Appendix G Answers

This appendix lists the answers for the quiz and exercise sections at the end of each chapter. Note that for many of the exercises, more than one solution is possible. In most cases, only one of many possible answers is listed. In other cases, you'll see additional information to help you solve the exercise.

Answers for Chapter 1

Quiz

1. C is powerful, popular, and portable.
2. The compiler translates C source code into machine-language instructions that your computer can understand.
3. Editing, compiling, linking, and testing.
4. The answer to this question depends on your compiler. Consult your compiler's manuals.
5. The answer to this question depends on your compiler. Consult your compiler's manuals.
6. The appropriate extension for C source files is .C (or .c).
Note: C++ uses the extension .CPP. You can write and compile your C programs with a .CPP extension, but it's more appropriate to use a .C extension.
7. FILENAME.TXT would compile. However, it's more appropriate to use a .C extension rather than .TXT.
8. You should make changes to the source code to correct the problems. You should then recompile and relink. After relinking, you should run the program again to see whether your corrections fixed the program.
9. Machine language is digital, or binary, instructions that the computer can understand. Because the computer can't understand C source code, a compiler translates source code to machine code, also called object code.
10. The linker combines the object code from your program with the object code from the function library and creates an executable file.

Exercises

1. When you look at the object file, you see many unusual characters and other gibberish. Mixed in with the gibberish, you also see pieces of the source file.
2. This program calculates the area of a circle. It prompts the user for the radius and then displays the area.
3. This program prints a 10*10 block made of the character X. A similar program is covered on Chapter 6, "Basic Program Control."
4. This program generates a compiler error. You should get a message similar to the following:

```
Error: ch1ex4.c: Declaration terminated incorrectly
```

This error is caused by the semicolon at the end of line 3. If you remove the semicolon, this program should compile and link correctly.
5. This program compiles OK, but it generates a linker error. You should get a message similar to the following:

```
Error: Undefined symbol _do_it in module...
```

This error occurs because the linker can't find a function called do_it. To fix this program, change do_it to printf.
6. Rather than printing a 10*10 block filled with the character X, the program now prints a 10*10 block of smiley faces.

Answers for Chapter 2

Quiz

1. A block.
2. The main() function.
3. Any text between /* and */ is a program comment and is ignored by the compiler. You use program comments to make notations about the program's structure and operation.
4. A function is an independent section of program code that performs a certain task and has been assigned a name. By using a function's name, a program can execute the code in the function.
5. A user-defined function is created by the programmer. A library function is supplied with the C compiler.

6. An #include directive instructs the compiler to add the code from another file into your source code during the compilation process.
7. Comments shouldn't be nested. Although some compilers let you to do this, others don't. To keep your code portable, you shouldn't nest comments.
8. Yes. Comments can be as long as needed. A comment starts with /* and doesn't end until a */ is encountered.
9. An include file is also known as a header file.
10. An include file is a separate disk file that contains information needed by the compiler to use various functions.

Exercises

1. Remember, only the main() function is required in C programs. The following is the smallest possible program, but it doesn't do anything:

```
void main()
{
}
```

This also could be written

```
void main(){}
```

2. a. Statements are on lines 8, 9, 10, 12, 20, and 21.
b. The only variable definition is on line 18.
c. The only function prototype (for display_line()) is on line 4.
d. The function definition for display_line() is on lines 16 through 22.
e. Comments are on lines 1, 15, and 23.
3. A comment is any text included between /* and */. Examples include the following:

```
/* This is a comment */
/*???*/
/*
This is a
     third comment */
```

4. This program prints the alphabet in all capital letters. You should understand this program better after you finish Chapter 10, "Characters and Strings."
The output is

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

5. This program counts and prints the number of characters and spaces you enter. This program also will be clearer after you finish Chapter 10.

Answers for Chapter 3

Quiz

1. An integer variable can hold a whole number (a number without a fractional part), and a floating-point variable can hold a real number (a number with a fractional part).
2. A type double variable has a greater range than type float (it can hold larger and smaller values). A type double variable also is more precise than type float.
3. a. The size of a char is 1 byte.
b. The size of a short is less than or equal to the size of an int.
c. The size of an int is less than or equal to the size of a long.
d. The size of an unsigned is equal to the size of an int.
e. The size of a float is less than or equal to the size of a double.
4. The names of symbolic constants make your source code easier to read. They also make it much easier to change the constant's value.
5. a. #define MAXIMUM 100
b. const int MAXIMUM = 100;
6. Letters, numerals, and underscores.
7. Names of variables and constants should describe the data being stored. Variable names should be in lowercase, and constant names should be in uppercase.
8. Symbolic constants are symbols that represent literal constants.

9. If it's an unsigned int that is two bytes long, the minimum value it can hold is 0. If it is signed, -32,768 is the minimum.

Exercises

1. a. Because a person's age can be considered a whole number, and a person can't be a negative age, an unsigned int is suggested.
b. unsigned int
c. float
d. If your expectations for yearly salary aren't very high, a simple unsigned int variable would work. If you feel you have the potential to go above $65,535, you probably should use a long. (Have faith in yourself; use a long.)
e. float. (Don't forget the decimal places for the cents.)
f. Because the highest grade will always be 100, it is a constant. Use either const int or a #define statement.
g. float. (If you're going to use only whole numbers, use either int or long.)
h. Definitely a signed field. Either int, long, or float. See answer 1.d.
i. double
2. Answers for exercises 2 and 3 are combined here.
Remember, a variable name should be representative of the value it holds. A variable declaration is the statement that initially creates the variable. The declaration might or might not initialize the variable to a value. You can use any name for a variable, except the C keywords.
a. unsigned int age;
b. unsigned int weight;
c. float radius = 3;
d. long annual_salary;
e. float cost = 29.95;
f. const int max_grade = 100; or #define MAX_GRADE 100
g. float temperature;
h. long net_worth = -30000;
i. double star_distance;
3. See answer 2.
4. The valid variable names are b, c, e, g, h, i, and j.
Notice that j is correct; however, it isn't wise to use variable names that are this long. (Besides, who would want to type them?) Most compilers wouldn't look at this entire name. Instead, they would look only at the first 31 characters or so.
The following are invalid:
a. You can't start a variable name with a number.
d. You can't use a pound sign (#) in a variable name.
f. You can't use a hyphen (-) in a variable name.

Answers for Chapter 4

Quiz

1. It is an assignment statement that instructs the computer to add 5 and 8, assigning the result to the variable x.
2. An expression is anything that evaluates to a numerical value.
3. The relative precedence of the operators.
4. After the first statement, the value of a is 10, and the value of x is 11. After the second statement, both a and x have the value 11. (The statements must be executed separately.)
5. 1
6. 19
7. (5 + 3) * 8 / (2 + 2)
8. 0
9. See the section "Operator Precedence Revisited" near the end of this chapter. It shows the C operators and their precedence.
a. < has higher precedence than ==
b. * has higher precedence than +
c. != and == have the same precedence, so they are evaluated from left to right.

d. >= has the same precedence as >. Use parentheses if you need to use more than one relational operator in a single statement or expression.

10. The compound assignment operators let you combine a binary mathematical operation with an assignment operation, thus providing a shorthand notation. The compound operators presented in this chapter are +=, -=, /=, *=, and %=.

Exercises

1. This listing should have worked even though it is poorly structured. The purpose of this listing is to demonstrate that white space is irrelevant to how the program runs. You should use white space to make your programs readable.

2. The following is a better way to structure the listing from exercise 1:

```
#include <stdio.h>
int x, y;
main()
{
    printf("\nEnter two numbers ");
    scanf( "%d %d",&x,&y);
    printf("\n\n%d is bigger\n",(x>y)?x:y);
    return 0;
}
```

This listing asks for two numbers, x and y, and then prints whichever one is bigger.

3. The only changes needed in Listing 4.1 are the following:

```
16:     printf("\n%d   %d", a++, ++b);
17:     printf("\n%d   %d", a++, ++b);
18:     printf("\n%d   %d", a++, ++b);
19:     printf("\n%d   %d", a++, ++b);
20:     printf("\n%d   %d", a++, ++b);
```

4. The following code fragment is just one of many possible answers. It checks to see if x is greater than or equal to 1 and less than or equal to 20. If these two conditions are met, x is assigned to y. If these conditions are not met, x is not assigned to y; therefore, y remains the same.

```
if ((x >= 1) && (x <= 20))
    y = x;
```

5. The code is as follows:

```
y = ((x >= 1) && (x <= 20)) ? x : y;
```

Again, if the statement is TRUE, x is assigned to y; otherwise, y is assigned to itself, thus having no effect.

6. The code is as follows:

```
if (x < 1 && x > 10 )
statement;
```

7. a. 7
b. 0
c. 9
d. 1 (true)
e. 5

8. a. TRUE
b. FALSE
c. TRUE. Notice that there is a single equals sign, making the if an assignment instead of a relation.
d. TRUE

9. The following is one possible answer:

```
if( age < 21 )
    printf( "You are not an adult" );
else if( age >= 65 )
        printf( "You are a senior citizen!");
     else
        printf( "You are an adult" );
```

10. This program had four problems. The first is on line 3. The assignment statement should end with a semicolon, not a colon. The second problem is the semicolon at the end of the if statement on line 6. The

third problem is a common one: The assignment operator (=) is used rather than the relational operator (==) in the if statement. The final problem is the word otherwise on line 8. This should be else. Here is the corrected code:

```
/* a program with problems... */
#include <stdio.h>
int x = 1;
main()
{
    if( x == 1)
       printf(" x equals 1" );
    else
       printf(" x does not equal 1");
    return 0;
 }
```

Answers for Chapter 5

Quiz

1. Yes! (Well, OK, this is a trick question, but you better answer "yes" if you want to become a good C programmer.)
2. Structured programming takes a complex programming problem and breaks it into a number of simpler tasks that are easier to handle one at a time.
3. After you've broken your program into a number of simpler tasks, you can write a function to perform each task.
4. The first line of a function definition must be the function header. It contains the function's name, its return type, and its parameter list.
5. A function can return either one value or no values. The value can be any of the C variable types. On Chapter 18, "Getting More from Functions," you'll see how to get more values back from a function.
6. A function that returns nothing should be type void.
7. A function definition is the complete function, including the header and all the function's statements. The definition determines what actions take place when the function executes. The prototype is a single line, identical to the function header, but it ends with a semicolon. The prototype informs the compiler of the function's name, return type, and parameter list.
8. A local variable is declared within a function.
9. Local variables are independent from other variables in the program.
10. main() should be the first function in your listing.

Exercises

1. float do_it(char a, char b, char c)
Add a semicolon to the end, and you have the function prototype. As a function header, it should be followed by the function's statements enclosed in braces.
2. void print_a_number(int a_number)
This is a void function. As in exercise 1, to create the prototype, add a semicolon to the end. In an actual program, the header is followed by the function's statements.
3. a. int
b. long
4. There are two problems. First, the print_msg() function is declared as a void; however, it returns a value. The return statement should be removed. The second problem is on the fifth line. The call to print_msg() passes a parameter (a string). The prototype states that this function has a void parameter list and therefore shouldn't be passed anything. The following is the corrected listing:

```
#include <stdio.h>
void print_msg (void);
main()
{
    print_msg();
    return 0;
```

```
}
void print_msg(void)
{
    puts( "This is a message to print" );
}
```
5. There should not be a semicolon at the end of the function header.
6. Only the larger_of() function needs to be changed:
```
21: int larger_of( int a, int b)
22: {
23:     int save;
24:
25:     if (a > b)
26:         save = a;
27:     else
28:         save = b;
29:
30:     return save;
31: }
```
7. The following assumes that the two values are integers and an integer is returned:
```
int product( int x, int y )
{
    return (x * y);
}
```
8. The following listing checks the second value passed to verify that it is not 0. Division by zero causes an error. You should never assume that the values passed are correct.
```
int divide_em( int a, int b )
{
    int answer = 0;
    if( b == 0 )
        answer = 0;
    else
        answer = a/b;
    return answer;
}
```
9. Although the following code uses main(), it could use any function. Lines 9, 10, and 11 show the calls to the two functions. Lines 13 through 16 print the values. To run this listing, you need to include the code from exercises 7 and 8 after line 19.

```
1:  #include <stdio.h>
2:
3:  main()
4:  {
5:     int number1 = 10,
6:         number2 = 5;
7:     int x, y, z;
8:
9:     x = product( number1, number2 );
10:    y = divide_em( number1, number2 );
11:    z = divide_em( number1, 0 );
12:
13:     printf( "\nnumber1 is %d and number2 is %d", number1, number2 );
14:     printf( "\nnumber1 * number2 is %d", x );
15:     printf( "\nnumber1 / number2 is %d", y );
16:     printf( "\nnumber1 / 0 is %d", z );
17:
18:     return 0;
```

```
19: }
```
10. The code is as follows:
```
/* Averages five float values entered by the user. */
#include <stdio.h>
float v, w, x, y, z, answer;
float average(float a, float b, float c, float d, float e);
main()
{
    puts("Enter five numbers:");
    scanf("%f%f%f%f%f", &v, &w, &x, &y, &z);
    answer = average(v, w, x, y, z);
    printf("The average is %f.\n", answer);
    return 0;
}
float average( float a, float b, float c, float d, float e)
{
    return ((a+b+c+d+e)/5);
}
```
11. The following is the answer using type int variables. It can run only with values less than or equal to 9. To use values larger than 9, you need to change the values to type long.
```
/* this is a program with a recursive function */
#include <stdio.h>
int three_powered( int power );
main()
{
    int a = 4;
    int b = 9;
    printf( "\n3 to the power of %d is %d", a,
    three_powered(a) );
    printf( "\n3 to the power of %d is %d\n", b,
    three_powered(b) );
    return 0;
}
int three_powered( int power )
{
    if ( power < 1 )
        return( 1 );
    else
        return( 3 * three_powered( power - 1 ));
}
```

Answers for Chapter 6

Quiz

1. The first index value of an array in C is 0.
2. A for statement contains initializing and increment expressions as parts of the command.
3. A do...while contains the while statement at the end and always executes the loop at least once.
4. Yes, a while statement can accomplish the same task as a for statement, but you need to do two additional things. You must initialize any variables before starting the while command, and you need to increment any variables as a part of the while loop.
5. You can't overlap the loops. The nested loop must be entirely inside the outer loop.
6. Yes, a while statement can be nested in a do...while loop. You can nest any command within any other command.

7. The four parts of a for statement are the initializer, the condition, the increment, and the statement(s).
8. The two parts of a while statement are the condition and the statement(s).
9. The two parts of a do...while statement are the condition and the statement(s).

Exercises

1. long array[50];
2. Notice that in the following answer, the 50th element is indexed to 49. Remember that arrays start at 0.

```
array[49] = 123.456;
```

3. When the statement is complete, x equals 100.
4. When the statement is complete, ctr equals 11. (ctr starts at 2 and is incremented by 3 while it is less than 10.)
5. The inner loop prints five Xs. The outer loop prints the inner loop 10 times. This totals 50 Xs.
6. The code is as follows:

```
int x;
for( x = 1; x <= 100; x += 3) ;
```

7. The code is as follows:

```
int x = 1;
while( x <= 100 )
    x += 3;
```

8. The code is as follows:

```
int ctr = 1;
do
{
    ctr += 3;
} while( ctr < 100 );
```

9. This program never ends. record is initialized to 0. The while loop then checks to see whether record is less than 100. 0 is less than 100, so the loop executes, thus printing the two statements. The loop then checks the condition again. 0 is still, and always will be, less than 100, so the loop continues. Within the brackets, record needs to be incremented. You should add the following line after the second printf() function call:

```
record++;
```

10. Using a defined constant is common in looping; you'll see examples similar to this code fragment in Weeks 2 and 3. The problem with this fragment is simple. The semicolon doesn't belong at the end of the for statement. This is a common bug.

Answers for Chapter 7

Quiz

1. There are two differences between puts() and printf():

- printf() can print variable parameters.
- puts() automatically adds a newline character to the end of the string it prints.

2. You should include the STDIO.H header file when using printf().
3. a. \\ prints a backslash.
b. \b prints a backspace.
c. \n prints a newline.
d. \t prints a tab.
e. \a (for "alert") sounds the beep.
4. a. %s is used for a character string.
b. %d is used for a signed decimal integer.
c. %f is used for a decimal floating-point number.
5. a. b prints the literal character b.
b. \b prints a backspace character.
c. \ looks at the next character to determine an escape character (see Table 7.1).
d. \\ prints a single backslash.

Exercises

1. puts() automatically adds the newline; printf() does not. The code is as follows:

```
printf( "\n" );
puts( " " );
```

2. The code is as follows:

```
char c1, c2;
int d1;
scanf( "%c %ud %c", &c1, &d1, &c2 );
```

3. Your answer might vary:

```
#include <stdio.h>
int x;
main()
{
    puts( "Enter an integer value" );
    scanf( "%d", &x );
    printf( "\nThe value entered is %d\n", x );
    return 0;
}
```

4. It's typical to edit a program to allow only specific values to be accepted. The following is one way to accomplish this exercise:

```
#include <stdio.h>
int x;
main()
{
    puts( "Enter an even integer value" );
    scanf( "%d", &x );
    while( x % 2 != 0)
    {
        printf( "\n%d is not even, Please enter an even
number: ", x );
        scanf( "%d", &x );
    }
    printf( "\nThe value entered is %d\n", x );
    return 0;
}
```

5. The code is as follows:

```
#include <stdio.h>
int array[6], x, number;
main()
{
    /* loop 6 times or until the last entered element is 99
*/
    for( x = 0; x < 6 && number != 99; x++ )
    {
        puts( "Enter an even integer value, or 99 to quit"
);
        scanf( "%d", &number );
        while( number % 2 == 1 && number != 99)
       {
            printf( "\n%d is not even, Please enter an even
number: ", number);
            scanf( "%d", &number );
```

```
        }
        array[x] = number;
    }
    /* now print them out... */
    for( x = 0; x < 6 && array[x] != 99; x++ )
    {
        printf( "\nThe value entered is %d", array[x] );
    }
    return 0;
}
```

6. The previous answers already are executable programs. The only change that needs to be made is in the final printf(). To print each value separated by a tab, change the printf() statement to the following:

```
printf( "%d\t", array[x]);
```

7. You can't have quotes within quotes. To print quotes within quotes, you must use the escape character \". Additionally, you must include a single slash at the end of the first line in order to have the text continued to the second line. The following is the corrected version:

```
printf( "Jack said, \"Peter Piper picked a peck of pickled
peppers.\"");
```

8. This listing has three errors. The first is the lack of quotes in the printf() statement. The second is the missing address-of operator in the answer variable in the scanf(). The final error is also in the scanf() statement. Instead of "%f", it should have "%d", because answer is a type int variable, not a type float. The following is corrected:

```
int get_1_or_2( void )
{
    int answer = 0;
    while( answer < 1 || answer > 2 )
    {
        printf("Enter 1 for Yes, 2 for No ");     /* corrected */
        scanf( "%d", &answer );                  /* corrected */
    }
    return answer;
}
```

9. Here is the completed print_report() function for Listing 7.1:

```
void print_report( void )
{
    printf( "\nSAMPLE REPORT" );
    printf( "\n\nSequence\tMeaning" );
    printf( "\n=========\t=======" );
    printf( "\n\\a\t\tbell (alert)" );
    printf( "\n\\b\t\tbackspace" );
    printf( "\n\\n\t\tnew line" );
    printf( "\n\\t\t\thorizontal tab" );
    printf( "\n\\\\\t\tbackslash" );
    printf( "\n\\\?\t\tquestion mark" );
    printf( "\n\\\'\t\tsingle quote" );
    printf( "\n\\\"\t\tdouble quote" );
    printf( "\n...\t\t...");
}
```

10. The code is as follows:

```
/* Inputs two floating-point values and */
/* displays their product. */
#include <stdio.h>
float x, y;
main()
{
```

```
    puts("Enter two values: ");
    scanf("%f %f", &x, &y);
    printf("\nThe product is %f\n", x * y);
    return 0;
}
```

11. The following program prompts for 10 integers and displays their sum:

```
/* Input 10 integers and display their sum. */
#include <stdio.h>
int count, temp;
long total = 0;     /* Use type long to ensure we don't */
/* exceed the maximum for type int. */
main()
{
    for (count = 1; count <=10; count++)
    {
        printf("Enter integer # %d: ", count);
        scanf("%d", &temp);
        total += temp;
    }
    printf("\n\nThe total is %d\n", total);
    return 0;
}
```

12. The code is as follows:

```
/* Inputs integers and stores them in an array, stopping */
/* when a zero is entered. Finds and displays the array's */
/* largest and smallest values */
#include <stdio.h>
#define MAX 100
int array[MAX];
int count = -1, maximum, minimum, num_entered, temp;
main()
{
    puts("Enter integer values one per line.");
    puts("Enter 0 when finished.");
    /* Input the values */
    do
    {
        scanf("%d", &temp);
        array[++count] = temp;
    } while ( count < (MAX-1) && temp != 0 );
    num_entered = count;
    /* Find the largest and smallest. */
    /* First set maximum to a very small value, */
    /* and minimum to a very large value. */
    maximum = -32000;
    minimum = 32000;
    for (count = 0; count <= num_entered && array[count] != 0; count++)
    {
        if (array[count] > maximum)
        maximum = array[count];
        if (array[count] < minimum )
            minimum = array[count];
    }
    printf("\nThe maximum value is %d", maximum);
    printf("\nThe minimum value is %d\n", minimum);
    return 0;
}
```

Answers for Chapter 8

Quiz

1. All of them, but one at a time. A given array can contain only a single data type.
2. 0. Regardless of the size of an array, all C arrays start with subscript 0.
3. *n*-1
4. The program compiles and runs but produces unpredictable results.
5. In the declaration statement, follow the array name with one set of brackets for each dimension. Each set of brackets contains the number of elements in the corresponding dimension.
6. 240. This is determined by multiplying 2 * 3 * 5 * 8.
7. array[0][0][1][1]

Exercises

1. int one[1000], two[1000], three[1000];
2. int array[10] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
3. This exercise can be solved in numerous ways. The first way is to initialize the array when it's declared:

```
int eightyeight[88] = {88,88,88,88,88,88,88,...,88};
```

However, this approach would require you to place 88 88s between the braces (instead of using ..., as I did). I don't recommend this method for initializing such a large array. The following is a better method:

```
int eightyeight[88];
int x;
for ( x = 0; x < 88; x++ )
    eightyeight[x] = 88;
```

4. The code is as follows:

```
int stuff[12][10];
int sub1, sub2;
for( sub1 = 0; sub1 < 12; sub1++ )
    for( sub2 = 0; sub2 < 10; sub2++ )
        stuff[sub1][sub2] = 0;
```

5. Be careful with this fragment. The bug presented here is easy to create. Notice that the array is 10 * 3 but is initialized as a 3 * 10 array.
To describe this differently, the left subscript is declared as 10, but the for loop uses x as the left subscript. x is incremented with three values. The right subscript is declared as 3, but the second for loop uses y as the right subscript. y is incremented with 10 values. This can cause unpredictable results. You can fix this program in one of two ways. The first way is to switch x and y in the line that does the initialization:

```
int x, y;
int array[10][3];
main()
{
   for ( x = 0; x < 3; x++ )
      for ( y = 0; y < 10; y++ )
         array[y][x] = 0;         /* changed */
   return 0;
}
```

The second way (which is recommended) is to switch the values in the for loops:

```
int x, y;
int array[10][3];
main()
{
   for ( x = 0; x < 10; x++ )      /* changed */
      for ( y = 0; y < 3; y++ )   /* changed */
         array[x][y] = 0;
   return 0;
}
```

6. This, I hope, was an easy bug to bust. This program initializes an element in the array that is out of bounds. If you have an array with 10 elements, their subscripts are 0 to 9. This program initializes elements with subscripts 1 through 10. You can't initialize array[10], because it doesn't exist. The for statement should be changed to one of the following examples:

```
for( x = 1; x <=9; x++ )  /* initializes 9 of the 10
elements */
for( x = 0; x <= 9; x++ )
```

Note that x <= 9 is the same as x < 10. Either is appropriate; x < 10 is more common.

7. The following is one of many possible answers:

```
/* Using two-dimensional arrays and rand() */
#include <stdio.h>
#include <stdlib.h>
/* Declare the array */
int array[5][4];
int a, b;
main()
{
   for ( a = 0; a < 5; a++ )
   {
      for ( b = 0; b < 4; b++ )
      {
         array[a][b] = rand();
      }
   }
   /* Now print the array elements */
   for ( a = 0; a < 5; a++ )
   {
      for ( b = 0; b < 4; b++ )
      {
         printf( "%d\t", array[a][b] );
      }
      printf( "\n" );    /* go to a new line */
   }
   return 0;
}
```

8. The following is one of many possible answers:

```
/* RANDOM.C: using a single-dimensional array */
#include <stdio.h>
#include <stdlib.h>
/* Declare a single-dimensional array with 1000 elements */
int random[1000];
int a, b, c;
long total = 0;
main()
{
   /* Fill the array with random numbers. The C library */
   /* function rand() returns a random number. Use one */
   /* for loop for each array subscript. */
   for (a = 0; a < 1000; a++)
   {
      random[a] = rand();
      total += random[a];
   }
```

```
   printf("\n\nAverage is: %ld\n",total/1000);
   /* Now display the array elements 10 at a time */
   for (a = 0; a < 1000; a++)
   {
      printf("\nrandom[%d] = ", a);
      printf("%d", random[a]);
      if ( a % 10 == 0 && a > 0 )
     {
         printf("\nPress Enter to continue, CTRL-C to
quit.");
         getchar();
      }
   }
   return 0;
}        /* end of main() */
```

9. The following are two solutions. The first initializes the array at the time it is declared, and the second initializes it in a for loop.

Answer 1:

```
#include <stdio.h>
/* Declare a single-dimensional array */
int elements[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int idx;
main()
{
   for (idx = 0; idx < 10; idx++)
   {
      printf( "\nelements[%d] = %d ", idx, elements[idx] );
   }
   return 0;
}        /* end of main() */
```

Answer 2:

```
#include <stdio.h>
/* Declare a single-dimensional array */
int elements[10];
int idx;
main()
{
   for (idx = 0; idx < 10; idx++)
      elements[idx] = idx ;
   for (idx = 0; idx < 10; idx++)
      printf( "\nelements[%d] = %d ", idx, elements[idx] );
   return 0;
}
```

10. The following is one of many possible answers:

```
#include <stdio.h>
/* Declare a single-dimensional array */
int elements[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int new_array[10];
int idx;
main()
{
   for (idx = 0; idx < 10; idx++)
```

```
    {
       new_array[idx] = elements[idx] + 10 ;
    }
    for (idx = 0; idx < 10; idx++)
    {
       printf( "\nelements[%d] = %d \tnew_array[%d] = %d",
       idx, elements[idx], idx, new_array[idx] );
    }
    return 0;
}
```

Answers for Chapter 9

Quiz

1. The address-of operator is the & sign.
2. The indirection operator * is used. When you precede the name of a pointer by *, it refers to the variable pointed to.
3. A pointer is a variable that contains the address of another variable.
4. Indirection is the act of accessing the contents of a variable by using a pointer to the variable.
5. They are stored in sequential memory locations, with lower array elements at lower addresses.
6. &data[0]
data
7. One way is to pass the length of the array as a parameter to the function. The second way is to have a special value in the array, such as NULL, signify the array's end.
8. Assignment, indirection, address-of, incrementing, differencing, and comparison.
9. Differencing two pointers returns the number of elements in between. In this case, the answer is 1. The actual size of the elements in the array is irrelevant.
10. The answer is still 1.

Exercises

1. char *char_ptr;
2. The following declares a pointer to cost and then assigns the address of cost (&cost) to it:

```
int *p_cost;
p_cost = &cost;
```

3. Direct access: cost = 100;
Indirect access: *p_cost = 100;
4. printf("Pointer value: %d, points at value: %d", p_cost, *p_cost);
5. float *variable = &radius;
6. The code is as follows:

```
data[2] = 100;
*(data + 2) = 100;
```

7. This code also includes the answer for exercise 8:

```
#include <stdio.h>
#define MAX1 5
#define MAX2 8
int array1[MAX1] = { 1, 2, 3, 4, 5 };
int array2[MAX2] = { 1, 2, 3, 4, 5, 6, 7, 8 };
int total;
int sumarrays(int x1[], int len_x1, int x2[], int len_x2);
main()
{
   total = sumarrays(array1, MAX1, array2, MAX2);
   printf("The total is %d\n", total);
   return 0;
```

```
}
int sumarrays(int x1[], int len_x1, int x2[], int len_x2)
{
   int total = 0, count = 0;
   for (count = 0; count < len_x1; count++)
      total += x1[count];
   for (count = 0; count < len_x2; count++)
      total += x2[count];
   return total;
}
```

8. See the answer for exercise 7.

9. The following is just one possible answer:

```
#include <stdio.h>
#define SIZE 10
/* function prototypes */
void addarrays( int [], int []);
main()
{
   int a[SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
   int b[SIZE] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
   addarrays(a, b);
   return 0;
}
void addarrays( int first[], int second[])
{
    int total[SIZE];
    int ctr = 0;
    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
       total[ctr] = first[ctr] + second[ctr];
       printf("%d + %d = %d\n", first[ctr], second[ctr], total[ctr]);
   }
}
```

Answers for Chapter 10

Quiz

1. The values in the ASCII character set range from 0 to 255. From 0 to 127 is the standard ASCII character set, and 128 to 255 is the extended ASCII character set.

2. As the character's ASCII code.

3. A string is a sequence of characters terminated by the null character.

4. A sequence of one or more characters enclosed in double quotation marks.

5. To hold the string's terminating null character.

6. As a sequence of ASCII values corresponding to the quoted characters, followed by 0 (the ASCII code for the null character).

7. a. 97
b. 65
c. 57
d. 32
e. 206
f. 6

8. a. I
b. a space
c. c
d. a
e. n
f. NUL
g. B

9. a. 9 bytes. (Actually, the variable is a pointer to a string, and the string requires 9 bytes of memory--8 for the string and 1 for the null terminator.)
b. 9 bytes
c. 1 byte
d. 20 bytes
e. 20 bytes
10. a. A
b. A
c. 0 (NUL)
d. This is beyond the end of the string, so it could have any value.
e. !
f. This contains the address of the first element of the string.

Exercises

1. char letter = `$';
2. char array[18] = "Pointers are fun!";
3. char *array = "Pointers are fun!";
4. The code is as follows:

```
char *ptr;
ptr = malloc(81);
gets(ptr);
```

5. The following is just one possible answer. A complete program is provided:

```
#include <stdio.h>
#define SIZE 10
/* function prototypes */
void copyarrays( int [], int []);
main()
{
    int ctr=0;
    int a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[SIZE];
    /* values before copy */
    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
       printf( "a[%d] = %d, b[%d] = %d\n",
               ctr, a[ctr], ctr, b[ctr]);
    }
    copyarrays(a, b);
    /* values after copy */
    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
       printf( "a[%d] = %d, b[%d] = %d\n",
               ctr, a[ctr], ctr, b[ctr]);
    }
    return 0;
}
void copyarrays( int orig[], int newone[])
{
    int ctr = 0;
    for (ctr = 0; ctr < SIZE; ctr ++ )
    {
        newone[ctr] = orig[ctr];
    }
}
```

6. The following is one of many possible answers:

```
#include <stdio.h>
#include <string.h>
/* function prototypes */
char * compare_strings( char *, char *);
main()
{
    char *a = "Hello";
    char *b = "World!";
    char *longer;
    longer = compare_strings(a, b);
    printf( "The longer string is: %s\n", longer );
    return 0;
}
char * compare_strings( char * first, char * second)
{
    int x, y;
    x = strlen(first);
    y = strlen(second);
    if( x > y)
       return(first);
    else
       return(second);
}
```

7. This exercise was on your own!

8. a_string is declared as an array of 10 characters, but it's initialized with a string larger than 10 characters. a_string needs to be bigger.

9. If the intent of this line of code is to initialize a string, it is wrong. You should use either char *quote or char quote[100].

10. No.

11. Yes. Although you can assign one pointer to another, you can't assign one array to another. You should change the assignment to a string-copying command such as strcpy().

Answers for Chapter 11

Quiz

1. The data items in an array must all be of the same type. A structure can contain data items of different types.

2. The structure member operator is a period. It is used to access members of a structure.

3. struct

4. A structure tag is tied to a template of a structure and is not an actual variable. A structure instance is an allocated structure that can hold data.

5. These statements define a structure and declare an instance called myaddress. This instance is then initialized. The structure member myaddress.name is initialized to the string "Bradley Jones", yaddress.add1 is initialized to "RTSoftware", myaddress.add2 is initialized to "P.O. Box 1213", myaddress.city is initialized to "Carmel", myaddress.state is initialized to "IN", and myaddress.zip is initialized to "46032-1213".

6. The following statement changes ptr to point to the second array element:

```
ptr++;
```

Exercises

1. The code is as follows:

```
struct time {
    int hours;
```

```
    int minutes;
    int seconds;
} ;
```
2. The code is as follows:
```
struct data {
    int value1;
    float value2;
    float value3;
} info ;
```
3. info.value1 = 100;
4. The code is as follows:
```
struct data *ptr;
ptr = &info;
```
5. The code is as follows:
```
ptr->value2 = 5.5;
(*ptr).value2 = 5.5;
```
6. The code is as follows:
```
struct data {
    char name[21];
    struct data *ptr;
};
```
7. The code is as follows:
```
typedef struct {
    char address1[31];
    char address2[31];
    char city[11];
    char state[3];
    char zip[11];
} RECORD;
```
8. The following uses the values from quiz question 5 for the initialization:
```
RECORD myaddress = {"RTSoftware",
                    "P.O. Box 1213",
                    "Carmel", "IN", "46032-1213" };
```
9. This code fragment has two problems. The first is that the structure should contain a tag. The second is the way sign is initialized. The initialization values should be in braces. Here is the corrected code:
```
struct zodiac {
    char zodiac_sign[21];
    int month;
} sign = {"Leo", 8};
```
10. The union declaration has only one problem. Only one variable in a union can be used at a time. This is also true of initializing the union. Only the first member of the union can be initialized. Here is the correct initialization:
```
/* setting up a union */
union data{
    char a_word[4];
    long a_number;
}generic_variable = { "WOW" };
```

Answers for Chapter 12

Quiz

1. The scope of a variable refers to the extent to which different parts of a program have access to the variable, or where the variable is visible.

2. A variable with local storage class is visible only in the function where it is defined. A variable with external storage class is visible throughout the program.
3. Defining a variable in a function makes it local; defining a variable outside of any function makes it external.
4. Automatic (the default) or static. An automatic variable is created each time the function is called and is destroyed when the function ends. A static local variable persists and retains its value between calls to the function.
5. An automatic variable is initialized every time the function is called. A static variable is initialized only the first time the function is called.
6. False. When declaring register variables, you're making a request. There is no guarantee that the compiler will honor the request.
7. An uninitialized global variable is automatically initialized to 0; however, it's best to initialize variables explicitly.
8. An uninitialized local variable isn't automatically initialized; it could contain anything. You should never use an uninitialized local variable.
9. Because the variable count is now local to the block, the printf() no longer has access to a variable called count. The compiler gives you an error.
10. If the value needs to be remembered, it should be declared as static. For example, if the variable were called vari, the declaration would be

```
static int vari;
```

11. The extern keyword is used as a storage-class modifier. It indicates that the variable has been declared somewhere else in the program.
12. The static keyword is used as a storage-class modifier. It tells the compiler to retain the value of a variable or function for the duration of a program. Within a function, the variable keeps its value between function calls.

Exercises

1. register int x = 0;
2. The code is as follows:

```
/* Illustrates variable scope. */
#include <stdio.h>
void print_value(int x);
main()
{
    int x = 999;
    printf("%d", x);
    print_value( x );
    return 0;
}
void print_value( int x)
{
    printf("%d", x);
}
```

3. Because you're declaring var as a global, you don't need to pass it as a parameter.

```
/* Using a global variable */
#include <stdio.h>
int var = 99;
void print_value(void);
main()
{
    print_value();
    return 0;
}
void print_value(void)
{
```

```
    printf( "The value is %d\n", var );
}
```
4. Yes, you need to pass the variable var in order to print it in a different function.
```
/* Using a local variable*/
#include <stdio.h>
void print_value(int var);
main( )
{
    int var = 99;
    print_value(  var );
    return 0;
}
void print_value(int var)
{
    printf( "The value is %d\n", var );
}
```
5. Yes, a program can have a local and global variable with the same name. In such cases, active local variables take precedence.
```
/* Using a global */
#include <stdio.h>
int var = 99;
void print_func(void);
main( )
{
    int var = 77;
    printf( "Printing in function with local and global:");
    printf( "\nThe Value of var is %d", var );
    print_func( );
    return 0;
}
void print_func( void )
{
    printf( "\nPrinting in function  only global:");
    printf( "\nThe value of var is %d\n", var );
}
```
6. There is only one problem with a_sample_function(). Variables can be declared at the beginning of any block, so the declarations of crtl and star are fine. The other variable, ctr2, is not declared at the beginning of a block; it needs to be. The following is the corrected function within a complete program.
Note: If you're using a C++ compiler instead of a C compiler, the listing with a bug might run and compile. C++ has different rules concerning where variables can be declared. However, you should still follow the rules for C, even if your compiler lets you get away with something different.
```
#include <stdio.h>
void a_sample_function( );
main()
{
    a_sample_function();
    return 0;
}
void a_sample_function( void )
{
    int ctr1;
    for ( ctr1 = 0; ctr1 < 25; ctr1++ )
        printf( "*" );
```

```
    puts( "\nThis is a sample function" );
    {
        char star = `*';
        int ctr2;     /* fix */
        puts( "\nIt has a problem\n" );
        for ( ctr2 = 0; ctr2 < 25; ctr2++ )
        {
            printf( "%c", star);
        }
     }
}
```

7. This program actually works properly, but it could be better. First of all, there is no need to initialize the variable x to 1, because it's initialized to 0 in the for statement. Also, declaring the variable tally to be static is pointless, because within the main() function, static keywords have no effect.

8. What is the value of star? What is the value of dash? These two variables are never initialized. Because they are both local variables, each could contain any value. Note that although this program compiles with no errors or warnings, there is still a problem.

There is a second issue that should be brought up about this program. The variable ctr is declared as global, but it's only used in print_function(). This isn't a good assignment. The program would be better if ctr were a local variable in print_function().

9. This program prints the following pattern forever. See exercise 10.

```
X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==X==...
```

10. This program poses a problem because of the global scope of ctr. Both main() and print_letter2() use ctr in loops at the same time. Because print_letter2() changes the value, the for loop in main() never completes. This could be fixed in a number of ways. One way is to use two different counter variables. A second way is to change the scope of the counter variable ctr. It could be declared in both main() and print_letter2() as a local variable.

An additional comment on letter1 and letter2. Because each of these is used in only one function, they should be declared as local. Here is the corrected listing:

```
#include <stdio.h>
void print_letter2(void);           /* function prototype */
main()
{
    char letter1 = `X';
    int ctr;
    for( ctr = 0; ctr < 10; ctr++ )
    {
       printf( "%c", letter1 );
       print_letter2();
    }
    return 0;
}
void print_letter2(void)
{
    char letter2 = `=';
    int ctr;            /* this is a local variable */
                        /* it is different from ctr in
main() */
    for( ctr = 0; ctr < 2; ctr++ )
        printf( "%c", letter2 );
}
```

Answers for Chapter 13

Quiz

1. Never. (Unless you are very careful)
2. When a break statement is encountered, execution immediately exits the for, do...while, or while loop that contains the break. When a continue statement is encountered, the next iteration of the enclosing loop begins immediately.
3. An infinite loop executes forever. You create one by writing a for, do...while, or while loop with a test condition that is always true.
4. Execution terminates when the program reaches the end of main() or the exit() function is called.
5. The expression in a switch statement can evaluate to a long, int, or char value.
6. The default statement is a case in a switch statement. When the expression in the switch statement evaluates to a value that doesn't have a matching case, control goes to the default case.
7. The exit() function causes the program to end. A value can be passed to the exit() function. This value is returned to the operating system.
8. The system() function executes a command at the operating system level.

Exercises

1. continue;
2. break;
3. For a DOS system, the answer would be

```
system("dir");
```

4. This code fragment is correct. You don't need a break statement after the printf() for `N', because the switch statement ends anyway.
5. You might think that the default needs to go at the bottom of the switch statement, but this isn't true. The default can go anywhere. There is a problem, however. There should be a break statement at the end of the default case.
6. The code is as follows:

```
if( choice == 1 )
    printf("You answered 1");
else if( choice == 2 )
          printf( "You answered 2");
     else
          printf( "You did not choose 1 or 2");
```

7. The code is as follows:

```
do {
    /* any C statements */
} while ( 1 );
```

Answers for Chapter 14

Quiz

1. A stream is a sequence of bytes. A C program uses streams for all input and output.
2. a. A printer is an output device.
b. A keyboard is an input device.
c. A modem is both an input and an output device.
d. A monitor is an output device. (Although a touch screen would be an input device and an output device.)
e. A disk drive can be both an input and an output device.
3. All C compilers support three predefined streams: stdin (the keyboard), stdout (the screen), and stderr (the screen). Some compilers, including DOS, also support stdprn (the printer), and stdaux (the serial port COM1). Note that the Macintosh doesn't support the stdprn function.
4. a. stdout
b. stdout
c. stdin
d. stdin

e. fprintf() can use any output stream. Of the five standard streams, it can use stdout, stderr, stdprn, and stdaux.
5. Buffered input is sent to the program only when the user presses Enter. Unbuffered input is sent one character at a time, as soon as each key is pressed.
6. Echoed input automatically sends each character to stdout as it is received; unechoed input does not.
7. You can "unget" only one character between reads. The EOF character can't be put back into the input stream with unget().
8. With the newline character, which corresponds to the user's pressing Enter.
9. a. Valid
b. Valid
c. Valid
d. Not valid. There is not an identifier of q.
e. Valid
f. Valid
10. stderr can't be redirected; it always prints to the screen. stdout can be redirected to somewhere other than the screen.

Exercises

1. printf("Hello World");
2. fprintf(stdout, "Hello World");
puts("Hello World");
3. fprintf(stdaux, "Hello Auxiliary Port");
4. The code is as follows:

```
char buffer[31];
scanf( "%30[^*]", buffer );
```

5. The code is as follows:

```
printf( "Jack asked, \"What is a backslash\?\"\nJill said, \
\"It is \'\\\'\"");
```

7. Hint: Use an array of 26 integers. To count each character, increment the appropriate array element for each character read.
9. Hint: Get a string at a time, and then print a formatted line number followed by a tab and the string.
Second hint: Check out the Print_It program in Type & Run 1!

Answers for Chapter 15

Quiz

1. The code is as follows:

```
float x;
float *px = &x;
float **ppx = &px;
```

2. The error is that the statement uses a single indirection operator and, as a result, assigns the value 100 to px instead of to x. The statement should be written with a double indirection operator:

```
**ppx = 100;
```

3. array is an array with two elements. Each of these elements is itself an array that contains three elements. Each of these elements is an array that contains four type int variables.
4. array[0][0] is a pointer to the first four-element array of type int.
5. The first and third comparisons are true; the second is not true.
6. void func1(char *p[]);
7. It has no way of knowing. This value must be passed to the function as another argument.
8. A pointer to a function is a variable that holds the address where the function is stored in memory.
9. char (*ptr)(char *x[]);
10. If you omit the parentheses surrounding *ptr, the line is a prototype of a function that returns a pointer to type char.
11. The structure must contain a pointer to the same type of structure.
12. It means that the linked list is empty.
13. Each element in the list contains a pointer that identifies the next element in the list. The first element in the list is identified by the head pointer.
14. a. var1 is a pointer to an integer.

b. var2 is an integer.
c. var3 is a pointer to a pointer to an integer.
15. a. a is an array of 36 (3 * 12) integers.
b. b is a pointer to an array of 12 integers.
c. c is an array of 12 pointers to integers.
16. a. z is an array of 10 pointers to characters.
b. y is a function that takes an integer (field) as an argument and returns a pointer to a character.
c. x is a pointer to a function that takes an integer (field) as an argument and returns a character.

Exercises

1. float (*func)(int field);
2. int (*menu_option[10])(char *title);
An array of function pointers can be used in conjunction with a menuing system. The number selected from a menu could correspond to the array index for the function pointer. For example, the function pointed to by the fifth element of the array would be executed if item 5 were selected from the menu.
3. char *ptrs[10];
4. ptr is declared as an array of 12 pointers to integers, not a pointer to an array of 12 integers. The correct code is

```
int x[3][12];
int (*ptr)[12];
ptr = x;
```

5. The following is one of many possible solutions:

```
struct friend {
   char name[35+1];
   char street1[30+1];
   char street2[30+1];
   char city[15+1];
   char state[2+1];
   char zipcode[9+1];
   struct friend *next;
```

Answers for Chapter 16

Quiz

1. A text-mode stream automatically performs translation between the newline character (\n), which C uses to mark the end of a line, and the carriage-return linefeed character pair that DOS uses to mark the end of a line. In contrast, a binary-mode stream performs no translations. All bytes are input and output without modification.
2. Open the file using the fopen() library function.
3. When using fopen(), you must specify the name of the disk file to open and the mode to open it in. The function fopen() returns a pointer to type FILE; this pointer is used in subsequent file access functions to refer to the specific file.
4. Formatted, character, and direct.
5. Sequential and random.
6. EOF is the end-of-file flag. It is a symbolic constant equal to -1.
7. EOF is used with text files to determine when the end of the file has been reached.
8. In binary mode, you must use the feof() function. In text mode, you can look for the EOF character or use feof().
9. The file position indicator indicates the position in a given file where the next read or write operation will occur. You can modify the file position indicator with rewind() and fseek().
10. The file position indicator points to the first character of the file, or offset 0. The one exception is if you open an existing file in append mode, in which case the position indicator points to the end of the file.

Exercises

1. fcloseall();
2. rewind(fp); and fseek(fp, 0, SEEK_SET);
3. You can't use the EOF check with a binary file. You should use the feof() function instead.

Answers for Chapter 17

Quiz

1. The length of a string is the number of characters between the start of the string and the terminating null character (not counting the null character). You can determine the length of a string using the strlen() function.
2. You must be sure to allocate sufficient storage space for the new string.
3. *Concatenate* means to join two strings, appending one string to the end of another.
4. When you compare strings, "greater than" means that one string's ASCII values are larger than the other string's ASCII values.
5. strcmp() compares two entire strings. strncmp() compares only a specified number of characters within the string.
6. strcmp() compares two strings, considering the case of the letters. (For example, `A' and `a' are different.) strcmpi() ignores case. (For example, `A' and `a' are the same.)
7. isascii() checks the value passed to see whether it's a standard ASCII character between 0 and 127. It doesn't check for extended ASCII characters.
8. isascii() and iscntrl() both return TRUE; all others return FALSE. Remember, these macros look at the character value.
9. 65 is equivalent to the ASCII character A. The following macros return TRUE: isalnum(), isalpha(), isascii(), isgraph(), isprint(), and isupper().
10. The character-test functions determine whether a particular character meets a certain condition, such as whether it is a letter, punctuation mark, or something else.

Exercises

1. TRUE (1) or FALSE (0)
2. a. 65
b. 81
c. -34
d. 0
e. 12
f. 0
3. a. 65.000000
b. 81.230000
c. -34.200000
d. 0.000000
e. 12.000000
f. 1000.000000
4. string2 wasn't allocated space before it was used. There is no way to know where strcpy() copies the value of string1.

Answers for Chapter 18

Quiz

1. Passing by value means that the function receives a copy of the value of the argument variable. Passing by reference means that the function receives the address of the argument variable. The difference is that passing by reference allows the function to modify the original variable, whereas passing by value does not.
2. A type void pointer can point to any type of C data object (in other words, it's a generic pointer).
3. By using a void pointer, you can create a generic pointer that can point to any object. The most common use of a void pointer is to declare function parameters. You can create a function that can handle different types of arguments.
4. A typecast provides information about the type of the data object that the void pointer is pointing to at the moment. You must cast a void pointer before dereferencing it.
5. A function that takes a variable argument list must be passed at least one fixed argument. This is done to inform the function of the number of arguments being passed each time it is called.
6. va_start() should be used to initialize the argument list. va_arg() should be used to retrieve the arguments. va_end() should be used to clean up after all the arguments have been retrieved.

7. Trick question! void pointers can't be incremented, because the compiler wouldn't know what value to add.
8. A function can return a pointer to any of the C variable types. A function can also return a pointer to such storage areas as arrays, structures, and unions.

Exercises

1. int function(char array[]);
2. int numbers(int *nbr1, int *nbr2, int *nbr3);
3. The code is as follows:

```
int number1 = 1, number2 = 2, number3 = 3;
numbers( &number1, &number2, &number3);
```

4. Although the code might look confusing, it is correct. This function takes the value being pointed to by nbr and multiplies it by itself.
5. When using variable parameter lists, you should use all the macro tools. This includes va_list, va_start(), va_arg(), and va_end(). See Listing 18.3 for the correct way to use variable parameter lists.

Answers for Chapter 19

Quiz

1. Type double.
2. On most compilers, it's equivalent to a long; however, this isn't guaranteed. Check the TIME.H file with your compiler or your reference manual to find out what variable type your compiler uses.
3. The time() function returns the number of seconds that have elapsed since midnight, January 1, 1970. The clock() function returns the number of $^{1}/_{100}$ seconds that have elapsed since the program began execution.
4. Nothing. It simply displays a message that describes the error.
5. Sort the array into ascending order.
6. 14
7. 4
8. 21
9. 0 if the values are equal, >0 if the value of element 1 is greater than element 2, and <0 if the value of element 1 is less than element 2.
10. NULL

Exercises

1. The code is as follows:

```
bsearch( myname, names, (sizeof(names)/sizeof(names[0])),
sizeof(names[0]), comp_names);
```

2. There are three problems. First, the field width isn't provided in the call to qsort(). Second, the parentheses shouldn't be added to the end of the function name in the call to qsort(). Third, the program is missing its comparison function. qsort() uses compare_function(), which isn't defined in the program.
3. The compare function returns the wrong values. It should return a positive number if element1 is greater than element2 and a negative number if element1 is less than element2.

Answers for Chapter 20

Quiz

1. malloc() allocates a specified number of bytes of memory, whereas calloc() allocates sufficient memory for a specified number of data objects of a certain size. calloc() also sets the bytes of memory to 0, whereas malloc() doesn't initialize them to any specific value.
2. To preserve the fractional part of the answer when dividing one integer by another and assigning the result to a floating-point variable.
3. a. long
b. int
c. char
d. float
e. float

4. Dynamically allocated memory is allocated at runtime--while the program is executing. Dynamic memory allocation lets you allocate exactly as much memory as is needed, only when it is needed.
5. memmove() works properly when the source and destination memory regions overlap, whereas memcpy() does not. If the source and destination regions don't overlap, the two functions are identical.
6. By defining a bit field member with a size of 3 bits. Since 2^3 equals 8, such a field is sufficient to hold values 1 through 7.
7. 2 bytes. Using bit fields, you could declare a structure as follows:

```
struct date{
unsigned month : 4;
unsigned Chapter   : 5;
unsigned year  : 7;
}
```

This structure stores the date in 2 bytes (16 bits). The 4-bit month field can hold values from 0 to 15, sufficient for holding 12 months. Likewise, the 5-bit Chapter field can hold values from 0 to 31, and the 7-bit year field can hold values from 0 to 127. We assume that the year value will be added to 1900 to allow year values from 1900 to 2027.
8. 00100000
9. 00001001
10. These two expressions evaluate to the same result. Using exclusive OR with 11111111 is the same as using the complement operator: Each bit in the original value is reversed.

Exercises

1. The code is as follows:

```
long *ptr;
ptr = malloc( 1000 * sizeof(long));
```

2. The code is as follows:

```
long *ptr;
ptr = calloc( 1000, sizeof(long));
```

3. Using a loop and assignment statement:

```
int count;
for (count = 0; count < 1000; count++)
data[count] = 0;
```

Using the memset() function:

```
memset(data, 0, 1000 * sizeof(float));
```

4. This code will compile and run without error; however, the results will be incorrect. Because number1 and number2 are both integers, the result of their division will be an integer, thus losing any fractional part of the answer. In order to get the correct answer, you need to cast the expression to type float:

```
answer = (float) number1/number2;
```

5. Because p is a type void pointer, it must be cast to the proper type before being used in an assignment statement. The third line should be as follows:

```
*(float*)p = 1.23;
```

6. No. When using bit fields, you must place them within a structure first. The following is correct:

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

Answers for Chapter 21

Quiz

1. *Modular programming* refers to the program development method that breaks a program into multiple source-code files.

2. The main module contains the main() function.
3. To avoid unwanted side effects by ensuring that complex expressions passed as arguments to the macro are fully evaluated first.
4. Compared to a function, a macro results in faster program execution but larger program size.
5. The defined() operator tests to see whether a particular name is defined, returning TRUE if the name is defined and FALSE if it isn't.
6. You must use #endif.
7. Compiled source files become object files with an .OBJ extension.
8. #include copies the contents of another file into the current file.
9. An #include statement with double quotes looks in the current directory for the include file. An include statement with <> searches the standard directory for the include file.
10. __DATE__ is used to place into the program the date that the program was compiled.
11. A string containing the name of the current program, including path information.