c` and `%d` are again the placeholders for the remaining parameters `ch` and `j`.

FIGURE 2.5
The memory model for the program of Figure 2.4.

(a) Fixed location. **(b)** Run-time stack.

The standard output device can be either the screen or a disk file. In a Unix environment, the default output device is the screen. You can redirect the output to go to a disk file when you execute the program. `\n` is the newline character. This output statement sends the value of variable `ch` to the output device, moves the cursor to the start of the next line, sends the value of variable `j` to the output device, and then moves the cursor to the start of the next line. The `printf()` function does not use the `&` character in front of the variables because it does not change the values of the variables. Instead, it outputs the values they already have.

FIGURE 2.5 shows the memory model for the program of Figure 2.4 just before the program terminates. Storage for the global variables `ch` and `j` is allocated at a fixed location in memory, as Figure 2.5(a) shows.

Remember that when a function is called, four items are allocated on the run-time stack: return value, parameters, return address, and local variables. Because the main function in this program has no parameters and no local variables, the only items allocated on the stack are storage for the return value, labeled `retVal`, and the return address, labeled `retAddr`, in Figure 2.5(b). The figure shows the value for the return address as ra0, which is the address of the instruction in the operating system that will execute when the program terminates. The details of the operating system at Level OS4 are hidden from us at Level HOL6.

Local Variables

Local variables are declared within `main()`.

Global variables are allocated at a fixed position in main memory. Local variables, however, are allocated on the run-time stack. In a C program, local variables are declared within the main program. The program in FIGURE 2.6 declares a constant and three local variables that represent two scores on exams for a course, and the total score computed as their average plus a bonus.

Before the first variable is the constant `bonus`. A constant is like a variable in that it has a name, a type, and a value. Unlike a variable, however,

FIGURE 2.6
A C program that processes three local integer values.

```
#include <stdio.h>

int main() {
   const int bonus = 10;
   int exam1;
   int exam2;
   int score;
   scanf("%d %d", &exam1, &exam2);
   score = (exam1 + exam2) / 2 + bonus;
   printf("score = %d\n", score);
   return 0;
}
```

Input

```
68 84
```

Output

```
score = 86
```

the value of a constant cannot change. The value of this constant is 10, as specified by the initialization operator =.

The first executable statement in Figure 2.6 is

```
scanf("%d %d", &exam1, &exam2);
```

which gives the first value in the input stream to `exam1` and the second value to `exam2`. The second executable statement is

```
score = (exam1 + exam2) / 2 + bonus;
```

which adds the values in `exam1` and `exam2`, divides the sum by 2 to get their average, adds the bonus to the average, and then assigns the value to the variable `score`. Because `exam1`, `exam2`, and `2` are all integers, the division operator `/` represents integer division. If either `exam1` or `exam2` is declared to be a floating-point value, or if the divisor is written as `2.0` instead of `2`, then the division operator represents floating-point division. Integer division truncates the remainder, whereas floating-point division maintains the fractional part. To output the value of a floating-point variable, use `%f` for the conversion specifier in the format string.

Integer versus floating-point division

Example 2.1 If the input of the program in Figure 2.6 is

```
68 85
```

then the output is still

```
score = 86
```

The sum of the exams is 153. If you divide 153 by 2.0, you get the floating-point value 76.5. But if you divide 153 by 2, the / operator represents integer division and the fractional part is truncated—in other words, chopped off—yielding 76.

Example 2.2 If you declare `score` to have a double-precision, floating-point type as follows

```
double score;
```

and if you force the division to be floating point by changing `2` to `2.0` as follows

```
score = (exam1 + exam2) / 2.0 + bonus;
```

then the output is

```
score = 86.5
```

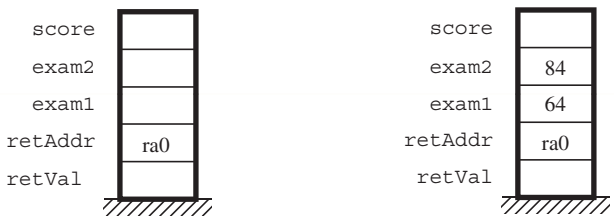
when the input is 68 and 85.

Floating-point division of two numbers produces only one value, the quotient. However, integer division produces two values—the quotient and the remainder—both of which are integers. You can compute the remainder of an integer division with the C modulus operator `%`. FIGURE 2.7 shows some examples of integer division and the modulus operation.

FIGURE 2.7
Some examples of integer division and the modulus operation.

Expression	Value	Expression	Value
15 / 3	5	15 % 3	0
14 / 3	4	14 % 3	2
13 / 3	4	13 % 3	1
12 / 3	4	12 % 3	0
11 / 3	3	11 % 3	2

FIGURE 2.8
The memory model for the local variables in the program of Figure 2.6.



(a) Before the input statement executes. **(b)** After the input statement executes.

FIGURE 2.8 shows the memory model for the local variables in the program of Figure 2.6. The computer allocates storage for all local variables on the run-time stack. When `main()` executes, storage for the return value, the return address, and local variables `exam1`, `exam2`, and `score` are pushed onto the stack. Because `bonus` is not a variable, it is not pushed onto the stack.

2.2 Flow of Control

A program operates by executing its statements sequentially—that is, one statement after the other. You can alter the sequence by changing the flow of control in two ways: selection and repetition. C has the `if` and `switch` statements for selection, and the `while`, `do`, and `for` statements for repetition. Each of these statements performs a test to possibly alter the sequential flow of control. The most common tests use one of the six relational operators shown in FIGURE 2.9.

FIGURE 2.9
The relational operators.

Operator	Meaning
==	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
!=	Not equal to

The If/Else Statement

FIGURE 2.10 shows a simple use of the C `if` statement to perform a test with the greater-than-or-equal-to relational operator, `>=`. The program inputs a value for the integer variable `num` and compares it with the constant integer `limit`. If the value of `num` is greater than or equal to the value of `limit`, which is 100, the word `high` is output. Otherwise, the word `low` is output. It is legal to write an `if` statement without an `else` part.

You can combine several relational tests with the Boolean operators shown in FIGURE 2.11. The double ampersand `(&&)` is the symbol for the AND operation, the double vertical bar `(||)` is for the OR operation, and the exclamation point `(!)` is for the NOT operation.

Example 2.3 If `age`, `income`, and `tax` are integer variables, the `if` statement

```
if ((age < 21) && (income <= 4000)) {
    tax = 0;
}
```

sets the value of `tax` to 0 if `age` is less than 21 and `income` is less than $4000. ■

FIGURE 2.10
The C `if` statement.

```
#include <stdio.h>

int main() {
   const int limit = 100;
   int num;
   scanf("%d", &num);
   if (num >= limit) {
      printf("high\n");
   }
   else {
      printf("low\n");
   }
   return 0;
}
```

Input
```
75
```

Output
```
low
```

The `if` statement in Figure 2.10 has a single statement in each alternative. If you want more than one statement to execute in an alternative, you must enclose the statements in braces, { }. Otherwise, the braces are optional.

Example 2.4 The `if` statement in Figure 2.10 can be written

```
if (num >= limit)
   printf("high\n");
else
   printf("low\n");
```

without the braces around the output statements. ∎

FIGURE 2.11
The Boolean operators.

Symbol	Meaning
&&	AND
||	OR
!	NOT

The Switch Statement

The program in FIGURE 2.12 uses the C `switch` statement to play a little guessing game with the user. It asks the user to pick a number. Then, depending on the number input, it outputs an appropriate message.

You can achieve the same effect yielded by the `switch` statement using the `if` statement. However, the equivalent `if` statement is not quite as efficient as `switch`.

FIGURE 2.12
The C `switch` statement.

```
#include <stdio.h>

int main() {
   int guess;
   printf("Pick a number 0..3: ");
   scanf("%d", &guess);
   switch (guess) {
      case 0: printf("Not close\n"); break;
      case 1: printf("Close\n"); break;
      case 2: printf("Right on\n"); break;
      case 3: printf("Too high\n");
   }
   return 0;
}
```

Interactive Input/Output

```
Pick a number 0..3: 1
Close
```

Example 2.5 The `switch` statement in Figure 2.12 can be written using the logically equivalent nested `if` statement:

```
if (guess == 0) {
   printf("Not close\n");
}
else if (guess == 1) {
   printf("Close\n");
}
else if (guess == 2) {
   printf("Right on\n");
}
else if (guess == 3) {
   printf("Too high\n");
}
```

However, this code is not as efficient as the `switch`. With this code, if the user guesses 3, all four tests will execute. With the `switch` statement, if the user guesses 3, the program jumps immediately to the `"Too high"` statement without having to compare `guess` with 0, 1, and 2. ∎

The While Loop

The program in FIGURE 2.13 takes as input a sequence of characters terminated with the asterisk, `*`. It outputs all the characters up to but not including the asterisk with each word on a separate line. An experienced C programmer would not use the asterisk as a sentinel character, so this example is unrealistic. Figure 2.13 and all the programs in this chapter are presented so that they can be analyzed at a lower level of abstraction in later chapters.

The program inputs the value of the first character into the global variable `letter` before entering the loop. The statement

```
while (letter != '*')
```

compares the value of `letter` with the asterisk character. If they are not equal, the body of the loop executes, which outputs either the character or a newline and then inputs the next character. Flow of control then returns to the test at the top of the loop.

This program would produce identical output if `letter` were local instead of global. Whether to declare a variable as local instead of global is a software design issue. The rule of thumb is to always declare variables to be local unless there is a good reason to do otherwise. Local variables enhance the modularity of software systems and make long programs easier to read and debug. The global variables in Figures 2.4 and 2.13 do not represent

```
#include <stdio.h>

char letter;

int main() {
   scanf("%c", &letter);
   while (letter != '*') {
      if (letter == ' ') {
         printf("\n");
      }
      else {
         printf("%c", letter);
      }
      scanf("%c", &letter);
   }
   return 0;
}
```

Input

```
Hello, world!*
```

Output

```
Hello,
world!
```

FIGURE 2.13
The C `while` loop.

good software design. They are presented because they illustrate the C memory model. Later chapters show how a C compiler would translate the programs presented in this chapter.

The Do Loop

The program in FIGURE 2.14 illustrates the `do` statement. It is unusual because it has no input. The program produces the same output each time it executes. This is another nonsense program whose purpose is to illustrate flow of control.

A police officer is initially at a position of 0 units when he begins to pursue a driver who is initially at a position of 40 units. Each execution of the loop represents one time interval, during which the officer travels 25 units and the driver 20. The statement

```
cop += 25;
```

is C shorthand for

```
cop = cop + 25;
```

FIGURE 2.14
The C `do` loop.

```
#include <stdio.h>

int cop;
int driver;

int main() {
   cop = 0;
   driver = 40;
   do {
      cop += 25;
      driver += 20;
   }
   while (cop < driver);
   printf("%d", cop);
   return 0;
}
```

Output

```
200
```

Unlike in the loop in Figure 2.13, the `do` statement has its test at the bottom of the loop. Consequently, the body of the loop is guaranteed to execute at least one time. When the statement

```
while (cop < driver);
```

executes, it compares the value of `cop` with the value of `driver`. If `cop` is less than `driver`, flow of control transfers to `do`, and the body of the loop repeats.

Arrays and the For Loop

The program in FIGURE 2.15 illustrates the `for` loop and the array. It allocates a local array of four integers, inputs values into the array, and then outputs the values in reverse order.

The statement

```
int vector[4];
```

declares variable `vector` to be an array of four integers. In C, all arrays have their first index at 0. Hence, this declaration allocates storage for array elements

```
vector[0] vector[1] vector[2] vector[3]
```

FIGURE 2.15
The C `for` loop with an array.

```
#include <stdio.h>

int vector[4];
int j;

int main() {
   for (j = 0; j < 4; j++) {
      scanf("%d", &vector[j]);
   }
   for (j = 3; j >= 0; j--) {
      printf("%d %d\n", j, vector[j]);
   }
   return 0;
}
```

Input

```
2 26 -3 9
```

Output

```
3 9
2 -3
1 26
0 2
```

The number in the declaration that specifies how many elements will be allocated is always one more than the index of the last element. In this program, 4, which is the number of elements, is one more than 3, which is the index of the last element.

Every `for` statement has a pair of parentheses whose interior is divided into three compartments, each compartment separated from its neighbor by a semicolon. The first compartment initializes, the second compartment tests, and the third compartment increments. In this program, the `for` statement

```
for (j = 0; j < 4; j++)
```

has `j = 0` for the initialization, `j < 4` for the test, and `j++` for the increment.

When the program enters the loop, `j` is set to 0. Because the test is at the top of the loop, the value of `j` is compared to 4. Because `j` is less than 4, the body of the loop

```
scanf("%d", &vector[j]);
```

executes. The first integer value from the input stream is read into `vector[0]`. Control returns to the `for` statement, which increments `j` because of the expression `j++` in the third compartment. The value of `j` is then compared to 4, and the process repeats.

The values are printed in reverse order by the second loop because of the decrement expression

```
j--
```

which is C shorthand for

```
j = j - 1
```

The programming style with `for` loops in Figure 2.15 is not the preferred C style. The control variable `j` would rarely be declared as a global variable. Instead, it would be contained in the scope of the `for` statement, the first of which would be written as

```
for (int j = 0; j < 4; j++)
```

This text eschews this preferred coding style to more effectively teach the concepts of global versus local allocation on the run-time stack. A description of the allocation process for the preferred style would add an extra level of complication in the exposition.

2.3 Functions

Procedures are functions that return void.

In C, there are two kinds of functions: those that return void and those that return some other type. Function `main()` returns an integer, not void. The operating system uses the integer to determine if the program terminated normally. Functions that return void perform their processing without returning a value at all. Functions that return void are also called *procedures*. One common use of void functions is to input or output a collection of values.

Void Functions and Call-by-Value Parameters

The program in FIGURE 2.16 uses a void function to print a bar chart of data values. The program reads the first value into the integer variable `numPts`. The global variable `j` controls the `for` loop in the main program, which executes `numPts` times. Each time the loop executes, it calls the void function `printBar()`. FIGURE 2.17 shows a trace of the beginning of execution of the program in Figure 2.16.

FIGURE 2.16
A program that prints a bar chart. The void function prints a single bar.

```
#include <stdio.h>

int numPts;
int value;
int j;

void printBar(int n) {
   int k;
   for (k = 1; k <= n; k++) {
      printf("*");
   }
   printf("\n");
}

int main() {
   scanf("%d", &numPts);
   for (j = 1; j <= numPts; j++) {
      scanf("%d", &value);
      printBar(value);
      //ra1
   }
   return 0;
}
```

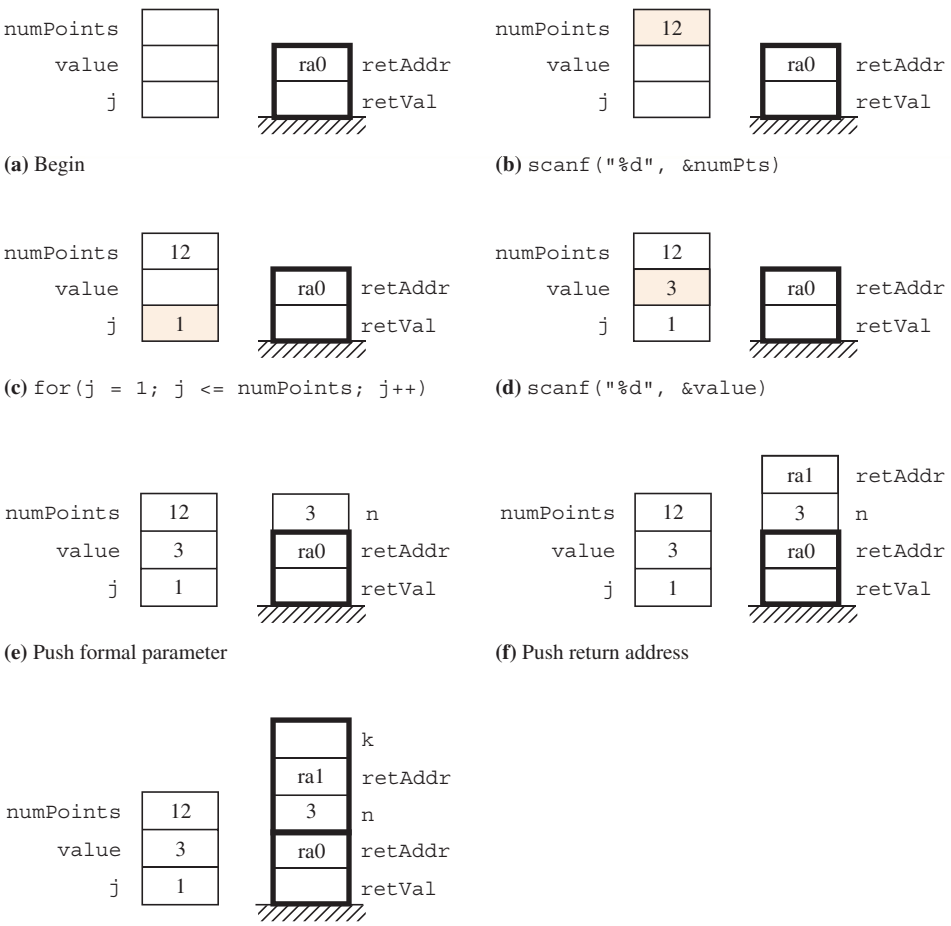
Input

```
12 3 13 17 34 27 23 25 29 16 10 0 2
```

Output

```
***
*************
*****************
**********************************
**************************
**********************
************************
****************************
***************
**********

**
```

FIGURE 2.17
The run-time stack for the program in Figure 2.16.



(a) Begin

(b) `scanf("%d", &numPts)`

(c) `for(j = 1; j <= numPoints; j++)`

(d) `scanf("%d", &value)`

(e) Push formal parameter

(f) Push return address

(g) Push storage for local variable `k`

The allocation process for a void function

Allocation takes place on the run-time stack in the following order when you call a void function:

- Push the actual parameters.
- Push the return address.
- Push storage for the local variables.

It is the same allocation as with a non-void function but without the initial push of storage for the return value. A *formal parameter* is the parameter in the function declaration. In Figure 2.16, `n` is the formal parameter. An *actual parameter* is the parameter in the function call. In Figure 2.16, `value` is the actual parameter.

Formal and actual parameters

Figure 2.17(e) is the start of the allocation process for Figure 2.16. The program pushes the value of actual parameter `value` for the formal parameter `n`. The effect is that formal parameter `n` gets the value of the actual parameter `value`. The program pushes the return address in Figure 2.17(f). In Figure 2.17(g), it pushes storage for the local variable, `k`. After the allocation process, the last local variable in the listing, `k`, is on top of the stack.

The collection of all the items pushed onto the run-time stack in a function call is known as a *stack frame* or *activation record*. In the program of Figure 2.16, the stack frame for the void function consists of three items—`n`, the return address, and `k`. The return address indicated by ra1 in the figure is the address of the end of the `for` statement of the main program. The stack frame for the `main()` function consists of two items—the return value and the return address.

Stack frames

After the procedure prints a single bar, control returns to the main program. The items on the run-time stack are deallocated in reverse order compared to their allocation. The process is:

The deallocation process for a void function

- Pop the local variables.
- Pop the return address and use it to determine the next instruction to execute.
- Pop the parameters.

The program uses the return address to know which statement to execute next in the main program after executing the last statement in the void function. Return address ra1 in the listing of the main program is the statement after the procedure call. It represents the point where `j` is incremented before the branch up to the test at the top of the loop.

Functions

The program in FIGURE 2.18 uses a function to compute the value of the factorial of an integer. It prompts the user for a small integer and passes that integer as a parameter to function `fact()`.

FIGURE 2.19 shows the allocation process for the function in Figure 2.18, which returns the factorial of the actual parameter. Figure 2.19(c) shows storage for the return value pushed first. Figure 2.19(d) shows the value of actual parameter `num`, which is 3, pushed for the formal parameter `n`.

FIGURE 2.18
A program to compute the factorial of an integer with a function.

```
#include <stdio.h>

int num;

int fact(int n) {
   int f, j;
   f = 1;
   for (j = 1; j <= n; j++) {
      f *= j;
   }
   return f;
}

int main() {
   printf("Enter a small integer: ");
   scanf("%d", &num);
   printf("Its factorial is: %d\n", fact(num)); // ra1
   return 0;
}
```

Interactive Input/Output

```
Enter a small integer: 3
Its factorial is: 6
```

The return address is pushed in Figure 2.19(e). Storage for local variables `f` and `j` is pushed in Figure 2.19(f) and (g).

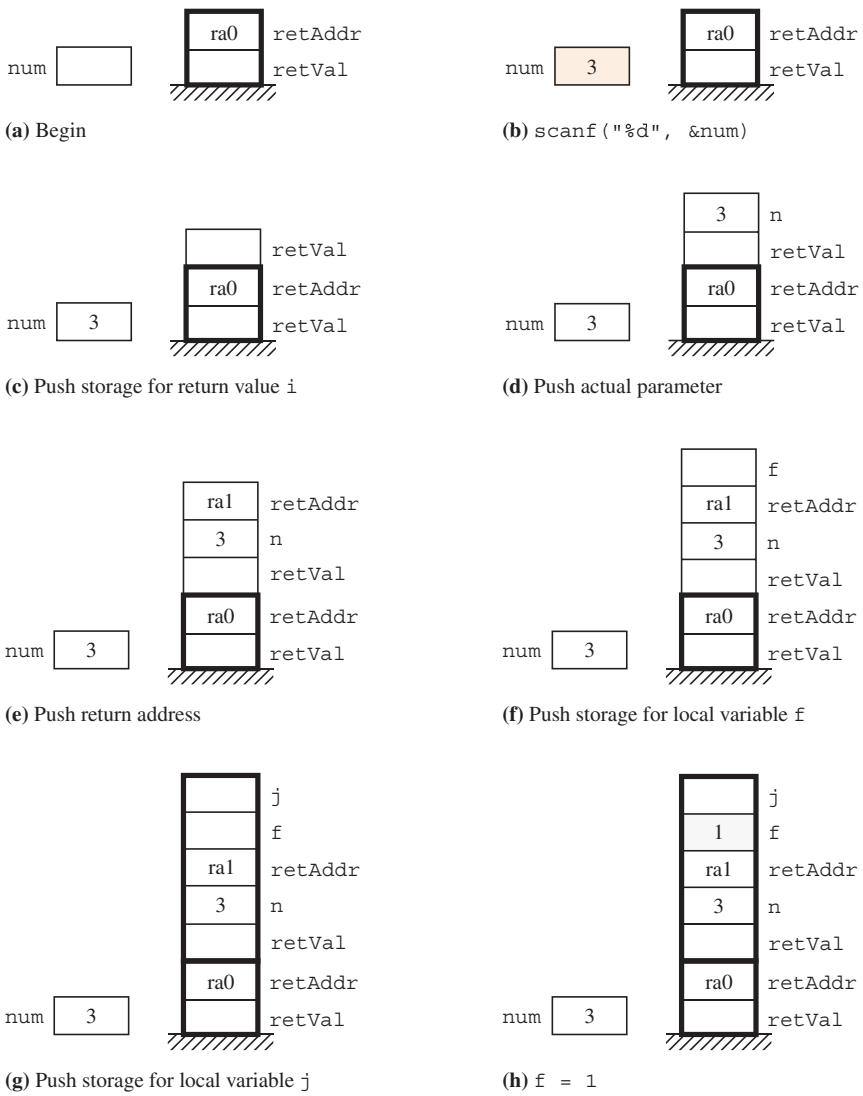
Returning control from void and nonvoid functions

The stack frame for this function has five items. The return address indicated by ra1 in the figure represents the address of the `printf()` function call in the main program. Control returns from the function to the calling statement. This is in contrast to a void function, in which control returns to the statement *following* the calling statement.

Call-by-Reference Parameters

The procedures and functions in the previous programs all pass their parameters by value. In call by value, the formal parameter gets the value of the actual parameter. If the called procedure changes the value of its formal parameter, the corresponding actual parameter in the calling program does not change. Any changes made by the called procedure are made to the value

FIGURE 2.19
The run-time stack for the program in Figure 2.18.



(a) Begin

(b) `scanf("%d", &num)`

(c) Push storage for return value `i`

(d) Push actual parameter

(e) Push return address

(f) Push storage for local variable `f`

(g) Push storage for local variable `j`

(h) `f = 1`

on the run-time stack. When the stack frame is deallocated, any changed values are deallocated with it.

If the intent of the procedure is to change the value of the actual parameter in the calling program, then call by reference is used instead of call by value. In call by reference, the formal parameter gets a reference to the actual parameter. If the called procedure changes the value of its formal parameter, the

The & address operator

corresponding actual parameter in the calling program changes. To specify that a parameter is called by reference, you pass the address of the actual parameter with the `&` address operator. The corresponding formal parameter is a pointer and must be preceded by the asterisk, `*`, symbol. In C, a pointer is an address.

The program in FIGURE 2.20 uses call by reference to change the values of the actual parameters in `main()`. It prompts the user for two integer values and puts them in order. It has one void function, `order()`, that

FIGURE 2.20
A program to put two values in order. The void functions pass parameters by reference.

```
#include <stdio.h>

int a, b;

void swap(int *r, int *s) {
   int temp;
   temp = *r;
   *r = *s;
   *s = temp;
}

void order(int *x, int *y) {
   if (*x > *y) {
      swap (x, y);
   }  // ra2
}

int main() {
   printf("Enter an integer: ");
   scanf("%d", &a);
   printf("Enter an integer: ");
   scanf("%d", &b);
   order(&a, &b);
   printf("Ordered they are: %d, %d\n", a, b); // ra1
   return 0;
}
```

Interactive Input/Output

```
Enter an integer: 6
Enter an integer: 2
Ordered they are: 2, 6
```

calls another void function, `swap()`. In `main()`, the call to `order()` has `&a` as the actual parameter and `*x` as the corresponding formal parameter. `x` is a pointer and, hence, is an address. Namely, `x` is the address of actual parameter `a`. When `order()` calls `swap()`, the actual parameter must again be an address. Because `x` is already an address, the `&` address operator is not prefixed to it in the actual parameter list in the call from `order()`.

Procedure `order()` shows how to access the value of the cell pointed to. The test for the `if` statement is

```
if (*x > *y)
```

Because `x` is a pointer, `*x` is the value in the memory cell to which `x` points. Variable `x` is a pointer to an `int`. The expression `*x` is an `int`. Similarly, `*y` is the value in the memory cell to which `y` points. The test

```
if (x > y)
```

would be an error because you would be testing whether the address of `a` is greater than the address of `b` instead of whether `a` is greater than `b`.

Procedure `swap()` also shows how the `*` operator is used to dereference a pointer. In the parameter list, the formal parameter

```
int *r
```

indicates that `r` is a pointer to an integer. That is, `r` is the address of an integer. In the assignment statement

```
temp = *r;
```

*The * pointer dereference operator*

the asterisk `*` dereferences `r`. Because `r` is a pointer to an integer, `*r` is the integer to which it points. The assignment statement gives the integer value `*r` to the integer variable temp.

FIGURE 2.21 shows the allocation and deallocation sequence for the entire program. The stack frame for `order()` in part (c) has three items. The formal parameters, `x` and `y`, are called by reference. The arrow pointing from `x` on the run-time stack to `a` in the main program indicates that `x` refers to `a`. Literally, `x` is the address of `a`. Similarly, the arrow from `y` to `b` indicates that `y` refers to `b`. The return address, indicated by ra1, is the address of the `printf()` statement that follows the call to `order()` in the main program.

The stack frame for `swap()` in Figure 2.21(d) has four items. `r` refers to `x`, which refers to `a`. Therefore, `r` refers to `a`. The arrow pointing from `r` on the run-time stack points to `a`, as does the arrow from `x`. Similarly, the arrow from `s` points to `b`, as does the arrow from `y`. The return address indicated by ra2 is the address after the last statement in `order()`. The statements in `swap()` exchange the values of `r` and `s`. Because `r` refers to `a` and `s` refers to `b`, they exchange the values of `a` and `b` in the main program.

FIGURE 2.21
The run-time stack for Figure 2.20.

When a void function terminates and it is time to deallocate its stack frame, the return address in the frame tells the computer which instruction to execute next. Figure 2.21(e) shows the return from void function `swap`, deallocating its stack frame. The return address in the stack frame for `swap` tells the computer to execute the statement labeled ra2 in `order()` after deallocation. Although the listing shows no statement at ra2 in Figure 2.20, there is an implied `return` statement at the end of the void function that is invisible at Level HOL6.

In Figure 2.21(f), the stack frame for `order()` is deallocated. The return address in the stack frame for `order()` tells the computer to execute the `printf()` function in the main program after deallocation.

Because a stack is a LIFO structure, the last stack frame pushed onto the run-time stack will be the first one popped off at the completion of a function. The return address will, therefore, return control to the most recent calling function. This LIFO property of the run-time stack will be basic to your understanding of recursion in Section 2.4.

A simplification for `main()` in this text

You may have noticed that `main()` is always a function that returns an integer and that all the programs thus far have returned 0 to the operating system. Furthermore, all the main program functions thus far have no parameters. Although it is common for a main program to have parameters, none of the programs in this text do. To keep the figures simple, from now on they will omit the `retVal` and `retAddr` for the main program. A real C compiler must account for both of them.

2.4 Recursion

Did you ever look up the definition of some unknown word in the dictionary, only to discover that the dictionary defined it in terms of another unknown word? Then, when you looked up the second word, did you discover that it was defined in terms of the first word? That is an example of circular or indirect recursion. The problem with the dictionary is that you did not know the meaning of the first word to begin with. Had the second word been defined in terms of a third word that you knew, you would have been satisfied.

Recursive definitions in mathematics

In mathematics, a *recursive definition* of a function is a definition that uses the function itself. For example, suppose a function, $f(n)$, is defined as follows:

$$f(n) = nf(n-1)$$

You want to use this definition to determine $f(4)$, so you substitute 4 for n in the definition:

$$f(4) = 4 \times f(3)$$

But now you do not know what $f(3)$ is. So you substitute 3 for n in the definition and get

$$f(3) = 3 \times f(2)$$

Substituting this into the formula for $f(4)$ gives

$$f(4) = 4 \times 3 \times f(2)$$

But now you do not know what $f(2)$ is. The definition tells you it is 2 times $f(1)$. So the formula for $f(4)$ becomes

$$f(4) = 4 \times 3 \times 2 \times f(1)$$

You can see the problem with this definition. With nothing to stop the process, you will continue to compute $f(4)$ endlessly.

$$f(4) = 4 \times 3 \times 2 \times 1 \times 0 \times (-1) \times (-2) \times (-3) \cdots$$

It is as if the dictionary gave you an endless string of definitions, each based on another unknown word. To be complete, the definition must specify the value of $f(n)$ for a specific value of n. Then the preceding process will terminate, and you can compute $f(n)$ for any n.

Here is a complete recursive definition of $f(n)$:

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ nf(n-1) & \text{if } n > 1. \end{cases}$$

This definition says you can stop the previous process at $f(1)$, which is called the *basis*. So $f(4)$ is

$$\begin{aligned} f(4) &= 4 \times f(4) \\ &= 4 \times 3 \times f(2) \\ &= 4 \times 3 \times 2 \times f(1) \\ &= 4 \times 3 \times 2 \times 1 \\ &= 24 \end{aligned}$$

You should recognize this definition as the factorial function.

A Factorial Function

Recursive functions in C

A *recursive function* in C is a function that calls itself. There is no special recursion statement with a new syntax to learn. The method of storage allocation on the run-time stack is the same as with nonrecursive functions. The only difference is that a recursive function contains a statement that calls itself.

The function in FIGURE 2.22 computes the factorial of a number recursively. It is a direct application of the preceding recursive definition of $f(n)$.

FIGURE 2.23 is a trace that shows the run-time stack with the simplification of not showing the stack frame of the main program. The first function call is from the main program. Figure 2.23(c) shows the stack frame for the first call, assuming the user entered 4. The return address is ra1, which represents the address of the `printf()` function in the main program.

The first statement in the function tests `n` for 1. Because the value of `n` is 4, the `else` part executes. But the statement in the `else` part

```
return n * fact(n - 1) // ra2
```

contains a call to function `fact()` on the right side of the return statement.

FIGURE 2.22
A program to compute the factorial recursively.

```
#include <stdio.h>

int num;

int fact(int n) {
   if (n <= 1) {
      return 1;
   }
   else {
      return n * fact(n - 1); // ra2
   }
}

int main() {
   printf("Enter a small integer: ");
   scanf("%d", &num);
   printf("Its factorial is: %d\n", fact(num)); // ra1
   return 0;
}
```

Interactive Input/Output

```
Enter a small integer: 4
Its factorial is: 24
```

This is a recursive call because it is a call to the function within the function itself. The same sequence of events happens as with any function call. A new stack frame is allocated, as Figure 2.23(d) shows. The return address in the second stack frame is the address of the calling statement in the function, represented by ra2.

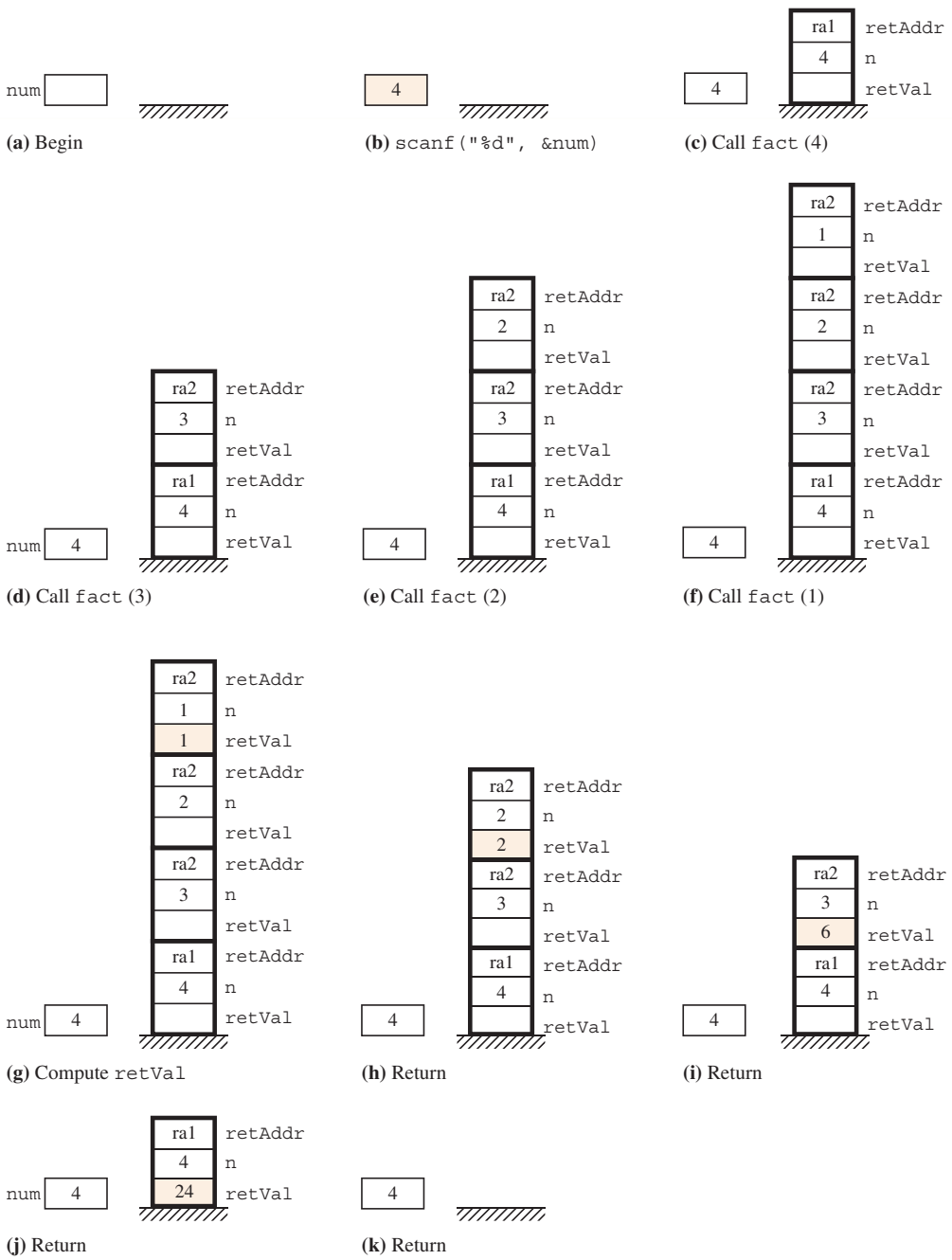
The actual parameter is `n - 1`, whose value is 3 because the value of `n` in Figure 2.23(c) is 4. The formal parameter, `n`, is called by value. Therefore, the formal parameter `n` in the top frame of Figure 2.23(d) gets the value 3.

Multiple instances of local variables and parameters

Figure 2.23(d) shows a curious situation that is typical of recursive calls. The program listing of Figure 2.22 shows only one declaration of `n` in the formal parameter list of `fact`. But Figure 2.23(d) shows two instances of `n`. The old instance of `n` has the value 4 from the main program. But the new instance of `n` has the value 3 from the recursive call.

The computer suspends the old execution of the function and begins a new execution of the same function from its beginning. The first statement

FIGURE 2.23
The run-time stack for Figure 2.22.



in the function tests n for 1. But which n? Figure 2.23(d) shows two ns on the run-time stack. The rule is that any reference to a local variable or formal parameter is to the one on the top stack frame. Because the value of n is 3, the else part executes.

But now the function makes another recursive call. It allocates a third stack frame, as Figure 2.23(e) shows, and then a fourth, as Figure 2.23(f) shows. Each time, the newly allocated formal parameter gets a value one less than the old value of n because the function call is

```
fact(n - 1)
```

Finally, in Figure 2.23(g), n has the value 1. The function gives 1 to the cell on the stack labeled retVal. It skips the else part and terminates. That triggers a return to the calling statement.

The same events transpire with a recursive return as with a nonrecursive return. retVal contains the return value, and the return address tells which statement to execute next. In Figure 2.22(g), retVal is 1 and the return address is the calling statement in the function. The top frame is deallocated, and the calling statement

```
return n * fact(n - 1) // ra2
```

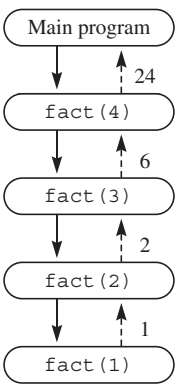
completes its execution. It multiplies its value of n, which is 2, by the value returned, which is 1, and assigns the result to retVal. So, retVal gets 2, as Figure 2.23(h) shows.

A similar sequence of events occurs on each return. Figure 2.23(i) and (j) show that the value returned from the second call is 6 and from the first call is 24. FIGURE 2.24 shows the calling sequence for Figure 2.22. The main program calls fact. Then fact calls itself three times. In this example, fact is called a total of four times.

You see that the program computes the factorial of 4 the same way you would compute $f(4)$ from its recursive definition. You start by computing $f(4)$ as 4 times $f(3)$. Then you must suspend your computation of $f(4)$ to compute $f(3)$. After you get your result for $f(3)$, you can multiply it by 4 to get $f(4)$.

Similarly, the program must suspend its execution of the function to call the same function again. The run-time stack keeps track of the current values of the variables so they can be used when that instance of the function resumes.

FIGURE 2.24
The calling sequence for Figure 2.22. The solid arrows represent function calls. The dotted arrows represent returns. The value returned is next to each return arrow.



Thinking Recursively

The microscopic and macroscopic viewpoints of recursion

You can take two different viewpoints when dealing with recursion: microscopic and macroscopic. Figure 2.23 illustrates the microscopic viewpoint and shows precisely what happens inside the computer during execution. It is the viewpoint that considers the details of the run-time stack

during a trace of the program. The macroscopic viewpoint does not consider the individual trees. It considers the forest as a whole.

You need to know the microscopic viewpoint to understand how C implements recursion. The details of the run-time stack will be necessary when you study how recursion is implemented at Level Asmb5. But to write a recursive function, you should think macroscopically, not microscopically.

The most difficult aspect of writing a recursive function is the assumption that you can call the procedure that you are in the process of writing. To make that assumption, you must think macroscopically and forget about the run-time stack.

Proof by mathematical induction can help you think macroscopically. The two key elements of proof by induction are

- Establish the basis.
- Given the formula for n, prove it for $n + 1$.

The relation between proof by mathematical induction and recursion

Similarly, the two key elements of designing a recursive function are

- Compute the function for the basis.
- Assuming the function for $n - 1$, write it for n.

Imagine you are writing function `fact()`. You get to this point:

```
int fact(int n) {
   if (n <= 1) {
      return 1;
   }
   else {
```

and wonder how to continue. You have computed the function for the basis, `n` = 1. But now you must assume that you can call function `fact()`, even though you have not finished writing `fact()`. You must assume that `fact(n - 1)` will return the correct value for the factorial.

The importance of thinking macroscopically when you design a recursive function

Here is where you must think macroscopically. If you start wondering how `fact(n - 1)` will return the correct value, and if visions of stack frames begin dancing in your head, you are not thinking correctly. In proof by induction, you must assume the formula for `n`. Similarly, in writing `fact()`, you must assume that you can call `fact(n - 1)` with no questions asked.

The divide and conquer strategy

Recursive programs are based on a divide-and-conquer strategy, which is appropriate when you can solve a large problem in terms of a smaller one. Each recursive call makes the problem smaller and smaller, until the program reaches the smallest problem of all, the basis, which is simple to solve.

Recursive Addition

Here is another example of a recursive problem. Suppose `list` is an array of integers. You want to find the sum of all integers in the list recursively.

The first step is to formulate the solution of the large problem in terms of a smaller problem. If you knew how to find the sum of the integers between `list[0]` and `list[n - 1]`, you could simply add it to `list[n]`. You would then have the sum of all the integers.

The next step is to design a function with the appropriate parameters. The function will compute the sum of `n` integers by calling itself to compute the sum of `n - 1` integers. So the parameter list must have a parameter that tells how many integers in the array to add. That should lead you to the following function head:

```
int sum(int a[], int n) {
// Returns the sum of the elements of a between a[0]
and a[n].
```

How do you establish the basis? That is simple. If `n` is `0`, the function should add the sum of the elements between `a[0]` and `a[0]`. The sum of one element is just `a[0]`.

Now you can write

```
if (n == 0) {
   return a[0];
}
else {
```

Now think macroscopically. You can assume that `sum(a, n - 1)` will return the sum of the integers between `a[0]` and `a[n - 1]`. Have faith. All you need to do is add that sum to `a[n]`. FIGURE 2.25 shows the function in a finished program.

Even though you write the function without considering the microscopic view, you can still trace the run-time stack. FIGURE 2.26 shows the stack frames for the first two calls to `sum`. The stack frame consists of the value returned, the parameters `a` and `n`, and the return address. Because there are no local variables, no storage for them is allocated on the run-time stack.

Arrays always called by reference

In C, arrays are always called by reference without the `&` address operator in the actual parameter list. In Figure 2.25, the actual parameter `list` in the call

```
sum(list, 3)
```

is not prefixed with the `&` address operator, and yet it is called by reference. Hence, variable `a` in procedure `sum` refers to `list` in the main program. It literally contains the address of the first cell of the array. In this program, `a` contains the address of `list[0]`. The arrows in Figure 2.26(b) and (c) that

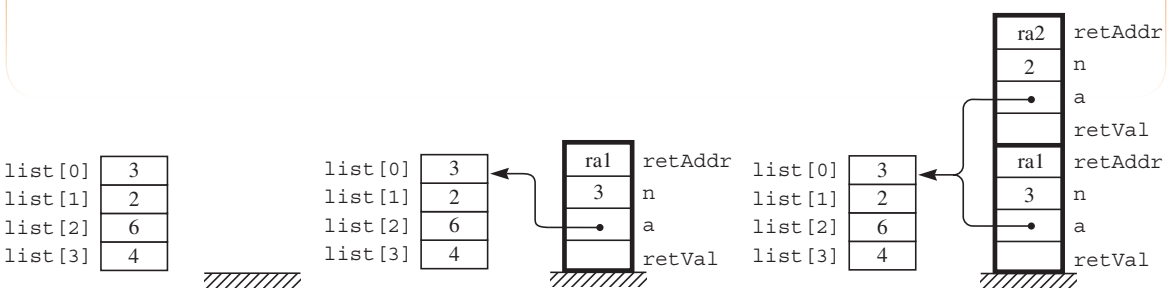
FIGURE 2.25
A recursive function that returns the sum of the first `n` numbers in an array.

```
#include <stdio.h>

int list[4];

int sum(int a[], int n) {
// Returns the sum of the elements of a between a[0] and a[n].
   if (n == 0) {
      return a[0];
   }
   else {
      return a[n] + sum(a, n - 1); // ra2
   }
}

int main() {
   printf("Enter four integers: ");
   scanf("%d %d %d %d", &list[0], &list[1], &list[2], &list[3]);
   printf("Their sum is: %d\n", sum(list, 3));
   return 0;
}
```

Interactive Input/Output

```
Enter four integers: 3 2 6 4
Their sum is: 15
```

FIGURE 2.26
The run-time stack for the program in Figure 2.25.



(a) *Input* `list` **(b)** Call sum (list, 3) **(c)** Call sum (list, 2)

point from the cells labeled `a` in the stack frame to the cell labeled `list[0]` indicate the reference of `a` to `list`.

In contrast to an array name without an index, an array name with an index is an individual element of an array and is treated like an individual variable. In Figure 2.25, the actual parameter `list[1]` in the call to `scanf()` is prefixed with the `&` address operator so it can be called by reference. To summarize, `list[1]` has type integer and requires the address operator to be called by reference. `list` has type array and is called by reference by default without the address operator.

A Binomial Coefficient Function

The next example of a recursive function has a more complex calling sequence. It is a function to compute the coefficient in the expansion of a binomial expression.

Consider the following expansions:

$(x + y)^1 = x + y$
$(x + y)^2 = x^2 + 2xy + y^2$
$(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$
$(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$

The coefficients of the terms are called *binomial coefficients*. If you write the coefficients without the terms, they form a triangle of values called *Pascal's triangle*. FIGURE 2.27 is Pascal's triangle for the coefficients up to the seventh power.

You can see from Figure 2.27 that each coefficient is the sum of the coefficient immediately above and the coefficient above and to the left. For example, the binomial coefficient in row 5, column 2, which is 10, equals 4 plus 6. Six is above 10, and 4 is above and to the left.

FIGURE 2.27
Pascal's triangle of binomial coefficients.

Power, n \ Term number, k	0	1	2	3	4	5	6	7
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Mathematically, the binomial coefficient $b(n, k)$ for power n and term k is

$$b(n, k) = b(n - 1, k) + b(n - 1, k - 1)$$

That is a recursive definition because it defines the function $b(n, k)$ in terms of itself. You can also see that if k equals 0, or if n equals k, the value of the binomial coefficient is 1. The complete mathematical definition, including the two base cases, is

$$b(n, k) = \begin{cases} 1 & \text{if } k = 0, \\ 1 & \text{if } n = k, \\ b(n - 1, k) + b(n - 1, k - 1) & \text{if } 0 < k < n. \end{cases}$$

FIGURE 2.28 computes the value of a binomial coefficient recursively. It is based directly on the recursive definition of $b(n, k)$. FIGURE 2.29 shows a trace of the run-time stack. Parts (b), (c), and (d) show the allocation of the first three stack frames. They represent calls to `binCoeff(3, 1)`,

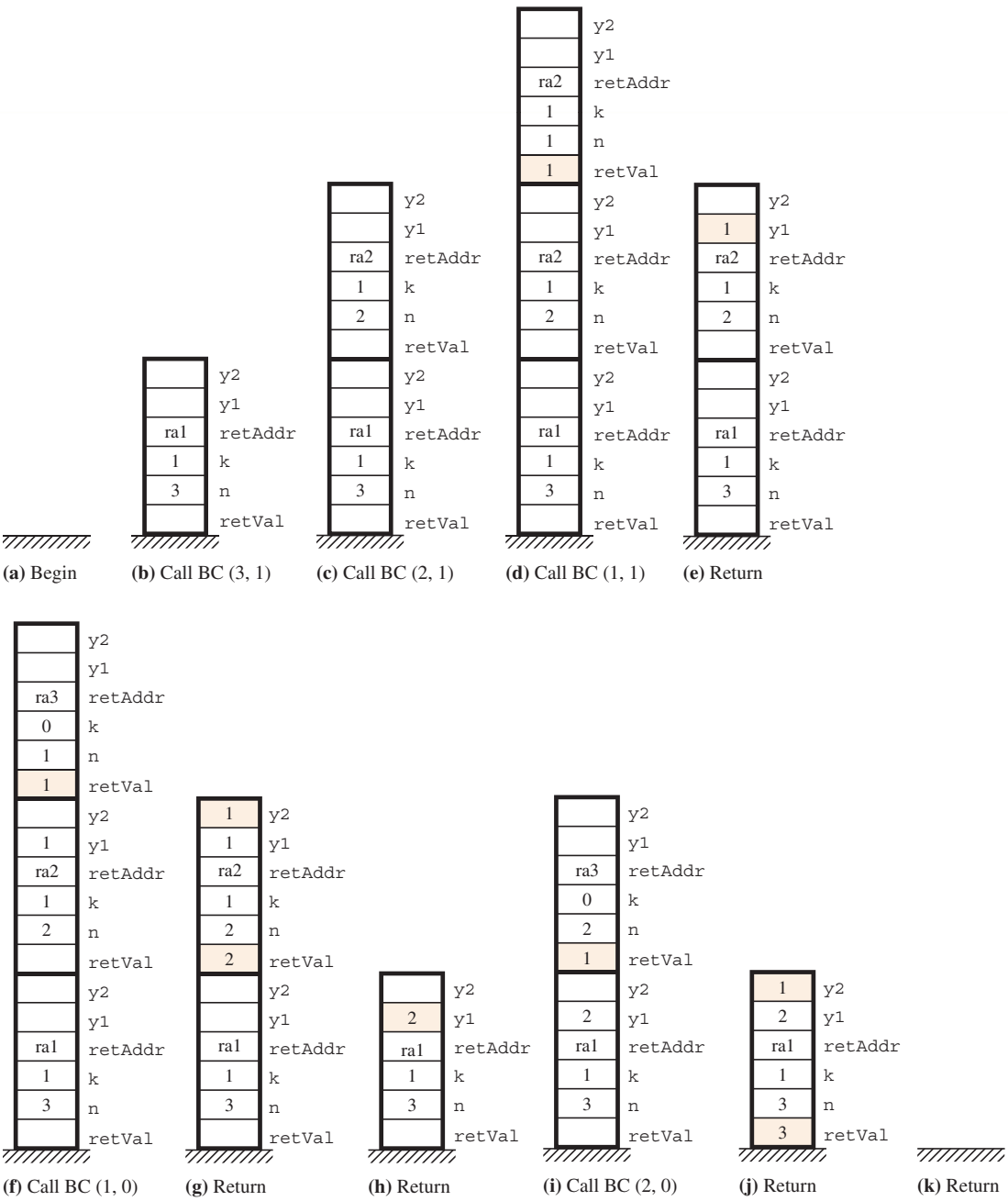
FIGURE 2.28
A recursive computation of the binomial coefficient.

```
#include <stdio.h>

int binCoeff(int n, int k) {
   int y1, y2;
   if ((k == 0) || (n == k)) {
      return 1;
   }
   else {
      y1 = binCoeff(n - 1, k); // ra2
      y2 = binCoeff(n - 1, k - 1); // ra3
      return y1 + y2;
   }
}

int main() {
   printf("binCoeff(3, 1) = %d\n", binCoeff(3, 1)); // ra1
   return 0;
}
```

Output

```
binCoeff(3, 1) = 3
```

FIGURE 2.29
The run-time stack for Figure 2.28.



`binCoeff(2,1)`, and `binCoeff(1,1)`. The first stack frame has the return address of the calling program in the main program. The next two stack frames have the return address of the `y1` assignment statement. ra2 represents that statement.

Figure 2.29(e) shows the return from `binCoeff(1,1)`. `y1` gets the value 1 returned by the function. Then the `y2` assignment statement calls the function `binCoeff(1,0)`. Figure 2.29(f) shows the run-time stack during execution of `binCoeff(1,0)`. Each stack frame has a different return address.

The calling sequence for this program is different from those of the previous recursive programs. The other programs keep allocating stack frames until the run-time stack reaches its maximum height. Then they keep deallocating stack frames until the run-time stack is empty. This program allocates stack frames until the run-time stack reaches its maximum height. It does not deallocate stack frames until the run-time stack is empty, however. From Figure 2.29(d) to (e), it deallocates, but from 2.29(e) to (f), it allocates. From 2.29(f) to (g) to (h), it deallocates, but from 2.29(h) to (i), it allocates. Why? Because this function has two recursive calls instead of one. If the basis step is true, the function makes no recursive call. But if the basis step is false, the function makes two recursive calls, one for `y1` and one for `y2`. FIGURE 2.30 shows the calling sequence for the program. Notice that it is in the shape of a tree. Each node of the tree represents a function call. Except for the main program, a node has either two children or no children, corresponding to two recursive calls or no recursive calls.

Referring to Figure 2.30, the sequence of calls and returns is

The sequence of calls and returns for the program in Figure 2.30

```
Main program
    Call BC(3, 1)
        Call BC(2, 1)
            Call BC(1, 1)
        Return to BC(2, 1)
            Call BC(1, 0)
        Return to BC(2, 1)
    Return to BC(3, 1)
        Call BC(2, 0)
    Return to BC(3, 1)
Return to main program
```

With this indentation style, each indent from one line to the next represents a function call, and each outdent from one line to the next represents a function return. You can visualize the order of execution on the call tree by imagining that the tree is a coastline in an ocean. A boat starts from the left side of the main program and sails along the coast, always

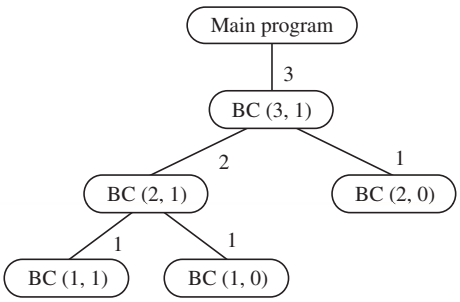


FIGURE 2.30
The call tree for the program in Figure 2.28.

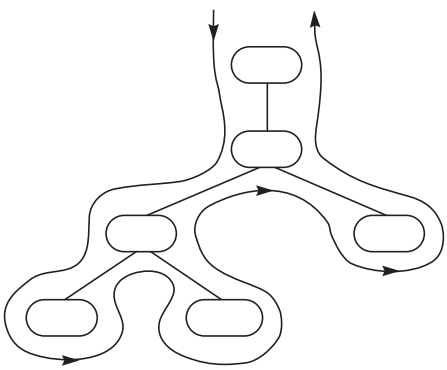


FIGURE 2.31
The order of execution of the program in Figure 2.28.

keeping the shore to its left. The boat visits the nodes in the same order from which they are called and returned. FIGURE 2.31 shows the visitation path.

When analyzing a recursive program from a microscopic point of view, it is easier to construct the call tree before you construct the trace of the run-time stack. Once you have the tree, it is easy to see the behavior of the run-time stack. Every time the boat visits a lower node in the tree, the program allocates one stack frame. Every time the boat visits a higher node in the tree, the program deallocates one stack frame.

You can determine the maximum height of the run-time stack from the call tree. Just keep track of the net number of stack frames allocated when you get to the lowest node of the call tree. That will correspond to the maximum height of the run-time stack.

Drawing the call tree in the order of execution is not the easiest way. The previous execution sequence started

```
Main program
    Call BC(3, 1)
        Call BC(2, 1)
            Call BC(1, 1)
```

You should not draw the call tree in that order. It is easier to start with

Main program
    Call BC(3, 1)
    Return to BC(3, 1)
    Return to BC(3, 1)
Return to main program

recognizing from the program listing that BC(3, 1) will call itself twice, BC(2, 1) once, and BC(2, 0) once. Then you can go back to BC(2, 1) and determine its children. In other words, determine all the children of a node before analyzing the deeper calls from any one of the children.

Constructing the call tree breadth first

This is a "breadth-first" construction of the tree, as opposed to the "depth-first" construction, which follows the execution sequence. The problem with the depth-first construction arises when you return up several levels in a complicated call tree to some higher node. You might forget the state of execution the node is in and not be able to determine its next child node. If you determine all the children of a node at once, you no longer need to remember the state of execution of the node.

Reversing the Elements of an Array

FIGURE 2.32 has a recursive procedure instead of a function. `reverse()` is a void function that reverses the elements in a local array of characters without returning a value. C allows the programmer to initialize an array of characters from a string constant enclosed in double quotes. In the main program, `word` is a local array of characters that is initialized from the string constant `"star"`. Because it is local, it is stored on the run-time stack in the stack frame for `main()`. The number of elements in the array is always one greater than the number of characters in the string constant because of an additional `\0` sentinel at the end of the string. In this program, `word` has five elements, four for the letters and one for the sentinel. In the `printf()` function calls, the placeholder `%s` causes the program to output the characters in array `word` from the first and up to, but not including, the sentinel.

The procedure reverses the characters in the array `str` between `str[j]` and `str[k]`. The main program wants to reverse the characters between `'s'` and `'r'`. So it calls `reverse()` with 0 for `j` and 3 for `k`.

The procedure solves this problem by breaking it down into a smaller problem. Because 0 is less than 3, the procedure knows that the characters between 0 and 3 need to be reversed. So it switches `str[0]` with `str[3]` and calls itself recursively to switch all the characters between `str[1]` and `str[2]`. If `j` is ever greater than or equal to `k`, no switching is necessary and the procedure does nothing. FIGURE 2.33 shows the beginning of a trace of the run-time stack.

FIGURE 2.32
A recursive procedure to reverse the elements of a local array.

```
#include <stdio.h>

void reverse(char *str, int j, int k) {
   char temp;
   if (j < k) {
      temp = str[j];
      str[j] = str[k];
      str[k] = temp;
      reverse(str, j + 1, k - 1);
   } // ra2
}

int main() {
   char word[5] = "star";
   printf("%s\n", word);
   reverse(word, 0, 3);
   printf("%s\n", word); // ra1
   return 0;
}
```

Output

```
star
rats
```

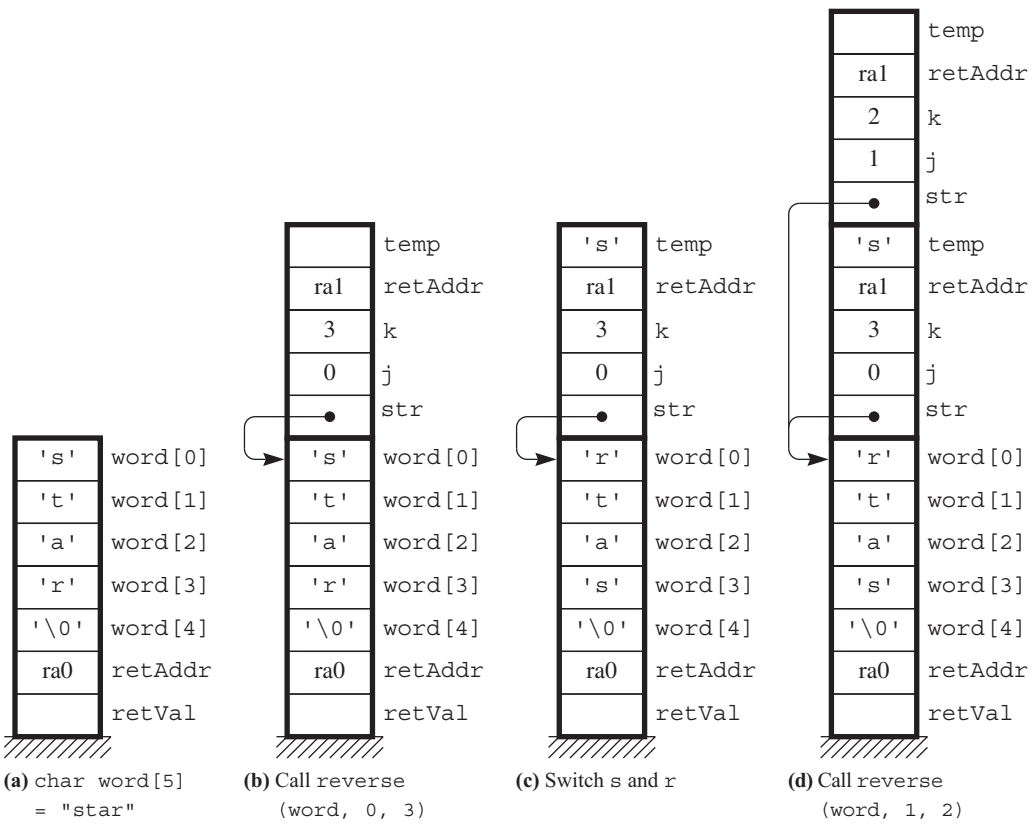
Towers of Hanoi

The Towers of Hanoi puzzle is a classic computer science problem that is conveniently solved by the recursive technique. The puzzle consists of three pegs and a set of disks with different diameters. The pegs are numbered 1, 2, and 3. Each disk has a hole at its center so that it can fit onto one of the pegs. The initial configuration of the puzzle consists of all the disks on one peg in a way that no disk rests directly on another disk with a smaller diameter. FIGURE 2.34 is the initial configuration for four disks.

The problem is to move all the disks from the starting peg to another peg under the following conditions:

- You may move only one disk at a time. It must be the top disk from one peg, which is moved to the top of another peg.
- You may not place one disk on another disk having a smaller diameter.

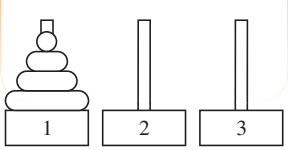
FIGURE 2.33
The run-time stack for the program in Figure 2.32.



The procedure for solving this problem has three parameters, `n`, `j`, and `k`, where

- `n` is the number of disks to move
- `j` is the starting peg
- `k` is the goal peg

FIGURE 2.34
The Towers of Hanoi puzzle.



`j` and `k` are integers that identify the pegs. Given the values of `j` and `k`, you can calculate the intermediate peg, which is the one that is neither the starting peg nor the goal peg, as `6 - j - k`. For example, if the starting peg is 1 and the goal peg is 3, then the intermediate peg is 6 – 1 – 3 = 2.

To move the `n` disks from peg `j` to peg `k`, first check whether `n = 1`. If it does, then simply move the one disk from peg `j` to peg `k`. But if it does not, then decompose the problem into several smaller parts:

- Move `n - 1` disks from peg `j` to the intermediate peg.
- Move one disk from peg `j` to peg `k`.
- Move `n - 1` disks from the intermediate peg to peg `k`.

FIGURE 2.35 shows this decomposition for the problem of moving four disks from peg 1 to peg 3.

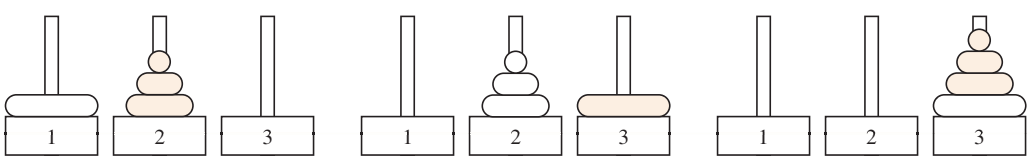
This procedure guarantees that a disk will not be placed on another disk with a smaller diameter, assuming that the original `n` disks are stacked correctly. Suppose, for example, that four disks are to be moved from peg 1 to peg 3, as in Figure 2.35. The procedure says that you should move the top three disks from peg 1 to peg 2, move the bottom disk from peg 1 to peg 3, and then move the three disks from peg 2 to peg 3.

In moving the top three disks from peg 1 to peg 2, you will leave the bottom disk on peg 1. Remember that it is the disk with the largest diameter, so any disk you place on it in the process of moving the other disks will be smaller. In order to move the bottom disk from peg 1 to peg 3, peg 3 must be empty. You will not place the bottom disk on a smaller disk in this step either. When you move the three disks from peg 2 to peg 3, you will place them on the largest disk, now on the bottom of peg 3. So the three disks will be placed on peg 3 correctly.

The procedure is recursive. In the first step, you must move three disks from peg 1 to peg 2. To do that, move two disks from peg 1 to peg 3, then one disk from peg 1 to peg 2, then two disks from peg 3 to peg 2. FIGURE 2.36 shows this sequence. Using the previous reasoning, these steps will be carried out correctly. In the process of moving two disks from peg 1 to peg 3, you may place any of these two disks on the bottom two disks of peg 1 without fear of breaking the rules.

Eventually you will reduce the problem to the basis step, where you need to move only one disk. But the solution with one disk is easy. Programming

FIGURE 2.35
The solution for moving four disks from peg 1 to peg 3, assuming that you can move three disks from one peg to any other peg.

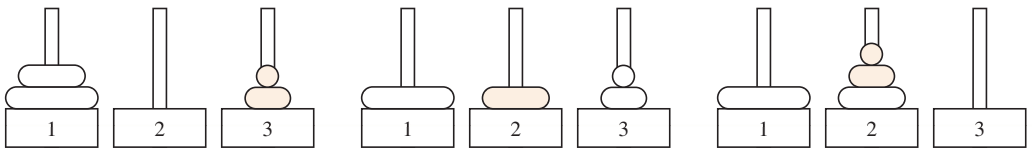


(a) Move three disks from peg 1 to peg 2.

(b) Move one disk from peg 1 to peg 3.

(c) Move three disks from peg 2 to peg 3.

FIGURE 2.36
The solution for moving three disks from peg 1 to peg 2, assuming that you can move two disks from one peg to any other peg.



(a) Move two disks from peg 1 to peg 3.

(b) Move one disk from peg 1 to peg 2.

(c) Move two disks from peg 3 to peg 2.

the solution to the Towers of Hanoi puzzle is a problem at the end of the chapter.

Mutual Recursion

Some problems are best solved by procedures that do not call themselves directly but that are recursive nonetheless. Suppose a main program calls procedure `a`, and procedure `a` contains a call to procedure `b`. If procedure `b` contains a call to procedure `a`, then `a` and `b` are mutually recursive. Even though procedure `a` does not call itself directly, it does call itself indirectly through procedure `b`.

There is nothing different about the implementation of mutual recursion compared to plain recursion. Stack frames are allocated on the run-time stack the same way, with the return value allocated first, followed by parameters, followed by the return address, followed by local variables.

There is one slight problem in specifying mutually recursive procedures in a C program, however. It arises from the fact that procedures must be declared before they are used. If procedure `a()` calls procedure `b()`, the declaration of procedure `b()` must appear before the declaration of procedure `a()` in the listing. But if procedure `b()` calls procedure `a()`, the declaration of procedure `a()` must appear before the declaration of procedure `b()` in the listing. The problem is that if each calls the other, each must appear before the other in the listing, an obvious impossibility.

The function prototype

For this situation, C provides the *function prototype*, which allows the programmer to write the first procedure heading without the body. In a function prototype, you include the complete formal parameter list, but in place of the body, you put `;`. After the function prototype comes the declaration of the second procedure, followed by the body of the first procedure.

Example 2.6 Here is an outline of the structure of the mutually recursive procedures `a()` and `b()` as just discussed:

```
    Constants, types, variables of main program
    void a(SomeType x);
    void b(SomeOtherType y) {
        Body for b
    }
    void a(SomeType x) {
        Body for a
    }
    int main() {
        Executable statements of main program
    }
```

If `b()` has a call to `a()`, the compiler will be able to verify that the number and types of the actual parameters match the formal parameters of `a` scanned earlier in the function prototype. If `a()` has a call to `b()`, the call will be in the body of `a()`. The compiler will have scanned the declaration of `b()` because it occurs before the block of `a()`. ∎

Mutual recursion in a recursive descent compiler

Although mutual recursion is not as common as recursion, some compilers are based on a technique called *recursive descent*, which uses mutual recursion heavily. You can get an idea of why this is so by considering the structure of C statements. It is possible to nest an `if` inside a `while`, which is nested in turn inside another `if`. A compiler that uses recursive descent has a procedure to translate `if` statements and another procedure to translate `while` statements. When the procedure that is translating the outer `if` statement encounters the `while` statement, it calls the procedure that translates `while` statements. But when that procedure encounters the nested `if` statement, it calls the procedure that translates `if` statements; hence the mutual recursion.

The Cost of Recursion

The selection of examples in this section was based on only one criterion: the ability of the example to illustrate recursion. You can see that recursive solutions require much storage for the run-time stack. It also takes time to allocate and deallocate the stack frames. Recursive solutions are expensive in both space and time.

If you can solve a problem easily without recursion, the nonrecursive solution will usually be better than the recursive solution. Figure 2.18, the nonrecursive function to calculate the factorial, is certainly better than the recursive factorial function of Figure 2.22. Both Figure 2.25, to find the

sum of the numbers in an array, and Figure 2.32 can easily be programmed nonrecursively with a loop.

The binomial coefficient $b(n, k)$ has a nonrecursive definition that is based on factorials:

$$b(n, k) = \frac{n!}{k!(n - k)!}$$

If you compute the factorials nonrecursively, a program based on this definition may be more efficient than the corresponding recursive program. Here the choice is a little less clear because the nonrecursive solution requires multiplication and division, but the recursive solution requires only addition.

Some problems are recursive by nature and can be solved only nonrecursively with great difficulty. The problem of solving the Towers of Hanoi puzzle is recursive by nature. You can try to solve it without recursion to see how difficult it would be. Quick sort, one of the best-known sorting algorithms, falls into this category also. It is much more difficult to program quick sort nonrecursively than recursively.

Integrated Development Environments

The C programs in this chapter are short, with few, if any, additional functions other than `main()`. Furthermore, the additional functions are all contained in the same file as the main function. A typical commercial application consists of tens or hundreds of additional functions, which makes it impractical for the entire application to be contained in a single file.

A common convention for organizing a big software project is to collect small groups of related functions into separate files. In C, there are two types of files for these collections of functions—header files with file extension `.h` and source files with file extension `.c`. For example, the `#include` statement at the beginning of the program instructs the compiler to refer to the header file `stdio.h` for the input/output library. The header file contains the head of each function declaration—that is, the return type, the name of the function, and the formal parameter list. The corresponding source file contains not only the head of each function but also the C code that implements it.

An integrated development environment (IDE) is an application for software developers to write large programs containing many header files and source files. A typical IDE has an integrated text editor for writing source code and a point-and-click interface for managing all the header files and source files in the project. The text editor provides syntax highlighting for the source code and code completion for keywords and identifiers. The IDE provides a graphical user interface for the compiler and an integrated pane for program input and output.

Examples of some of the more popular IDEs are NetBeans, Eclipse, Qt Creator, Visual Studio, and Xcode. NetBeans is an open-source IDE that originated as a student project and was picked up by Sun Microsystems, the company that originated the Java programming language. When Oracle acquired Sun, they continued the NetBeans project and still maintain it. Eclipse is also an open-source IDE, originated by IBM. Both NetBeans and Eclipse are written in Java but provide developers with a choice of different programming languages to code in, including Java, C, and C++. Qt Creator is another open-source IDE and is maintained by The Qt Company, a subsidiary of Digia Plc. The Pep/9 software that accompanies this text is written in C++ with Qt Creator. In contrast to these open-source, cross-platform IDEs, Visual Studio and Xcode are proprietary

FIGURE 2.37
The NetBeans IDE with the program from Figure 2.32.

IDEs. Visual Studio is a Microsoft product for software development, primarily for the Windows operating system, and Xcode is an Apple product, primarily for the OS X and iOS operating systems.

FIGURE 2.37 shows the NetBeans IDE with the program from Figure 2.32. The left pane with the tab labeled `Projects` is for accessing all the files in a project. You can see that the IDE is managing four projects, with access to the project labeled `fig0232`. The IDE groups header files and source files separately. The programmer has opened the source file in the upper-right pane, which is the integrated text editor. She compiled and ran the program with a single click, producing the output in the bottom-right pane.

Another capability integrated into an IDE is version control. When many developers work on a single software project simultaneously, they need a way to manage potential conflicts when they modify the same source file. Version control systems manage the process by keeping track of all the changes made to the code and providing a systematic, documented process to resolve any conflicts. The two most popular version control systems are Subversion (SVN) and Git. Of the two, Git is the newer and more widespread. IDEs typically provide access to these version control systems through their graphical user interfaces. The Pep/9 software is maintained using Git and is available on GitHub, which is a software hosting service on the Internet.

2.5 Dynamic Memory Allocation

In C, values are stored in three distinct areas of main memory:

The C memory model

- Fixed locations in memory for global variables
- The run-time stack for local variables and parameters
- The heap for dynamically allocated variables

You do not control allocation and deallocation from the heap during procedure calls and returns. Instead, you allocate from the heap with the help of pointer variables. Allocation on the heap, which is not triggered automatically on the run-time stack by procedure calls, is known as *dynamic memory allocation*.

Dynamic memory allocation

Pointers

When you declare a global or local variable, you specify its type. For example, you can specify the type to be an integer, or a character, or an array. Similarly, when you declare a pointer, you must declare that it points to some type. The pointer itself can be global or local. The value to which it points, however, resides in the heap and is neither global nor local.

C provides two functions to control dynamic memory allocation:

Two operators that control dynamic memory allocation

- `malloc()`, to allocate from the heap
- `free()`, to deallocate from the heap

Although memory deallocation with the `free()` function is important, this text does not describe how it operates. The programs that use pointers in this text are bad examples of software design because of this omission. The intent of the programs is to show the relationship between Levels HOL6 and Asmb5. This relationship will become evident in Chapter 6, which describes the translation of the programs.

Omitting `free()` is a simplification.

The `malloc()` function expects the number of bytes of memory to allocate for its actual parameter. It does two things when it executes:

The two actions of the `malloc()` function

- It allocates a memory cell from the heap equal in size to the number of bytes as specified in its parameter.
- It returns a pointer to the newly allocated storage.

Two assignments are possible with pointers. You can assign a value to a pointer, or you can assign a value to the cell to which the pointer points. The first assignment is called a *pointer assignment*, which behaves according to the following rule:

The pointer assignment rule

- If `p` and `q` are pointers, the assignment `p = q` makes `p` point to the same cell to which `q` points.

```
#include <stdio.h>
#include <stdlib.h>
int *a, *b, *c;
int main() {
   a = (int *) malloc(sizeof(int));
   *a = 5;
   b = (int *) malloc(sizeof(int));
   *b = 3;
   c = a;
   a = b;
   *a = 2 + *c;
   printf("*a = %d\n", *a);
   printf("*b = %d\n", *b);
   printf("*c = %d\n", *c);
   return 0;
}
```

Output

```
*a = 7
*b = 7
*c = 5
```

FIGURE 2.38
A C nonsense program that illustrates the pointer type.

FIGURE 2.38 is a nonsense program that illustrates the actions of the `malloc()` function and the pointer assignment rule. It uses global pointers, but the output would be the same if the pointers were local. If they were local, they would be allocated on the run-time stack instead of being at a fixed location in memory.

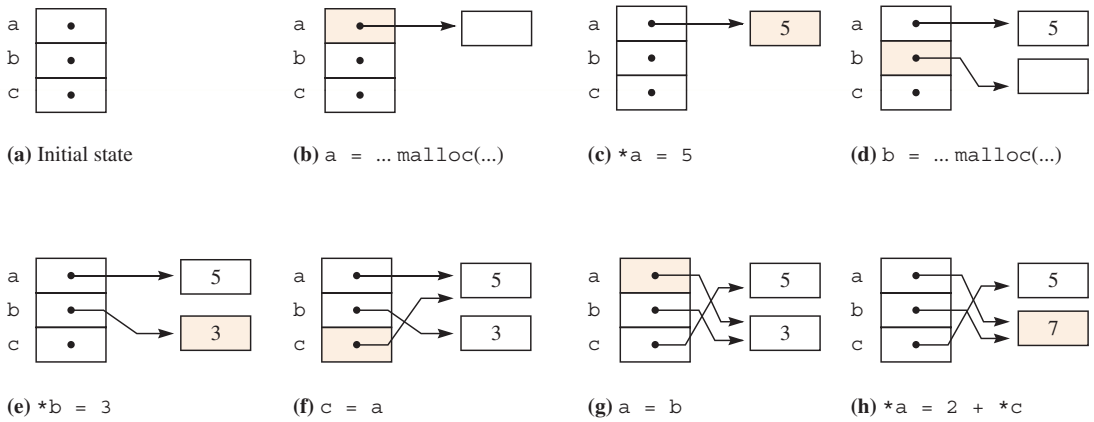
In the declaration of the global pointers

```
int *a, *b, *c;
```

the asterisk before the variable name indicates that the variable, instead of being an integer, is a pointer to an integer. In FIGURE 2.39, Figure 2.39(a) shows the pictorial representation of a pointer value to be a small black dot.

Figure 2.39(b) illustrates the action of the `malloc()` function. The `sizeof()` function takes a type and returns the number of bytes necessary to hold a value of that type. The expression `(int *)` in the first assignment is a type cast. Because `malloc()` returns a generic pointer, the type cast is necessary to convert the generic pointer to a pointer to an `int`. Consequently, the call to `malloc()` allocates a cell from the heap large enough to store an integer value, and it returns a pointer to the value. The assignment makes

FIGURE 2.39
A trace of the program in Figure 2.38.



(a) Initial state **(b)** `a = ... malloc(...)` **(c)** `*a = 5` **(d)** `b = ... malloc(...)`

(e) `*b = 3` **(f)** `c = a` **(g)** `a = b` **(h)** `*a = 2 + *c`

`a` point to the newly allocated cell. Figure 2.39(c) shows how to access the cell to which a pointer points. Because `a` is a pointer, `*a` is the cell to which `a` points. Figure 2.39(f) illustrates the pointer assignment rule. The assignment `c = a` makes `c` point to the same cell to which `a` points. Similarly, the assignment `a = b` makes `a` point to the same cell to which `b` points. In Figure 2.39 (h), the assignment is not to pointer `a` but to the cell to which `a` points.

Structures

Structures are the key to data abstraction in C. They let the programmer consolidate variables with primitive types into a single abstract data type. Both arrays and structures are groups of values. However, all cells of an array must have the same type. Each cell is accessed by the numeric integer value of the index. With a structure, the cells can have different types. C provides the `struct` construct to group the values. The C programmer gives each cell, called a *field*, a field name.

FIGURE 2.40 shows a program that declares a `struct` named `person` that has four fields named `first`, `last`, `age`, and `gender`. The program declares a global variable named `bill` that has type `person`. Fields `first`, `last`, and `gender` have type `char`, and field `age` has type `int`.

To access the field of a structure, you place a period between the name of the variable and the name of the field you want to access. For example, the test of the `if` statement

```
if (bill.gender == 'm')
```

accesses the field named `gender` in the variable named `bill`.

FIGURE 2.40
The C structure.

```
#include <stdio.h>

struct person {
   char first;
   char last;
   int age;
   char gender;
};
struct person bill;
int main() {
   scanf("%c%c%d %c", &bill.first, &bill.last, &bill.age, &bill.gender);
   printf("Initials: %c%c\n", bill.first, bill.last);
   printf("Age: %d\n", bill.age);
   printf("Gender: ");
   if (bill.gender == 'm') {
      printf("male\n");
   }
   else {
      printf("female\n");
   }
   return 0;
}
```

Input

```
bj 32 m
```

Output

```
Initials: bj
Age: 32
Gender: male
```

Linked Data Structures

Programmers frequently combine pointers and structures to implement linked data structures. The `struct` is usually called a *node*, a pointer points to a node, and the node has a field that is a pointer. The pointer field of the node serves as a link to another node in the data structure. FIGURE 2.41 is a program that implements a linked list data structure. The first loop inputs a sequence of integers terminated by the sentinel value –9999, placing the

FIGURE 2.41
A C program to input and output a linked list.

```
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node *next;
};
int main() {
   struct node *first, *p;
   int value;
   first = 0;
   scanf("%d", &value);
   while (value != -9999) {
      p = first;
      first = (struct node *) malloc(sizeof(struct node));
      first->data = value;
      first->next = p;
      scanf("%d", &value);
   }
   for (p = first; p != 0; p = p->next) {
      printf("%d ", p->data);
   }
   return 0;
}
```

Input

```
10 20 30 40 -9999
```

Output

```
40 30 20 10
```

first value in the input stream at the end of the linked list. The second loop outputs each element of the linked list. FIGURE 2.42 is a trace of the first few statement executions of the program in Figure 2.41.

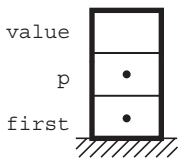
0 is a special pointer value.

The value 0 for a pointer is a special value that is guaranteed to point to no cell at all. It is commonly used in C programs as a sentinel value of linked structures. The statement

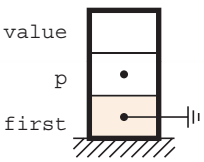
```
first = 0;
```

assigns this special value to local pointer `first`. Figure 2.42(b) shows the value pictorially as a dashed triangle.

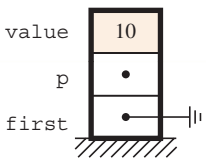
FIGURE 2.42
A trace of the first few statement executions of the program in Figure 2.41.



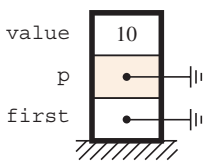
(a) Initial state in `main()`



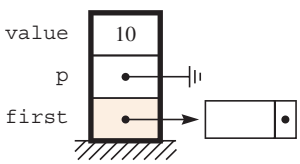
(b) `first = 0`



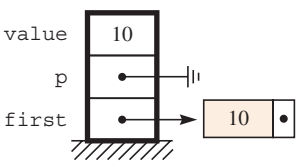
(c) `scanf("%d", &value)`



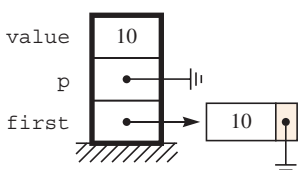
(d) `p = first`



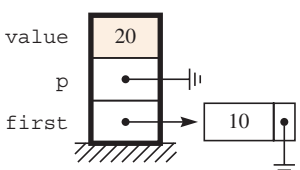
(e) `first = … malloc(…)`



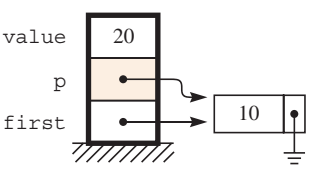
(f) `first->data = value`



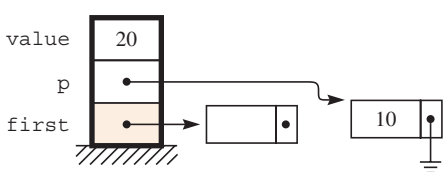
(g) `first->next = p`



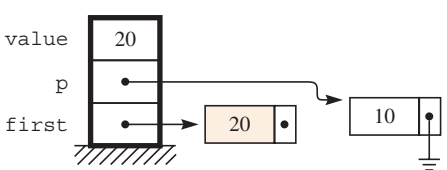
(h) `scanf("%d", &value)`



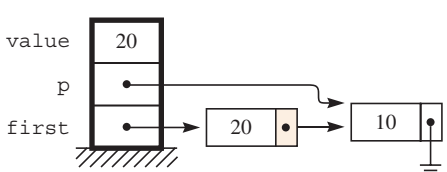
(i) `p = first`



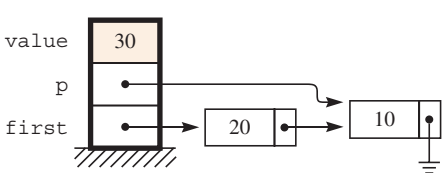
(j) `first = … malloc(…)`



(k) `first->data = value`



(l) `first->next = p`



(m) `scanf("%d", &value)`

You use an asterisk to access the cell to which a pointer points, and a period to access the field of a structure. If a pointer points to a `struct`, you access a field of the `struct` using both the asterisk and the period.

Example 2.7 The following statement assigns the value of variable `value` to the data field of the structure to which it points.

```
(*first).data = value;
```

■

The -> operator

Because this combination of asterisk and period is so common, C provides the arrow operator `->` formed by a hyphen followed immediately by a greater-than symbol. The statement in Example 2.7 can be written using this abbreviation as

```
first->data = value;
```

which Figure 2.42(f) and (k) shows. The program uses the same abbreviation to access the `next` field, which Figure 2.42(g) and (l) shows.

Chapter Summary

In C, values are stored in three distinct areas of main memory: fixed locations in memory for global variables, the run-time stack for local variables and parameters, and the heap for dynamically allocated variables. The two ways in which flow of control can be altered from the normal sequential flow are through selection and repetition. The C `if` and `switch` statements implement selection, and the `while`, `do`, and `for` statements implement repetition. All five statements use the relational operators to test the truth of a condition.

The LIFO nature of the run-time stack is required to implement function and procedure calls. The allocation process for a function is the following: Push storage for the return value, push the actual parameters, push the return address, and push storage for the local variables. The allocation process for a procedure is identical except that storage for the return value is not pushed. The stack frame consists of all the items pushed onto the run-time stack in one function or procedure call.

A recursive procedure is one that calls itself. To avoid calling itself endlessly, a recursive procedure must have an `if` statement that serves as an escape hatch to stop the recursive calls. Two different viewpoints in thinking about recursion are the microscopic and the macroscopic viewpoints. The microscopic viewpoint considers the details of the run-time stack during execution. The macroscopic viewpoint is based on a higher level of abstraction

and is related to proof by mathematical induction. The microscopic viewpoint is useful for analysis; the macroscopic viewpoint is useful for design.

Allocation on the heap with the `malloc()` function is known as *dynamic memory allocation*. The `malloc()` function allocates a memory cell from the heap and returns a pointer to the newly allocated cell. A structure, indicated as `struct` in C, is a collection of values that need not all be the same type. Each value is stored in a field, and each field has a name. Linked data structures consist of nodes, which are structures that have pointers to other nodes. The node for a linked list has a field for a value and a field usually named `next` that points to the next node in the list.

Exercises

Section 2.4

1. The function `sum()` in Figure 2.25 is called for the first time by the main program. From the second time on, it is called by itself. ***(a)** How many times is it called altogether, including the call from `main()`? **(b)** Draw a picture of the main program variables and the run-time stack just after the function is called for the third time. Do not draw the stack frame for `main()`. You should have three stack frames. **(c)** Draw a picture of the main program variables and the run-time stack just before the return from the call of part (b). You should have three stack frames, but with different contents from part (b).

2. Each exercise below has five parts, as follows: (1) Draw the call tree in the style of Figure 2.30 for the function `binCoeff()` of Figure 2.28 assuming the given call statement from the main program. (2) Write down the sequence of calls and returns using the indentation notation on page 92. (3) How many times is the function called, including the call from the main program? (4) What is the maximum number of stack frames on the run-time stack during the execution, not counting the frame for the main program? (5) Draw the run-time stack in the style of Figure 2.29 at the given point during execution.

- ***(a)** Call statement `binCoeff(4,1)` from the main program. For part (5), draw the run-time stack just before the return from `binCoeff(2, 1)`.
- **(b)** Call statement `binCoeff(5,1)` from the main program. For part (5), draw the run-time stack just before the return from `binCoeff(3, 1)`.
- **(c)** Call statement `binCoeff(3,2)` from the main program. For part (5), draw the run-time stack just before the return from `binCoeff(1, 0)`.

(d) Call statement `binCoeff(4,4)` from the main program. For part (5), draw the run-time stack just before the return from `binCoeff(4,4)`.

(e) Call statement `binCoeff(4,2)` from the main program. For part (5), `binCoeff(2,1)` is called twice. Draw the run-time stack just before the return from the second call of the function.

3. Draw the call tree in the style of Figure 2.30 for the program in Figure 2.32 to reverse the letters of an array of characters, assuming the initial string is `"Backward"`. How many times is function `reverse()` called, including the call from `main()`? What is the maximum number of stack frames allocated on the run-time stack, including the stack frame for `main()`? Draw the run-time stack just after the third call to function `reverse()`, including the stack frame for `main()`.

4. The Fibonacci sequence is

0 1 1 2 3 5 8 13 21 . . .

Each Fibonacci number is the sum of the preceding two Fibonacci numbers. The sequence starts with the first two Fibonacci numbers and is defined recursively as

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n < 1. \end{cases}$$

Draw the call tree in the style of Figure 2.30 for the following Fibonacci numbers:

(a) `fib(3)` **(b)** `fib(4)` **(c)** `fib(5)`

For each of these calls, how many times is `fib()` called, including the call from `main()`? What is the maximum number of stack frames allocated on the run-time stack, not including the stack frame for `main()`?

5. For your solution to the Towers of Hanoi in Problem 2.14, draw the call tree for the four-disk problem. How many times is your procedure called, including the call from `main()`? What is the maximum number of stack frames on the run-time stack, not including the stack frame for `main()`?

6. The mystery numbers are defined recursively as

$$\text{myst}(n) = \begin{cases} 2 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ 2 \times \text{myst}(n-1) + \text{myst}(n-2) & \text{if } n > 1. \end{cases}$$

(a) Draw the call tree in the style of Figure 2.30 for `myst(4)`.

(b) What is the value of `myst(4)`?

7. Examine the C program that follows: **(a)** Draw the run-time stack just after the procedure is called for the last time, including the stack frame for `main()`. **(b)** What is the output of the program?

```
#include <stdio.h>
void what(char *word, int j) {
   if (j > 1) {
      word[j] = word[3 - j];
      what(word, j - 1);
   }  // ra2
}
int main() {
   char str[5] = "abcd";
   what(str, 3);
   printf("%s\n", str); // ra1
   return 0;
}
```

Problems

Section 2.1

8. Write a C program that inputs two integers and outputs their quotient and remainder. To output the `%` character, you must write it as `%%` in the format string.

Sample Input

```
13  4
```

Sample Output

```
13/4 has value 3.
13%4 has value 1.
```

Section 2.2

9. Write a C program that inputs an integer and outputs whether the integer is even.

Sample Input

```
15
```

Sample Output

```
15 is not even.
```

10. Write a C program that inputs two integers and outputs the sum of the integers between them.

Sample Input

```
9  12
```

Sample Output

```
The sum of the numbers between 9 and 12 inclusive is 42.
```

Section 2.3

11. Write a C function

```
int rectArea (int len, int wid)
```

that returns the area of a rectangle with length `len` and width `wid`. Test it with a main program that inputs the length and width of a rectangle and outputs its area. Output the value in the main program, not in the function.

Sample Input

```
6  10
```

Sample Output

```
The area of a 6 by 10 rectangle is 60.
```

12. Write a C function

```
void rect(int *ar, int *per, int len, int wid)
```

that computes the area `ar` and perimeter `per` of a rectangle with length `len` and width `wid`. Test it with a main program that inputs the length and width of a rectangle and outputs its area and perimeter. Output the value in the main program, not in the procedure.

Sample Input

```
6  10
```

Sample Output

```
Length: 6
Width: 10
Area: 60
Perimeter: 32
```

Section 2.4

13. Write a C program that asks the user to input a small integer. Then use a recursive function that returns the value of that Fibonacci number as defined in Exercise 4. Do not use a loop. Output the value in the main program, not in the function.

Sample Input/Output

```
Which Fibonacci number? 8
The number is 21.
```

14. Write a C program that prints the solution to the Towers of Hanoi puzzle. It should ask the user to input the number of disks in the puzzle, the peg on which all the disks are placed initially, and the peg on which the disks are to be moved.

Sample Input/Output

```
How many disks do you want to move? 3
From which peg? 3
To which peg? 2

Move a disk from peg 3 to peg 2.
Move a disk from peg 3 to peg 1.
Move a disk from peg 2 to peg 1.
Move a disk from peg 3 to peg 2.
Move a disk from peg 1 to peg 3.
Move a disk from peg 1 to peg 2.
Move a disk from peg 3 to peg 2.
```

15. Write a recursive void function called `rotateLeft()` with two parameters, an array and an integer count *n*, that rotates the first *n* integers in the array to the left. To rotate *n* items left, rotate the first *n* – 1 items left recursively, and then exchange the last two items. For example, to rotate the five items

```
50   60   70   80   90
```

to the left, recursively rotate the first four items to the left:

```
60   70   80   50   90
```

and then exchange the last two items:

```
60   70   80   90   50
```

Test it with a main program that takes as input an integer count followed by the values to rotate. Output the original values and the rotated values. Do not use a loop in `rotateLeft()`. Output the value in the main program, not in the procedure.

Sample Input

```
5   50 60 70 80 90
```

Sample Output

```
Original list: 50  60  70  80  90
Rotated list: 60  70  80  90  50
```

16. Write a function

```
int maximum (int list[], int n)
```

that recursively finds the largest integer between `list[0]` and `list[n]`. Assume at least one element is in the list. Test it with a main program that takes as input an integer count followed by the values. Output the original values followed by the maximum. Do not use a loop in maximum. Output the value in the main program, not in the function.

Sample Input

```
5   50 30 90 20 80
```

Sample Output

```
Original list: 50  30  90  20  80
Largest value: 90
```

Section 2.5

17. The program in Figure 2.41 creates a linked list whose elements are in reverse order compared to their input order. Modify the first loop of the program to create the list in the same order as the input order. Do not modify the second loop.

Sample Input

```
10 20 30 40 -9999
```

Sample Output

```
10 20 30 40
```

18. Declare the following node for a binary search tree:

```
struct node {
   node *left;
   int data;
   node *right;
};
```

where `left` is a pointer to the left subtree and `right` is a pointer to the right subtree. Write a C program that inputs a sequence of integers with –9999 as a sentinel and inserts them into a binary search tree. Output them in ascending order with a recursive procedure that makes an inorder traversal of the search tree.

Sample Input

```
40 90 50 10 80 30 70 60 20 -9999
```

Sample Output

```
10 20 30 40 50 60 70 80 90
```