Pointers On C

Instructor's Guide

Contents

Chapter 1	A Quick Start	1
Chapter 2	Basic Concepts	7
Chapter 3	Data	11
Chapter 4	Statements	15
Chapter 5	Operators and Expressions	23
Chapter 6	Pointers	29
Chapter 7	Functions	37
Chapter 8	Arrays	43
Chapter 9	Strings, Characters, and Bytes	55
Chapter 10	Structures and Unions	69
Chapter 11	Dynamic Memory Allocation	75
Chapter 12	Using Structures and Pointers	79
Chapter 13	Advanced Pointer Topics	87
Chapter 14	The Preprocessor	93
Chapter 15	Input/Output Functions	95
Chapter 16	Standard Library	119
Chapter 17	Classic Abstract Data Types	129
Chapter 18	Runtime Environment	145

1

A Quick Start

1.1 Questions

1. To make the program easier to read, which in turn makes it easier to maintain later.

3. It is easier to see what a named constant represents, if it is well named, than a literal constant, which merely displays its value.

4. `"%d %s %g\n"`

6. The programmer can put in subscript checks where they are needed; in places where the subscript is already known to be correct (for example, from having been checked earlier), there is no overhead expended in checking it again. But the real reason they are omitted is the fact that subscripts are implemented as pointer expressions, which are described in Chapter 8.

7. More characters would be copied than are actually needed; however, the `output_col` would be updated properly, so the next range of characters would be copied into the output array at the proper place, replacing any extra characters from the preceding operation. The only potential problem is that the unbounded `strcpy` might copy more characters into the output array than it has room to hold, destroying some other variables.

1.2 Programming Exercises

1. Watch the solutions for proper use of `void` declarations and a reasonable style. The first program is no place to begin learning bad habits. The program will compile and run on most systems without the `#include` statement.

```
/*
** Print the message "Hello world!" to the standard output.
*/

#include <stdio.h>

void
main( void )
```

Solution 1.1 *continued . . .*

```
{
        printf( "Hello world!\n" );
}
```

Solution 1.1 hello_w.c

3. Many students will attempt to read the input file line by line, which is unnecessarily complicated. Other common errors are to forget to initialize the sum to -1, or to declare it an integer rather than a character. Finally, be sure the variable used to read the characters is an integer; if it is a character variable, the program will stop on systems with signed characters when the input contains the binary value 0377 (which, when promoted to an integer, is -1 and equal to `EOF`). Note that the overflow renders this program nonportable, but we don't know enough yet to avoid it.

```
/*
** This program copies its standard input to the standard output, and computes
** a checksum of the characters.  The checksum is printed after the input.
*/

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

int
main( void )
{
        int     c;
        char    sum = –1;

        /*
        ** Read the characters one by one, and add them to the sum.
        */
        while( (c = getchar()) != EOF ){
                putchar( c );
                sum += c;
        }

        printf( "%d\n", sum );

        return EXIT_SUCCESS;
}
```

Solution 1.3 checksum.c

4. The basis of this program is an array which holds the longest string found so far, but a second array is required to read each line. The buffers are declared 1001 characters long to hold the data plus its terminating NUL byte. The only tricky thing is the initialization to prevent garbage from being printed when the input is empty.

```
/*
** Reads lines of input from the standard input and prints the longest line that
** was found to the standard output.  It is assumed that no line will exceed
```

Solution 1.4 *continued . . .*

```
** 1000 characters.
*/

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

#define MAX_LEN         1001    /* Buffer size for longest line */

int
main( void )
{
        char    input[ MAX_LEN ];
        int     len;
        char    longest[ MAX_LEN ];
        int     longest_len;

        /*
        ** Initialize length of the longest line found so far.
        */
        longest_len = -1;

        /*
        ** Read input lines, one by one.
        */
        while( gets( input ) != NULL ){
                /*
                ** Get length of this line.  If it is longer than the previous
                ** longest line, save this line.
                */
                len = strlen( input );
                if( len > longest_len ){
                        longest_len = len;
                        strcpy( longest, input );
                }
        }

        /*
        ** If we saved any line at all from the input, print it now.
        */
        if( longest_len >= 0 )
                puts( longest );

        return EXIT_SUCCESS;
}
```

Solution 1.4 longest.c

6. The statements

```
/*
** Make sure we have an even number of inputs ...
*/
if( num % 2 != 0 ){
        puts( "Last column number is not paired." );
```

```
                exit( EXIT_FAILURE );
        }
```

are removed from the `read_column_numbers` function, and the `rearrange` function is modified as follows. Note that the computation of `nchars` is moved after the test that checks whether the starting column is within the bounds of the input string.

```
/*
** Process a line of input by concatenating the characters from the indicated
** columns.  The output line is then NUL terminated.
*/
void
rearrange( char *output, char const *input, int const n_columns,
    int const columns[] )
{
        int     col;            /* subscript for columns array */
        int     output_col;     /* output column counter */
        int     len;            /* length of input line */

        len = strlen( input );
        output_col = 0;

        /*
        ** Process each pair of column numbers.
        */
        for( col = 0; col < n_columns; col += 2 ){
                int     nchars;

                /*
                ** If the input line isn't this long or the output array is
                ** full, we're done.
                */
                if( columns[col] >= len || output_col == MAX_INPUT - 1 )
                        break;

                /*
                ** Compute how many characters to copy.
                */
                if( col + 1 < n_columns )
                        nchars = columns[col + 1] - columns[col] + 1;
                else
                        nchars = len - columns[col + 1];

                /*
                ** If there isn't room in the output array, only copy what will
                ** fit.
                */
                if( output_col + nchars > MAX_INPUT - 1 )
                        nchars = MAX_INPUT - output_col - 1;

                /*
                ** Copy the relevant data.
```

Solution 1.6 *continued . . .*

```
                */
                strncpy( output + output_col, input + columns[col], nchars );
                output_col += nchars;
        }

        output[output_col] = '\0';
}
```

Solution 1.6 rearran3.c

2

Basic Concepts

2.1 Questions

1. The outside comment ends at the end of the first enclosed comment. This makes the variable `i` undefined in the rest of the function; the phrase `End of commented-out code` will be a syntax error, and the final closing `*/` will be illegal.
2. Advantages:
   a. When you want to modify a function, it is easy to determine which file it is in.
   b. You can safely use longer function names. Depending on the limits of your particular system, internal must be distinct from one another somewhere in the first 31 characters or more, whereas external names must be distinct from one another in the first six characters.

   Disadvantages:
   a. Depending on how powerful your editor is, it may be harder to locate a particular piece of code in a large file than a small one.
   b. Depending on your operating system, the type of editor you are using, and the size of the file, it may be more time consuming to edit a large file than a small one.
   c. If you make a mistake in your editor, it is easy to lose the whole program.
   d. Even if you change only one function, the entire program must be recompiled, which takes longer than recompiling only the modified function.
   e. It is harder to reuse general purpose functions from the program if they are buried in with all the code that is specific to that problem.
3. `"\"Blunder\?\?!??\""`
   Note that the final pair of questions marks need not be escaped, as the next character does not form a trigraph.
5. The preprocessor replaces the comment with a single space, which makes the resulting statement illegal.
6. Nothing. There are no conflicts with the C keywords or between the last two identifiers because they are all differ in the case of their characters.

9. The answer will vary from system to system, but on a UNIX system

   ```
   cc main.c list.c report.c
   ```

   is one command that does the job.

10. The answer will vary from system to system, but on a UNIX system you would add

    ```
    -lparse
    ```

    to the end of the command.

2.2 Programming Exercises

1. The sole purpose of this exercise is to make use of separate compilation, so the hardest thing about this program is accepting that it is as trivial as it sounds. The main program may also be implemented with a loop.

```
/*
** Main program to test the increment and negate functions.
*/

#include <stdio.h>

main( void )
{
        printf( "%d %d\n", increment( 10 ), negate( 10 ) );
        printf( "%d %d\n", increment( 0 ), negate( 0 ) );
        printf( "%d %d\n", increment( -10 ), negate( -10 ) );
}
```

Solution 2.1a main.c

```
/*
** A function to increment a value and return it.
*/

int
increment( int value )
{
        return value + 1;
}
```

Solution 2.1b increment.c

```
/*
** A function to negate a value and return it.
*/

int
negate( int value )
```

Solution 2.1c *continued . . .*

```
{
        return –value;
}
```

Solution 2.1c negate.c

3

Data

3.1 Questions

Note to instructor: The first few questions ask the students to investigate some behaviors on their own implementation. The answers given describe what the Standard says about each situation; this is far more detailed than students are likely discover on their own. You may wish to use some of the material in these answers in your presentation.

1. Depends; look in `<limits.h>` for the definitions. The location of this include file may vary; on UNIX systems it is typically found in the directory `/usr/include`. For Borland compilers, look in the `include` directory found where the compiler was installed.

2. Depends; look in `<float.h>` for the definitions. See above for the location of this file.

4. Many compilers will give a warning message. The Standard defines the runtime behavior roughly this way: if the value to be assigned is small enough to fit in the shorter variable, its value is preserved; otherwise, it is implementation dependent. The carefully worded description implies that the implementation may simply discard the high-order bits that don't fit, which on most machines gives the most efficient object code. This is obviously not portable.

5. When you compile it, you may get a warning message. The runtime behavior is defined in much the same manner as for integers: If the value fits in the smaller variable, it works; otherwise it is implementation dependent. With floating-point values, though, a value "doesn't fit" only if its exponent is larger than the shorter type can hold. If the exponent fits, there is still the mantissa, which might have more significance than the shorter type can maintain. In this case, the value is replaced with nearest value that can be represented in the shorter variable; it is implementation dependent whether this rounds, truncates, or does something else.

6.
```
enum Change { PENNY = 1, NICKEL = 5, DIME = 10,
    QUARTER = 25, HALF_DOLLAR = 50, DOLLAR = 100 };
```

   Though the problem did not require it, this declaration gives each symbol a value indicating its actual worth. This would facilitate arithmetic on these values.

8. Depends on the implementation; consult the documentation.

9. It is needed only for characters, and even then only on machines where the default character is unsigned. It is allowed in other contexts (e.g., `signed int`) only for consistency.

12. There is no difference.
13. The left declaration still does what it previously did, but the statements on the right are in error; you cannot assign to a constant variable.
14. True, except when a nested block declares another variable with the same name, which hides the earlier variable and makes it inaccessible from within the nested block.
15. False. The only automatic variables are those with block scope, and these cannot ever be accessed by name from other blocks.
16. No, that changes its storage class, but not its scope; the answer is still false.
17. None is needed.
18. No; it still has internal linkage, so every function that comes later in the file may access it.
19. `extern int x;`
20. Yes; now there is no declaration that will enable you to access `x` from a different source file.
22. Inside: the variable is automatic, it is reinitialized each time the function is called, its scope is limited to the function, it has no linkage. Outside: the variable is static, it is initialized only once before the program begins to run, it has file scope, and external linkage.
23. The trick to this one is to realize that function `y` can be put ahead of `x` in the file; after that, the rest is straightforward. Watch for assignment statements though; the problem specifies no executable statements in the functions.

```
static  char    b = 2;

void
y( void )
{
}

int     a = 1;

void
x( void )
{
        int     c = 3;
        static  float   d = 4;
}
```

24. There is one error: The declaration of `c` in line 6 conflicts with the function parameter `c`. Some compilers have been seen to flag line 24 as an error, saying it conflicts with the declaration in line 20. This should not be an error, as the scope of `y` from line 20 runs out at line 22, so there is no conflict.

Name (Line)	*Storage Class*	*Scope*	*Linkage*	*Initial Value*
w (1)	static	1–8, 17–31	internal	5
x (2)	static	2–18, 23–31	external	Note a
func1 (4)	–	4–31	external	–
a (4)	auto	5–18, 23	none	Note b

Name (Line)	*Storage Class*	*Scope*	*Linkage*	*Initial Value*
b, c (4)	auto	5–11, 16–23	none	Note b
d (4)	auto	6–8, 17, 23	none	garbage
e (4)	auto	6–8, 17–23	none	1
d (9)	auto	9–11, 16	none	garbage
e, w (9)	auto	9–16	none	garbage
b, c, d (12)	auto	12–15	none	garbage
y (12)	static	13–15	none	2
a, d, x (19)	register	19–22	none	garbage
y (19)	static	20–22	external	Note a
y (24)	static	24–31	internal	zero
func2 (26)	–	26–31	external	–
a (26)	auto	27–31	none	Note b
y (28)	static	28–31	Note c	see y (24)
z (29)	static	29–31	none	zero

Note a:
If the variable is not initialized in any other declaration, it will have an initial value of zero.

Note b:
The initial value of a function parameter is the argument that was passed when the function was called.

Note c:
The `extern` keyword doesn’t change the linkage of `y` that was declared in line 24.

4

Statements

4.1 Questions

2. Trick question! There is no “assignment statement” in C. Assignments are done with an expression statement that uses the assignment operator, as in:

```
x = y + z;
```

3. Yes, it is legal. This is useful if you need to introduce a temporary variable for the enclosed statements, but wish to restrict access to the variable to only those statements.

5. The integers starting at 0 and ending with 9. The value 10 is not printed.

6. When there are no initialization or adjustment expressions.

7. Despite the indentation, the call to `putchar` is not in the loop because there are no braces. As a result, the input is not printed, only the checksum. Note that the only thing printed by `putchar` is the end of file indicator; on most systems, this is not a valid character.

8. When the body of the loop must be executed once even if the condition is initially false.

10. Either a `for` or a `while` may be used, but not a `do`.

```
int n_blank;
int counter;

scanf( "%d", &n_blank );
for( counter = 0; counter < n_blank;
    counter = counter + 1 ){
        putchar( '\n' );
}
```

11.
```
if( x < y || a >= b )
        printf( "WRONG" );
else
        printf( "RIGHT );
```

13. Note: a better solution would be to declare an array of these string literals, as discussed in Chapter 8.

```
switch( which_word ){
case 1:
        printf( "who" );
        break;

case 2:
        printf( "what" );
        break;

case 3:
        printf( "when" );
        break;

case 4:
        printf( "where" );
        break;

case 5:
        printf( "why" );
        break;

default:
        printf( "don't know" );
        break;
}
```

14.
```
while( hungry() )
        eat_hamburger();
```

15.
```
do
        eat_hamburger();
while( hungry() );
```

16.
```
if( precipitating )
        if( temperature < 32 )
                printf( "snowing" );
        else
                printf( "raining" );
else
        if( temperature < 60 )
                printf( "cold" );
        else
                printf( "warm" );
```

4.2 Programming Exercises

2. This solution exploits the fact that two is the only even number that is prime. Other than that, it is pretty simplistic.

```
/*
** Compute and print all the prime numbers from 1 to 100.
*/

#include <stdlib.h>

int
main()
{
        int     number;
        int     divisor;

        /*
        ** One and two are easy.
        */
        printf( "1\n2\n" );

        /*
        ** No other even numbers are prime; look at the remaining odd ones.
        */
        for( number = 3; number <= 100; number = number + 2 ){
                /*
                ** See if any divisor from 3 up to the number evenly divides the
                ** number.
                */
                for( divisor = 3; divisor < number; divisor = divisor + 2 ){
                        if( number % divisor == 0 )
                                break;
                }

                /*
                ** If the loop above stopped because the divisor got too big,
                ** we've got a prime.
                */
                if( divisor >= number )
                        printf( "%d\n", number );
        }
}
```

Solution 4.2 prime.c

3. The program should look for numbers that are not a triangle at all: negative numbers and disconnected lines such as 25, 1, and 1. If they're sharp, they will also look for degenerate triangles such as 10, 6, and 4, which have no area and are thus really just a line. The program should use floating-point numbers, since there is no reason why a triangle must have sides of integral length. The logic is greatly simplified by sorting the three numbers before trying to classify them.

```
/*
** Classify the type of a triangle given the lengths of its sides.
*/

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

int
main()
{
        float   a;
        float   b;
        float   c;
        float   temp;

        /*
        ** Prompt for and read the data.
        */
        printf( "Enter the lengths of the three sides of the triangle: " );
        scanf( "%f %f %f", &a, &b, &c );

        /*
        ** Rearrange the values so that a is the longest and c is the shortest.
        */
        if( a < b ){
                temp = a;
                a = b;
                b = temp;
        }
        if( a < c ){
                temp = a;
                a = c;
                c = temp;
        }
        if( b < c ){
                temp = b;
                b = c;
                c = temp;
        }

        /*
        ** Now see what kind of triangle it is.  Note that if any of the sides
        ** is <= 0 (and we really only have to check the shortest one for this),
        ** or if the two shorter sides together are shorter than the longest
        ** side, it isn't a triangle at all.
        */
        if( c <= 0 || b + c < a )
                printf( "Not a triangle.\n" );

        else if( a == b && b == c )
                printf( "Equilateral.\n" );
```

Solution 4.3

continued . . .

```
        else if( a == b || b == c )
                printf( "Isosceles.\n" );

        else
                printf( "Scalene.\n" );

        return EXIT_SUCCESS;
}
```

Solution 4.3 triangle.c

5. The `LINE_SIZE` need only be 128 if `gets` is used, because the newlines are stripped off. If they read the input character by character and store the newline, then it must be 129. The initialization of the `previous_line` array to the first line of the input avoids the special case for the first line that would otherwise be needed.

```
/*
** Print one line from each set of duplicate lines in the standard input.
*/
#include <stdio.h>

#define TRUE            1
#define FALSE           0

#define LINE_SIZE       129

main()
{
        char    input[LINE_SIZE], previous_line[LINE_SIZE];
        int     printed_from_group = FALSE;

        if( gets( previous_line ) != NULL ){
                while( gets( input ) != NULL ){
                        if( strcmp( input, previous_line ) != 0 ){
                                printed_from_group = FALSE;
                                strcpy( previous_line, input );
                        }
                        else if( !printed_from_group ){
                                printed_from_group = TRUE;
                                printf( "%s\n", input );
                        }
                }
        }
}
```

Solution 4.5 pr_dup.c

6. You must look character by character to get to the starting position to ensure that it is not beyond the end of the string.

```
/*
** Extract the specified substring from the string in src.
*/
int
substr( char dst[], char src[], int start, int len )
{
        int     srcindex;
        int     dstindex;

        dstindex = 0;

        if( start >= 0 && len > 0 ){
                /*
                ** Advance srcindex to right spot to begin, but stop if we reach
                ** the terminating NUL byte.
                */
                for( srcindex = 0;
                    srcindex < start && src[srcindex] != '\0';
                    srcindex += 1 )
                        ;

                /*
                ** Copy the desired number of characters, but stop at the NUL if
                ** we reach it first.
                */
                while( len > 0 && src[srcindex] != '\0' ){
                        dst[dstindex] = src[srcindex];
                        dstindex += 1;
                        srcindex += 1;
                        len -= 1;
                }
        }

        /*
        ** Null-terminate the destination.
        */
        dst[dstindex] = '\0';
        return dstindex;
}
```

Solution 4.6 substr.c

7. Watch for solutions that check for spaces but not for tabs.

   This solution is implemented with pointers, but could easily have been done using subscripts. The use of ++ makes the code more compact, but harder to read. Its approach is to keep two pointers into the string: one from which we get characters, and one to which they are written. When white space is encountered, the source pointer is advanced but the destination isn't. The process continues until the NUL byte is reached in the source. The destination string is then terminated.

   Note that it is ok to overwrite the string because the destination can never become longer than it originally was.

```
/*
** Shrink runs of white space in the given string to a single space.
*/
#define NUL     '\0'

void
deblank( char *string )
{
        char    *dest;
        char    *src;
        int     ch;

        /*
        ** Set source and destination pointers to beginning of the string, then
        ** move to 2nd character in string.
        */
        src = string;
        dest = string++;

        /*
        ** Examine each character from the source string.
        */
        while( (ch = *src++) != NUL ){
                if( is_white( ch ) ){
                        /*
                        ** We found white space.  If we're at the beginning of
                        ** the string OR the previous char in the dest is not
                        ** white space, store a blank.
                        */
                        if( src == string || !is_white( dest[-1] ) )
                                *dest++ = ' ';
                }
                else {
                        /*
                        ** Not white space: just store it.
                        */
                        *dest++ = ch;
                }
        }
        *dest = NUL;
}

int
is_white( int ch )
{
        return ch == ' ' || ch == '\t' || ch == '\v' || ch == '\f' || ch == '\n'
            || ch == '\r';
}
```

Solution 4.7 deblank.c

5

Operators and Expressions

5.1 Questions

1. The cast is applied to the result of the division, and because both operands are integers, a truncating integer division is done. The value therefore is 2.0. If you wanted to do a floating-point division, use this expression instead:

   ```
   (float)25 / 10
   ```

   The same applies to other integer expressions.

3. They are often used when programming device controllers to set or test specific bits in specific locations. If anyone comes up with other good answers, please mail them to me at `kar@cs.rit.edu`!

5. Note that the parentheses are not actually needed, as the higher precedence of the `&&` operator already gives this result.

   ```
   leap_year = year % 400 == 0 ||
       ( year % 100 != 0 && year % 4 == 0 );
   ```

7. It prints `In range`. This is because `1 <= a` is true and evaluates to 1; the expression then tests `1 <= 10`, which is also true.

   It is legal, but doesn't do what it appears to do. This is a potential pitfall for new programmers, but those coming from another language usually don't make this mistake because it is not legal in other languages.

8. ```
   for( a = f1( x );
       b = f2( x + a ), c = f3( a, b ), c > 0;
       a = f1( ++x ) ){
           statements
   }
   ```

9. No, it fails if the array contains nonzero values that happen to sum to zero.

10. a. -25
   ```

b. -25, `b` = -24
c. 9, `a` = 9
d. 1
e. 4
f. -5
g. 40
h. -4
i. 1
j. 10, `b` = 10
k. 0
l. 2
m. -19
n. -17
o. 24
p. 0
q. 1
r. 10
s. 12, `a` = 12
t. 4, `b` = 4
u. -4, `b` = -4
v. 4, `a` = 4
w. 1, `d` = 1
x. 3, `a` = 3, `b` = 3, `c` = 3
y. -27, `c` = -15, `e` = -27
z. -65
aa. -1
bb. 1
cc. 1
dd. 1
ee. 0
ff. 1
gg. 0
hh. -25590
ii. 1
jj. 1, `a` = 11
kk. 0, `b` = -24
ll. 1, `b` = -24
mm. 17, `c` = 3
nn. 5, `a` = 5
oo. 80, `a` = 80, `d` = 2

11. a. `a + b / c`
    b. `( a + b ) / c`
    c. `a * b % 6`
    d. `a * ( b % 6 )`
    e. `a + b == 6`
    f. `! ( a >= '0' && a <= '9' )`
    g. `( a & 0x2f ) == ( b | 1 ) && ~ c > 0`
    h. `( a << b ) – 3 < b << a + 3`
    i. `~ a ++`
    j. `( a == 2 || a == 4 ) && ( b == 2 || b == 4 )`
    k. `a & b ^ ( a | b )`
    l. `a + ( b + c )`

12. On a two's complement machine, declare a signed integer, assign it a negative value, right shift it one bit, and then print the result. If it is negative, an arithmetic shift was used; positive indicates a logical shift.

5.2 Programming Exercises

2. This program is not portable to EBCDIC-based systems for the reasons described in Programming Exercise 1. However, note the use of a separate function to do the character wraparound independent of case.

```
/*
** Encrypt the text on the standard input by rotating the alphabetic characters
** 13 positions through the alphabet.  (Note: this program decrypts as well.)
*/
#include <stdio.h>

/*
**      Encrypt a single character.  The base argument is either an upper or
**      lower case A, depending on the case of the ch argument.
*/
int
encrypt( int ch, int base )
{
        ch –= base;
        ch += 13;
        ch %= 26;
        return ch + base;
}

/*
**      Main program.
*/
main( void )
{
        int     ch;

        while( (ch = getchar()) != EOF ){
                if( ch >= 'A' && ch <= 'Z' )
                        ch = encrypt( ch, 'A' );
                else if( ch >= 'a' && ch <= 'z' )
                        ch = encrypt( ch, 'a' );
                putchar( ch );
        }
}
```

Solution 5.2 crypt.c

4. With the hints provided in the discussion on bitwise operators, this is not difficult. Watch for separate functions to isolate the arithmetic that locates the specific bit in one place. A complete implementation should also have a header file defining the function prototypes.

```
/*
** Prototypes for a suite of functions that implement an array of bits in a
** character array.
*/
```

Solution 5.4a *continued . . .*

```
/*
**      Set a specific bit
*/
void    set_bit( char bit_array[], unsigned bit_number );

/*
**      Clear a specific bit
*/
void    clear_bit( char bit_array[], unsigned bit_number );

/*
**      Assign a value to a bit
*/
void    assign_bit( char bit_array[], unsigned bit_number,
            int value );

/*
**      Test a specific bit
*/
int     test_bit( char bit_array[], unsigned bit_number );
```

Solution 5.4a bitarray.h

```
/*
** Implements an array of bits in a character array.
*/

#include <limits.h>
#include "bitarray.h"

/*
**      Prototypes for internal functions
*/
unsigned character_offset( unsigned bit_number );
unsigned bit_offset( unsigned bit_number );

/*
**      Set a specific bit
*/
void
set_bit( char bit_array[], unsigned bit_number )
{
        bit_array[ character_offset( bit_number ) ] |=
            1 << bit_offset( bit_number );
}

/*
**      Clear a specific bit
*/
void
clear_bit( char bit_array[], unsigned bit_number )
{
```

Solution 5.4b *continued . . .*

```
        bit_array[ character_offset( bit_number ) ] &=
            ~ ( 1 << bit_offset( bit_number ) );
}

/*
**      Assign a value to a bit
*/
void
assign_bit( char bit_array[], unsigned bit_number, int value )
{
        if( value != 0 )
                set_bit( bit_array, bit_number );
        else
                clear_bit( bit_array, bit_number );
}

/*
**      Test a specific bit
*/
int
test_bit( char bit_array[], unsigned bit_number )
{
        return (
                bit_array[ character_offset( bit_number ) ]
                & 1 << bit_offset( bit_number )
            ) != 0;
}

/*
**      Compute the character that will contain the desired bit
*/
unsigned
character_offset( unsigned bit_number )
{
        return bit_number / CHAR_BIT;
}

/*
**      Compute the bit number within the desired character
*/
unsigned
bit_offset( unsigned bit_number )
{
        return bit_number % CHAR_BIT;
}
```

Solution 5.4b bitarray.c

5. This program makes use of the `limits.h` file to figure out how many bits there are in an integer. This could be hard coded if portability were not an issue.

```
/*
** Store a value in an arbitrary field in an integer.
*/
#include <limits.h>

#define INT_BITS        ( CHAR_BIT * sizeof( int ) )

int
store_bit_field( int original_value, int value_to_store, unsigned starting_bit,
    unsigned ending_bit )
{
        unsigned mask;

        /*
        ** Validate the bit parameters.  If an error is found, do nothing.  This
        ** is not great error handling.
        */
        if( starting_bit < INT_BITS && ending_bit < INT_BITS &&
            starting_bit >= ending_bit ){

                /*
                ** Construct the mask, which is unsigned to ensure that we get a
                ** logical, not arithmetic shift.
                */
                mask = (unsigned)-1;
                mask >>= INT_BITS - ( starting_bit - ending_bit + 1 );
                mask <<= ending_bit;

                /*
                ** Clear the field in the original value.
                */
                original_value &= ~mask;

                /*
                ** Shift the value to store to the right position
                */
                value_to_store <<= ending_bit;

                /*
                ** Mask excess bits off of the value, and store it.
                */
                original_value |= value_to_store & mask;

        }
        return original_value;
}
```

Solution 5.5 bitfield.c

6

Pointers

6.1 Questions

2. They are rarely used because you can't tell ahead of time where the compiler will put variables.
3. The value is an integer, so the compiler will not generate instructions to dereference it.
5. Even if `offset` has the same value as the literal in the next expression, it is more time consuming to evaluate the first expression because the multiplication to scale `offset` to the size of an integer must be done at run time. This is because the variable might contain any value, and the compiler has no way of knowing ahead of time what the value might actually be. On the other hand, the literal three can be scaled to an integer by multiplying it at compile time, and the result of that multiplication is simply added to `p` at run time. In other words, the second expression can be implemented by simply adding 12 to `p` (on a machine with four-byte integers); no runtime multiplication is needed.
7. Many of the expressions are not legal L-values.

	Integers		***Pointers to Integers***	
	R-value	***L-value addr***	***R-value***	***L-value addr***
a.	1008	1016	1008	1016
b.	1037	illegal	1040	illegal
c.	996	illegal	984	illegal
d.	12	illegal	3	illegal
e.	-24	illegal	-6	illegal
f.	1056	illegal	1056	illegal
g.	1036	illegal	1036	illegal
h.	1080	illegal	1080	illegal
i.	illegal	illegal	illegal	illegal
j.	illegal	illegal	1000	1064
k.	illegal	illegal	1045	illegal
l.	illegal	illegal	1012	1060
m.	illegal	illegal	1076	illegal
n.	illegal	illegal	1056	1076
o.	illegal	illegal	illegal	illegal
p.	illegal	illegal	52	illegal
q.	illegal	illegal	-80	illegal
r.	illegal	illegal	illegal	illegal
s.	0	illegal	0	illegal

	Integers		Pointers to Integers	
	R-value	L-value addr	R-value	L-value addr
t.	illegal	illegal	illegal	illegal
u.	illegal	illegal	1	illegal
v.	illegal	illegal	1080	1020
w.	1076	illegal	1076	illegal
x.	1077	illegal	1080	illegal
y.	illegal	illegal	1080	1072
z.	illegal	illegal	1080	illegal
aa.	illegal	illegal	1056	1076
bb.	illegal	illegal	1081	illegal
cc.	illegal	illegal	1072	1084
dd.	illegal	illegal	1021	illegal

6.2 Programming Exercises

1. The program requires two nested loops: The outer selects each character in the first string one by one, and the inner checks that character against each of the characters from the second string. The loops can also be nested the other way around.

```
/*
** Find the first occurrence in 'str' of any of the characters in 'chars' and
** return a pointer to that location.  If none are found, or if 'str' or 'chars'
** are NULL pointers, a NULL pointer is returned.
*/

#define NULL    0

char *
find_char( char const *str, char const *chars )
{
        char    *cp;

        /*
        ** Check arguments for NULL
        */
        if( str != NULL && chars != NULL ){

                /*
                ** Look at 'str' one character at a time.
                */
                for( ; *str != '\0'; str++ ){

                        /*
                        ** Look through 'chars' one at a time for a
                        ** match with *str.
                        */
                        for( cp = chars; *cp != '\0'; cp++ )
                                if( *str == *cp )
                                        return str;
```

Solution 6.1 *continued . . .*

```
                }
        }

        return NULL;
}
```

Solution 6.1 findchar.c

The inner loop could also have been written like this:

```
for( cp = chars; *cp != '\0'; )
        if( *str == *cp++ )
                return str;
```

2. This solution has two functions for clarity. Because it is only called once, the `match` function could also have been written in-line.

```
/*
** If the string "substr" appears in "str", delete it.
*/
#define NULL    0               /* null pointer */
#define NUL     '\0'            /* null byte */
#define TRUE    1
#define FALSE   0

/*
**      See if the substring beginning at 'str' matches the string 'want'.  If
**      so, return a pointer to the first character in 'str' after the match.
*/
char *
match( char *str, char *want )
{
        /*
        ** Keep looking while there are more characters in 'want'.  We fall out
        ** of the loop if we get a match.
        */
        while( *want != NUL )
                if( *str++ != *want++ )
                        return NULL;

        return str;
}

int
del_substr( char *str, char const *substr )
{
        char    *next;

        /*
        ** Look through the string for the first occurrence of the substring.
        */
```

Solution 6.2 *continued . . .*

```
        while( *str != NUL ){
                next = match( str, substr );
                if( next != NULL )
                        break;
                str++;
        }

        /*
        ** If we reached the end of the string, then the substring was not
        ** found.
        */
        if( *str == NUL )
                return FALSE;

        /*
        ** Delete the substring by copying the bytes after it over the bytes of
        ** the substring itself.
        */
        while( *str++ = *next++ )
                ;
        return TRUE;
}
```

Solution 6.2 delsubst.c

4. This solution uses the hint to eliminate all the even numbers from the array.

```
/*
** Sieve of Eratosthenes -- compute prime numbers using an array.
*/

#include <stdlib.h>

#define SIZE    1000

#define TRUE    1
#define FALSE   0

int
main()
{
        char    sieve[ SIZE ];  /* the sieve */
        char    *sp;            /* pointer to access the sieve */
        int     number;         /* number we're computing */

        /*
        ** Set the entire sieve to TRUE.
        */
        for( sp = sieve; sp < &sieve[ SIZE ]; )
                *sp++ = TRUE;

        /*
```

Solution 6.4 *continued . . .*

```
        ** Process each number from 3 to as many as the sieve holds.  (Note: the
        ** loop is terminated from inside.)
        */
        for( number = 3; ; number += 2 ){
                /*
                ** Set the pointer to the proper element in the sieve, and stop
                ** the loop if we've gone too far.
                */
                sp = &sieve[ 0 ] + ( number – 3 ) / 2;
                if( sp >= &sieve[ SIZE ] )
                        break;

                /*
                ** Now advance the pointer by multiples of the number and set
                ** each subsequent entry FALSE.
                */
                while( sp += number, sp < &sieve[ SIZE ] )
                        *sp = FALSE;
        }

        /*
        ** Go through the entire sieve now and print the numbers corresponding
        ** to the locations that remain TRUE.
        */
        printf( "2\n" );
        for( number = 3, sp = &sieve[ 0 ];
            sp < &sieve[ SIZE ];
            number += 2, sp++ ){
                if( *sp )
                        printf( "%d\n", number );
        }

        return EXIT_SUCCESS;
}
```

Solution 6.4 sieve.c

5. This program also computes only the odd numbers, so the bit array is half the size that it would otherwise be. Note the use of `CHAR_BIT` in the declaration of `SIZE` to get the number of bits per character. The addition of one in this declaration ensures that enough characters appear in the array even if the number of bits required is not evenly divisible by `CHAR_BIT`. The initial loop to set the entire array TRUE could have been done bit by bit, but it is faster to do it byte by byte. It would be faster still if the array were declared as integers, however when you called the bit array functions you would have to cast the argument to a character pointer.

```
/*
** Sieve of Eratosthenes –– compute prime numbers using a bit array.
*/

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

Solution 6.5 *continued . . .*

```
#include <limits.h>
#include "bitarray.h"

/*
**      An optimization that was not described in the problem statement: 2 is
**      the only even number that is prime, so to save space and time the bit
**      array only represents the odd values.
*/

/*
**      MAX_VALUE is the largest number in our "list".
**
**      MAX_BIT_NUMBER is the bit number corresponding to MAX_VALUE, considering
**      that we only keep bits to represent the odd numbers starting with 3.
**
**      SIZE is the number of characters needed to get enough bits.
*/
#define MAX_VALUE       10000
#define MAX_BIT_NUMBER  ( ( MAX_VALUE – 3 ) / 2 )
#define SIZE            ( MAX_BIT_NUMBER / CHAR_BIT + 1 )

int
main()
{
        char    sieve[ SIZE ];  /* the sieve */
        int     number;         /* number we’re computing */
        int     bit_number;     /* corresponding bit in the sieve */
        char    *sp;            /* for initializing the array */

        /*
        ** Set the entire sieve to TRUE.
        */
        for( sp = sieve; sp < &sieve[ SIZE ]; )
                *sp++ = 0xff;

        /*
        ** Process each number from 3 to as many as the sieve holds.
        */
        for( number = 3; number <= MAX_VALUE; number += 2 ){
                /*
                ** Compute bit number corresponding to this number.
                */
                bit_number = ( number – 3 ) / 2;

                /*
                ** An optimization that was not described in the problem
                ** statement: If the bit for this value was already cleared by
                ** an earlier number, then skip it –– all of its multiples will
                ** have been cleared, too.
                */
                if( !test_bit( sieve, bit_number ) )
                        continue;
```

Solution 6.5 *continued . . .*

```
                /*
                ** Now advance the pointer by multiples of the number and set
                ** each subsequent entry FALSE.  Note that we advance by
                ** "number" rather than "number / 2" in order to skip the even
                ** numbers that aren't represented in the bit array.
                */
                while( ( bit_number += number ) <= MAX_BIT_NUMBER )
                        clear_bit( sieve, bit_number );
        }

        /*
        ** Go through the entire sieve now, and print the numbers corresponding
        ** to the locations that remain TRUE.
        */
        printf( "2\n" );
        for( bit_number = 0, number = 3;
            number <= MAX_VALUE;
            bit_number += 1, number += 2 ){
                if( test_bit( sieve, bit_number ) )
                        printf( "%d\n", number );
        }

        return EXIT_SUCCESS;
}
```

Solution 6.5 sieve2.c

6. The changes needed to the program are minor: Instead of printing the primes, they are counted.

```
        /*
        ** Go through the entire sieve now and count how many primes there are
        ** per thousand numbers.
        */
        n_primes = 1;
        limit = 1000;

        for( bit_number = 0, number = 3;
            number <= MAX_VALUE;
            bit_number += 1, number += 2 ){
                if( number > limit ){
                        printf( "%d–%d: %d\n", limit – 1000, limit, n_primes );
                        n_primes = 0;
                        limit += 1000;
                }
                if( test_bit( sieve, bit_number ) )
                        n_primes += 1;
        }
        printf( "%d–%d: %d\n", limit – 1000, limit, n_primes );
```

Solution 6.6 s3_frag.c

Counting the primes less than 1,000,000 yields these results:

Range of Numbers	*Average # of Primes per Thousand Numbers*
0–100,000	95.92
100,000–200,000	83.92
200,000–300,000	80.13
300,000–400,000	78.63
400,000–500,000	76.78
500,000–600,000	75.6
600,000–700,000	74.45
700,000–800,000	74.08
800,000–900,000	73.23
900,000–1,000,000	72.24

The number of primes per thousand numbers is decreasing, but slower and slower. This trend continues to at least one billion:

Range of Numbers	*Average # of Primes per Thousand Numbers*
1,000,000–2,000,000	70.435
2,000,000–3,000,000	67.883
3,000,000–4,000,000	66.33
4,000,000–5,000,000	65.367
5,000,000–6,000,000	64.336
6,000,000–7,000,000	63.799
7,000,000–8,000,000	63.129
8,000,000–9,000,000	62.712
9,000,000–10,000,000	62.09
10,000,000–20,000,000	60.603
20,000,000–30,000,000	58.725
30,000,000–40,000,000	57.579
40,000,000–50,000,000	56.748
50,000,000–60,000,000	56.098
60,000,000–70,000,000	55.595
70,000,000–80,000,000	55.132
80,000,000–90,000,000	54.757
90,000,000–100,000,000	54.45
100,000,000–200,000,000	53.175
200,000,000–300,000,000	51.734
300,000,000–400,000,000	50.84
400,000,000–500,000,000	50.195
500,000,000–600,000,000	49.688
600,000,000–700,000,000	49.292
700,000,000–800,000,000	48.932
800,000,000–900,000,000	48.63
900,000,000–1,000,000,000	48.383

To compute prime numbers this large requires a bit array of 62.5 megabytes. Unless your system has this much memory, the performance of the program deteriorates as the operating system thrashes pages in and out. The speed can be improved by running the outer loop over and over, each time modifying only as much of the array as will fit in physical memory.

7

Functions

7.1 Questions

2. An advantage is that it allows you to be lazy; there is less code to write. The other consequences, such as being able to call functions with the wrong numbers or types of arguments, are all disadvantages.
3. The value is converted to the type specified by the function. The Standard indicates that this is done the same as if the value had been assigned to a variable of that type.
4. This is not allowed; the compiler should give an error message.
5. The value returned is interpreted as if it were an integer.
6. The argument values are interpreted as the types of the formal parameters, not their real types.
9. a. It is easier to use a `#include` in several source files than to copy the prototype.

   b. There is only one copy of the prototype itself.

   c. `#include`ing the prototype in the file that defines the function ensures that they match.
10. The progression is indeed related to the Fibonacci numbers: each count is the sum of the two preceding counts plus one. Here are the values requested, plus some additional counts to show how bad the recursive function really is.

Fibonacci(n)	*Number of Calls*
1	1
2	1
3	3
4	5
5	9
6	15
7	25
8	41
9	67
10	109
11	177
15	1,219
20	13,529
25	150,049

Fibonacci(n)	*Number of Calls*
30	1,664,079
40	204,668,309
50	25,172,538,049
75	4,222,970,155,956,099
100	708,449,696,358,523,830,149

7.2 Programming Exercises

2. Another simple recursion exercise.

```
/*
** Return the greatest common divisor of the arguments m and n (recursively).
*/

int
gcd( int m, int n )
{
        int     r;

        if( m <= 0 || n <= 0 )
                return 0;

        r = m % n;
        return r > 0 ? gcd( n , r ) : n;
}
```

Solution 7.2a gcd1.c

However, this is tail recursion, so here is an iterative solution:

```
/*
** Return the greatest common divisor of the arguments m and n (iteratively).
*/

int
gcd( int m, int n )
{
        int     r;

        if( m <= 0 || n <= 0 )
                return 0;

        do {
                r = m % n;
                m = n;
                n = r;
        } while( r > 0 );
```

Solution 7.2b *continued . . .*

```
        return m;
}
```

Solution 7.2b gcd2.c

4. This problem is interesting because the argument list terminates itself; the problem does not require any named arguments, but the `stdarg` macros do. Watch for the limiting case of no arguments at all.

```
/*
** Return the largest value from the argument list.  The list is terminated by a
** negative value.
*/

#include <stdarg.h>

int
max_list( int first_arg, ... )
{
        va_list var_arg;
        int     max = 0;

        /*
        ** Get the first arg if there is one and save it as the max.
        */
        if( first_arg >= 0 ){
                int     this_arg;

                max = first_arg;

                /*
                ** Get the remaining arguments and save each one if it is
                ** greater than the current max.
                */
                va_start( var_arg, first_arg );
                while( ( this_arg = va_arg( var_arg, int ) ) >= 0 )
                        if( this_arg > max )
                                max = this_arg;

                va_end( var_arg );
        }
        return max;
}
```

Solution 7.4 max.c

5. This is ambitious even in its stripped down form, primarily because of the concept of parsing the format string. This solution chooses to ignore undefined format codes. The test in the `switch` statement takes care of format strings that end with a `%`.

```
/*
** Bare-bones printf function: handles the %d, %f, %s, and %c format codes.
*/

#include <stdarg.h>

void
printf( char *format, ... )
{
        va_list arg;
        char    ch;
        char    *str;

        va_start( arg, format );

        /*
        ** Get the format characters one by one.
        */
        while( ( ch = *format++ ) != '\0' ){
                if( ch != '%' ){
                        /*
                        ** Not a format code -- print the character verbatim.
                        */
                        putchar( ch );
                        continue;
                }

                /*
                ** We got a % -- now get the format code and use it to format
                ** the next argument.
                */
                switch( *format != '\0' ? *format++ : '\0' ){
                case 'd':
                        print_integer( va_arg( arg, int ) );
                        break;

                case 'f':
                        print_float( va_arg( arg, float ) );
                        break;

                case 'c':
                        putchar( va_arg( arg, int ) );
                        break;

                case 's':
                        str = va_arg( arg, char * );
                        while( *str != '\0' )
                                putchar( *str++ );
                        break;
                }
        }
}
```

Solution 7.5 printf.c

6. Having the flexibility to choose how to print values that have more than one legal output is convenient! This solution uses a recursive helper function to do the work. The magnitude table is for a 32-bit machine and will have to be expanded for machines with larger integers. It is not unreasonable to require the caller to make the buffer large enough, as the maximum possible length of the output is easily calculated.

```
/*
** Convert a numeric value to words.
*/

static  char    *digits[] = {
        "", "ONE ", "TWO ", "THREE ", "FOUR ", "FIVE ", "SIX ", "SEVEN ",
        "EIGHT ", "NINE ", "TEN ", "ELEVEN ", "TWELVE ", "THIRTEEN ",
        "FOURTEEN ", "FIFTEEN ", "SIXTEEN ", "SEVENTEEN ", "EIGHTEEN ",
        "NINETEEN "
};

static  char    *tens[] = {
        "", "", "TWENTY ", "THIRTY ", "FORTY ", "FIFTY ", "SIXTY ", "SEVENTY ",
        "EIGHTY ", "NINETY "
};

static  char    *magnitudes[] = {
        "", "THOUSAND ", "MILLION ", "BILLION "
};

/*
**      Convert the last 3-digit group of amount to words.  Amount is the value
**      to be converted, buffer is where to put the words, and magnitude is the
**      name of the 3-digit group we're working on.
*/

static void
do_one_group( unsigned int amount, char *buffer, char **magnitude )
{
        int     value;

        /*
        ** Get all the digits beyond the last three.  If we have any value
        ** there, process those digits first.  Note that they are in the next
        ** magnitude.
        */
        value = amount / 1000;
        if( value > 0 )
                do_one_group( value, buffer, magnitude + 1 );

        /*
        ** Now process this group of digits.  Any hundreds?
        */
        amount %= 1000;
        value = amount / 100;
        if( value > 0 ){
```

Solution 7.6 *continued . . .*

```
                strcat( buffer, digits[ value ] );
                strcat( buffer, "HUNDRED " );
        }

        /*
        ** Now do the rest of the value.  If less than 20, treat it as a single
        ** digit to get the teens names.
        */
        value = amount % 100;
        if( value >= 20 ){
                /*
                ** Greater than 20.  Do a tens name and leave the units to be
                ** printed next.
                */
                strcat( buffer, tens[ value / 10 ] );
                value %= 10;
        }
        if( value > 0 )
                strcat( buffer, digits[ value ] );

        /*
        ** If we had any value in this group at all, print the magnitude.
        */
        if( amount > 0 )
                strcat( buffer, *magnitude );
}

void
written_amount( unsigned int amount, char *buffer )
{
        if( amount == 0 )
                /*
                ** Special case for zero.
                */
                strcpy( buffer, "ZERO " );
        else {
                /*
                ** Store an empty string in the buffer, then begin.
                */
                *buffer = '\0';
                do_one_group( amount, buffer, magnitudes );
        }
}
```

Solution 7.6 written.c

8

Arrays

8.1 Questions

2. No, the second one is the same as `array[ i ] + j` due to the precedence of the operators.
3. The assignment is illegal, as the pointer it attempts to compute is off the left end of the array; the technique should be avoided for this reason. Nevertheless, it will work on most machines.
4.
```
char    buffer[SIZE];
char    *front, *rear;
...
front = buffer;
rear = buffer + strlen( buffer ) – 1;
while( front < rear ){
        if( *front != *rear )
                break;
        front++;
        rear––;
}
if( front >= rear ){
        printf( "It is a palindrome!\n" );
}
```

   This question borders on entrapment! If you try and get fancy like this:

```
if( *front++ != *rear–– )
        break;
```

   the program can fail because the test after the loop ends is no longer valid. This is a good example of the discussion surrounding the statement “experienced C programmers will have little trouble with the pointer loop...”
6. This depends on the machine. Borland C++ for the 80x86 family produces results similar to the 68000 code shown in the text. Some RISC machines have been shown to produce better code with `try1` than with `try5`; this is because RISC architectures do not implement the fancy addressing modes that the `*x++` expression exploits.

7. This depends entirely on the particular machine and compiler being used.

9. ```
   int     coin_values[] = { 1, 5, 10, 25, 50, 100 };
   ```

10. With two-byte elements, each of the four rows occupies four bytes.

Expression	*Value*
`array`	1000
`array + 2`	1008
`array[3]`	1012
`array[2] – 1`	1006
`&array[1][2]`	1008
`&array[2][0]`	1008

11. This exercise is tedious, but not really difficult.

Expression	*Value*	*Type of* `x`
`array`	1000	`int (*x)[2][3][6];`
`array + 2`	1288	`int (*x)[2][3][6];`
`array[3]`	1432	`int (*x)[3][6];`
`array[2] – 1`	1216	`int (*x)[3][6];`
`array[2][1]`	1360	`int (*x)[6];`
`array[1][0] + 1`	1168	`int (*x)[6];`
`array[1][0][2]`	1192	`int *x;`
`array[0][1][0] + 2`	1080	`int *x;`
`array[3][1][2][5]`	can't tell	`int x;`
`&array[3][1][2][5]`	1572	`int *x;`

13.

Expression	*Subscript expression*
`*array`	`array[0]`
`*( array + 2 )`	`array[2]`
`*( array + 1 ) + 4`	`array[1] + 4`
`*( *( array + 1 ) + 4 )`	`array[1][4]`
`*( *( *( array + 3 ) + 1 ) + 2 )`	`array[3][1][2]`
`*( *( *array + 1 ) + 2 )`	`array[0][1][2]`
`*( **array + 2 )`	`array[0][0][2]`
`**( *array + 1 )`	`array[0][1][0]`
`***array`	`array[0][0][0]`

14. If `i` were declared as a pointer to an integer, there is no error.

15. The second makes more sense. If `which` is out of range, using it as a subscript could crash the program.

16. There are several differences. Being an argument, `array1` is actually a pointer variable; it points to the array passed as the actual argument, and its value can be changed by the function. No space for this array is allocated in this function, and there is no guarantee that the argument actually passed has ten elements.

    On the other hand, `array2` is a pointer constant, so its value cannot be changed. It points to the space allocated in this function for ten integers.

18. There are two ways:

    ```
    void function( int array[][2][5] );
    ```
    ```

```
void function( int (*array)[2][5] );
```

The second and third sizes cannot be omitted or the compiler will have no idea how large each of those dimensions is. The first size can be omitted because the subscript calculation does not depend on it.

Technically, there are millions of additional ways—just give arbitrary integers for the first size. The value is ignored, so they all have the same effect. This isn't very useful, though.

19. Simply append an empty string to it. The end of the table can then be checked like this:

```
for( kwp = keyword_table; **kwp != '\0'; kwp++ )
```

8.2 Programming Exercises

1. The key to this is that the initialization be done statically, not with assignment statements. This means that the array must be in static memory, even though the problem did not specifically state that.

   Also, the problem purposely avoids specifying any locations that have a zero for any subscript. This simply tests whether the student remembers that subscripts begin at zero.

   The solution below exploits incomplete initialization to avoid having to enter each value explicitly.

```
unsigned char   char_values[3][6][4][5] = {
        { /* 0 */
                { /* 0,0 */
                        { 0 },
                }
        },
        { /* 1 */
                { /* 1,0 */
                        { 0 }
                },
                { /* 1,1 */
                        { 0 },
                        { 0, ' ' }
                },
                { /* 1,2 */
                        { 0 },
                        { 0 },
                        { 0, 0, 0, 'A' },
                        { 0, 0, 0, 0, 'X' }
                },
                { /* 1,3 */
                        { 0 },
                        { 0 },
                        { 0, 0, 0363 }
                },
                { /* 1,4 */
                        { 0 },
```

Solution 8.1 *continued . . .*

```
                        { 0 },
                        { 0, 0, 0, '\n' }
                }
        },
        { /* 2 */
                { /* 2,0 */
                        { 0 }
                },
                { /* 2,1 */
                        { 0 },
                        { 0, 0, 0320 }
                },
                { /* 2,2 */
                        { 0 },
                        { 0, '0' },
                        { 0, 0, '\'' },
                        { 0, '\121' }
                },
                { /* 2,3 */
                        { 0 }
                },
                { /* 2,4 */
                        { 0 },
                        { 0 },
                        { 0 },
                        { 0, 0, '3', 3 }
                },
                { /* 2,X */
                        { 0 },
                        { 0 },
                        { 0 },
                        { 0, 0, 0, 0, '}' }
                }
        },
};
```

Solution 8.1 arr_init.c

3. This problem is simple because the argument must be a specific size. The test for zero or one is simple, though perhaps not immediately obvious.

```
/*
** Test a 10 by 10 matrix to see if it is an identity matrix.
*/

#define FALSE   0
#define TRUE    1

int
identity_matrix( int matrix[10][10] )
{
        int     row;
```

Solution 8.3 *continued . . .*

```
        int     column;

        /*
        ** Go through each of the matrix elements.
        */
        for( row = 0; row < 10; row += 1 ){
                for( column = 0; column < 10; column += 1 ){
                        /*
                        ** If the row number is equal to the column number, the
                        ** value should be 1, else 0.
                        */
                        if( matrix[ row ][ column ] != ( row == column ) )
                                return FALSE;
                }
        }

        return TRUE;
}
```

Solution 8.3 identty1.c

4. The storage order of the array elements allows a pointer to an integer to be walked sequentially through the elements. Separate counters keep track of the row and column, using the size argument provided. Note that there is no way to check whether the matrix is actually the size indicated by the second argument.

```
/*
** Test a square matrix to see if it is an identity matrix.
*/

#define FALSE   0
#define TRUE    1

int
identity_matrix( int *matrix, int size )
{
        int     row;
        int     column;

        /*
        ** Go through each of the matrix elements.
        */
        for( row = 0; row < size; row += 1 ){
                for( column = 0; column < size; column += 1 ){
                        /*
                        ** If the row number is equal to the column number, the
                        ** value should be 1, else 0.
                        */
                        if( *matrix++ != ( row == column ) )
                                return FALSE;
                }
```

Solution 8.4 *continued . . .*

```
        }

        return TRUE;
}
```

Solution 8.4 identty2.c

6. If you have some method for the students to submit their programs electronically, giving bonus points for achieving a program that is smaller than some threshold gives students strong motivation to write compact code.

```
/*
** Compute an array offset from a set of subscripts and dimension information.
*/

#include <stdarg.h>
#define reg     register

int
array_offset( reg int *arrayinfo, ... )
{
        reg     int     ndim;
        reg     int     offset;
        reg     int     hi, lo;
        reg     int     i;
        int     s[10];
        va_list subscripts;

        /*
        ** Check the number of dimensions.
        */
        ndim = *arrayinfo++;
        if( ndim >= 1 && ndim <= 10 ){

                /*
                ** Copy the subscript values to an array.
                */
                va_start( subscripts, arrayinfo );
                for( i = 0; i < ndim; i += 1 )
                        s[i] = va_arg( subscripts, int );
                va_end( subscripts );

                /*
                ** Compute the offset one dimension at a time.
                */
                offset = 0;
                for( i = 0; ndim; ndim--, i++ ){
                        /*
                        ** Get the limits for the next subscript.
                        */
                        lo = *arrayinfo++;
                        hi = *arrayinfo++;
```

Solution 8.6 *continued . . .*

```
                        /*
                        ** Note that it is not necessary to test for hi < lo
                        ** because if this is true, then at least one of the
                        ** tests below will fail.
                        */
                        if( s[i] < lo || s[i] > hi )
                                return –1;

                        /*
                        ** Compute the offset.
                        */
                        offset *= hi – lo + 1;
                        offset += s[i] – lo;
                }
                return offset;
        }
        return –1;
}
```

Solution 8.6 subscrp1.c

7. This function is very similar to the previous one except that the subscripts must be processed from right to left.

```
/*
** Compute an array offset from a set of subscripts and dimension information.
*/

#include <stdarg.h>
#define reg     register

int
array_offset2( reg int *arrayinfo, ... )
{
        reg     int     ndim;
        reg     int     hi;
        reg     int     offset;
        reg     int     lo;
        reg     int     *sp;
        int     s[10];
        va_list subscripts;

        /*
        ** Check number of dimensions
        */
        ndim = *arrayinfo++;
        if( ndim >= 1 && ndim <= 10 ){

                /*
                ** Copy subscripts to array
                */
```

Solution 8.7 *continued . . .*

```
                va_start( subscripts, arrayinfo );
                for( offset = 0; offset < ndim; offset += 1 )
                        s[offset] = va_arg( subscripts, int );
                va_end( subscripts );

                /*
                ** Compute offset, starting with last subscript and working back
                ** towards the first.
                */
                offset = 0;
                arrayinfo += ndim * 2;
                sp = s + ndim;
                while( ndim-- >= 1 ){
                        /*
                        ** Get the limits for the next subscript.
                        */
                        hi = *--arrayinfo;
                        lo = *--arrayinfo;

                        /*
                        ** Note that it is not necessary to test for hi < lo
                        ** because if this is true, then at least one of the
                        ** tests below will fail.
                        */
                        if( *--sp > hi || *sp < lo ){
                                return -1;
                        }

                        /*
                        ** Compute the offset.
                        */
                        offset *= hi - lo + 1;
                        offset += *sp - lo;
                }
                return offset;
        }
        return -1;
}
```

Solution 8.7 subscrp2.c

8. There are 92 valid solutions. The key to making the program simple is to separate the functionality into separate functions. The search for conflicts is limited to only those rows with queens in them, as described in the comments. Doing this eliminates roughly 3/8 of the work.

```
/*
** Solve the Eight Queens Problem.
*/

#include <stdlib.h>

#define TRUE    1
```

Solution 8.8 *continued . . .*

```
#define FALSE   0

/*
**      The chessboard.  If an element is TRUE, there is a queen on that square;
**      if FALSE, no queen.
*/
int     board[8][8];

/*
** print_board
**
**      Print out a valid solution.
*/
void
print_board()
{
        int     row;
        int     column;
        static  int     n_solutions;

        n_solutions += 1;
        printf( "Solution #%d:\n", n_solutions );

        for( row = 0; row < 8; row += 1 ){
                for( column = 0; column < 8; column += 1 ){
                        if( board[ row ][ column ] )
                                printf( " Q" );
                        else
                                printf( " +" );
                }
                putchar( '\n' );
        }
        putchar( '\n' );
}

/*
** conflicts
**
**      Check the board for conflicts with the queen that was just placed.
**      NOTE: because the queens are placed in the rows in order, there is no
**      need to look at rows below the current one as there are no queens there!
*/
int
conflicts( int row, int column )
{
        int     i;

        for( i = 1; i < 8; i += 1 ){
                /*
                ** Check up, left, and right.  (Don't have to check down; no
                ** queens in those rows yet!)
                */
```

Solution 8.8

continued . . .

```
                if( row – i >= 0 && board[ row – i ][ column ] )
                        return TRUE;
                if( column – i >= 0 && board[ row ][ column – i ] )
                        return TRUE;
                if( column + i < 8 && board[ row ][ column + i ] )
                        return TRUE;

                /*
                ** Check the diagonals: up and left, up and right.  (Don't have
                ** to check down; no queens there yet!)
                */
                if( row – i >= 0 && column – i >= 0
                    && board[ row – i ][ column – i ] )
                        return TRUE;
                if( row – i >= 0 && column + i < 8
                    && board[ row – i ][ column + i ] )
                        return TRUE;
        }

        /*
        ** If we get this far, there were no conflicts!
        */
        return FALSE;
}

/*
** place_queen
**
**      Try to place a queen in each column of the given row.
*/
void
place_queen( int row )
{
        int     column;

        /*
        ** Try each column, one by one.
        */
        for( column = 0; column < 8; column += 1 ){
                board[ row ][ column ] = TRUE;

                /*
                ** See if this queen can attack any of the others (don't need to
                ** check this for the first queen!).
                */
                if( row == 0 || !conflicts( row, column ) )
                        /*
                        ** No conflicts -- if we're not yet done, place the next
                        ** queen recursively.  If done, print the solution!
                        */
                        if( row < 7 )
                                place_queen( row + 1 );
```

Solution 8.8 *continued . . .*

```
                        else
                                print_board();

                /*
                ** Remove the queen from this position.
                */
                board[ row ][ column ] = FALSE;
        }

}

int
main()
{
        place_queen( 0 );
        return EXIT_SUCCESS;
}
```

Solution 8.8 8queens.c

9

Strings, Characters, and Bytes

9.1 Questions

2. It is more appropriate because the length of a string simply cannot be negative. Also, using an unsigned value allows longer string lengths (which would be negative in a signed quantity) to be represented. It is less appropriate because arithmetic involving unsigned expressions can yield unexpected results. The "advantage" of being able to report the length of longer strings is only rarely of value: On machines with 16 bit integers, it is needed only for strings exceeding 32,767 characters in length. On machines with 32 bit integers, it is needed only for strings exceeding 2,147,483,647 bytes in length (which is rare indeed).
3. Yes, then subsequent concatenations could be done more efficiently because the work of finding the end of the string would not need to be repeated.
5. Only if the last character in the array is already NUL. A string must be terminated with a NUL byte, and `strncpy` does not guarantee that this will occur. However, the statement does not let `strncpy` change the last position in the array, so if that contains a NUL byte (either through an assignment or by the default initialization of static variables), then the result will be a string.
6. First, the former will work regardless of the character set in use. The latter will work with the ASCII character set but will fail with the EBCDIC character set. Second, the former will work properly whether or not the locale has been changed; the latter may not.
7. The main thing is to eliminate the test for `islower`: this is unnecessary because `toupper` includes such a test already. After that, the loop can be made more efficient (but not simpler!) by saving a copy of the character being processed, like this:

```
register int    ch;
...
for( pstring = message; ( ch = *pstring ) != '\0'; ){
        *pstring++ = toupper( ch );
}
```

9.2 Programming Exercises

1. This is quite a simple program, but contains a lot of similar sequences of code. A related problem in Chapter 13 addresses this problem.

```
/*
** Compute the percentage of characters read from the standard input that are in
** each of several character categories.
*/
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

int     n_cntrl;
int     n_space;
int     n_digit;
int     n_lower;
int     n_upper;
int     n_punct;
int     n_nprint;
int     total;

int
main()
{
        int     ch;
        int     category;

        /*
        ** Read and process each character
        */
        while( (ch = getchar()) != EOF ){
                total += 1;

                /*
                ** Call each of the test functions with this char; if true,
                ** increment the associated counter.
                */
                if( iscntrl( ch ) )
                        n_cntrl += 1;
                if( isspace( ch ) )
                        n_space += 1;
                if( isdigit( ch ) )
                        n_digit += 1;
                if( islower( ch ) )
                        n_lower += 1;
                if( isupper( ch ) )
                        n_upper += 1;
                if( ispunct( ch ) )
                        n_punct += 1;
                if( !isprint( ch ) )
                        n_nprint += 1;
```

Solution 9.1 *continued . . .*

```
        }

        /*
        ** Print the results
        */
        if( total == 0 )
                printf( "No characters in the input!\n" );
        else {
                printf( "%3.0f%% %s control characters\n",
                    n_cntrl * 100.0 / total );
                printf( "%3.0f%% %s whitespace characters\n",
                    n_space * 100.0 / total );
                printf( "%3.0f%% %s digit characters\n",
                    n_digit * 100.0 / total );
                printf( "%3.0f%% %s lower case characters\n",
                    n_lower * 100.0 / total );
                printf( "%3.0f%% %s upper case characters\n",
                    n_upper * 100.0 / total );
                printf( "%3.0f%% %s punctuation characters\n",
                    n_punct * 100.0 / total );
                printf( "%3.0f%% %s non-printable characters\n",
                    n_nprint * 100.0 / total );
                }

        return EXIT_SUCCESS;
}
```

Solution 9.1 char_cat.c

3. In order to do what this question asks, the size of the destination array must be known. The problem did not specify this explicitly, but it cannot be avoided. Also be sure that the function returns its first argument, which is required by the "similar to `strcpy`" specification.

```
/*
** Safe string copy.
*/

#include <string.h>

char *
my_strcpy( char *dst, char const *src, int size )
{
        strncpy( dst, src, size );
        dst[ size - 1 ] = '\0';

        return dst;
}
```

Solution 9.3 mstrcpy.c

4. Again, the size of the destination array must be known, and the first argument must be returned. Note the danger in trying to determine the length of the existing string in the first argument: if it is not terminated with a NUL byte, then `strlen` will run past the end of the array looking for

one. The logic later in the function handles this case properly, but there is still the possibility that a runaway `strlen` will crash the program. For this reason, this solution uses the `my_strlen` function from question one.

Notice also that the value returned by `my_strlen` is being cast to an integer; this prevents the result of the subtraction from being promoted to unsigned.

```
/*
** Safe string concatenation.
*/

#include <string.h>
#include "string_len.h"

char *
my_strcat( char *dst, char const *src, int size )
{
        int     length;

        size -= 1;
        length = size - (int)my_strlen( dst, size );
        if( length > 0 ){
                strncat( dst, src, length );
                dst[ size ] = '\0';
        }

        return dst;
}
```

Solution 9.4 mstrcat.c

5. This function calls `strncat` to do the work after adjusting the length parameter.

```
/*
** Append a string to the end of an existing one, ensuring that the destination
** buffer does not overflow.
*/

void
my_strncat( char *dest, char *src, int dest_len )
{
        register int    len;

        /*
        ** Get length of existing string in destination buffer; deduct this
        ** length from dest_len.  The "+1" accounts for the terminating NUL byte
        ** that is appended.
        */
        len = strlen( dest );
        dest_len -= len + 1;

        /*
```

Solution 9.5 *continued . . .*

```
        ** If there is any room left, call strncpy to do the work.
        */
        if( dest_len > 0 )
                strncat( dest + len, src, dest_len );
}
```

Solution 9.5 mstrncat.c

7. This function can be implemented with either of the two approaches described in the answer to Programming Exercise 6. Only the first approach, calling the library string functions to do some of the work, is shown here.

```
/*
** Find the last occurrence of a character in a string and return a pointer to
** it.
*/

#include <string.h>
#include <stdio.h>

char *
my_strrchr( char const *str, int ch )
{
        char    *prev_answer = NULL;

        for( ; ( str = strchr( str, ch ) ) != NULL; str += 1 )
                prev_answer = str;

        return prev_answer;
}
```

Solution 9.7 mstrrchr.c

8. This function can also be implemented with either of the two approaches described above. Only the first approach is shown here.

```
/*
** Find the specified occurrence of a character in a string and return a
** pointer to it.
*/

#include <string.h>
#include <stdio.h>

char *
my_strnchr( char const *str, int ch, int which )
{
        char    *answer = NULL;

        while( --which >= 0 && ( answer = strchr( str, ch ) ) != NULL )
                str = answer + 1;
```

Solution 9.8 *continued . . .*

```
        return answer;
}
```

Solution 9.8 mstrnchr.c

9. There are no routines in the string library that perform this task directly, so it must be written from scratch. The string library does offer some help, though.

```
/*
** Count the number of characters in the first argument that also appear in the
** second argument.
*/

int
count_chars( char const *str, char const *chars )
{
        int     count = 0;

        while( ( str = strpbrk( str, chars ) ) != NULL ){
                count += 1;
                str++;
        }

        return count;
}
```

Solution 9.9 count_ch.c

10. The fact that the string can begin or end with white space forces the loops that skip it to come first. This will not occur to everyone.

```
/*
** Determine whether or not a string is a palindrome.  Nonalphabetic characters
** are ignored, and the comparison is not case sensitive.
*/

#include <ctype.h>

#define TRUE    1
#define FALSE   0

int
palindrome( char *string )
{
        char    *string_end;

        string_end = string + strlen( string ) - 1;

        while( TRUE ){
                /*
```

Solution 9.10 *continued . . .*

```
                ** Advance the beginning pointer to skip any nonletters.
                ** Retreat the ending pointer likewise.
                */
                while( !isalpha( *string ) )
                        string++;

                while( !isalpha( *string_end ) )
                        string_end--;

                /*
                ** If the pointers have passed each other, we're done, and it is
                ** a palindrome.
                */
                if( string_end <= string )
                        return TRUE;

                /*
                ** Otherwise, compare the characters to see if they match.
                ** Converting them both to lower case before comparing makes it
                ** case insensitive.
                */
                if( tolower( *string ) != tolower( *string_end ) )
                        return FALSE;

                /*
                ** These characters done -- move to the next ones.
                */
                string++;
                string_end--;
        }
}
```

Solution 9.10 palindrm.c

12. This solution makes extensive use of library functions. The use of the string functions requires that the array always be a string, which is why it is NUL-terminated after each character is appended in the final loop.

```
/*
** Convert a key word to the scrambled alphabet used with encrypt and decrypt.
*/

#include <ctype.h>
#include <string.h>

#define TRUE    1
#define FALSE   0

int
prepare_key( char *key )
{
        register char   *keyp;
```

Solution 9.12 *continued . . .*

```
        register char   *dup;
        register int    character;

        /*
        ** Make sure the key is not empty.
        */
        if( *key == '\0' )
                return FALSE;

        /*
        ** First, convert the word to upper case (lower case would be just as
        ** good, as long as we're consistent).
        */
        for( keyp = key; ( character = *keyp ) != '\0'; keyp++ ){
                if( !islower( character ) ){
                        if( !isupper( character ) )
                                return FALSE;
                        *keyp = tolower( character );
                }
        }

        /*
        ** Now eliminate all duplicate characters from the word.
        */
        for( keyp = key; ( character = *keyp ) != '\0'; ){
                dup = ++keyp;
                while( ( dup = strchr( dup, character ) ) != NULL )
                        strcpy( dup, dup + 1 );
        }

        /*
        ** Now add the remaining letters of the alphabet to the key.  This makes
        ** use of the fact that the loop above leaves keyp pointing at the
        ** terminating NULL byte.
        */
        for( character = 'a'; character <= 'z'; character += 1 ){
                if( strchr( key, character ) == NULL ){
                        *keyp++ = character;
                        *keyp = '\0';
                }
        }

        return TRUE;
}
```

Solution 9.12 prep_key.c

13. This solution assumes that the codes for alphabetic characters are consecutive, so it works on ASCII machines but fails on EBCDIC machines. This could be solved by using a second array containing an ordinary alphabet; this is illustrated by the solution to the next program.

```
/*
** Encrypts a string of characters according to the key provided.
*/

#include <ctype.h>

void
encrypt( char *data, char const *key )
{
        register int    character;

        /*
        ** Process the data one character at a time.  This depends on the key
        ** being all lower case.
        */
        for( ; ( character = *data ) != '\0'; data++ ){
                if( islower( character ) )
                        *data = key[ character - 'a' ];
                else if( isupper( character ) )
                        *data = toupper( key[ character - 'A' ] );
        }
}
```

Solution 9.13 encrypt.c

14. The solution presented here makes use of an alphabet array so that it is not restricted to ASCII machines.

```
/*
** Decrypts a string of characters according to the key provided.
*/

#include <ctype.h>
#include <string.h>

static  char    alphabet[] = "abcdefghijklmnopqrstuvwxyz";

void
decrypt( char *data, char const *key )
{
        register int    character;

        /*
        ** Process the data one character at a time.  This depends on the key
        ** being all lower case.
        */
        for( ; ( character = *data ) != '\0'; data++ ){
                if( islower( character ) )
                        *data = alphabet[ strchr( key, character ) - key ];
                else if( isupper( character ) )
                        *data = toupper( alphabet[ strchr( key,
                                tolower( character ) ) - key ] );
```

Solution 9.14 *continued . . .*

```
        }
}
```

Solution 9.14 decrypt.c

16. This solution violates the Standard. Can you see where?

```
/*
** Take a pattern string and a digit string and copy the digits into the
** pattern.
*/

#include <stdio.h>
#define TRUE    1
#define FALSE   0

int
format( char *pattern, char *digits )
{
        char    *patternp, *digitp;

        /*
        ** Check for NULL arguments.
        */
        if( pattern == NULL || digits == NULL )
                return FALSE;

        /*
        ** Find end of both strings and see if digit string is empty.
        */
        patternp = pattern + strlen( pattern ) - 1;
        digitp = digits + strlen( digits ) - 1;
        if( digitp < digits )
                return FALSE;

        /*
        ** Continue until either pattern or digits have been used up.
        */
        while( patternp >= pattern && digitp >= digits ){
                if( *patternp == '#' ){
                        *patternp-- = *digitp--;
                        continue;
                }
                patternp--;
        }

        /*
        ** If there are more characters in the pattern, replace them with
        ** blanks.
        */
        while( patternp >= pattern ){
                if( *patternp == '.' ){
```

Solution 9.16 *continued . . .*

```
                        /*
                        ** Extend zeros out to the left of the dot.
                        */
                        char    *p0;

                        for( p0 = patternp + 1; *p0 == ' '; *p0++ = '0' )
                                ;
                        /*
                        ** Put a zero to the left of the dot.
                        */
                        *--patternp = '0';
                        --patternp;
                        continue;
                }
                *patternp-- = ' ';
        }

        /*
        ** If there are digits left over, it is an error.
        */
        return digitp < digits;
}
```

Solution 9.16 format.c

Technically, the test

```
while( patternp >= pattern && digitp >= digits ){
```

is illegal, because when it is false, either the `patternp` or the `digitp` pointer has run off the left end of the corresponding array. On the other hand, it is possible (though unlikely) that the caller has invoked the function like this:

```
if( format( p_array + 1, d_array + 1 ) ) ...
```

in which case there is no violation.

The interesting part of this is that the pointer arithmetic needed to make the comparison work in the legal case shown above cause it to work in the "illegal" case as well. In fact, the only way that the program could fail is if one of these arrays began at location zero: computing a pointer to the location "before" the beginning of the array would actually wrap around and produce a pointer to the end of the address space. This would make the comparison fail.

On most systems this cannot happen: zero is the null pointer, so the compiler may not put any data there. However, there is still a danger: the Standard allows the null pointer to actually be any value, even though zero is always used to represent it in source code. With such an implementation, it is possible that the problem described above could occur, and it would be the devil to debug.

The function should really be fixed to remove this dependency; the erroneous version was given here solely to spark this discussion.

17. This is probably the longest single function in this book. Sloppy design would make it considerably longer. For example, much of the processing for the digit selector and significance starter is identical, and should not be repeated.

```
/*
** Process a pattern string by copying digits from a digit string into it, ala
** the IBM 360 "edit" instruction.
*/

#define NULL                    0
#define NUL                     '\0'

#define DIGIT_SELECTOR          '#'
#define SIGNIFICANCE_START      '!'

#define TRUE                    1
#define FALSE                   0

#define reg                     register

char *
edit( reg char *pattern, reg char *digits )
{
        reg     int     digit;
        reg     int     pat_char;
        reg     int     fill;
        reg     int     significance;
        reg     char    *first_digit;

        /*
        ** Check for missing data, and get fill character.
        */
        if( pattern == NULL || digits == NULL || ( fill = *pattern ) == '\0' )
                return NULL;

        first_digit = NULL;
        significance = FALSE;

        /*
        ** Process pattern string one by one.
        */
        while( ( pat_char = *++pattern ) != NUL ){

                /*
                ** See what meaning the pattern character has.
                */
                switch( pat_char ){
                case DIGIT_SELECTOR:
                case SIGNIFICANCE_START:
                        if( ( digit = *digits++ ) == NUL ){
                                *pattern = NUL;
                                return first_digit;
                        }

                        if( digit == ' ' )
                                digit = '0';
```

Solution 9.17 *continued . . .*

```
                        if( digit != '0' || pat_char == SIGNIFICANCE_START ){
                                if( !significance )
                                        first_digit = pattern;
                                significance = TRUE;
                        }

                        break;

                default:
                        digit = pat_char;
                        break;
                }

                /*
                ** Store the proper character in the result.
                */
                *pattern = significance ? digit : fill;
        }

        return first_digit;
}
```

Solution 9.17 ibm_edit.c

10

Structures and Unions

10.1 Questions

1. Structure members can be all different types; they are accessed by name; and unused memory may be between adjacent members to enforce boundary alignment requirements. Array elements must all be the same type; they are accessed with a subscript; and no space is ever lost between elements for boundary alignment.

3. First, a declaration where all components are given:

```
struct  S       {
        int     a;
        float   b;
} x;
```

This declares `x` to be a structure having two members, `a` and `b`. In addition, the structure tag `S` is created for use in future declarations.

Omitting the tag field gives:

```
struct  {
        int     a;
        float   b;
} z;
```

which has the same effect as before, except that no tag is created. While other declarations may created more structure variables with identical members, it is not possible to create any more variables with the same type as `z`.

Omitting the member list gives:

```
struct  S       y;
```

which declares another structure variable `y` with the same type as `x`.

Omitting the variable list gives:

```
struct  S       {
        int     a;
        float   b;
};
```

which simply defines the tag `S` for use in later declarations. Finally, there is the incomplete declaration

```
struct  S;
```

which informs the compiler that `S` is a structure tag to be defined later.

4. `abc` is the structure tag, not the name of a variable, so the assignment statements are illegal.
5. `abc` is a type name, not the name of a variable, so the assignment statements are illegal.
6. Because `x` is stored in static memory, the initializer for `c` need not be given; the example below omits it.

```
struct  {
        int     a;
        char    b[10];
        float   c;
} x = { 3, "hello" };
```

8. With 16 bit integers, two bytes are wasted, one after each character. With 32 bit integers, six are wasted. Note that space is lost after `c` in order to guarantee that the structure ends at the most stringent boundary. If this were not done, the next variable allocated might not begin at the proper boundary.
9. The following are all implementation dependent:
   a. whether the fields are allocated right to left or left to right;
   b. whether fields too large to fit in the remaining bits of a word begin there anyway and cross the boundary to the next word or begin in the next word;
   c. whether signed or unsigned arithmetic is used for fields declared signed; and
   d. the maximum size of an individual field.
10. One or the other of the following declarations will be correct, depending on whether the compiler allocates bit fields from left to right or from right to left.

```
struct  FLOAT_FORMAT {
        unsigned int    sign:1;
        unsigned int    exponent:7;
        unsigned int    fraction:24;
};

struct  FLOAT_FORMAT {
        unsigned int    fraction:24;
        unsigned int    exponent:7;
        unsigned int    sign:1;
};
```

12. It can either be 2 or -2, depending on whether the compiler uses signed or unsigned arithmetic.
13. A union is being used as if it were a structure. On a machine with 32 bit integers and floats, the second assignment will completely replace the value stored by the first, and the last assignment will replace the first eight bits of the value stored by the second. The integer and floating-point members therefore print as garbage, but the character prints correctly.

14. The same member that was used to store the data must also be used to read it.

15. First, the member `s` would store the actual value of the string rather than a pointer to the value. This means that the value would not have to be allocated elsewhere, which is an advantage. But this entails a terrible disadvantage: The structure now contains enough space to store the largest possible string, and nearly all of this space is wasted when integer and floating-point values are stored. The original structure did not have this problem because it only contained a pointer to the string value, not the value itself.

10.2 Programming Exercises

2. Though not mentioned in the problem, a correct solution *must* have some kind of `type` field. The first solution is implemented exactly as the instructions specify.

```
/*
** Structure declaration for auto dealership sales records.
*/
struct  INFO1   {
        char    cust_name[21];
        char    cust_addr[41];
        char    model[21];
        enum    { PURE_CASH, CASH_LOAN, LEASE } type;
        union   {
                struct  {
                        float   msrp;
                        float   sales_price;
                        float   sales_tax;
                        float   licensing_fee;
                } pure_cash;
                struct  {
                        float   msrp;
                        float   sales_price;
                        float   sales_tax;
                        float   licensing_fee;
                        float   down_payment;
                        int     loan_duration;
                        float   interest_rate;
                        float   monthly_payment;
                        char    bank[21];
                } cash_loan;
                struct  {
                        float   msrp;
                        float   sales_price;
                        float   down_payment;
                        float   security_deposit;
                        float   monthly_payment;
                        float   lease_term;
                } lease;
        } info;
};
```

Solution 10.2a info1.h

The second factors fields common to all three types of sale out of the variant portion of the record. There are still some duplicated fields, but none are common to all three variants.

```
/*
** Improved structure declaration for auto dealership sales records.
*/
struct  INFO2   {
        char    cust_name[21];
        char    cust_addr[41];
        char    model[21];
        float   msrp;
        float   sales_price;
        enum    { PURE_CASH, CASH_LOAN, LEASE } type;
        union   {
                struct  {
                        float   sales_tax;
                        float   licensing_fee;
                } pure_cash;
                struct  {
                        float   sales_tax;
                        float   licensing_fee;
                        float   down_payment;
                        int     loan_duration;
                        float   interest_rate;
                        float   monthly_payment;
                        char    bank[21];
                } cash_loan;
                struct  {
                        float   down_payment;
                        float   security_deposit;
                        float   monthly_payment;
                        float   lease_term;
                } lease;
        } info;
};
```

Solution 10.2b info2.h

3. The correct answer depends on whether the machine you are using allocates bit fields from left to right or from right to left. This answer is for the former; the order of the bit fields (but not the union members) would be reversed for the latter.

```
/*
** Declaration of a structure to access the various parts of a machine
** instruction for a particular machine.
*/
typedef union
{
        unsigned short  addr;
        struct {
                unsigned opcode:10;
```

Solution 10.3 *continued . . .*

```
                unsigned dst_mode:3;
                unsigned dst_reg:3;
        } sgl_op;
        struct {
                unsigned opcode:4;
                unsigned src_mode:3;
                unsigned src_reg:3;
                unsigned dst_mode:3;
                unsigned dst_reg:3;
        } dbl_op;
        struct {
                unsigned opcode:7;
                unsigned src_reg:3;
                unsigned dst_mode:3;
                unsigned dst_reg:3;
        } reg_src;
        struct {
                unsigned opcode:8;
                unsigned offset:8;
        } branch;
        struct {
                unsigned opcode:16;
        } misc;
} machine_inst;
```

Solution 10.3 machinst.h

11

Dynamic Memory Allocation

11.1 Questions

1. This will vary from system to system. There are several things that may affect the result on PC-based systems, including the memory model in use, the amount of space in the data and/or stack segment, the amount of available memory on the system, and so forth. The result on Unix systems will depend on the amount of available swap space, among other things.
2. There are two explanations possible. Requesting smaller chunks may allow more memory to be allocated because the amount of memory left over after the last allocation will be smaller. This would make the total for the smaller requests larger. More likely, though, is that the total for the smaller requests is smaller: this is due to the overhead of the extra space that `malloc` attaches to the memory in order to keep track of the size of each allocated chunk.

6. Yes, dynamic allocation will use less stack space because the memory for the arrays will be taken from the heap rather than the stack. Dynamic allocation of scalars will help only if the values being allocated are larger than the size of a pointer, as it would be with a large structure. There is no gain in dynamically allocating an integer because the pointer variable you must have to keep track of it takes just as much space as the integer itself.
7. Memory leaks would be possible, but only when either the second or third allocations failed, meaning that the program had nearly run out of memory anyway.

11.2 Programming Exercises

1. This interesting part of this is the need to declare a pointer other than `void *` in order to clear the memory.

```
/*
** A function that performs the same job as the library 'calloc' function.
*/

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

Solution 11.1 *continued . . .*

```
void *
calloc( size_t n_elements, size_t element_size )
{
        char    *new_memory;

        n_elements *= element_size;
        new_memory = malloc( n_elements );
        if( new_memory != NULL ){
                char    *ptr;

                ptr = new_memory;
                while( --n_elements >= 0 )
                        *ptr++ = '\0';
        }

        return new_memory;
}
```

Solution 11.1 calloc.c

3. The strategy used in this solution is to keep a dynamically allocated buffer in the `readstring` function. If this fills while reading a string, it is enlarged. The increment, `DELTA`, can be tuned to achieve a balance between minimizing wasted space and minimizing the number of reallocations that occur. The `assert` macro is used to abort the program if any memory allocation fails. A new chunk of memory is allocated for the copy of the string that is returned to the caller—this avoids the overhead of dynamically growing the array from scratch each time a string is read.

```
/*
** Read a string and return a copy of it in dynamically allocated memory.  There
** is no limit (other than the amount of dynamic memory available) on the size
** of the string.
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#define DELTA   256

char *
readstring()
{
        static  char    *buffer = NULL;
        static  int     buffer_size = 0;
        int     ch;
        int     len;
        char    *bp;

        bp = buffer;
        len = 0;
```

Solution 11.3 *continued . . .*

```
        /*
        ** Get characters one at a time until a newline is read or EOF is
        ** reached.
        */
        do {
                ch = getchar();
                if( ch == '\n' || ch == EOF )
                        ch = '\0';

                /*
                ** If the buffer is full, make it bigger.
                */
                if( len >= buffer_size ){
                        buffer_size += DELTA;
                        buffer = realloc( buffer, buffer_size );
                        assert( buffer != NULL );
                        bp = buffer + len;
                }

                *bp++ = ch;
                len += 1;
        } while( ch != '\0' );

        /*
        ** Make a copy of the string to return.
        */
        bp = malloc( len );
        assert( bp != 0 );
        strcpy( bp, buffer );
        return bp;
}
```

Solution 11.3 readstr.c

4. There are many variations on this program that are equally good. If the list were to be much longer, it would be worth while to have a pointer to the end of the list in order to simplify the insertions.

```
/*
** Create a particular linked list of dynamically allocated nodes.
*/

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

typedef struct NODE {
        int     value;
        struct  NODE    *link;
} Node;

Node *
```

Solution 11.4 *continued . . .*

```
newnode( int value ){
        Node    *new;

        new = (Node *)malloc( sizeof( Node ) );
        assert( new != 0 );
        new->value = value;
        return new;
}

main()
{
        Node    *head;

        head = newnode( 5 );
        head->link = newnode( 10 );
        head->link->link = newnode( 15 );
        head->link->link->link = 0;
}
```

Solution 11.4 linklist.c

Another good approach is to construct the list in reverse order.

12

Using Structures and Pointers

12.1 Questions

1. Yes, it can.

```
/*
** Look for the right place.
*/
while( *linkp != NULL && (*linkp)->value < value )
        linkp = &(*linkp)->link;

/*
** Allocate a new node and insert it.
*/
new = malloc( sizeof( Node ) );
if( new == NULL )
        return FALSE;
new->value = value;
new->link = *linkp;
*linkp = new;
return TRUE;
```

   This version of the program uses one fewer variable, but it has three extra indirections so it will take slightly longer to execute. It is also harder to understand, which is its major drawback.

3. Ahead of other nodes with the same value. If the comparison were changed, duplicate values would be inserted after other nodes with the same value.

5. Each attempt to add a duplicate value to the list would result in a memory leak: A new node would be allocated, but not added to the list.

6. Yes, but it is very inefficient. The simplest strategy is to take the nodes off the list one by one and insert them into a new, ordered list.

12.2 Programming Exercises

2. First, the program for the unordered list.

```
/*
** Find a specific value in node an unordered singly linked list.
*/

#include "singly_linked_list_node.h"
#include <stdio.h>

struct NODE *
sll_find( struct NODE *first, int desired_value )
{
        for( ; first != NULL; first = first->link )
                if( first->value == desired_value )
                        return first;

        return NULL;
}
```

Solution 12.2 sll_find.c

Technically, no change is *required* to search an ordered list, though the function can be made more efficient with a minor change. If nodes are found whose values are greater than the desired value, there is no need to continue searching. This is implemented by changing the test in the `for` loop to

```
first != NULL && first->value <= value
```

3. This makes the function more complex, primarily because the root pointers can no longer be manipulated in the same way as the node pointers.

```
/*
** Insert a value into a doubly linked list.  frontp is a pointer to the root
** pointer to the first node; rearp is a pointer to the root pointer to the last
** node; and value is the new value to be inserted.  Returns: 0 if the value is
** already in the list, -1 if there was no memory to create a new node, 1 if the
** value was added successfully.
*/
#include <stdlib.h>
#include <stdio.h>
#include "doubly_linked_list_node.h"

int
dll_insert( Node **frontp, Node **rearp, int value )
{
        register Node   **fwdp;
        register Node   *next;
        register Node   *newnode;

        /*
```

Solution 12.3 *continued . . .*

```
        ** See if value is already in the list; return if it is.  Otherwise,
        ** allocate a new node for the value ("newnode" will point to it), and
        ** "next" will point to the one after where the new node goes.
        */
        fwdp = frontp;
        while( (next = *fwdp) != NULL ){
                if( next->value == value )
                        return 0;
                if( next->value > value )
                        break;
                fwdp = &next->fwd;
        }

        newnode = (Node *)malloc( sizeof( Node ) );
        if( newnode == NULL )
                return -1;
        newnode->value = value;

        /*
        ** Add the new node to the list.
        */
        newnode->fwd = next;
        *fwdp = newnode;

        if( fwdp != frontp )
                if( next != NULL )
                        newnode->bwd = next->bwd;
                else
                        newnode->bwd = *rearp;
        else
                newnode->bwd = NULL;

        if( next != NULL )
                next->bwd = newnode;
        else
                *rearp = newnode;

        return 1;
}
```

Solution 12.3 dll_insb.c

4. This is a good exercise in examining a problem critically to determine exactly what needs to be done. If implemented properly, the function is quite simple.

```
/*
** Reverse the order of the nodes in a singly linked list, returning a pointer
** to the new head of the list.
*/

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

Solution 12.4 *continued . . .*

```
struct NODE *
sll_reverse( struct NODE *current )
{
        struct NODE     *previous;
        struct NODE     *next;

        for( previous = NULL; current != NULL; current = next ){
                next = current->link;
                current->link = previous;
                previous = current;
        }

        return previous;
}
```

Solution 12.4 sll_rev.c

6. Note the chains of indirection in this program: `this->bwd->fwd` and `this->fwd->bwd`: if either of these expressions had been used more than once, it might have been better to create temporary variables to hold the values instead.

```
/*
** Remove a specified node from a doubly linked list.  The first argument points
** to the root node for the list, and the second points to the node to be
** removed. TRUE is returned if it can be removed, otherwise FALSE is returned.
*/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "doubly_linked_list_node.h"

#define FALSE   0
#define TRUE    1

int
dll_remove( struct NODE *rootp, struct NODE *delete )
{
        register Node   *this;

        assert( delete != NULL );

        /*
        ** Find the node in the list and return FALSE if it is not there.
        ** Otherwise, delete it and return TRUE.
        */
        for( this = rootp->fwd; this != NULL; this = this->fwd )
                if( this == delete )
                        break;

        if( this == delete ){
```

Solution 12.6 *continued . . .*

```
                /*
                ** Update fwd pointer of the previous node.
                */
                if( this->bwd == NULL )
                        rootp->fwd = this->fwd;
                else
                        this->bwd->fwd = this->fwd;

                /*
                ** Update bwd pointer of the next node.
                */
                if( this->fwd == NULL )
                        rootp->bwd = this->bwd;
                else
                        this->fwd->bwd = this->bwd;

                free( this );
                return TRUE;
        }
        else
                return FALSE;
}
```

Solution 12.6 dll_remv.c

7. The fact that nodes in the two lists differ in structure precludes using a single insertion function for both lists; this is unfortunate, as the same logic is needed in each case. The declarations in the solution below should be in an include file if they are needed by any other source files.

```
/*
** Insert a word into a concordance list.  The arguments are a pointer to the
** location that points to the first node in the list, and a string containing
** the word to be inserted.  The function returns TRUE if the string begins with
** a letter and was inserted into the list, else FALSE.
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <malloc.h>

#define TRUE    1
#define FALSE   0

/*
** Declaration for the secondary node that holds an individual word.
*/
typedef struct  WORD{
        char            *word;
        struct  WORD    *next;
} Word;
```

Solution 12.7 *continued . . .*

```
/*
** Declaration for the primary node that heads a list of words.
*/
typedef struct  LIST    {
        char            letter;
        struct  LIST    *next;
        Word            *word_list;
} List;

int
concordance_insert( List **listp, char *the_word )
{
        int     first_char;
        List    *current_list;
        Word    *current_word;
        Word    **wordp;
        int     comparison;
        Word    *new_word;

        /*
        ** Get the first character of the word and make sure it is valid.
        */
        first_char = *the_word;
        if( !islower( first_char ) )
                return FALSE;

        /*
        ** First, find the word list that begins with the right letter. If it
        ** does not exist, create a new one and add it.
        */
        while( (current_list = *listp) != NULL
            && current_list->letter < first_char )
                listp = &current_list->next;

        /*
        ** If current_list is NULL or points to a node with a letter larger
        ** than what we want, we've got to create a new word list and insert it
        ** here in the primary list.
        */
        if( current_list == NULL || current_list->letter > first_char ){
                List    *new_list;

                new_list = (List *)malloc( sizeof( List ) );
                if( new_list == NULL )
                        return FALSE;

                new_list->letter = first_char;
                new_list->word_list = NULL;
                new_list->next = current_list;
                *listp = new_list;
                current_list = new_list;
        }
```

Solution 12.7

continued . . .

```
        /*
        ** current_list now points to the node that heads the proper word list.
        ** Search down through it looking for our word.
        */
        wordp = &current_list->word_list;
        while( ( current_word = *wordp ) != NULL ){
                comparison = strcmp( current_word->word, the_word );
                if( comparison >= 0 )
                        break;
                wordp = &current_word->next;
        }

        /*
        ** If current_word not NULL and comparison is 0, the word already is in
        ** the list.
        */
        if( current_word != NULL && comparison == 0 )
                return FALSE;

        /*
        ** Create a new node for the word.
        */
        new_word = (Word *)malloc( sizeof( Word ) );
        if( new_word == NULL )
                return FALSE;

        new_word->word = malloc( strlen( the_word ) + 1 );
        if( new_word->word == NULL )
                return;

        strcpy( new_word->word, the_word );

        /*
        ** Link the new node into the list.
        */
        new_word->next = current_word;
        *wordp = new_word;

        return TRUE;
}
```

Solution 12.7 concord.c

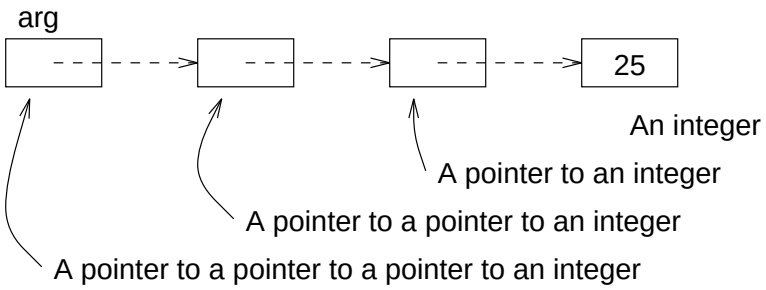
13

Advanced Pointer Topics

13.1 Questions

2. As with all pointer arithmetic, the value one is scaled to the size of whatever the pointer is pointing at, which in this case is a pointer to a character. The result is that `ptr` is advanced to point at the next element of the array.

3. It is a pointer to a pointer to a pointer to an integer, and is used like this:



   The expression `***arg` gets the value of the integer.

5. a. The address of `p`.

   b. The entire value (both fields) of `p`.

   c. The value of the `x` field of `p`.

   d. The address of the variable `a`.

   e. Illegal, as `p` is not a pointer to a structure.

   f. Illegal, as `p` is not a pointer to a structure.

   g. The value stored in `a`, which is the address of `p`.

   h. `->` has higher precedence than `*`, so this is the same as `*(b->a)`, which is illegal because `b` is not a pointer to a structure.

   i. `->` has higher precedence than `*`, so this is the same as `*(b->x)`, which is illegal because `b` is not a pointer to a structure.

j. Illegal, as `p` is not a pointer to a structure.

k. `(*b)` gives the value of `a`, so this is equivalent to `a->a`, which is illegal because the structure to which `a` points has no field which is named `a`.

l. `(*b)` gives the value of `a`, so this is equivalent to `a->x`, which gives the value of the `x` field of `p`.

m. `*b` gives the value of `a`, so this is equivalent to `*a`, which gives the entire value (both fields) of `p`.

6. a. Copies the values from `y` into `x`.

b. Illegal, as `a` is a pointer but `y` is not.

c. Copies the pointer value from `b` into `a`.

d. Illegal, as a is `a` pointer but `*b` is not.

e. Copies the values from `y` into `x`.

8. If the filename exceeds 15 characters (this one does not), it will overflow the memory allocated to hold the string literal. If the initial value of the pathname is ever changed, you must remember to change the literal constant ten also. But the real problem is that it overwrites the string literal, the effects of which are not defined by the Standard. The solution is to declare `pathname` as an array, not a pointer.

9. The array is only as large as needed to hold its initial value, so appending anything to it overflows the memory array, probably overwriting some other variables. The solution is to make the array enough larger than the initial value to hold the longest filename that will be appended to it later. A potential problem is the use of `strcat`; if this pathname prefix is supposed to be used with several different filenames, appending the next one with `strcat` will not produce the desired result.

10. If the string in `filename` is longer than nine characters, it will overflow the array. If the length of a pathname has an upper bound, the array should be declared at least that large. Otherwise, the length of the filename can be computed and used to obtain memory dynamically.

13.2 Programming Exercises

2. This function is probably not terribly useful, as it is nearly as easy to traverse the list on your own as it is to call a general purpose function to do it. On the other hand, the paradigm is valuable enough to justify this exercise.

Note that the function still needs to know where the link is in the node structure, so it isn't all that general after all.

```
/*
** Traverse a singly linked list.  The callback function will be called for each
** node in the list, and it will be passed a pointer to the current node.
*/
#include "node.h"
#include <stdio.h>
```

Solution 13.2 *continued . . .*

```
void
sll_traverse( Node *current, void (*func)( Node *node ) ){
        while( current != NULL ){
                func( current );
                current = current->link;
        }
}
```

Solution 13.2 sll_trav.c

3. Several of the cases do their work directly; this code must be moved into functions. The main difficulty here is to come up with a common prototype that will serve the needs of all of the necessary functions; it must contain each argument used by any function. The new functions must use the common prototype, and the existing functions must be modified to match it. Most of the functions will end up with at least one argument that they do not need. A pointer must be passed to `current` so that the appropriate functions can modify it.

```
/*
** Modified prototypes for existing functions (the functions themselves must be
** modified to match).
*/
void    add_new_trans( Node *list, Node **current, Transaction *trans );
void    delete_trans( Node *list, Node **current, Transaction *trans );
void    search( Node *list, Node **current, Transaction *trans );
void    edit( Node *list, Node **current, Transaction *trans );

/*
**      Definitions of the new functions that are needed.
*/
void
forward( Node *list, Node **current, Transaction *trans )
{
        *current = (*current)->next;
}

void
backward( Node *list, Node **current, Transaction *trans )
{
        *current = (*current)->prev;
}

/*
**      The jump table itself.
*/
void    (*function[])( Node *list, Node **current, Transaction *trans ) = {
        add_new_trans,
        delete_trans,
        forward,
        backward,
        search,
        edit
```

Solution 13.3 *continued . . .*

```
};
#define N_TRANSACTIONS  ( sizeof( function ) / sizeof( function[ 0 ] ) )

/*
**      Invoke the proper function to perform a transaction (this is where the
**      switch statement used to be).
*/
        if( transaction->type < 0 || transaction_type >= N_TRANSACTIONS )
                printf( "Illegal transaction type!\en" );
        else
                function[ transaction->type ]( list, &current, transaction );
```

Solution 13.3 jump_tbl.c

Note that the solution is considerably longer than the original code. This is primarily due to the overhead of creating a function to perform a one-line-of-code task. As the number of transactions grows, though, the jump table combined with the separate transaction functions will be more manageable than a large `switch` statement.

4. This function is patterned after the `qsort` function provided in the ANSI C library. The only two tricks are locating the beginning of each array element and interchanging two elements. The element length is used for both tasks.

```
/*
** Sort an array of arbitrary fixed-length elements.  The caller passes a
** pointer to a function which compares two elements.
*/

/*
**      Exchange two elements with each other.
*/
swap( char *i, char *j, int recsize )
{
        char    x;

        while( recsize-- > 0 ){
                x = *i;
                *i++ = *j;
                *j++ = x;
        }
}

/*
**      Sort the array.
*/
void
sort( char *base, int nel, int recsize, int (*comp)( char *, char * ) )
{
        reg     char    *i;
        reg     char    *j;
        reg     char    *last;
```

Solution 13.4 *continued . . .*

```
        last = base + ( nel – 1 ) * recsize;
        for( i = base; i < last; i += recsize )
                for( j = i + recsize; j <= last; j += recsize )
                        if( comp( i, j ) > 0 )
                                swap( i, j, recsize );
}
```

Solution 13.4 sort.c

5. A common mistake is to modify either the `argv` pointer list or the arguments themselves. This solution uses register declarations for efficiency. The `argtype` function, called from one place, was written as a separate function solely for clarity.

```
/*
** Parse the command line arguments for processing by the caller's functions.
*/

#define TRUE    1
#define FALSE   0

#define NULL    0
#define NUL     '\0'

enum { NONE, FLAG, ARG };

/*
**      Determine whether the argument is a flag or one that requires a value.
*/
argtype( register int ch, register int control )
{
        while( *control != NUL )
                if( ch == *control++ )
                        return *control == '+' ? ARG : FLAG;

        return NONE;
}

/*
**      Process the arguments.
*/
char **
do_args( int ac, reg char **av, char *control,
    void (*do_arg)( int, char * ), void (*illegal_arg)( int ) )
{
        register char   *argp;
        register int    ch;
        register int    skip_arg;

        while( (argp = *++av) != NULL && *argp == '-' ){
                skip_arg = FALSE;
                while( !skip_arg && ( ch = *++argp ) != NUL ){
                        switch( argtype( ch, control ) ){
```

Solution 13.5 *continued . . .*

```
                        case FLAG:
                                (*do_arg)( ch, NULL );
                                break;

                        case ARG:
                                if( *++argp != NUL || (argp = *++av) != NULL ){
                                        (*do_arg)( ch, argp );
                                        skip_arg = TRUE;
                                        break;
                                }
                                (*illegal_arg)( ch );
                                return av;

                        case NONE:
                                (*illegal_arg)( ch );
                                break;
                        }
                }
        }
        return av;
}
```

Solution 13.5 do_args.c

14

The Preprocessor

14.1 Questions

2. First, a well chosen name gives the reader some idea of the meaning of a quantity, whereas a literal constant conveys only its value. Second, literal constants that are used more than once are easier to change if they are defined in a single place.

3.
```
#define DEBUG_PRINT( fmt, expr )        \
        printf( "File %s, line %d: %s = " \
            fmt "\n", \
            __FILE__, __LINE__, \
            #epxr, expr )
```

   The macro is invoked like this:

```
DEBUG_PRINT( "%d", x * y + 3 );
```

   and produces output like this:

```
File main.c, line 25:  x * y + 3  = -4
```

4. a. 3 2 3

   b. 5 3 5

   c. 2 4 20

   d. 2 4 12

5. Because `putchar` is invoked so often, speed of invocation was considered of primary importance. Implementing it as a macro eliminates the overhead of a function call.

8. Nothing is wrong. They each include the other, and it first appears that the compiler will read them alternately until its include nesting limit is reached. In fact this does not happen because of the conditional compilation directives. Whichever file is included first defines its own symbol and then causes the other to be included. When it tries to include the first one again, the entire file is skipped.

9. Unfortunately, `sizeof` is evaluated after the preprocessor has finished its work, which means that this won't work. An alternate approach would be to use the values defined in the `limits.h` include file.

14.2 Programming Exercises

2. The function would be more complex if it had to verify that exactly one of the cpu-type symbols was defined. It does not, though, so a simple `#if/#elif` sequence does the job. Note that many compilers predefine symbols for precisely this purpose. Consult your compiler's documentation for details.

```
/*
** Return a code indicating which type of cpu the program is running on.  This
** depends on the proper symbol being defined when the program was compiled.
*/
#include "cpu_types.h"

cpu_type()
{
#if defined( VAX )
        return CPU_VAX;
#elif defined( M68000 )
        return CPU_68000;
#elif defined( M68020 )
        return CPU_68020;
#elif defined( I80386 )
        return CPU_80386;
#elif defined( X6809 )
        return CPU_6809;
#elif defined( X6502 )
        return CPU_6502;
#elif defined( U3B2 )
        return CPU_3B2;
#else
        return CPU_UNKNOWN;
#endif
}
```

Solution 14.2 cpu_type.c

15

Input/Output Functions

15.1 Questions

3. The fact that it failed at all indicates that something is wrong; the most likely possibility is a bug in the program. By not checking, this bug goes undetected and may cause problems later. Also, the `FILE` structure used for that stream will not be released. There is a limited number of these available, so if this happens very often, the program will run out of them and be unable to open any more files.
5. Space is always left for the NUL byte, so with a buffer size of one there is no room for any characters from the stream. With a buffer size of two, characters are read one by one.
6. The first value can take up to six characters, the second at most one, and the third at most four. Counting the two spaces and the terminating NUL byte, the buffer must be at least 14 bytes long.
7. This is quite unsafe as there is no way to tell how large the buffer must be; strings may be any length, so if the length of `a` is not checked prior to this statement, the buffer might be overrun no matter how large it is.
8. It rounds. If 3.14159 is printed with a code of `%.3f` the result is `3.142`.
9. Write a program to store all possible integer values in `errno` and then call `perror`. You must watch the output, as garbage may be produced for values that are not legitimate error codes.
10. Because they change the state of the stream. The call by value semantics of C would not allow the caller's stream variable to be changed if a copy if it were passed as the argument.
11. The mode `r+` does the job. The `w` modes truncate the file, and the `a` modes restrict writing to the end of the file.
12. It allows a particular stream to be reopened to a new file. For example, in order for a program that uses `printf` to begin writing to a different file, the program would have to reopen `stdout`. This function is the reliable way to do this.
13. It is not worth it. Only if a program is not fast enough or not small enough should you spend time thinking about things like this.
14. The results depend on the system, but it won't be 3!

15. The strings will be left justified. At least six characters, but no more than ten, will be printed.

15.2 Programming Exercises

1. This program is quite straightforward.

```
/*
** Copy standard input to standard output, one character at a time.
*/

#include <stdio.h>

main()
{
        int     ch;

        while( (ch = getchar()) != EOF )
                putchar( ch );

        return EXIT_SUCCESS;
}
```

Solution 15.1 prog1.c

3. The buffer need not be made any larger for this, but it is reasonable to increase it. The main change here is to use `fgets` to ensure that the buffer is not overrun by longer input lines.

```
/*
** Copy standard input to standard output, one line at a time.  Lines > 256
** bytes long are copied 256 bytes at a time.
*/

#include <stdio.h>

#define BUFSIZE 256     /* I/O buffer size */

main()
{
        char    buf[BUFSIZE];

        while( fgets( buf, BUFSIZE, stdin ) != NULL )
                fputs( buf, stdout );

        return EXIT_SUCCESS;
}
```

Solution 15.3 prog3.c

4. Because two filenames must be entered, this is simplified by writing a function to read and open a file. The file mode must then be passed as an argument; this solution also passes a string used as a prompt.

```
/*
** This program reads two file names and then copies data from the input file
** to the output file one line at a time.
*/

#include <stdio.h>

#define BUFSIZE 256     /* I/O buffer size */

/*
**      This function prompts for a filename, reads the name, and then tries to
**      open the file.  Any errors encountered are reported before the program
**      terminates.  Note that the gets function strips the trailing newline off
**      of the file name so we don't have to do it ourselves.
*/
FILE *
openfile( char *prompt, char *mode )
{
        char    buf[BUFSIZE];
        FILE    *file;

        printf( "%s filename? ", prompt );
        if( gets( buf ) == NULL ){
                fprintf( stderr, "Missing %s file name.\n", prompt );
                exit( EXIT_FAILURE );
        }

        if( (file = fopen( buf, mode )) == NULL ){
                perror( buf );
                exit( EXIT_FAILURE );
        }

        return file;
}

/*
**      Main function.
*/
int
main()
{
        char    buf[BUFSIZE];
        FILE    *input;
        FILE    *output;
        FILE    *openfile();

        input = openfile( "Input", "r" );
        output = openfile( "Output", "w" );

        while( fgets( buf, BUFSIZE, input ) != NULL )
                fputs( buf, output );
```

Solution 15.4 *continued . . .*

```
        fclose( input );
        fclose( output );

        return EXIT_SUCCESS;
}
```

Solution 15.4 prog4.c

5. The only change here is that the lines are analyzed in a specific manner before being written; this change is all that is shown below.

```
/*
** Same program as before, but now the sum is computed of all lines that begin
** with an integer.
*/

        int     value, total = 0;

        while( fgets( buf, BUFSIZE, input ) != NULL ){
                if( sscanf( buf, "%d", &value ) == 1 )
                        total += value;
                fputs( buf, output );
        }

        fprintf( output, "%d\en", total );
```

Solution 15.5 prog5.c

6. The fact that the string palindrome function exists makes this problem trivial. The only reason this short program rates two stars is that it takes an open mind to think of the approach. A 64 bit integer would use at most 20 digits, so the 50 character buffer is large enough to handle anything that contemporary computers can dish out.

```
/*
** Determine whether or not an integer value is a palindrome.
*/

/*
**      Prototype for the string palindrome function from chapter 9.
*/
extern  int     palindrome( char *string );

int
numeric_palindrome( int value )
{
        char    string[ 50 ];

        /*
        ** Convert the number to a string and then check the string!
        */
        sprintf( string, "%d", value );
```

Solution 15.6 *continued . . .*

```
        return palindrome( string );
}
```

Solution 15.6 n_palind.c

7. The limitation to at most 10 members makes it possible (if tedious) to do this with `fgets` and `sscanf`. Assuming a maximum of three digits (plus one separating space) per age, a buffer of 40 characters would be adequate. However, the problem specification does not say that the ages will be separated by exactly one whitespace character, so this solution uses a buffer of 512 characters instead. If you know something about the nature of the input (e.g., it was created with an editor whose maximum line size is 512 bytes), then this approach is fine. Otherwise, it is risky and you should dynamically allocate a buffer that can be extended whenever a line is found that is too long.

```
/*
** Compute the average age of family members.  Each line of input is for one
** family; it contains the ages of all family members.
*/

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

#define BUFFER_SIZE     512     /* size of input buffer */

int
main()
{
        char    buffer[ BUFFER_SIZE ];

        /*
        ** Get the input one line at a time.
        */
        while( fgets( buffer, BUFFER_SIZE, stdin ) != NULL ){
                int     age[ 10 ];
                int     members;
                int     sum;
                int     i;

                /*
                ** Decode the ages, remembering how many there were.
                */
                members = sscanf( buffer, "%d %d %d %d %d %d %d %d %d %d",
                     age, age + 1, age + 2, age + 3, age + 4, age + 5, age + 6,
                         age + 7, age + 8, age + 9 );

                if( members == 0 )
                        continue;

                /*
                ** Compute the average and print the results.
                */
```

Solution 15.7 *continued . . .*

```
                sum = 0;
                for( i = 0; i < members; i += 1 )
                        sum += age[ i ];

                printf( "%5.2f: %s", (double)sum / members, buffer );
        }
}
```

Solution 15.7 ages.c

8. While not explicitly specified in the problem statement, an important consideration is what to do when the file being dumped does not contain an even multiple of 16 bytes. A simplistic approach is to simply report the missing bytes as zero. The solution below constructs each output line in a buffer in memory; this makes it easier to print only the data that appears in a partial last line while still maintaining the proper format. The program would be easier to modify when someone comes along and wants the format changed (as is inevitable) had defined names been used for the numbers related to the format rather than literal constants.

```
/*
** Print an hexadecimal dump of the specified file.  If no filename argument is
** given, print a dump of the standard input instead.
*/

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <ctype.h>

#define BUFFER_SIZE     64

/*
**      Function to dump the contents of a stream.
*/
void
dump( FILE *stream )
{
        long    offset;
        unsigned char   data[ 16 ];
        int     len;
        char    buffer[ BUFFER_SIZE ];

        /*
        ** Initialize the buffer that will be used for output.
        */
        memset( buffer, ' ', BUFFER_SIZE - 1 );
        buffer[ 45 ] = '*';
        buffer[ 62 ] = '*';
        buffer[ BUFFER_SIZE - 1 ] = '\0';

        offset = 0;
        while( ( len = fread( data, 1, 16, stream ) ) > 0 ){
```

Solution 15.8 *continued . . .*

```
char    *hex_ptr;
char    *char_ptr;
int     i;

/*
** Start building the output buffer with the offset.
*/
sprintf( buffer, "%06X  ", offset );

/*
** Prepare pointers to the hex and character portions of the
** buffer and initialize them to spaces.
*/
hex_ptr = buffer + 8;
char_ptr = buffer + 46;
memset( hex_ptr, ' ', 35 );
memset( char_ptr, ' ', 16 );

/*
** Now translate the data to both of the output forms and store
** it in the buffer.
*/
for( i = 0; i < len; i += 1 ){
        /*
        ** Convert the next character to hex.  Must overwrite
        ** the NUL that sprintf inserts with a space.
        */
        sprintf( hex_ptr, "%02X", data[ i ] );
        hex_ptr += 2;
        *hex_ptr = ' ';

        /*
        ** Leave a space between each group of 4 values in the
        ** hex portion of the line.
        */
        if( i % 4 == 3 )
                hex_ptr++;

        /*
        ** If the character is printable, put it in the char
        ** portion of the line, else put a dot in.
        */
        if( isprint( data[ i ] ) || data[ i ] == ' ' )
                *char_ptr++ = data[ i ];
        else
                *char_ptr++ = '.';
}

/*
** Print the line and then update the offset for the next time
** through the loop.
*/
```

Solution 15.8

continued . . .

```
                puts( buffer );

                offset += len;
        }
}

/*
**      Main program.  Dump the file (if there is an argument) or stdin.
*/
int
main( int ac, char **av )
{
        if( ac <= 1 )
                dump( stdin );
        else {
                FILE    *stream;

                stream = fopen( av[ 1 ], "rb" );
                if( stream == NULL ){
                        perror( av[ 1 ] );
                        exit( EXIT_FAILURE );
                }
                dump( stream );
                fclose( stream );
        }

        return EXIT_SUCCESS;
}
```

Solution 15.8 hex_dump.c

10. This is a fair sized program; here is one way of doing it.

```
/*
** This program computes a checksum for each of the specified input files.  The
** result is printed either to the standard output, or to a file.
*/

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

#define TRUE    1
#define FALSE   0

/*
**      Output option flag
*/
char    file_output     = FALSE;

/*
**      Function prototypes
*/
```

Solution 15.10 *continued . . .*

```
char            **do_args( char ** );
unsigned short  process( FILE * );
void            print( unsigned short, char * );

/*
**      Main function.  Parse arguments and process each file specified.
*/
main( int ac, char **av )
{
        FILE    *f;             /* stream to read from */
        unsigned short  sum;    /* checksum value */

        /*
        ** Process option arguments.  do_args returns a pointer to the first
        ** file name in the argument list.
        */
        av = do_args( av );

        /*
        ** Process the input files.
        */
        if( *av == NULL ){
                /*
                ** No files were given, so read the standard input.
                */
                if( file_output ){
                        fprintf( stderr, "-f illegal with standard input\n" );
                        exit( EXIT_FAILURE );
                }
                sum = process( stdin );
                print( sum, NULL );
        }
        else
                /*
                ** For each file given: open it, process its contents, and print
                ** the answer.
                */
                for( ; *av != NULL; ++av ){
                        f = fopen( *av, "r" );
                        if( f == NULL )
                                perror( *av );
                        else {
                                sum = process( f );
                                fclose( f );
                                print( sum, *av );
                        }
                }

        return EXIT_SUCCESS;
}

/*
```

Solution 15.10 *continued . . .*

```
**      Process a file by reading its contents, character by character, and
**      calling the appropriate summing function.
*/
unsigned short
process( FILE *f )
{
        unsigned short  sum;    /* current checksum value */
        int     ch;             /* current char from the file */

        sum = 0;

        while( (ch = getc( f )) != EOF )
                sum += ch;

        return sum;
}

/*
**      Print the checksum.  This either goes to the standard output or to a
**      file whose name is derived from the input file name.
*/
void
print( unsigned short sum, char *in_name )
{
        char    *out_name;      /* name of output file */
        FILE    *f;             /* stream for opening output file */

        if( !file_output )
                printf( "%u\n", sum );
        else {
                /*
                ** Allocate space to hold output file name.  It needs to be 5
                ** bytes longer than input name to hold ".cks" and the
                ** terminating NUL byte.
                */
                out_name = malloc( strlen( in_name ) + 5 );
                if( out_name == NULL ){
                        fprintf( stderr, "malloc: out of memory\n" );
                        exit( EXIT_FAILURE );
                }

                strcpy( out_name, in_name );
                strcat( out_name, ".cks" );

                f = fopen( out_name, "w" );
                if( f == NULL )
                        perror( out_name );
                else {
                        fprintf( f, "%u\n", sum );
                        fclose( f );
                }
                free( out_name );
```

Solution 15.10 *continued . . .*

```
        }
}

/*
**      Process option arguments.  Return a pointer to the first nonoption
**      argument, which is the beginning of the list of file names.
*/
char **
do_args( char **av )
{
        /*
        ** Look at each command line argument, one by one.
        */
        while( *++av != NULL && **av == '-' ){
                /*
                ** Look at each character in each argument, one by one.
                */
                while( *++*av != '\0' ){
                        /*
                        ** Record each option that was given.
                        */
                        switch( **av ){
                        case 'f':
                                file_output = TRUE;
                                break;

                        default:
                                fprintf( stderr, "Illegal option: %c\n", **av );
                                break;
                        }
                }
        }

        /*
        ** Value to be returned is the pointer to the place in the argument list
        ** just after the options ended.
        */
        return av;
}
```

Solution 15.10 cksum.c

11. There are innumerable details that were not specified in the description of the program, so student's answers may vary in many areas. The solution shown here contains three modules: `main.c` obtains the command line argument and implements the transaction processing loop, `process.c` implements the functions needed to decode and process the transactions, and `io.c` implements the functions to open, read, write, and close the inventory file. Function prototypes and other definitions needed by these modules are found in the associated header files.

    There is an additional header file, `part.h`, which contains definitions relating to the structure that holds information about a part. `typedef`s are used to help ensure that variables are declared with their proper types. This also makes it easier to change the type of a variable later if the need arises. The declarations for `TRUE` and `FALSE` don't really have anything to do with a

part, and appear in `part.h` only because there was no better place to put them. A larger program might have enough global definitions to justify putting them in a separate include file.

```
/*
** Declarations for the simple inventory system.
*/

/*
**      The part structure holds all the information about one part, except for
**      the part number that determines where the part is stored in the file.
**      Part descriptions may be at most 20 characters long.
*/
#define MAX_DESCRIPTION         20
#define DESCRIPTION_FIELD_LEN   ( MAX_DESCRIPTION + 1 )

typedef unsigned long   Part_number;
typedef unsigned short  Quantity;
typedef double          Value;

typedef struct {
        char    description[ DESCRIPTION_FIELD_LEN ];
        Quantity quantity;
        Value   total_value;
} Part;

/*
**      Boolean constants
*/
#define TRUE    1
#define FALSE   0
```

Solution 15.11a part.h

The transaction loop in `main.c` has no body because `process_request` does all the work of reading and processing one transaction. Another common paradigm is to separate the reading and the processing of transactions into different functions; in that case, the main transaction loop will look something like this:

```
while( (trans = read_request()) != NULL )
        process_request( trans );
```

with the first function returning a pointer to a structure containing all the information on one transaction.

```
/*
** Main program for simple inventory system.  Opens the inventory file and
** performs the main transaction processing loop.
*/

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

Solution 15.11b *continued . . .*

```
#include "part.h"
#include "io.h"
#include "process.h"

int
main( int ac, char **av )
{
        if( ac != 2 ){
                fprintf( stderr, "Usage: inventory inv-filename\n" );
                return EXIT_FAILURE;
        }

        if( open_file( av[ 1 ] ) ){
                while( process_request() )
                        ;

                close_file();
        }

        return EXIT_SUCCESS;
}
```

Solution 15.11b main.c

Note the use of explicit field widths in the `print_all` function in `process.c`. This produces a neat table with all the columns lined up. Using `%-*.*s` as the format for the description field allows the width (`MAX_DESCRIPTION`) to be given in the arguments to `printf` rather than being hard-coded in the format. This simplifies changing the width in the future.

The `process_request` function decodes transactions by reading a line of input and attempting to decode it with various `sscanf` statements. End of file is detected and handled as a synonym for the `end` command.

Requests whose formats are identical are grouped together and handled by a single `sscanf`. Each `sscanf` attempts to decode one field more than there ought to be in order to detect the entry of extra data.

The length of the description given in a `new` transaction is limited by the format code used; if a description is entered that exceeds this length, the scan will fail. It is unfortunate that `sscanf`, unlike `printf`, has no provision for giving the field width as an argument; the consequence of this is that the description length must be hard-coded in the format code, making future changes more difficult.

```
/*
** Functions to process a transaction.
*/

/*
**      Process one transaction.  Prompts on the standard output and reads the
**      next request from the standard input and processes it.  Returns FALSE if
**      the request was "end" (meaning there is no more work to do) and TRUE
**      otherwise.
*/
```

Solution 15.11c *continued . . .*

```
int     process_request( void );

/*
**      Size of the longest request we will accept.  This is large enough for
**      the request type, a 20 character description, and a few long integers.
*/
#define MAX_REQUEST_LINE_LENGTH 100
```

Solution 15.11c process.h

```
/*
** Transaction decoding and processing for the inventory system.
*/

#include <stdio.h>
#include "part.h"
#include "io.h"
#include "process.h"

/*
**      These functions implement the various transactions.  They are static
**      because they are called only by the transaction decoding function that
**      appears later in this module.
*/

/*
**      total
*/
static void
total( void )
{
        Part_number p;
        Part    part;
        Value   total_value;

        /*
        ** Read each part and add its value to the total.
        */
        total_value = 0;
        for( p = last_part_number(); p > 0; p -= 1 )
                if( read_part( p, &part ) )
                        total_value += part.total_value;

        printf( "Total value of inventory = %.2f\n", total_value );
}

/*
**      new_part
*/
static void
new_part( char const *description, Quantity quantity, Value price_each )
{
```

Solution 15.11d *continued . . .*

```
        Part    part;
        Part_number part_number;

        /*
        ** Copy the arguments into the Part structure.
        */
        strcpy( part.description, description );
        part.quantity = quantity;
        part.total_value = quantity * price_each;

        /*
        ** Get the smallest part number that is not currently being used and
        ** write the information for this part.
        */
        part_number = next_part_number();
        write_part( part_number, &part );
        printf( "%s is part number %lu\n", description, part_number );
}

/*
**      buy and sell
*/
static void
buy_sell( char request_type, Part_number part_number, Quantity quantity,
    Value price_each )
{
        Part    part;

        if( !read_part( part_number, &part ) )
                fprintf( stderr, "No such part\n" );
        else {
                if( request_type == 'b' ){
                        /*
                        ** Buy.
                        */
                        part.quantity += quantity;
                        part.total_value += quantity * price_each;
                }
                else {
                        /*
                        ** Sell: make sure we've got enough to sell.  If so,
                        ** compute the profit on this sale.
                        */
                        Value   unit_value;

                        if( quantity > part.quantity ){
                                printf( "Sorry, only %hu in stock\n",
                                    part.quantity );
                                return;
                        }

                        unit_value = part.total_value / part.quantity;
```

Solution 15.11d *continued . . .*

```
                        part.total_value -= quantity * unit_value;
                        part.quantity -= quantity;
                        printf( "Total profit: $%.2f\n",
                            quantity * ( price_each - unit_value ) );
                }
                write_part( part_number, &part );
        }
}

/*
**      "delete"
*/
static void
delete( Part_number part_number )
{
        Part    part;

        if( !read_part( part_number, &part ) )
                fprintf( stderr, "No such part\n" );
        else {
                part.description[ 0 ] = '\0';
                write_part( part_number, &part );
        }
}

/*
**      "print"
*/
static void
print( Part_number part_number )
{
        Part    part;

        if( !read_part( part_number, &part ) )
                fprintf( stderr, "No such part\n" );
        else {
                printf( "Part number %lu\n", part_number );
                printf( "Description: %s\n", part.description );
                printf( "Quantity on hand: %hu\n", part.quantity );
                printf( "Total value: %.2f\n", part.total_value );
        }
}

/*
**      "print all"
*/
static void
print_all( void )
{
        Part_number p;
        Part    part;
```

Solution 15.11d *continued . . .*

```
        printf( "Part number  Description             Quantity  "
            "Total value\n" );
        printf( "-----------  --------------------  ----------  "
            "-----------\n" );
        for( p = 1; p <= last_part_number(); p += 1 )
                if( read_part( p, &part ) )
                        printf( "%11lu  %-*.*s  %10hu  %11.2f\n",
                            p, MAX_DESCRIPTION, MAX_DESCRIPTION,
                            part.description, part.quantity,
                            part.total_value );
}

/*
**      Decode and process one transaction.
*/
int
process_request( void )
{
        char    request[ MAX_REQUEST_LINE_LENGTH ];
        char    request_type[ 10 ];
        char    description[ DESCRIPTION_FIELD_LEN ];
        Part_number part_number;
        Quantity quantity;
        Value   price_each;
        char    left_over[ 2 ];

        /*
        ** Prompt for and read the request.  If end of file is reached, return
        ** FALSE to stop the main transaction loop.
        */
        fputs( "\nNext request? ", stdout );
        if( fgets( request, MAX_REQUEST_LINE_LENGTH, stdin ) == NULL )
                return FALSE;

        /*
        ** See what type of request it is and decode the arguments.  Note the
        ** attempt to extract one extra string (left_over) from each request to
        ** ensure that the user doesn't enter too much data.
        */

        /*
        ** "end" and "total": take no arguments
        */
        if( sscanf( request, "%10s %1s", request_type, left_over ) == 1 &&
            ( strcmp( request_type, "end" ) == 0
            || strcmp( request_type, "total" ) == 0 ) ){
                if( request_type[ 0 ] == 'e' )
                        /*
                        ** 'end' request: return FALSE to stop the main
                        ** transaction loop.
                        */
```

Solution 15.11d *continued . . .*

```
                return FALSE;
        else
                total();
}

/*
** "new": requires description, quantity, cost each.  It uses the next
** available part number.
*/
else if( sscanf( request, "new %20[^,],%hu,%lf %1s",
    description, &quantity, &price_each, left_over ) == 3 ){
        new_part( description, quantity, price_each );
}

/*
** "buy" and "sell": require part-number, quantity, price each.
*/
else if( sscanf( request, "%10s %lu,%hu,%lf %1s", request_type,
    &part_number, &quantity, &price_each, left_over ) == 4 &&
    ( strcmp( request_type, "buy" ) == 0
    || strcmp( request_type, "sell" ) == 0 ) ){
        buy_sell( request_type[ 0 ], part_number, quantity,
            price_each );
}

/*
** "delete" and "print": require a part number.
*/
else if( sscanf( request, "%10s %lu %1s", request_type, &part_number,
    left_over ) == 2 &&
    ( strcmp( request_type, "delete" ) == 0
    || strcmp( request_type, "print" ) == 0 ) ){
        if( request_type[ 0 ] == 'd' )
                delete( part_number );
        else
                print( part_number );
}

/*
** "print all": takes no arguments.
*/
else if( sscanf( request, "print %10s %1s", request_type, left_over ) == 1
    && strcmp( request_type, "all" ) == 0 ){
        print_all();
}

/*
** If nothing else worked, it must be an error.  Print an error message
** if the input line was not empty.
*/
else {
        if( sscanf( request, "%10s", request_type ) == 1 )
```

Solution 15.11d

continued . . .

```
                        fprintf( stderr, "Invalid request: %s\n", request_type );
        }

        /*
        ** Return TRUE so that the main transaction loop keeps going.
        */
        return TRUE;
}
```

Solution 15.11d process.c

Defining all of the I/O functions in one module simplifies accessing the inventory file from other parts of the program as well as enhancing future maintainability. This module uses the part number to determine the position in the file where a part is stored; `fseek` is used to position the file to the proper location before reading or writing. Because there is no part number zero, that location in the file is used to store information about the inventory file itself, specifically, the largest part number currently in use and the part number of the earliest deleted part in the file. In principle, these values need not be written to the file until just before it is closed. This program writes them whenever they change so that no information will be lost if the program is interrupted or aborts.

Note that the file is opened in binary mode because we are writing binary data. On some systems, forgetting this will cause end-of-line processing to be done, damaging the file. For example, the quantity twelve is indistinguishable from a newline character; if this were in fact written as a carriage-return/newline pair, the resulting record would be longer than it ought to and would overwrite the beginning of the next part.

```
/*
** Declarations and prototypes for I/O functions.
*/

/*
**      Open the inventory file.  Takes the filename as the only argument and
**      returns a boolean: TRUE if successful, else FALSE.
*/
int     open_file( char const *filename );

/*
**      Close the inventory file.
*/
void    close_file( void );

/*
**      Return the number of the last part on file.
*/
Part_number last_part_number( void );

/*
**      Return the next available part number.
*/
Part_number next_part_number( void );
```

Solution 15.11e *continued . . .*

```
/*
**      Read an inventory record.  Takes the part number and a pointer to a Part
**      structure as arguments.  Returns TRUE if the part exists and FALSE
**      otherwise.
*/
int     read_part( Part_number part_number, Part *part );

/*
**      Write an inventory record.  Takes the part number and a pointer to a
**      Part structure as arguments.
*/
void    write_part( Part_number part_number, Part const *part );
```

Solution 15.11e io.h

```
/*
** Functions to access the inventory file.
*/

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include "part.h"
#include "io.h"

/*
**      Stream used for the inventory file.  The part_number array contains the
**      largest part number used and the number of the first deleted part.  The
**      latter makes it more efficient to add new parts by avoiding the need to
**      scan the entire file to locate a previously deleted entry.  These are
**      all static because all the functions that need it are in this module.
*/
static  FILE            *inv_file;
static  Part_number     part_number[ 2 ];
static  enum            { LAST, NEXT };

/*
**      Write the last and next part numbers to the file.
*/
static void
write_part_numbers( void )
{
        fseek( inv_file, 0, SEEK_SET );
        fwrite( part_number, sizeof( Part_number ), 2, inv_file );
}

/*
**      Open the inventory file.
*/
int
open_file( char const *filename )
```

Solution 15.11f *continued . . .*

```
{
        /*
        ** Try opening the file.
        */
        inv_file = fopen( filename, "r+b" );
        if( inv_file == NULL ){
                /*
                ** It failed.  If it was because the file did not exist, try to
                ** create it.
                */
                if( errno == ENOENT ){
                        inv_file = fopen( filename, "w+b" );
                        if( inv_file != 0 ){
                                part_number[ LAST ] = 0;
                                part_number[ NEXT ] = 1;
                                write_part_numbers();
                        }
                }

                /*
                ** If we could not open (or create) the file, print a message.
                */
                if( inv_file == NULL )
                        perror( filename );
        }
        else
                /*
                ** File opened ok -- read the part number data.
                */
                fread( part_number, sizeof( Part_number ), 2, inv_file );

        /*
        ** Return the status of whether we were able to open the file.
        */
        return inv_file != NULL;
}

/*
**      Close the inventory file.
*/
void
close_file( void )
{
        fclose( inv_file );
}

/*
**      Return the number of the last part on file.
*/
Part_number
last_part_number( void )
{
```

Solution 15.11f

continued . . .

```
        return part_number[ LAST ];
}

/*
**      Return the next part number to use.
*/
Part_number
next_part_number( void )
{
        Part    part;

        /*
        ** If the "next" part number is in the range of existing parts, start
        ** reading the file from that point to find the first deleted part.
        ** Otherwise (or if no deleted parts are found), return the part number
        ** immediately after the last one used up till now.
        */
        while( part_number[ NEXT ] <= part_number[ LAST ] ){
                if( !read_part( part_number[ NEXT ], &part ) )
                        break;
                part_number[ NEXT ] += 1;
        }
        write_part_numbers();
        return part_number[ NEXT ];
}

/*
**      Read a part from the inventory file.
*/
int
read_part( Part_number p, Part *part )
{
        /*
        ** If the part number is legal, try to read the file.  Then verify that
        ** the part was not deleted by checking for a nonempty description.
        */
        if( p > 0 && p <= part_number[ LAST ] ){
                fseek( inv_file, p * sizeof( Part ), SEEK_SET );
                if( fread( part, sizeof( Part ), 1, inv_file ) != 1 ){
                        perror( "Cannot read part" );
                        exit( EXIT_FAILURE );
                }
                return *part->description != '\0';
        }

        return FALSE;
}

/*
**      Write a part to the inventory file.  Update the "last part number" if
**      this part is after the old "last" part.
*/
```

Solution 15.11f

continued . . .

```
void
write_part( Part_number p, Part const *part )
{
        /*
        ** Make sure the part number is legal (a brand new part may have the
        ** next number past "part_number[ LAST ]").
        */
        if( p > 0 && p <= part_number[ LAST ] + 1 ){
                fseek( inv_file, p * sizeof( Part ), SEEK_SET );
                if( fwrite( part, sizeof( Part ), 1, inv_file ) != 1 ){
                        perror( "Cannot write part" );
                        exit( EXIT_FAILURE );
                }

                /*
                ** Update the part number status.  If the part number is larger
                ** than part_number[ LAST ], we just created a new part.
                */
                if( p > part_number[ LAST ] ){
                        part_number[ LAST ] = p;
                        write_part_numbers();
                }

                /*
                ** If the description is empty, this part is being deleted.
                */
                if( part->description[ 0 ] == '\0' && p < part_number[ NEXT ] ){
                        part_number[ NEXT ] = p;
                        write_part_numbers();
                }
        }
}
```

Solution 15.11f io.c

16

Standard Library

16.1 Questions

2. Yes, all that is needed are numbers that have no apparent relationship to one another.

4. It is not easy to be absolutely sure. Some implementations provide a `sleep` function that suspends a program for a period of time; if the value of `clock` continues to increase during a sleep, then it is measuring elapsed time. Lacking that, try reading from the standard input but not entering anything for ten seconds. If `clock` continues to increase during that time, it is either measuring elapsed time or the operating system on your machine isn't very good at managing its I/O devices. Another alternative is to start another program; if `clock` in the first program increases while the second one is running, it is measuring elapsed time.

6. The major problem is that `longjmp` is called after the function that called `setjmp` has returned. This means that the state information saved in the jump buffer is no longer valid, so the result is unpredictable. Compared to this, the fact that the main function does not check whether the right number of command line argument were given is minor. What happens when this executes? It depends on the particular machine. Some will abort on an illegal return address from the no-longer-active `set_buffer` function. Others will go off into an infinite loop somewhere. Still others, particularly RISC machines which do not store function arguments on the stack along with the return address, will go into an infinite loop.

7. The results, of course, will depend on the specific implementation. A Sun Sparc II running Sun OS 4.1.4 raise a `SIGFPE` signal for integer division by zero but not for floating-point division by zero. Borland C++ version 4.0 on an Intel Pentium processor raises the `SIGFPE` signal for a floating-point division by zero, but not for integer division by zero.

   Different implementations behave differently due in large measure to how (or whether) the CPU detects such errors. This behavior cannot be standardized without the cooperation of the CPU manufacturers. The moral of the story is that signal handling is inherently non-portable, despite the best efforts of the Standard.

8. Yes, it would cause the array to be sorted into descending order.

16.2 Programming Exercises

1. This program makes use of the `div` function to obtain both the quotient and remainder from an integer division. On machines without hardware instructions to do integer division, this will be considerably faster than doing a / and a % operation.

```
/*
** For a given input age, compute the smallest radix in the range 2 – 36 for
** which the age appears as a number less than or equal to 29.
*/

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

int
main( int ac, char **av )
{
        int     age;
        int     radix;
        div_t   result;

        if( ac != 2 ){
                fputs( "Usage: age_radix your–age\n", stderr );
                exit( EXIT_FAILURE );
        }

        /*
        ** Get the age argument.
        */
        age = atoi( av[ 1 ] );

        /*
        ** Find the smallest radix that does the job.
        */
        for( radix = 2; radix <= 36; radix += 1 ){
                result = div( age, radix );
                if( result.quot <= 2 && result.rem <= 9 )
                        break;
        }

        /*
        ** Print the results.
        */
        if( radix <= 36 ){
                printf( "Use radix %d when telling your age; "
                    "%d in base %d is %d%d\n",
                    radix, age, radix, result.quot, result.rem );
                return EXIT_SUCCESS;
        }
        else {
                printf( "Sorry, even in base 36 your age "
                    "is greater than 29!\n" );
```

Solution 16.1 *continued . . .*

```
                return EXIT_FAILURE;
        }
}
```

Solution 16.1 age.c

3. The biggest problem with this is the rounding required to show which number the hour hand is actually closer to.

```
/*
** Give the current time as a young child would.
*/
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int
main()
{
        time_t  now;
        struct  tm      *tm;
        int     hour;
        int     minute;

        /*
        ** Get current hour and minute.
        */
        now = time( NULL );
        tm = localtime( &now );
        hour = tm->tm_hour;
        minute = tm->tm_min;

        /*
        ** Round and normalize the hour, convert the minute, and then print them.
        */
        if( minute >= 30 )
                hour += 1;
        hour %= 12;
        if( hour == 0 )
                hour = 12;
        minute += 2;
        minute /= 5;
        if( minute == 0 )
                minute = 12;

        printf( "The big hand is on the %d, and the little hand is on the %d.\n",
            minute, hour );

        return EXIT_SUCCESS;
}
```

Solution 16.3 clock.c

4. The month and year must be adjusted to the proper range, and a time other than midnight should be chosen to avoid ambiguity. Other than that, the program is straightforward. On a Silicon Graphics Indy, the range of valid dates handled by this program is December 14, 1901 through January 18, 2038. This suggests that the reason `time_t` is defined as a signed quantity is so that it can represent 68 years before 1970 as well as 68 years after 1970.

```
/*
** Read a month, day, and year from the command line and determine the day of
** the week for that date.
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

char    *month[] = { "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "December"
};

char    *day[] = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday"
};

int
main( int ac, char **av )
{
        struct  tm      tm;

        if( ac != 4 ){
                fputs( "Usage: day_of_seek month day year\n", stderr );
                exit( EXIT_FAILURE );
        }

        /*
        ** Store the arguments in the struct tm variable.
        */
        tm.tm_sec = 0;
        tm.tm_min = 0;
        tm.tm_hour = 12;
        tm.tm_mday = atoi( av[ 2 ] );
        tm.tm_mon = atoi( av[ 1 ] ) - 1;
        tm.tm_year = atoi( av[ 3 ] ) - 1900 ;
        tm.tm_isdst = 0;

        /*
        ** Normalize it, then print the answer.
        */
        mktime( &tm );
        printf( "%s %d, %d is a %s\n", month[ tm.tm_mon ], tm.tm_mday,
            tm.tm_year + 1900, day[ tm.tm_wday ] );

        return EXIT_SUCCESS;
}
```

Solution 16.4 weekday.c

5. The most common mistake in this program is to use the temperature directly rather than Δt.

```
/*
** Compute wind chill given the temperature in degrees Celsius and wind velocity
** in meters per second.
*/
#include <math.h>

#define A       10.45
#define B       10.00
#define C       -1.0
#define X       1.78816

double
wind_chill( double temp, double velocity )
{
        temp = 33 - temp;
        return 33 - ( ( A + B * sqrt( velocity ) + C * velocity ) * temp ) /
            ( A + B * sqrt( X ) + C * X );
}
```

Solution 16.5 windchil.c

6. There are several common errors: passing the number of payments argument as an integer to `pow`; failing to convert the years and interest to their monthly equivalents; failing to convert the interest to a decimal; and failing to round to the nearest penny.

```
/*
** Compute the monthly mortgage payment given the loan amount, annual interest
** rate, and loan term.
*/
#include <math.h>

double
payment( double amount, double interest, int periods )
{
        interest /= 1200;
        periods *= 12;
        amount = amount * interest /
            ( 1 - pow( 1 + interest, (double)( -periods ) ) );
        return floor( amount * 100 + 0.5 ) / 100;
}
```

Solution 16.6 mortgage.c

8. `sscanf` will no longer work, as there may be any number of ages. Instead, we will use `strtol`.

```
/*
** Compute the average age of family members.  Each line of input is for one
** family; it contains the ages of all family members.
```

Solution 16.8 *continued . . .*

```
*/

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

int
main()
{
        char    buffer[ 512 ];

        /*
        ** Get the input one line at a time.
        */
        while( fgets( buffer, 512, stdin ) != NULL ){
                char    *bp;
                int     members;
                long    sum;
                long    age;

                /*
                ** Decode the ages, one by one.
                */
                sum = 0;
                members = 0;

                bp = buffer;
                while( ( age = strtol( bp, &bp, 10 ) ) > 0 ){
                        members += 1;
                        sum += age;
                }

                if( members == 0 )
                        continue;

                /*
                ** Compute the average and print the results.
                */
                printf( "%5.2f: %s", (double)sum / members, buffer );
        }
}
```

Solution 16.8 ages.c

9. The answers are surprising: In a group of 30 people, the odds are around 70% that at least two of them share a birthday. It only takes a group of around 23 to get even odds. This program, on the other hand, is unremarkable.

```
/*
** Determine the odds of two people in a class of 30 having the same birthday.
*/

#include <stdio.h>
```

Solution 16.9 *continued . . .*

```
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
#include <time.h>

#define TRUE    1
#define FALSE   0

#define TRIALS  10000

int
main( int ac, char **av )
{
        int     n_students = 30;
        int     *birthdays;
        int     test;
        int     match;
        int     total_matches = 0;

        /*
        ** See how many students in the class (default 30).
        */
        if( ac > 1 ){
                n_students = atoi( av[ 1 ] );
                assert( n_students > 0 );
        }

        /*
        ** Seed the random number generator.
        */
        srand( (unsigned int)time( 0 ) );

        /*
        ** Allocate an array for the students' birthdays.
        */
        birthdays = (int *)malloc( n_students * sizeof( int ) );
        assert( birthdays != NULL );

        /*
        ** Run the tests a bunch of times!
        */
        for( test = 0; test < TRIALS; test += 1 ){
                int     i;

                /*
                ** Generate the birthdays and check for matches.
                */
                match = FALSE;
                for( i = 0; i < n_students && !match; i += 1 ){
                        int     j;

                        /*
```

Solution 16.9 *continued . . .*

```
                        ** Generate the next birthday.
                        */
                        birthdays[ i ] = rand() % 365;

                        /*
                        ** See if it matches any of the existing
                        ** ones; quit as soon as we find a match.
                        */
                        for( j = 0; !match && j < i; j += 1 )
                                if( birthdays[ i ] == birthdays[ j ] )
                                        match = TRUE;
                }

                /*
                ** Count the results.
                */
                if( match )
                        total_matches += 1;
        }

        printf( "The odds of any two people in a group of %d\n"
            "having the same birthday are %g\n", n_students,
            (double)total_matches / TRIALS );

        free( birthdays );
}
```

Solution 16.9 birthday.c

10. The only difference between this program and the algorithm described is that the sorted part of the array begins with one element in it.

```
/*
** Do an insertion sort to order the elements in an array.
*/

#include <stdio.h>
#include <assert.h>
#include <malloc.h>

void
insertion_sort( void *base, size_t n_elements, size_t el_size,
    int (*compare)( void const *x, void const *y ) )
{
        char    *array;
        char    *temp;
        int     i;
        int     next_element;

        /*
        ** Copy base address into a char * so we can do pointer arithmetic.
        ** Then get a temporary array large enough to hold a single element.
```

Solution 16.10 *continued . . .*

```
        */
        array = base;
        temp = malloc( el_size );
        assert( temp != NULL );

        /*
        ** The first element in the array is already sorted.  Insert the
        ** remaining ones one by one.
        */
        for( next_element = 1; next_element < n_elements; next_element += 1 ){
                char    *i_ptr = array;
                char    *next_ptr = array + next_element * el_size;

                /*
                ** Find the right place to insert the next element.
                */
                for( i = 0; i < next_element; i += 1, i_ptr += el_size )
                        if( compare( next_ptr, i_ptr ) < 0 )
                                break;

                /*
                ** If we went all the way to the end of the sorted part of the
                ** array, then the next element should go after those that are
                ** already sorted.  That's where it is right now, so we're done.
                */
                if( i == next_element )
                        continue;

                /*
                ** Otherwise, we must insert the next element before the one
                ** that i points to.  First, copy the next element into the
                ** temporary array.
                */
                memcpy( temp, next_ptr, el_size );

                /*
                ** Now copy the elements from i to the end of the sorted part of
                ** the array to the right one place.
                */
                memmove( i_ptr + el_size, i_ptr, ( next_element – i ) * el_size );

                /*
                ** Finally, the next element is inserted.
                */
                memcpy( i_ptr, temp, el_size );
        }

        free( temp );
}
```

Solution 16.10 ins_sort.c

17

Classic Abstract Data Types

17.1 Questions

1. A stack, because the values popped off of a stack come off in the opposite order from which they were pushed.
2. A queue is best because it keeps the oldest stock at the front so that it will be purchased before it goes bad. A stack would put the new stock at the front, and the cartons in the back might languish until the milk turned solid.
4. No, because the client can easily do this already:

```
while( !is_empty() )
        pop();
```

   The only advantage to be gained by having a function in the stack module to empty the stack is that it could do the job more efficiently; for example,

```
void
empty( void )
{
        top_element = -1;
}
```

5. `top_element` would have to be initialized to zero so that the first value pushed would be stored in the array. Then `top` would have to be changed appropriately:

```
return stack[ top_element - 1 ];
```

   With these changes, the module would work correctly, albeit slightly less efficiently.

6. Clients would be able to push too many values, overflowing the array. They could pop more values than the stack actually held, which would cause subsequent calls to `top` to access memory outside of the array.
8. No. If the function were written like this:

```
assert( !is_empty() );
free( stack );
```

```
stack = stack->next;
```

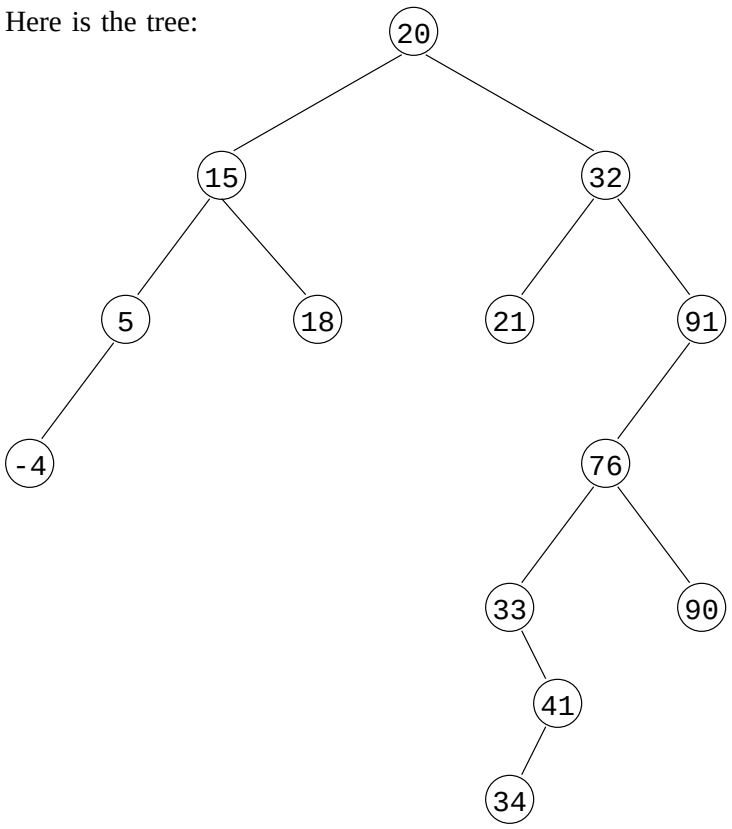
the last statement would be incorrect, as it accesses dynamically allocated memory that has already been freed.

10. They will both work, though the logic for using a separate counter is simpler. Leaving one array element unused wastes that space; if the elements are large this could be considerable.

11. This is tricky, as there are four cases to consider: the queue may be empty or nonempty, and `front` may be before or after `rear`. The modulo operation gives the correct answer when the queue is empty.

```
if( front <= rear )
        n_values = rear – front + 1;
else
        n_values = queue_size – front + rear + 1;
n_values %= QUEUE_SIZE;
```

13. Here is the tree:



14. It is the same as a sequential search of a linked list, because that is what the tree actually is.

15. Pre-order: 54 36 22 16 25 41 40 51 72 61 80 73.
In-order: 16 22 25 36 40 41 51 54 61 72 73 80.
Post-order: 16 25 22 40 51 41 36 61 73 80 72 54.
Breadth-first: 54 36 72 22 41 61 80 16 25 40 51 73.

16.
```
if( current < ARRAY_SIZE && tree[ current ] != 0 ){
        do_pre_order_traverse(
            left_child( current ), callback );
        callback( tree[ current ] );
        do_pre_order_traverse(
            right_child( current ), callback );
}
```

17.
```
if( current < ARRAY_SIZE && tree[ current ] != 0 ){
        do_pre_order_traverse(
            left_child( current ), callback );
        do_pre_order_traverse(
            right_child( current ), callback );
        callback( tree[ current ] );
}
```

19. A post-order traversal is the most convenient because it processes (deletes) the children of a node before deleting the node itself.

17.2 Programming Exercises

1. This function must allocate space for the new stack and copy the values from the old stack to the new. It must then free the old array. An assertion is used to ensure that the new array is large enough to hold all the data currently on the stack.

```
/*
** Resize the array holding the stack.
*/
void
resize_stack( size_t new_size )
{
        STACK_TYPE      *old_stack;
        int             i;

        /*
        ** Make sure the new size is large enough to hold the data already on
        ** the stack.  Then save the pointer to the old array and create a new
        ** one of the right size.
        */
        assert( new_size > top_element );
        old_stack = stack;
        stack = (STACK_TYPE *)malloc( new_size * sizeof( STACK_TYPE ) );
        assert( stack != NULL );
        stack_size = new_size;

        /*
        ** Copy values from the old array to the new one and then free the old
        ** memory.
        */
        for( i = 0; i <= top_element; i += 1 )
```

Solution 17.1 *continued . . .*

```
                stack[ i ] = old_stack[ i ];
        free( old_stack );
}
```

Solution 17.1 resize.c

2. This module is converted the same as the stack module was. The resize function is much more interesting: Not every location in the array need be copied, and it is easy for `front` and `rear` to become incorrect when the data has wrapped around the end of the array.

```
/*
** A queue implemented with a dynamically allocated array.  The array size is
** given when create is called, which must happen before any other queue
** operations are attempted.
*/
#include "queue.h"
#include <stdio.h>
#include <assert.h>

/*
**      The array that holds the values on the queue, its size, and pointers to
**      the front and rear of the queue.
*/
static  QUEUE_TYPE      *queue;
static  size_t          queue_size;
static  size_t          front = 1;
static  size_t          rear = 0;

/*
**      create_queue
*/
void
create_queue( size_t size )
{
        assert( queue_size == 0 );
        queue_size = size;
        queue = (QUEUE_TYPE *)malloc( queue_size * sizeof( QUEUE_TYPE ) );
        assert( queue != NULL );
}

/*
**      destroy_queue
*/
void
destroy_queue( void )
{
        assert( queue_size > 0 );
        queue_size = 0;
        free( queue );
        queue = NULL;
}
```

Solution 17.2 *continued . . .*

```
/*
**      resize_queue
*/
void
resize_queue( size_t new_size )
{
        QUEUE_TYPE      *old_queue;
        int             i;
        int             rear_plus_one;

        /*
        ** Make sure the new size is large enough to hold the data already on
        ** the queue.  Then save the pointer to the old array and create a new
        ** one of the right size.
        */
        if( front <= rear )
                i = rear - front + 1;
        else
                i = queue_size - front + rear + 1;
        i %= queue_size;
        assert( new_size >= i );
        old_queue = queue;
        queue = (QUEUE_TYPE *)malloc( new_size * sizeof( QUEUE_TYPE ) );
        assert( queue != NULL );
        queue_size = new_size;

        /*
        ** Copy values from the old array to the new one and then free the old
        ** memory.
        */
        i = 0;
        rear_plus_one = ( rear + 1 ) % queue_size;
        while( front != rear_plus_one ){
                queue[ i ] = old_queue[ front ];
                front = ( front + 1 ) % queue_size;
                i += 1;
        }
        front = 0;
        rear = ( i + queue_size - 1 ) % queue_size;

        free( old_queue );
}

/*
**      insert
*/
void
insert( QUEUE_TYPE value )
{
        assert( !is_full() );
        rear = ( rear + 1 ) % queue_size;
        queue[ rear ] = value;
```

Solution 17.2

continued . . .

```
}

/*
**      delete
*/
void
delete( void )
{
        assert( !is_empty() );
        front = ( front + 1 ) % queue_size;
}

/*
**      first
*/
QUEUE_TYPE first( void )
{
        assert( !is_empty() );
        return queue[ front ];
}

/*
**      is_empty
*/
int
is_empty( void )
{
        assert( queue_size > 0 );
        return ( rear + 1 ) % queue_size == front;
}

/*
**      is_full
*/
int
is_full( void )
{
        assert( queue_size > 0 );
        return ( rear + 2 ) % queue_size == front;
}
```

Solution 17.2 d_queue.c

4. This is straightforward; the stack data is simply declared as arrays, and the stack number passed as an argument selects which element to manipulate.

```
/*
** Interface for a module that manages 10 stacks.
*/

#include <stdlib.h>
#define STACK_TYPE      int     /* Type of value on the stack */
```

Solution 17.4a *continued . . .*

```
/*
** push
**      Pushes a new value on the stack.  The first argument selects which
**      stack, and the second argument is the value to push.
*/
void    push( size_t stack, STACK_TYPE value );

/*
** pop
**      Pops a value off of the selected stack, discarding it.
*/
void    pop( size_t stack );

/*
** top
**      Returns the topmost value on the selected stack without changing the
**      stack.
*/
STACK_TYPE top( size_t stack );

/*
** is_empty
**      Returns TRUE if the selected stack is empty, else FALSE
*/
int     is_empty( size_t stack );

/*
** is_full
**      Returns TRUE if the selected stack is full, else FALSE
*/
int     is_full( size_t stack );
```

Solution 17.4a 10stack.h

```
/*
** A stack implemented with a dynamically allocated array.  The array size is
** given when create is called, which must happen before any other stack
** operations are attempted.
*/
#include "10stack.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>

/*
**      The maximum number of stacks handled by the module.  This can be changed
**      only by recompiling the module.
*/
#define N_STACKS        10
```

Solution 17.4b *continued . . .*

```
/*
**      This structure holds all the information for one stack: the array that
**      holds the values, its size, and a pointer to the topmost value.
*/
typedef struct  {
        STACK_TYPE      *stack;
        size_t          size;
        int             top_element;
} StackInfo;

/*
**      Here are the actual stacks.
*/
StackInfo       stacks[ N_STACKS ];

/*
**      create_stack
*/
void
create_stack( size_t stack, size_t size )
{
        assert( stack < N_STACKS );
        assert( stacks[ stack ].size == 0 );
        stacks[ stack ].size = size;
        stacks[ stack ].stack =
            (STACK_TYPE *)malloc( size * sizeof( STACK_TYPE ) );
        assert( stacks[ stack ].stack != NULL );
        stacks[ stack ].top_element = -1;
}

/*
**      destroy_stack
*/
void
destroy_stack( size_t stack )
{
        assert( stack < N_STACKS );
        assert( stacks[ stack ].size > 0 );
        stacks[ stack ].size = 0;
        free( stacks[ stack ].stack );
        stacks[ stack ].stack = NULL;
}

/*
**      push
*/
void
push( size_t stack, STACK_TYPE value )
{
        assert( !is_full( stack ) );
        stacks[ stack ].top_element += 1;
        stacks[ stack ].stack[ stacks[ stack ].top_element ] = value;
```

Solution 17.4b *continued . . .*

```
}

/*
**      pop
*/
void
pop( size_t stack )
{
        assert( !is_empty( stack ) );
        stacks[ stack ].top_element –= 1;
}

/*
**      top
*/
STACK_TYPE top( size_t stack )
{
        assert( !is_empty( stack ) );
        return stacks[ stack ].stack[ stacks[ stack ].top_element ];
}

/*
**      is_empty
*/
int
is_empty( size_t stack )
{
        assert( stack < N_STACKS );
        assert( stacks[ stack ].size > 0 );
        return stacks[ stack ].top_element == –1;
}

/*
**      is_full
*/
int
is_full( size_t stack )
{
        assert( stack < N_STACKS );
        assert( stacks[ stack ].size > 0 );
        return stacks[ stack ].top_element == stacks[ stack ].size – 1;
}
```

Solution 17.4b d_10stak.c

5. Because two subtrees might have to be traversed, a recursive function is appropriate. This function is for the linked implementation.

```
/*
** Count the number of nodes in a linked binary search tree.
*/
```

Solution 17.5 *continued . . .*

```
/*
**      This helper function takes the root of the tree we're currently working
**      on as an argument.
*/
int
count_nodes( TreeNode *tree )
{
        if( tree == NULL )
                return 0;

        return 1 + count_nodes( tree->left ) + count_nodes( tree->right );
}

int
number_of_nodes()
{
        return count_nodes( tree );
}
```

Solution 17.5 count.c

7. Each node's value must be checked to see that it is not too large or too small. One way of doing this is to pass the minimum and maximum allowable values to each recursive call of the function. The problem with this is initialization for the first call: the solution shown uses the maximum and minimum integer constants to solve the problem. It may not be this easy with other data types. This function is written for the linked implementation.

```
/*
** Check a linked binary search tree for validity.
*/

/*
**      This helper function checks the validity of one node, using minimum and
**      maximum values passed in from the caller.
*/
int
check_bst_subtree( TreeNode *node, int min, int max )
{
        /*
        ** Empty trees are always valid.
        */
        if( node == NULL )
                return TRUE;

        /*
        ** Check the validity of this node.
        */
        if( node->value < min || node->value > max )
                return FALSE;

        /*
```

Solution 17.7 *continued . . .*

```
        ** Check the validity of the subtrees.
        */
        if( !check_bst_subtree( node->left, min, node->value - 1 ) ||
            !check_bst_subtree( node->right, node->value + 1, max ) )
                return FALSE;

        return TRUE;
}

/*
**      Check the validity of a binary search tree.
*/
int
check_bst_tree()
{
        return check_bst_subtree( tree, INT_MIN, INT_MAX );
}
```

Solution 17.7 chk_bst.c

8. This is difficult, but recursion helps considerably.

```
/*
** Delete a node from an arrayed binary search tree
*/
void
delete( TREE_TYPE value )
{
        int     current;
        int     left;
        int     right;
        int     left_subtree_empty;
        int     right_subtree_empty;

        /*
        ** First, locate the value.  It must exist in the tree or this routine
        ** will abort the program.
        */
        current = 1;

        while( tree[ current ] != value ){
                if( value < tree[ current ] )
                        current = left_child( current );
                else
                        current = right_child( current );
                assert( current < ARRAY_SIZE );
                assert( tree[ current ] != 0 );
        }

        /*
        ** We've found the value.  If it is a leaf, simply set it to zero.
        ** Otherwise, if its left subtree is not empty, replace the node's value
```

Solution 17.8 *continued . . .*

```
        ** with the rightmost (largest) child from its left subtree, and then
        ** delete that node. Otherwise, replace the value with the leftmost
        ** (smallest) child from its right subtree, and delete that node.
        */
        left = left_child( current );
        right = right_child( current );
        left_subtree_empty = left > ARRAY_SIZE || tree[ left ] == 0;
        right_subtree_empty = right > ARRAY_SIZE || tree[ right ] == 0;

        if( left_subtree_empty && right_subtree_empty )
                /*
                ** The value has no children; simply set it to zero.
                */
                tree[ current ] = 0;
        else {
                int     this_child;
                int     next_child;

                if( !left_subtree_empty ){
                        /*
                        ** The left subtree is nonempty.  Find its rightmost
                        ** child.
                        */
                        this_child = left;
                        next_child = right_child( this_child );

                        while( next_child < ARRAY_SIZE
                            && tree[ next_child ] != 0 ){
                                this_child = next_child;
                                next_child = right_child( this_child );
                        }
                }
                else {
                        /*
                        ** The right subtree is nonempty.  Find its leftmost
                        ** child.
                        */
                        this_child = right;
                        next_child = left_child( this_child );

                        while( next_child < ARRAY_SIZE
                            && tree[ next_child ] != 0 ){
                                this_child = next_child;
                                next_child = left_child( this_child );
                        }
                }

                /*
                ** Delete the child and replace the current value with
                ** this_child's value.
                */
                value = tree[ this_child ];
```

Solution 17.8 *continued . . .*

```
                delete( value );
                tree[ current ] = value;
        }
}
```

Solution 17.8 a_t_del.c

9. This is best done with a post-order traversal. Unfortunately, the interface for the traversal functions passes a pointer to the node's value rather than a pointer to the node containing the value, so we must write our own.

```
/*
** Destroy a linked binary search tree.
*/

/*
**      Do one level of a post-order traverse to destroy the tree.  This helper
**      function is needed to save the information of which node we're currently
**      processing; this is not a part of the client's interface.
*/
static void
do_destroy_tree( TreeNode *current )
{
        if( current != NULL ){
                do_destroy_tree( current->left );
                do_destroy_tree( current->right );
                free( current );
        }
}

/*
**      Destroy the entire tree.
*/
void
destroy_tree()
{
        do_destroy_tree( tree );
}
```

Solution 17.9 l_t_dstr.c

10. This is slightly easier than the arrayed tree deletion because we can rearrange values merely by changing pointers; the values do not need to be moved in the array.

```
/*
** Delete a node from a linked binary search tree
*/
void
delete( TREE_TYPE value )
{
        TreeNode        *current;
```

Solution 17.10 *continued . . .*

```
        TreeNode        **link;

        /*
        ** First, locate the value.  It must exist in the tree or this routine
        ** will abort the program.
        */
        link = &tree;

        while( (current = *link) != NULL && value != current->value ){
                if( value < current->value )
                        link = &current->left;
                else
                        link = &current->right;
        }
        assert( current != NULL );

        /*
        ** We've found the value.  See how many children it has.
        */
        if( current->left == NULL && current->right == NULL ){
                /*
                ** It is a leaf; no children to worry about!
                */
                *link = NULL;
                free( current );
        }
        else if( current->left == NULL || current->right == NULL ){
                /*
                ** The node has only one child, so the parent will simply
                ** inherit it.
                */
                if( current->left != NULL )
                        *link = current->left;
                else
                        *link = current->right;
                free( current );
        }
        else {
                /*
                ** The node has two children!  Replace its value with the
                ** largest value from its left subtree, and then delete that
                ** node instead.
                */
                TreeNode        *this_child;
                TreeNode        *next_child;

                this_child = current->left;
                next_child = this_child->right;
                while( next_child != NULL ){
                        this_child = next_child;
                        next_child = this_child->right;
                }
```

Solution 17.10 *continued . . .*

```
                /*
                ** Delete the child and replace the current value with
                ** this_child's value.
                */
                value = this_child->value;
                delete( value );
                current->value = value;
        }
}
```

Solution 17.10 l_t_del.c

```
/*
** GENERIC implementation of a stack with a static array.  The array size is
** given as one of the arguments when the stack is instantiated.
*/
#include <assert.h>

/*
** Interface
**
**      This macro declares the prototypes and data types needed for a stack of
**      one specific type.  It should be invoked ONCE (per source file) for each
**      different stack type used in that source file.  The suffix is appended
**      to each of the function names; it must be chosen by the user so as to
**      give unique names for each different type used.
*/
#define GENERIC_STACK_INTERFACE( STACK_TYPE, SUFFIX )           \
        typedef struct {                                        \
                STACK_TYPE      *stack;                         \
                int             top_element;                    \
                int             stack_size;                     \
        } STACK##SUFFIX;                                        \
        void push##SUFFIX( STACK##SUFFIX *stack, STACK_TYPE value );\
        void pop##SUFFIX( STACK##SUFFIX *stack );               \
        STACK_TYPE top##SUFFIX( STACK##SUFFIX *stack );         \
        int is_empty##SUFFIX( STACK##SUFFIX *stack );           \
        int is_full##SUFFIX( STACK##SUFFIX *stack );

/*
** Implementation
**
**      This macro defines the functions to manipulate a stack of a specific
**      type.  It should be invoked ONCE (per entire program) for each different
**      stack type used in the program.  The suffix must be the same one used in
**      the interface declaration.
*/
#define GENERIC_STACK_IMPLEMENTATION( STACK_TYPE, SUFFIX )      \
                                                                \
        void                                                    \
        push##SUFFIX( STACK##SUFFIX *stack, STACK_TYPE value )  \
```

Solution 17.11 *continued . . .*

```
        {                                                       \
                assert( !is_full##SUFFIX( stack ) );            \
                stack->top_element += 1;                        \
                stack->stack[ stack->top_element ] = value;     \
        }                                                       \
                                                                \
        void                                                    \
        pop##SUFFIX( STACK##SUFFIX *stack )                     \
        {                                                       \
                assert( !is_empty##SUFFIX( stack ) );           \
                stack->top_element -= 1;                        \
        }                                                       \
                                                                \
        STACK_TYPE top##SUFFIX( STACK##SUFFIX *stack )          \
        {                                                       \
                assert( !is_empty##SUFFIX( stack ) );           \
                return stack->stack[ stack->top_element ];      \
        }                                                       \
                                                                \
        int                                                     \
        is_empty##SUFFIX( STACK##SUFFIX *stack )                \
        {                                                       \
                return stack->top_element == -1;                \
        }                                                       \
                                                                \
        int                                                     \
        is_full##SUFFIX( STACK##SUFFIX *stack )                 \
        {                                                       \
                return stack->top_element == stack->stack_size - 1; \
        }

/*
** Stacks
**
**      This macro creates the data needed for a single stack.  It is invoked
**      once per stack.  NAME is the name by which you refer to the stack in
**      subsequent function calls, and STACK_SIZE is the size you want the stack
**      to be.  STACK_TYPE is the type of data stored on the stack, and the
**      SUFFIX must be the same one given in the interface declaration for this
**      STACK_TYPE.
*/
#define GENERIC_STACK( NAME, STACK_SIZE, STACK_TYPE, SUFFIX )   \
        static  STACK_TYPE      NAME##stack[ STACK_SIZE ];      \
        STACK##SUFFIX   NAME = { NAME##stack, -1, STACK_SIZE };
```

Solution 17.11 g_stack2.h

18

Runtime Environment

18.1 Questions

1. The answer depends on the specific environment. RISC architectures often have interesting strategies for handling parameter passing, though the semantics of C do not always allow the compiler writer to take advantage of all of them.
2. The answer depends on the specific environment.
3. The answer depends on the specific environment.
4. The answer depends on the specific environment.
7. Assembly language programs *can be* more efficient in C programs only in their size and speed. What is far more important is that the programmer is much more efficient using a high level language than using an assembly language. The assembly language programmer is unlikely to ever complete a piece of software of the scale common today, so the fact that the code is small and fast is irrelevant.

18.2 Programming Exercises

1. The answer depends on the specific environment.
2. The answer depends on the specific environment.