

Other C Topics



1

We'll use a signal I have tried and found far-reaching and easy to yell. Waa-hoo! —Zane Grey

It is quite a three-pipe problem. —Sir Arthur Conan Doyle

OBJECTIVES

In this chapter you will learn:

- To redirect keyboard input to come from a file.
- To redirect screen output to be placed in a file.
- To write functions that use variable-length argument lists.
- To process command-line arguments.
- To assign specific types to numeric constants.
- To use temporary files.
- To process external asynchronous events in a program.
- To allocate memory dynamically for arrays.
- To change the size of memory that was dynamically allocated previously.



14.1 Introduction

- 14.2 Redirecting Input/Output on Linux/UNIX and Windows Systems
- **14.3 Variable-Length Argument Lists**
- **14.4 Using Command-Line Arguments**
- 14.5 Notes on Compiling Multiple-Source-File Programs
- **14.6** Program Termination with exit and at exit





14.7 volatile Type Qualifier

- 14.8 Suffixes for Integer and Floating-Point Constants
- 14.9 More on Files
- 14.10 Signal Handling
- 14.11 Dynamic Memory Allocation: Functions calloc and realloc
- **14.12 Unconditional Branching with goto**



14.1 Introduction

- Several advanced topics in this chapter
- Operating system specific
 - Usually UNIX or DOS



14.2 Redirecting Input/Output on UNIX and DOS Systems

- Standard I/O keyboard and screen
 - Redirect input and output
- Redirect symbol(<)</p>
 - Operating system feature, not a C feature
 - UNIX and DOS
 - \$ or % represents command line
 - Example:
 - \$ sum < input</pre>
 - Rather than inputting values by hand, read them from a file
- Pipe command(|)
 - Output of one program becomes input of another
 - \$ random | sum
 - Output of random goes to sum



14.2 Redirecting Input/Output on UNIX and DOS Systems

- Redirect output (>)
 - Determines where output of a program goes
 - Example:
 - \$ random > out
 - Output goes into out (erases previous contents)
- Append output (>>)
 - Add output to end of file (preserve previous contents)
 - Example:
 - \$ random >> out
 - Output is added onto the end of out



14.3 Variable-Length Argument Lists

- Functions with unspecified number of arguments
 - Load <stdarg. h>
 - Use ellipsis(. . .) at end of parameter list
 - Need at least one defined parameter
 - Example:

double myfunction (int i, ...);

- The ellipsis is only used in the prototype of a function with a variable length argument list
- printf is an example of a function that can take multiple arguments
- The prototype of printf is defined as

int printf(const char* format, ...);



Identifier	Explanation
va_list	A type suitable for holding information needed by macros va_start, va_arg and va_end. To access the arguments in a variable-length argument list, an object of type va_l ist must be defined.
va_start	A macro that is invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with va_l i st for use by the va_arg and va_end macros.
va_arg	A macro that expands to an expression of the value and type of the next argument in the variable-length argument list. Each invocation of va_arg modifies the object declared with va_l i st so that the object points to the next argument in the list.
va_end	A macro that facilitates a normal return from a function whose variable- length argument list was referred to by the va_start macro.

Fig. 14.1 | stdarg. h variable-length argument list type and macros.



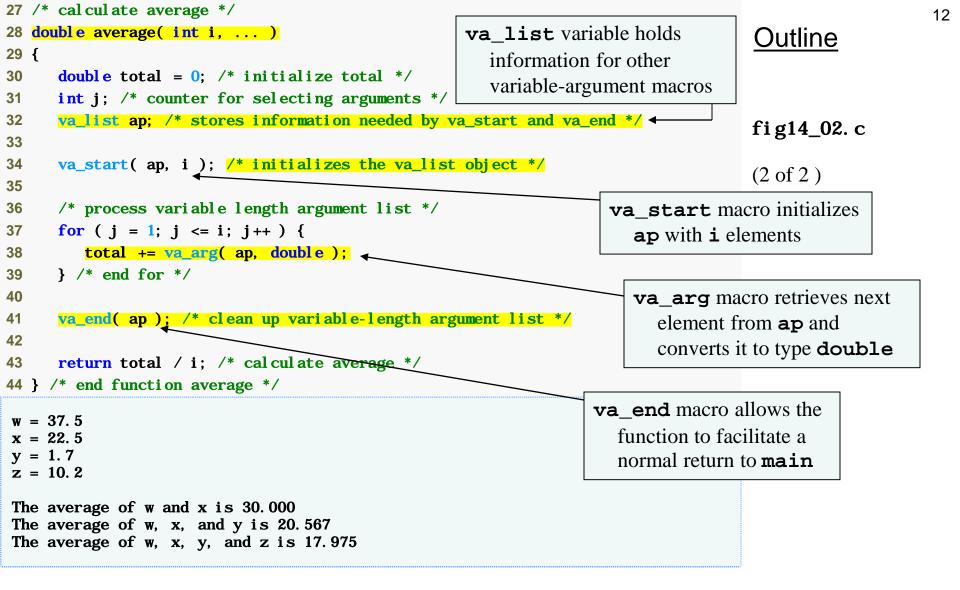
```
/* Fig. 14.2: fig14_02.c
1
      Using variable-length argument lists */
2
  #include <stdio. h>
3
  #include <stdarg. h>
4
5
  double average( int i, ... ); /* prototype */
6
7
8 int main( void )
                                 Function average takes an integer i and an
9 {
                                   unspecified number of additional arguments
      double w = 37.5;
10
      double x = 22.5;
11
      double y = 1.7;
12
      double z = 10.2;
13
14
      printf( "%s%. 1f\n%s%. 1f\n%s%. 1f\n/n",
15
16
         "w = ", w, "x = ", x, "y = ", y, "z = ", z);
      printf( "%s%. 3f\n%s%. 3f\n%s%. 3f\n",
17
         "The average of w and x is ", average(2, w, x),
18
         "The average of w, x, and y is ", average(3, w, x, y),
19
         "The average of w, x, y, and z is ",
20
         average( 4, w, x, y, z) );
21
22
23
      return 0; /* indicates successful termination */
24
25 } /* end main */
26
```

<u>Outline</u>

```
fig14_02. c
```

(1 of 2)







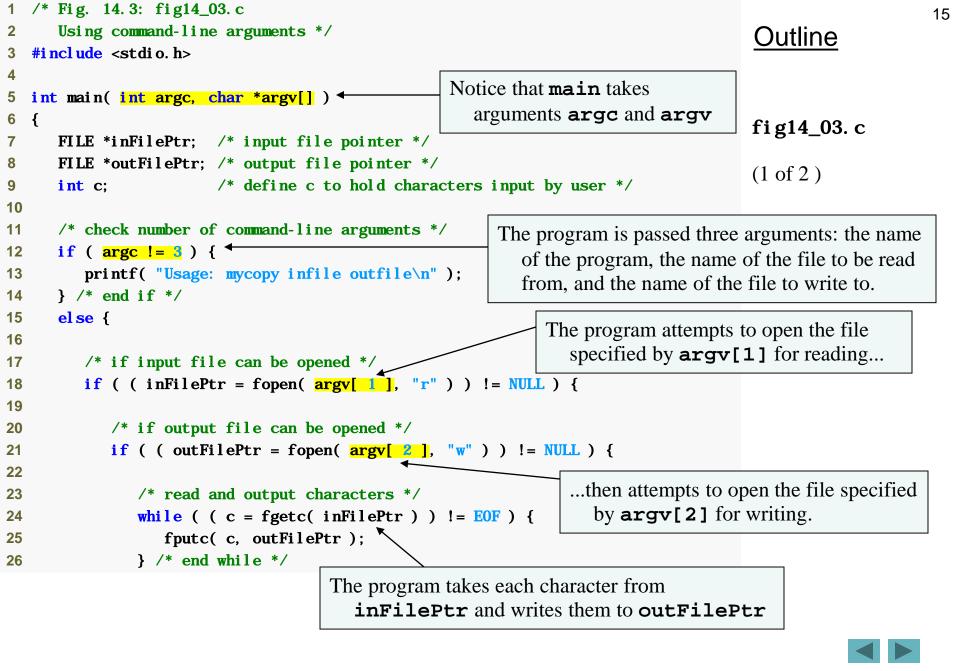
Common Programming Error 14.1

Placing an ellipsis in the middle of a function parameter list is a syntax error. An ellipsis may only be placed at the end of the parameter list.



14.4 Using Command-Line Arguments

- Pass arguments to main on DOS or UNIX
 - Define main as
 - int main(int argc, char *argv[])
 - int argc
 - Number of arguments passed
 - char *argv[]
 - Array of strings
 - Has names of arguments in order argv[0] is first argument
 - Example: \$ mycopy input output
 - argc: 3
 - argv[0]: "mycopy"
 - argv[1]: "input"
 - argv[2]: "output"



```
27
            } /* end if */
28
                                                                                        Outline
            else { /* output file could not be opened */
29
               printf( "File \"%s\" could not be opened\n", argv[ 2 ] );
30
            } /* end else */
31
32
                                                                                       fig14_03. c
33
         } /* end if */
34
         else { /* input file could not be opened */
                                                                                        (2 \text{ of } 2)
            printf( "File \"%s\" could not be opened\n", argv[ 1 ] );
35
         } /* end else */
36
37
     } /* end else */
38
39
      return 0; /* indicates successful termination */
40
41
42 } /* end main */
```



14.5 Notes on Compiling Multiple Source-File Programs

- Programs with multiple source files
 - Function definition must be in one file (cannot be split up)
 - Global variables accessible to functions in same file
 - Global variables must be defined in every file in which they are used
 - Example:
 - If integer fl ag is defined in one file
 - To use it in another file you must include the statement extern int flag;
 - extern
 - States that the variable is defined in another file
 - Function prototypes can be used in other files without an extern statement
 - Have a prototype in each file that uses the function



14.5 Notes on Compiling Multiple Source-File Programs

- Keyword static
 - Specifies that variables can only be used in the file in which they are defined
- Programs with multiple source files
 - Tedious to compile everything if small changes have been made to only one file
 - Can recompile only the changed files
 - Procedure varies on system
 - UNIX: make utility



Software Engineering Observation 14.1

Global variables should be avoided unless application performance is critical because they violate the principle of least privilege.



Software Engineering Observation 14.2

Creating programs in multiple source files facilitates software reusability and good software engineering. Functions may be common to many applications. In such instances, those functions should be stored in their own source files, and each source file should have a corresponding header file containing function prototypes. This enables programmers of different applications to reuse the same code by including the proper header file and compiling their applications with the corresponding source file.



14.6 Program Termination with exit and atexit

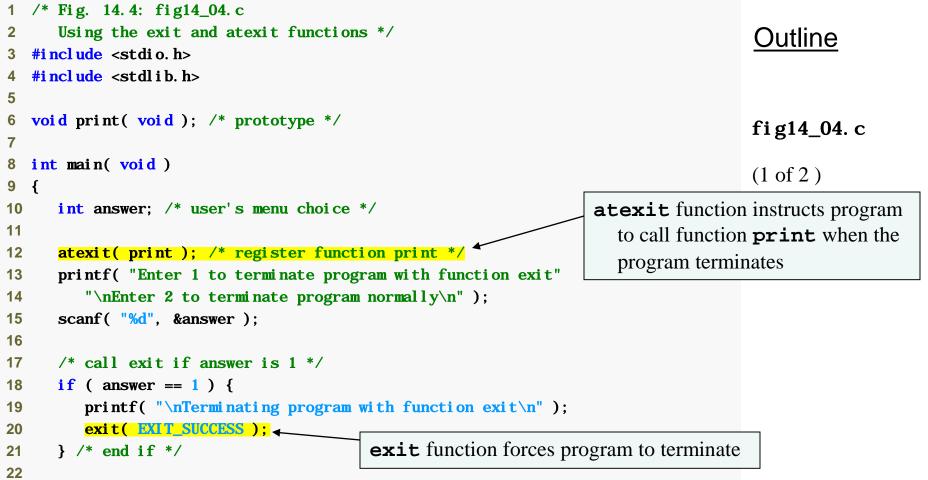
- Function exi t
 - Forces a program to terminate
 - Parameters symbolic constants EXI T_SUCCESS or EXI T_FAI LURE
 - Returns an implementation-defined value
 - Example:

```
exit( EXIT_SUCCESS );
```

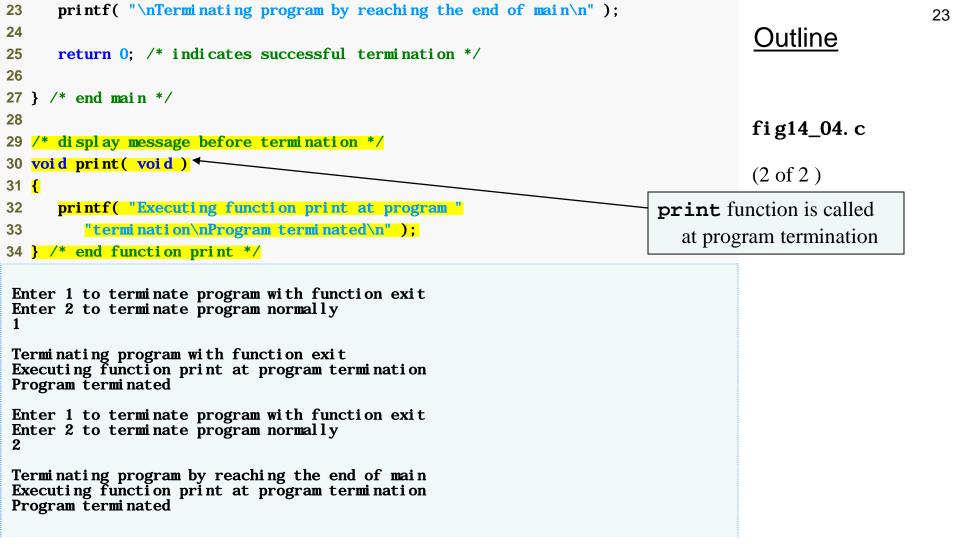
Function atexit

atexit(functionToRun);

- Registers functionToRun to execute upon successful program termination
 - atexit itself does not terminate the program
- Register up to 32 functions (multiple atexit() statements)
 - Functions called in reverse register order
- Called function cannot take arguments or return values









14.7 vol at i l e Type Qualifier

vol ati l e qualifier

- Variable may be altered outside program
- Variable not under control of program
- Variable cannot be optimized



14.8 Suffixes for Integer and Floating-Point Constants

- C provides suffixes for constants
 - unsi gned integer u or U
 - long integer l or L
 - unsi gned l ong integer ul , l u, UL or LU
 - float f or F
 - long doubl e l or L
 - Examples:

```
174u
467L
3451ul
```

- If integer constant is not suffixed type determined by first type capable of storing a value of that size (int, long int, unsigned long int)
- If floating point not suffixed of type doubl e



14.9 More on Files

C can process binary files

- Not all systems support binary files
 - Files opened as text files if binary mode not supported
- Binary files should be used when rigorous speed, storage, and compatibility conditions demand it
- Otherwise, text files are preferred
 - Inherent portability, can use standard tools to examine data
- Function tmpfile
 - Opens a temporary file in mode "wb+"
 - Some systems may process temporary files as text files
 - Temporary file exists until closed with fclose or until program terminates
- Function rewind
 - Positions file pointers to the beginning of the file



Mode Description

rb	Open an existing binary file for reading.
wb	Create a binary file for writing. If the file already exists, discard the current contents.
ab	Append; open or create a binary file for writing at end-of-file.
rb+	Open an existing binary file for update (reading and writing).
wb+	Create a binary file for update. If the file already exists, discard the current contents.
ab+	Append; open or create a binary file for update; all writing is done at the end of the file

Fig. 14.5 | Binary file open modes.



Performance Tip 14.1

Consider using binary files instead of text files in applications that demand high performance.



_

28

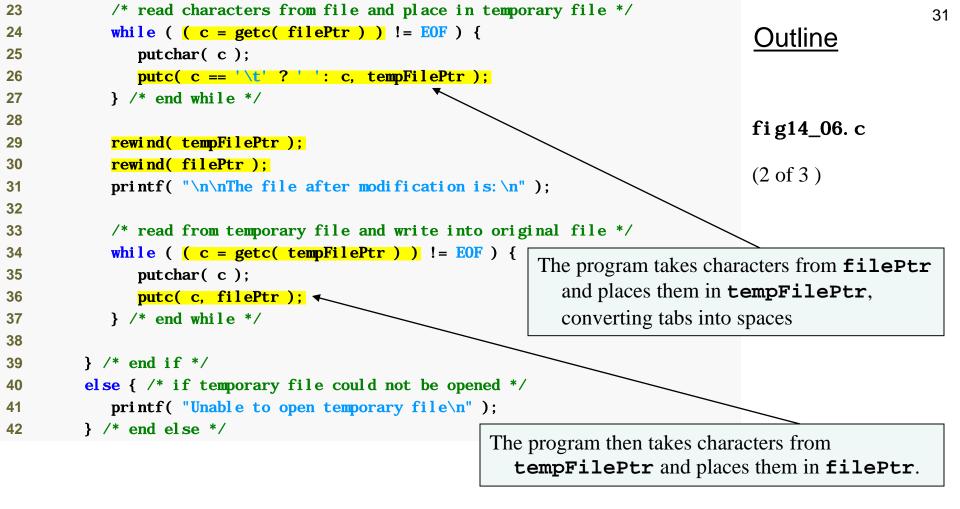
Portability Tip 14.1

Use text files when writing portable programs.



```
/* Fig. 14.6: fig14_06.c
1
      Using temporary files */
2
                                                                                       Outline
  #include <stdio. h>
3
4
5 int main( void )
6
  {
                                                                                      fig14_06. c
     FILE *filePtr; /* pointer to file being modified */
7
      FILE *tempFilePtr; /* temporary file pointer */
8
                                                                                      (1 \text{ of } 3)
      int c; /* define c to hold characters read from a file */
9
      char fileName[ 30 ]; /* create char array */
10
11
      printf( "This program changes tabs to spaces. n"
12
         "Enter a file to be modified: " );
13
      scanf( "%29s", fileName );
14
15
16
      /* fopen opens the file */
      if ( ( filePtr = fopen( fileName, "r+" ) ) != NULL ) {
17
                                                                    tmpfile function creates a
18
                                                                       temporary file
         /* create temporary file */
19
         if ( ( tempFilePtr = tmpfile() ) != NULL ) {
20
21
            printf( "\nThe file before modification is: \n" );
22
```







```
43
     } /* end if */
44
                                                                                      Outline
      else { /* if file could not be opened */
45
         printf( "Unable to open %s\n", fileName );
46
     } /* end else */
47
48
                                                                                     fig14_06. c
     return 0; /* indicates successful termination */
49
50
                                                                                      (3 of 3)
51 } /* end main */
This program changes tabs to spaces.
Enter a file to be modified: data.txt
The file before modification is:
0
                 2
                         3
         1
                                 4
         5
                6
                         7
                                 8
                                         9
The file after modification is:
0 1 2 3 4
 56789
```



14.10 Signal Handling

Signal

- Unexpected event, can terminate program
 - Interrupts (*<ctrl> c*), illegal instructions, segmentation violations, termination orders, floating-point exceptions (division by zero, multiplying large floats)

Function si gnal

- Traps unexpected events
- Header <si gnal . h>
- Receives two arguments: a signal number and a pointer to the signal handling function

Function rai se

- Takes an integer signal number and creates a signal



Signal	Explanation
SI GABRT	Abnormal termination of the program (such as a call to function abort).
SI GFPE	An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
SIGILL	Detection of an illegal instruction.
SI GI NT	Receipt of an interactive attention signal.
SI GSEGV	An invalid access to storage.
SI GTERM	A termination request set to the program.

Fig. 14.7 | si gnal . h standard signals.

34

```
/* Fig. 14.8: fig14_08.c
1
      Using signal handling */
2
                                                                                       Outline
  #include <stdio. h>
3
  #i ncl ude <si gnal. h>
4
  #include <stdlib.h>
5
  #include <time.h>
6
                                                                                       fig14_08. c
7
  void signalHandler( int signalValue ); /* prototype */
8
                                                                                       (1 \text{ of } 3)
9
10 int main( void )
11 {
     int i; /* counter used to loop 100 times */
12
     int x; /* variable to hold random values between 1-50 */
13
14
     signal ( SIGINT, signal Handler ); /* register signal handler */
15
16
      srand( clock() );
                                                 signal function instructs program to call
17
                                                    signalHandler if a SIGINT signal is detected
      /* output numbers 1 to 100 */
18
     for ( i = 1; i <= 100; i++ ) {
19
         x = 1 + rand() % 50; /* generate random number to raise SIGINT */
20
21
         /* raise SIGINT when x is 25 */
22
23
         if (x == 25) {
            raise( SIGINT ); 
24
                                 raise function causes a SIGINT signal to occur
         } /* end if */
25
26
```

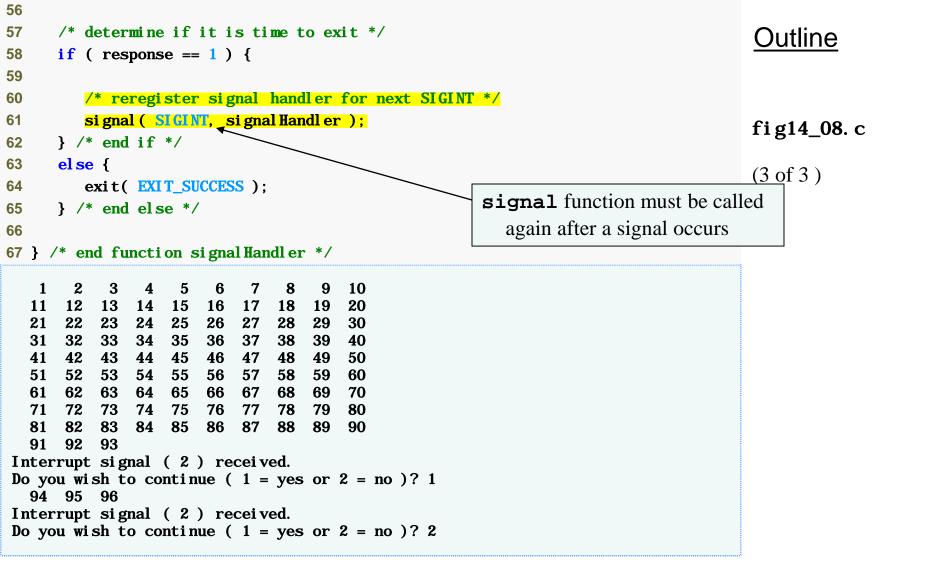


35

```
27
         printf( "%4d", i );
28
         /* output \n when i is a multiple of 10 */
29
         if ( i % 10 == 0 ) {
30
            printf( "\n" );
31
         } /* end if */
32
33
34
      } /* end for */
35
      return 0; /* indicates successful termination */
36
37
38 } /* end main */
39
40 /* handles signal */
41 void signal Handler(int signal Value)
42 {
      int response; /* user's response to signal (1 or 2) */
43
44
45
      printf( "%s%d%s\n%s",
         "\nInterrupt signal (", signalValue, ") received.",
46
         "Do you wish to continue (1 = yes \text{ or } 2 = no)?");
47
48
49
      scanf( "%d", &response );
50
      /* check for invalid responses */
51
      while ( response != 1 && response != 2 ) {
52
         printf( "( 1 = yes or 2 = no )? " );
53
         scanf( "%d", &response );
54
      } /* end while */
55
```

<u>Outline</u>

fig14_08. c (2 of 3)





14.11 Dynamic Memory Allocation: Functions calloc and realloc

Dynamic memory allocation

- Can create dynamic arrays
- calloc(nmembers, size)
 - *nmembers* number of elements
 - *size* size of each element
 - Returns a pointer to a dynamic array
- realloc(pointerToObject, newSize)
 - *pointerToObject* pointer to the object being reallocated
 - *newSize* new size of the object
 - Returns pointer to reallocated memory
 - Returns NULL if cannot allocate space
 - If newSi ze equals 0 then the object pointed to is freed
 - If pointerToObj ect equals 0 then it acts like mall oc



14.12 Unconditional Branching with goto

Unstructured programming

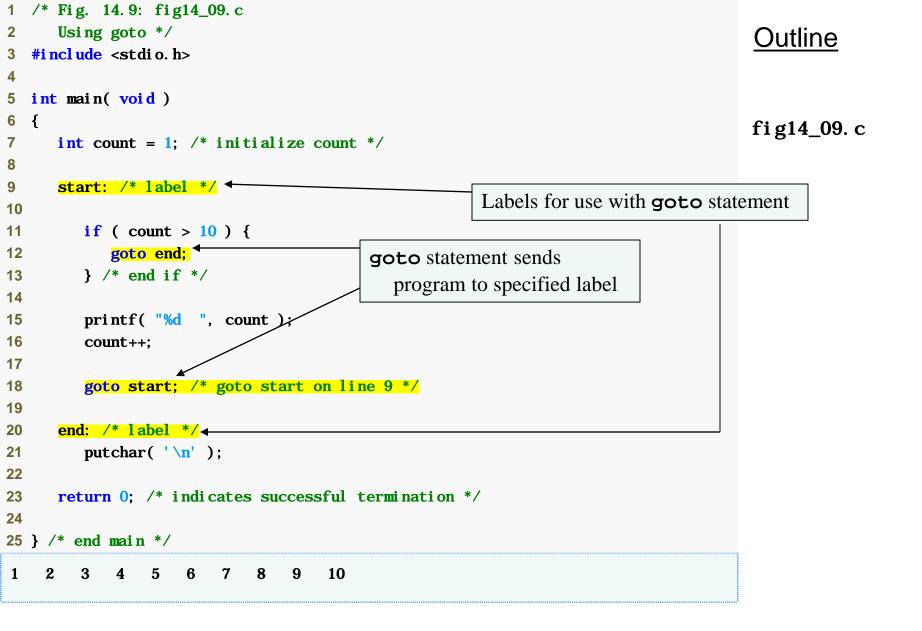
- Use when performance crucial
- break to exit loop instead of waiting until condition becomes fal se
- goto statement
 - Changes flow control to first statement after specified label
 - A label is an identifier followed by a colon (i.e. start:)
 - Quick escape from deeply nested loop goto start;



Performance Tip 14.2

The **goto** statement can be used to exit deeply nested control structures efficiently.







Software Engineering Observation 14.3

The goto statement should be used only in performance-oriented applications. The goto statement is unstructured and can lead to programs that are more difficult to debug, maintain and modify.

