Chapter 14
Other C Topics

C How to Program, 8/e, GE

Objectives

In this chapter, you'll:

- Redirect program input to come from a file.
- Redirect program output to be placed in a file.
- Write functions that use variable-length argument lists.
- Process command-line arguments.
- Compile multiple-source-file programs.
- Assign specific types to numeric constants.
- Terminate programs with `exit` and `atexit`.
- Process external asynchronous events in a program.
- Dynamically allocate arrays and resize memory that was dynamically allocated previously.

14.1 Introduction
14.2 Redirecting I/O
14.2.1 Redirecting Input with <
14.2.2 Redirecting Input with |
14.2.3 Redirecting Output
14.3 Variable-Length Argument Lists
14.4 Using Command-Line Arguments
14.5 Compiling Multiple-Source-File Programs
14.5.1 `extern` Declarations for Global Variables in Other Files
14.5.2 Function Prototypes
14.5.3 Restricting Scope with `static`
14.5.4 Makefiles
14.6 Program Termination with `exit` and `atexit`
14.7 Suffixes for Integer and Floating-Point Literals
14.8 Signal Handling
14.9 Dynamic Memory Allocation: Functions `calloc` and `realloc`
14.10 Unconditional Branching with `goto`

14.1 Introduction

- Many of the capabilities discussed here are specific to particular operating systems, especially Linux/UNIX and Windows.

14.2 Redirecting I/O

- In command-line applications, normally the input is received from the *keyboard* (standard input), and the output is displayed on the *screen* (standard output).
- On most computer systems—Linux/UNIX, Mac OS X and Windows systems in particular—it's possible to redirect inputs to come from a *file* rather than the keyboard and redirect outputs to be placed in a *file* rather than on the screen.
- Both forms of redirection can be accomplished without using the file-processing capabilities of the standard library.

14.2 Redirecting I/O (Cont.)

- There are several ways to redirect input and output from the command line——that is, a **Command Prompt** window in Windows, a shell in Linux or a **Terminal** window in Mac OS X.
- Consider the executable file `sum` (on Linux/UNIX systems) that inputs integers one at a time and keeps a running total of the values until the end-of-file indicator is set, then prints the result.
- Normally the user inputs integers from the keyboard and enters the end-of-file key combination to indicate that no further values will be input.
- With input redirection, the input can be stored in a file.

14.2 Redirecting I/O (Cont.)

- For example, if the data is stored in file `input`, the command line
  - `$ sum < input`

  executes the program `sum`; the redirect input symbol (`<`) indicates that the data in file `input` is to be used as input by the program.
- Redirecting input on a Windows system is performed identically.
- The character `$` is a typical Linux/UNIX command-line prompt (some systems use a `%` prompt or other symbol).
- The second method of redirecting input is piping.
- A pipe (`|`) causes the output of one program to be redirected as the input to another program.

14.2 Redirecting I/O (Cont.)

- Suppose program `random` outputs a series of random integers; the output of `random` can be "piped" directly to program `sum` using the command line
  - `$ random | sum`
- This causes the sum of the integers produced by `random` to be calculated.
- Piping is performed identically in Linux/UNIX and Windows.
- The standard output stream can be redirected to a file by using the redirect output symbol (>).
- For example, to redirect the output of program `random` to file `out`, use
  - `$ random > out`

14.2 Redirecting I/O (Cont.)

- Finally, program output can be appended to the end of an existing file by using the append output symbol (>>).
- For example, to append the output from program `random` to file `out` created in the preceding command line, use the command line
    - `$ random >> out`

14.3 Variable-Length Argument Lists

- It's possible to create functions that receive an unspecified number of arguments.
- Most programs in the text have used the standard library function `printf`, which, as you know, takes a variable number of arguments.
- As a minimum, `printf` must receive a string as its first argument, but `printf` can receive any number of additional arguments.
- The function prototype for `printf` is
  - `int printf(const char *format, ...);`
- The ellipsis (...) in the prototype indicates that the function receives a *variable number of arguments of any type*.

14.3 Variable-Length Argument Lists (Cont.)

- The ellipsis must always be placed at the *end* of the parameter list.
- The macros and definitions of the variable arguments headers `<stdarg.h>` (Fig. 14.1) provide the capabilities necessary to build functions with variable-length argument lists.
- Figure 14.2 demonstrates function `average` that receives a variable number of arguments.
- The first argument of `average` is always the number of values to be averaged.

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_list` must be defined.
`va_start`	A *macro* that's invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with `va_list` for use by the `va_arg` and `va_end` macros.
`va_arg`	A *macro* that expands to the value of the next argument in the variable-length argument list—the value has the type specified as the macro's second argument. Each invocation of `va_arg` modifies the object declared with `va_list` so that it 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.

```c
// Fig. 14.2: fig14_02.c
// Using variable-length argument lists
#include <stdio.h>
#include <stdarg.h>

double average(int i, ...); // prototype

int main(void)
{
   double w = 37.5;
   double x = 22.5;
   double y = 1.7;
   double z = 10.2;

   printf("%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
      "w = ", w, "x = ", x, "y = ", y, "z = ", z);
   printf("%s%.3f\n%s%.3f\n%s%.3f\n",
      "The average of w and x is ", average(2, w, x),
      "The average of w, x, and y is ", average(3, w, x, y),
      "The average of w, x, y, and z is ",
      average(4, w, x, y, z));
}
```

Fig. 14.2 | Using variable-length argument lists. (Part 1 of 3.)

```
24  // calculate average
25  double average(int i, ...)
26  {
27     double total = 0; // initialize total
28     va_list ap; // stores information needed by va_start and va_end
29
30     va_start(ap, i); // initializes the va_list object
31
32     // process variable-length argument list
33     for (int j = 1; j <= i; ++j) {
34        total += va_arg(ap, double);
35     }
36
37     va_end(ap); // clean up variable-length argument list
38     return total / i; // calculate average
39  }
```

Fig. 14.2 | Using variable-length argument lists. (Part 2 of 3.)

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Fig. 14.2 | Using variable-length argument lists. (Part 3 of 3.)

14.3 Variable-Length Argument Lists (Cont.)

- Function `average` uses all the definitions and macros of header `<stdarg.h>`.
- Object `ap`, of type `va_list`, is used by macros `va_start`, `va_arg` and `va_end` to process the variable-length argument list of function `average`.
- The function begins by invoking macro `va_start` to initialize object `ap` for use in `va_arg` and `va_end`.
- The macro receives two arguments—object `ap` and the identifier of the rightmost argument in the argument list *before* the ellipsis—`i` in this case (`va_start` uses `i` here to determine where the variable-length argument list begins).

14.3 Variable-Length Argument Lists (Cont.)

- Next, function `average` repeatedly adds the arguments in the variable-length argument list to `total`.
- The value to be added to `total` is retrieved from the argument list by invoking macro `va_arg`.
- Macro `va_arg` receives two arguments—object `ap` and the type of the value expected in the argument list—`double` in this case.
- The macro returns the value of the argument.
- Function `average` invokes macro `va_end` with object `ap` as an argument to facilitate a normal return to `main` from `average`.
- Finally, the average is calculated and returned to `main`.

Common Programming Error 14.1

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

14.3 Variable-Length Argument Lists (Cont.)

- The reader may question how function `printf` and function `scanf` know what type to use in each `va_arg` macro.
- The answer is that they scan the format conversion specifiers in the format control string to determine the type of the next argument to be processed.

14.4 Using Command-Line Arguments

- On many systems, it's possible to pass arguments to `main` from a command line by including parameters `int argc` and `char *argv[]` in the parameter list of `main`.
- Parameter `argc` receives the number of command-line arguments that the user has entered.
- Parameter `argv` is an array of strings in which the actual command-line arguments are stored.
- Common uses of command-line arguments include passing options to a program and passing filenames to a program.
- Figure 14.3 copies a file into another file one character at a time.

14.4 Using Command-Line Arguments (Cont.)

- We assume that the executable file for the program is called `mycopy`.
- A typical command line for the `mycopy` program on a Linux/UNIX system is
  - `$ mycopy input output`
- This command line indicates that file `input` is to be copied to file `output`.
- When the program is executed, if `argc` is not `3` (`mycopy` counts as one of the arguments), the program prints an error message and terminates.
- Otherwise, array `argv` contains the strings `"mycopy"`, `"input"` and `"output"`.

14.4 Using Command-Line Arguments (Cont.)

- The second and third arguments on the command line are used as file names by the program.
- The files are opened using function `fopen`.
- If both files are opened successfully, characters are read from file `input` and written to file `output` until the end-of-file indicator for file `input` is set.
- Then the program terminates.
- The result is an exact copy of file `input` (if no errors occur during processing.
- See your system documentation for more information on command-line arguments.

14.4 Using Command-Line Arguments (Cont.)

- In Visual C++, you can specify the command-line arguments by right clicking the project name in the Solution Explorer and selecting Properties, then expanding Configuration Properties, selecting Debugging and entering the arguments in the textbox to the right of Command Arguments.

```c
// Fig. 14.3: fig14_03.c
// Using command-line arguments
#include <stdio.h>

int main(int argc, char *argv[])
{
   // check number of command-line arguments
   if (argc != 3) {
      puts("Usage: mycopy infile outfile");
   }
   else {
      FILE *inFilePtr; // input file pointer

      // try to open the input file
      if ((inFilePtr = fopen(argv[1], "r")) != NULL) {
         FILE *outFilePtr; // output file pointer

         // try to open the output file
         if ((outFilePtr = fopen(argv[2], "w")) != NULL) {
            int c; // holds characters read from source file

```

Fig. 14.3 | Using command-line arguments. (Part 1 of 2.)

```
22              // read and output characters
23              while ((c = fgetc(inFilePtr)) != EOF) {
24                 fputc(c, outFilePtr);
25              }
26
27              fclose(outFilePtr); // close the output file
28           }
29           else { // output file could not be opened
30              printf("File \"%s\" could not be opened\n", argv[2]);
31           }
32
33           fclose(inFilePtr); // close the input file
34        }
35        else { // input file could not be opened
36           printf("File \"%s\" could not be opened\n", argv[1]);
37        }
38     }
39  }
```

Fig. 14.3 | Using command-line arguments. (Part 2 of 2.)

14.5 Notes on Compiling Multiple-Source-File Programs

- It's possible to build programs that consist of multiple source files.
- There are several considerations when creating programs in multiple files.
- For example, the definition of a function must be entirely contained in one file—it cannot span two or more files.
- In Chapter 5, we introduced the concepts of storage class and scope.
- We learned that variables declared *outside* any function definition are referred to as *global variables*.
- Global variables are accessible to any function defined in the same file after the variable is declared.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- Global variables also are accessible to functions in other files.
- However, the global variables must be declared in each file in which they're used.
- For example, to refer to global integer variable `flag` in another file, you can use the declaration
  - `extern int flag;`
- This declaration uses the storage-class specifier `extern` to indicate that variable `flag` is defined either *later in the same file or in a different file*.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- The compiler informs the linker that unresolved references to variable `flag` appear in the file.
- If the linker finds a proper global definition, the linker resolves the references by indicating where `flag` is located.
- If the linker cannot locate a definition of `flag`, it issues an error message and does not produce an executable file.
- Any identifier that's declared at file scope is `extern` by default.

Software Engineering Observation 14.1

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

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- Just as `extern` declarations can be used to declare *global variables* to other program files, *function prototypes can extend the scope of a function beyond the file in which it's defined* (the `extern` specifier is not required in a function prototype).
- Simply include the function prototype in each file in which the function is invoked and compile the files together (see Section 13.2).
- Function prototypes indicate to the compiler that the specified function is defined either later in the *same* file or in a *different* file.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- Again, the compiler does not attempt to resolve references to such a function—that task is left to the linker.
- If the linker cannot locate a proper function definition, the linker issues an error message.
- As an example of using function prototypes to extend the scope of a function, consider any program containing the preprocessor directive `#include <stdio.h>`, which includes a file containing the function prototypes for functions such as `printf` and `scanf`.
- Other functions in the file can use `printf` and `scanf` to accomplish their tasks.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- The `printf` and `scanf` functions are defined in other files.
- We do *not* need to know *where* they're defined.
- We're simply reusing the code in our programs.
- The linker resolves our references to these functions automatically.
- This process enables us to use the functions in the standard library.

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.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- *It's possible to restrict the scope of a global variable or function to the file in which it's defined.*
- The storage-class specifier `static`, when applied to a global variable or a function, prevents it from being used by any function that's not defined in the same file.
- This is referred to as internal linkage.
- Global variables and functions that are *not* preceded by `static` in their definitions have external linkage—they can be accessed in other files if those files contain proper declarations and/or function prototypes.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- The global variable declaration
  - `static const double PI = 3.14159;`

  creates constant variable `PI` of type `double`, initializes it to `3.14159` and indicates that `PI` is known *only* to functions in the file in which it's defined.
- The `static` specifier is commonly used with utility functions that are called only by functions in a particular file.
- If a function is not required outside a particular file, the principle of least privilege should be enforced by using `static`.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- If a function is defined *before* it's used in a file, `static` should be applied to the function definition.
- Otherwise, `static` should be applied to the function prototype.
- When building large programs in multiple source files, compiling the program becomes tedious if small changes are made to one file and the entire program must be recompiled.
- Many systems provide special utilities that recompile *only* the modified program file.

14.5 Notes on Compiling Multiple-Source-File Programs (Cont.)

- On Linux/UNIX systems the utility is called `make`.
- Utility `make` reads a file called `makefile` that contains instructions for compiling and linking the program.
- Products such as Eclipse™ and Microsoft® Visual C++® provide similar utilities as well.

14.6 Program Termination with `exit` and `atexit`

- The general utilities library (`<stdlib.h>`) provides methods of terminating program execution by means other than a conventional return from function `main`.
- Function `exit` causes a program to terminate.
- The function often is used to terminate a program when an input error is detected, or when a file to be processed by the program cannot be opened.
- Function `atexit` registers a function that should be called upon *successful termination* of the program—i.e., either when the program terminates by reaching the end of `main`, or when `exit` is invoked.

14.6 Program Termination with `exit` and `atexit` (Cont.)

- Function `atexit` takes as an argument a pointer to a function (i.e., the *function name*).
- *Functions called at program termination cannot have arguments and cannot return a value.*
- Function `exit` takes one argument.
- The argument is normally the symbolic constant `EXIT_SUCCESS` or the symbolic constant `EXIT_FAILURE`.
- If `exit` is called with `EXIT_SUCCESS`, the implementation-defined value for successful termination is returned to the calling environment.

14.6 Program Termination with `exit` and `atexit` (Cont.)

- If `exit` is called with `EXIT_FAILURE`, the implementation-defined value for unsuccessful termination is returned.
- When function `exit` is invoked, any functions previously registered with `atexit` are invoked in the *reverse* order of their registration, all streams associated with the program are flushed and closed, and control returns to the host environment.
- Figure 14.4 tests functions `exit` and `atexit`.
- The program prompts the user to determine whether the program should be terminated with `exit` or by reaching the end of `main`.
- Function `print` is executed at program termination.

```c
// Fig. 14.4: fig14_04.c
// Using the exit and atexit functions
#include <stdio.h>
#include <stdlib.h>

void print(void); // prototype

int main(void)
{
   atexit(print); // register function print
   puts("Enter 1 to terminate program with function exit"
      "\nEnter 2 to terminate program normally");
   int answer; // user's menu choice
   scanf("%d", &answer);

   // call exit if answer is 1
   if (answer == 1) {
      puts("\nTerminating program with function exit");
      exit(EXIT_SUCCESS);
   }

   puts("\nTerminating program by reaching the end of main");
}

```

Fig. 14.4 | Using the `exit` and `atexit` functions. (Part 1 of 2.)

```
25  // display message before termination
26  void print(void)
27  {
28     puts("Executing function print at program "
29        "termination\nProgram terminated");
30  }
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

Fig. 14.4 | Using the `exit` and `atexit` functions. (Part 2 of 2.)

14.7 Suffixes for Integer and Floating-Point Literals

- C provides integer and floating-point *suffixes* for explicitly specifying the types of integer and floating-point constants.
- The integer suffixes are: `u` or `U` for an `unsigned int`, `l` or `L` for a `long int`, and `LL` or `ll` for a `long long int`.
- The following literals are of type `unsigned int`, `long int`, `unsigned long int` and `unsigned long long int`, respectively:
  - `174u`
    `8358L`
    `28373ul`
    `9876543210llu`

14.7 Suffixes for Integer and Floating-Point Constants (Cont.)

- The floating-point suffixes are: `f` or `F` for a `float`, and `l` or `L` for a `long double`.
- The following constants are of type `float` and `long double`, respectively:
  - `1.28f`
    `3.14159L`

14.8 Signal Handling

- An external asynchronous event, or signal, can cause a program to terminate prematurely.
- Some events include interrupts (typing *<Ctrl> c* on a Linux/UNIX or Windows system), illegal instructions, segmentation violations, termination orders from the operating system and floating-point exceptions (division by zero or multiplying large floating-point values).

14.8 Signal Handling (Cont.)

- The signal-handling library (`<signal.h>`) provides the capability to trap unexpected events with function `signal`.
- Function `signal` receives two arguments—an integer *signal number* and a *pointer to the signal-handling function*.
- Signals can be generated by function `raise` which takes an integer signal number as an argument.
- Figure 14.5 summarizes the *standard signals* defined in header file `<signal.h>`.

Signal	Explanation
SIGABRT	Abnormal termination of the program (such as a call to function `abort`).
SIGFPE	An erroneous arithmetic operation, such as a divide-by-zero or an operation resulting in overflow.
SIGILL	Detection of an illegal instruction.
SIGINT	Receipt of an interactive attention signal (<*Ctrl*> *c* or <*command*> *c*).
SIGSEGV	An attempt to access memory that is not allocated to a program.
SIGTERM	A termination request sent to the program.

Fig. 14.5 | `signal.h` standard signals.

14.8 Signal Handling (Cont.)

- Figure 14.6 uses function `signal` to *trap* a `SIGINT`.
- The program calls `signal` with `SIGINT` and a pointer to function `signalHandler` (remember that the name of a function is a pointer to the beginning of the function).
- When a signal of type `SIGINT` occurs, control passes to function `signalHandler`, which prints a message and gives the user the option to continue normal execution of the program.
- If the user wishes to continue execution, the signal handler is reinitialized by calling `signal` again and control returns to the point in the program at which the signal was detected.

14.8 Signal Handling (Cont.)

- In this program, function `raise` is used to simulate a `SIGINT`.
- A random number between `1` and `50` is chosen.
- If the number is `25`, `raise` is called to generate the signal.
- Normally, `SIGINT`s are initiated outside the program.
- For example, typing *<Ctrl> c* during program execution on a Linux/UNIX or Windows system generates a `SIGINT` that *terminates* program execution.
- Signal handling can be used to trap the `SIGINT` and prevent the program from being terminated.

```c
// Fig. 14.6: fig14_06.c
// Using signal handling
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <time.h>

void signalHandler(int signalValue); // prototype

int main(void)
{
   signal(SIGINT, signalHandler); // register signal handler
   srand(time(NULL));

   // output numbers 1 to 100
   for (int i = 1; i <= 100; ++i) {
      int x = 1 + rand() % 50; // generate random number to raise SIGINT

      // raise SIGINT when x is 25
      if (x == 25) {
         raise(SIGINT);
      }

```

Fig. 14.6 | Using signal handling. (Part 1 of 4.)

```
24          printf("%4d", i);
25
26          // output \n when i is a multiple of 10
27          if (i % 10 == 0) {
28             printf("%s", "\n");
29          }
30       }
31    }
32
33    // handles signal
34    void signalHandler(int signalValue)
35    {
36       printf("%s%d%s\n%s",
37          "\nInterrupt signal (", signalValue, ") received.",
38          "Do you wish to continue (1 = yes or 2 = no)? ");
39       int response; // user's response to signal (1 or 2)
40       scanf("%d", &response);
41
42       // check for invalid responses
43       while (response != 1 && response != 2) {
44          printf("%s", "(1 = yes or 2 = no)? ");
45          scanf("%d", &response);
46       }
47
```

Fig. 14.6 | Using signal handling. (Part 2 of 4.)

```
48      // determine whether it's time to exit
49      if (response == 1) {
50         // reregister signal handler for next SIGINT
51         signal(SIGINT, signalHandler);
52      }
53      else {
54         exit(EXIT_SUCCESS);
55      }
56   }
```

Fig. 14.6 | Using signal handling. (Part 3 of 4.)

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 1

  94  95  96
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2
```

Fig. 14.6 | Using signal handling. (Part 4 of 4.)

14.9 Dynamic Memory Allocation: Functions `calloc` and `realloc`

- Chapter 12 introduced the notion of dynamically allocating memory using function `malloc`.
- As we stated in Chapter 12, arrays are better than linked lists for rapid sorting, searching and data access.
- However, arrays are normally static data structures.
- The general utilities library (`stdlib.h`) provides two other functions for dynamic memory allocation—`calloc` and `realloc`.
- These functions can be used to create and modify dynamic arrays.

14.9 Dynamic Memory Allocation: Functions `calloc` and `realloc` (Cont.)

- As shown in Chapter 7, a pointer to an array can be subscripted like an array.
- Thus, a pointer to a contiguous portion of memory created by `calloc` can be manipulated as an array.
- Function `calloc` dynamically allocates memory for an array.
- The prototype for `calloc` is
  - `void *calloc(size_t nmemb, size_t size);`
- Its two arguments represent the *number of elements* (`nmemb`) and the size of each element (`size`).
- Function `calloc` also initializes the elements of the array to zero.

14.9 Dynamic Memory Allocation: Functions `calloc` and `realloc` (Cont.)

- The function returns a pointer to the allocated memory, or a `NULL` pointer if the memory is *not* allocated.
- The primary difference between `malloc` and `calloc` is that `calloc` *clears the memory* it allocates and `malloc` *does not*.
- Function `realloc` *changes the size* of an object allocated by a previous call to `malloc`, `calloc` or `realloc`.
- The original object's contents are *not modified* provided that the amount of memory allocated is *larger* than the amount allocated previously.

14.9 Dynamic Memory Allocation: Functions `calloc` and `realloc` (Cont.)

- Otherwise, the contents are unchanged up to the size of the new object.
- The prototype for `realloc` is
  - `void *realloc(void *ptr, size_t size);`
- The two arguments are a pointer to the original object (`ptr`) and the *new size* of the object (`size`).
- If `ptr` is `NULL`, `realloc` works identically to `malloc`.

14.9 Dynamic Memory Allocation: Functions `calloc` and `realloc` (Cont.)

- If `ptr` is not `NULL` and size is greater than zero, `realloc` tries to *allocate a new block of memory* for the object.
- If the new space cannot be allocated, the object pointed to by `ptr` is unchanged.
- Function `realloc` returns either a pointer to the reallocated memory, or a `NULL` pointer to indicate that the memory was not reallocated.

Error-Prevention Tip 14.1

Avoid zero-sized allocations in calls to `malloc`, `calloc` *and* `realloc`.

14.10 Unconditional Branching with `goto`

- Throughout the text we've stressed the importance of using structured programming techniques to build reliable software that's easy to debug, maintain and modify.
- In some cases, performance is more important than strict adherence to structured programming techniques.
- In these cases, some unstructured programming techniques may be used.
- For example, we can use `break` to terminate execution of a repetition structure before the loop-continuation condition becomes false.
- This saves unnecessary repetitions of the loop if the task is completed before loop termination.

14.10 Unconditional Branching with goto (Cont.)

- Another instance of unstructured programming is the goto statement—an unconditional branch.
- The result of the goto statement is a change in the flow of control to the first statement after the label specified in the goto statement.
- A label is an identifier followed by a colon.
- A label must appear in the same function as the goto statement that refers to it.
- Figure 14.7 uses goto statements to loop ten times and print the counter value each time.

14.10 Unconditional Branching with `goto` (Cont.)

- After initializing `count` to `1`, we test `count` to determine whether it's greater than `10` (the label `start:` is skipped because labels do not perform any action).
- If so, control is transferred from the `goto` to the first statement after the label `end:` Otherwise, we print and increment `count`, and control transfers from the `goto` to the first statement after the label `start:`

```c
// Fig. 14.7: fig14_07.c
// Using the goto statement
#include <stdio.h>

int main(void)
{
   int count = 1; // initialize count

   start: // label

      if (count > 10) {
         goto end;
      }

      printf("%d  ", count);
      ++count;

      goto start; // goto start on line 9

   end: // label
      putchar('\n');
}
```

Fig. 14.7 | Using the goto statement. (Part 1 of 2.)

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 14.7 | Using the `goto` statement. (Part 2 of 2.)

14.10 Unconditional Branching with `goto` (Cont.)

- In Chapter 3, we stated that only three control structures are required to write any program—sequence, selection and repetition.
- When the rules of structured programming are followed, it's possible to create deeply nested control structures from which it's difficult to escape efficiently.
- Some programmers use `goto` statements in such situations as a quick exit from a deeply nested structure.

14.10 Unconditional Branching with `goto` (Cont.)

- This can eliminate the need to test multiple conditions to escape from a control structure.
- There are some additional situations where `goto` is actually recommended—see, for example, CERT recommendation MEM12-C, "Consider using a Goto-Chain when leaving a function on error when using and releasing resources."

Performance Tip 14.1

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

Software Engineering Observation 14.3

The `goto` *statement is unstructured and can lead to programs that are more difficult to debug, maintain and modify.*