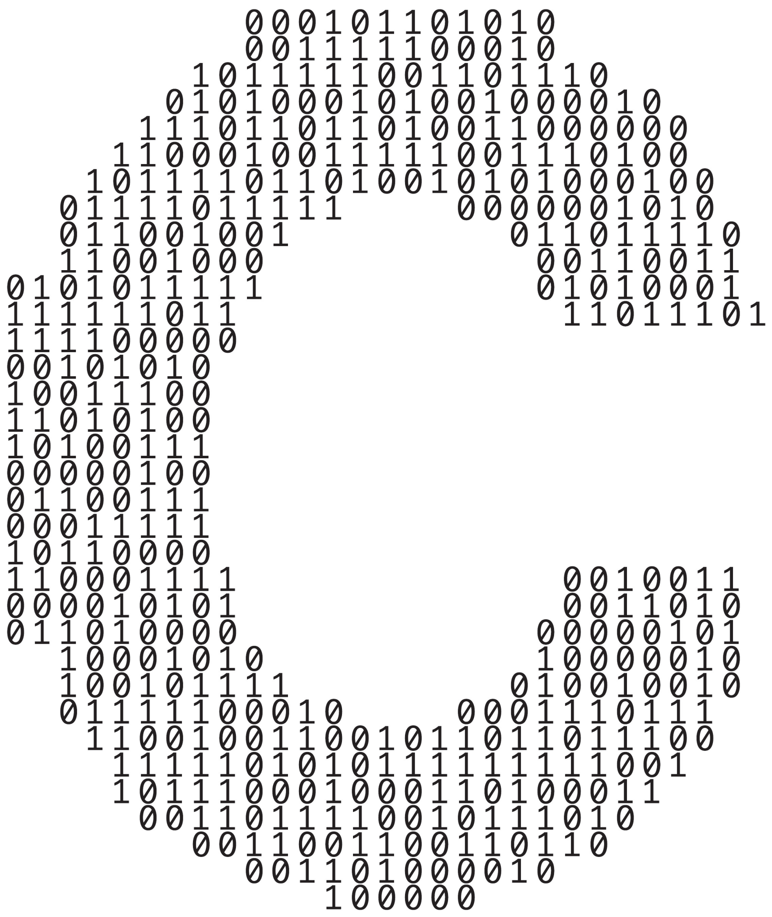
Other Topics

15



Objectives

In this chapter, you'll:

- 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.

Outline

15.1 Introduction

This chapter presents additional topics not ordinarily covered in introductory courses. Some capabilities discussed here are specific to particular operating systems, especially macOS/Linux and Windows.

15.2 Variable-Length Argument Lists

Most programs in the text have used the standard-library function `printf`. At 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 function prototype indicates that the function receives a *variable number of arguments of any type*. You can use this syntax to define your own functions with **variable-length argument lists**. The ellipsis must be the last parameter. ERR ⊗ Placing the ellipsis in the middle of the parameter list is a syntax error.

The following table contains the **variable arguments (`<stdarg.h>`) header's** macros and definitions for building functions with variable-length argument lists:

Identifier	Explanation
`va_list`	A type 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 you must invoke before accessing a variable-length argument list's arguments. This 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 variable-length argument list's next argument value. The value has the type you specify as the macro's second argument. Each use of `va_arg` modifies the object declared with `va_list` to point to the next argument.
`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.

Figure 15.1 demonstrates a function average (lines 23–36) with a variable-length argument list. The function's first argument is the number of values to average.

```
 1  // fig15_01.c
 2  // Using variable-length argument lists
 3  #include <stdarg.h>
 4  #include <stdio.h>
 5
 6  double average(int i, ...); // ... represents variable arguments
 7
 8  int main(void) {
 9     double w = 37.5;
10     double x = 22.5;
11     double y = 1.7;
12     double z = 10.2;
13
14     printf("%s%.1f; %s%.1f; %s%.1f; %s%.1f\n\n",
15        "w = ", w, "x = ", x, "y = ", y, "z = ", z);
16     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 ", average(4, w, x, y, z));
20  }
21
22  // calculate average
23  double average(int i, ...) {
24     double total = 0; // initialize total
25     va_list ap; // stores information needed by va_start and va_end
26
27     va_start(ap, i); // initializes the va_list object
28
29     // process variable-length argument list
30     for (int j = 1; j <= i; ++j) {
31        total += va_arg(ap, double);
32     }
33
34     va_end(ap); // clean up variable-length argument list
35     return total / i; // calculate average
36  }
```

```
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. 15.1 | Using variable-length argument lists.

The average function (lines 23–36) uses all the definitions and macros of header <stdarg.h>, except va_copy (Section C.7.8), which was added in C11. The **va_list** variable ap (short for "argument pointer"; line 25) is used by macros **va_start**, **va_arg** and **va_end** to process function average's variable-length argument list. First, the function invokes macro va_start (line 27) to initialize object ap for use by va_arg and va_end. The va_start macro receives:

- the object ap, and
- the identifier of the rightmost argument in the parameter list *before* the ellipsis (i in this example)—va_start uses this argument to determine where the variable-length argument list begins.

Next, the average function repeatedly adds the variable-length argument list's arguments to the variable total (lines 30–32). The macro va_arg retrieves the next value to add to total. The macro receives two arguments:

- the object ap, and
- the value *type* expected in the argument list—double in this case.

The macro returns the argument's value. Line 34 invokes the macro va_end with the object ap as an argument to facilitate a normal return to the caller from average. Finally, line 35 calculates the average and returns it to main.

You might wonder how functions with variable-length argument lists like printf and scanf know what type to use in each va_arg macro call. The answer is that, as the program executes, they scan the format conversion specifiers in the format control string to determine the type of the next argument to process.

✓ Self Check

1 *(Fill-In)* The function prototype for printf is

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

The ellipsis (...) in the prototype indicates that the function receives ________.
Answer: a variable number of arguments of any type.

2 *(Multiple Choice)* Which macro corresponds to the description: "To access the arguments in a variable-length argument list, an object of this type must be defined."

a) va_start
b) va_end
c) va_list
d) va_arg

Answer: c.

15.3 Using Command-Line Arguments

Command-line arguments are commonly used to pass options and filenames to a program. The main function may receive arguments from a command line if the function's parameter list contains the parameters int argc and char *argv[]:

- The **argc** parameter receives the number of command-line arguments that the user has entered.
- The **argv** parameter is an array of strings containing the command-line arguments.

Figure 15.2 copies a file one character at a time into another file. Assume that the executable file for this program is called `mycopy`. A typical command line for executing this program is

```
mycopy input output
```

This command line indicates that the file `input` should be copied to the file `output`. When the program executes, if `argc` is not `3` (`mycopy` counts as one of the arguments), the program prints an error message (line 8) and terminates. Otherwise, the array `argv` contains the strings `"mycopy"`, `"input"` and `"output"`. This program uses its second and third command-line arguments as filenames.

```
 1  // fig15_02.c
 2  // Using command-line arguments
 3  #include <stdio.h>
 4  
 5  int main(int argc, char *argv[]) {
 6     // check number of command-line arguments
 7     if (argc != 3) {
 8        puts("Usage: mycopy infile outfile");
 9     }
10     else {
11        FILE *inFilePtr = NULL; // input file pointer
12  
13        // try to open the input file
14        if ((inFilePtr = fopen(argv[1], "r")) != NULL) {
15           FILE *outFilePtr = NULL; // output file pointer
16  
17           // try to open the output file
18           if ((outFilePtr = fopen(argv[2], "w")) != NULL) {
19              int c = 0; // holds characters read from source file
20  
21              // read and output characters
22              while ((c = fgetc(inFilePtr)) != EOF) {
23                 fputc(c, outFilePtr);
24              }
25  
26              fclose(outFilePtr); // close the output file
27           }
28           else { // output file could not be opened
29              printf("File \"%s\" could not be opened\n", argv[2]);
30           }
31  
32           fclose(inFilePtr); // close the input file
33        }
34        else { // input file could not be opened
35           printf("File \"%s\" could not be opened\n", argv[1]);
36        }
37     }
38  }
```

Fig. 15.2 | Using command-line arguments.

We use the function `fopen` to open these files for reading (line 14) and writing (line 18), respectively. If the program opens both files successfully, lines 22–24 read characters from the file `input` and write them to the file `output`. This process continues until the end of `input` is reached. Then the program terminates. The result is an exact copy of the file `input`—if no errors occur during processing.

[*Note:* In Visual C++, you specify 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**.]

✓ Self Check

1 *(Fill-In)* You can pass command-line arguments to `main` by including parameters `int argc` and ________ in `main`'s parameter list.
Answer: `char *argv[]`.

2 *(Discussion)* Assuming `inFilePtr` represents a successfully opened input file and `outFilePtr` represents a successfully opened output file, what does the following code segment do?

```
while ((c = fgetc(inFilePtr)) != EOF) {
   fputc(c, outFilePtr);
}
```

Answer: This loop reads one character at a time from the input file and writes it to the output file until the end-of-file indicator for the input file is set.

15.4 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.

15.4.1 extern Declarations for Global Variables in Other Files

Chapter 5 introduced storage class and scope concepts. We learned that variables declared *outside* any function definition are *global variables*. Global variables are accessible to any function defined in the same file after the variable is declared. Global variables also can be accessible to functions in other files if they're declared in each file that uses them. For example, to refer to the global integer variable `flag` in another file, you can use the declaration

```
extern int flag;
```

The storage-class specifier **`extern`** indicates that `flag` is defined either *later in the same file or in a different file*. The compiler informs the linker that unresolved references to the variable `flag` appear in the file. If the linker finds a proper global definition, the linker resolves the references to `flag`. 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. You should avoid global variables unless application performance is critical because they violate the principle of least privilege. SE

15.4.2 Function Prototypes

Just as `extern` declarations can be used to declare that global variables are defined in 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 that calls the function, and compile the files together (see Section 14.2). Function prototypes indicate that the specified function is defined either later in the same file or in a different file. Again, the compiler does not resolve references to such a function—the linker performs that task. 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 `printf`, `scanf` and many other functions. A file that `#includes <stdio.h>` can use `printf` and `scanf`, even though they're defined in other files. We do not need to know where they're defined. The linker resolves our references to these functions automatically.

Software Reusability

Creating programs in multiple source files facilitates software reusability and good software engineering. Functions that are common to many applications should be stored in their own source files. Each source file should have a corresponding header containing the 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. SE

15.4.3 Restricting Scope with `static`

It's possible to restrict a global variable's or function's scope to the file in which it's defined. Applying the storage-class specifier `static` to a global variable or function prevents it from being used outside the file that defines it. This is known as **internal linkage**. Global variables and functions *not* preceded by `static` in their definitions have **external linkage**. They can be accessed in other files containing proper declarations.

The global variable definition

```
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 called only within a particular file. If a function is not required outside a file, enforce the principle of least privilege by applying `static` to both the function's definition and prototype.

✓ Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?

a) Variables declared outside any function definition are global variables.
b) Global variables are accessible to any function defined in the same file after the variable is declared.
c) Global variables also are accessible to functions in other files.
d) Once a global variable is defined, it's known to all the application's files.

Answer: d) is *false*. Actually, global variables must be declared in *each* file in which they're used.

2 *(Fill-In)* Any identifier that's declared at file scope is ________ by default.
Answer: `extern`.

3 *(True/False)* You should prefer global variables to local variables because global variables enforce the principle of least privilege.
Answer: False. Actually, you should avoid global variables unless application performance is critical because they *violate* the principle of least privilege.

4 *(Fill-In)* Applying `static` 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 called ________ linkage.
Answer: internal.

5 *(True/False)* The following global variable definition indicates that `PI` is known only to functions in the file in which it's defined:

```
static const double PI = 3.14159;
```

Answer: *True.*

15.5 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`.

exit Function

The **`exit`** function terminates a program immediately. This function often is used to terminate a program when an error is detected. The function takes one argument—normally, **`EXIT_SUCCESS`** or **`EXIT_FAILURE`**: These contain implementation-defined values for successful and unsuccessful termination.

atexit Function

The **`atexit`** function registers a function to call when the program terminates by reaching the end of `main` or when `exit` is invoked. This function takes as an argument another function's name. Recall that a function name is a pointer to that function. Functions called at program termination cannot have arguments and cannot return a value. When a program terminates, any functions previously registered with `atexit` are invoked in the reverse order of their registration.

Using Functions exit and atexit

Figure 15.3 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 in each case.

```c
// fig15_03.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\n"
        "Enter 2 to terminate program normally");
   int answer = 0; // user
   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");
}

// display message before termination
void print(void) {
   puts("Executing function print at program termination\n"
        "Program terminated");
}
```

```
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. 15.3 | Using the exit and atexit functions.

✓ Self Check

1 *(True/False)* Function `atexit` terminates a program immediately.
Answer: *False*. Actually, function `exit` causes a program to terminate immediately. Function `atexit` registers a function to call when a program terminates by reaching the end of `main` or when `exit` is invoked.

2 *(True/False)* Calling `exit` with `EXIT_SUCCESS` returns 1 to the calling environment, and calling `exit` with `EXIT_FAILURE` returns 0.
Answer: *False*. Actually, calling `exit` with `EXIT_SUCCESS` or `EXIT_FAILURE` returns *implementation-defined values* for successful or unsuccessful termination.

15.6 Suffixes for Integer and Floating-Point Literals

Integer and floating-point *suffixes* enable you to specify explicitly the data types of literal values. By default, an integer literal's type is determined by the first type capable of storing the value—`int`, then `long int`, then `unsigned long int`, etc. A floating-point literal with no suffix has type `double`.

The integer suffixes are `u` or `U` for `unsigned int`s, `l` or `L` for `long int`s, and `ll` or `LL` for `long long int`s. `L` and `LL` are preferred for readability, as a lowercase `l` can be mistaken as a `1` (one). You can combine `u` or `U` with those for `long int` and `long long int` to create unsigned literals for the larger integer types. The following literals have types `unsigned int`, `long int`, `unsigned long int` and `unsigned long long int`:

```
174u
8358L
28373ul
9876543210llu
```

The floating-point suffixes are `f` or `F` for `float`s, and `l` or `L` for `long double`s. Again, `L` is preferred for readability. The following are `float` and `long double` literals:

```
1.28f
3.14159L
```

✓ Self Check

1 *(Fill-In)* C provides integer and floating-point ________ for explicitly specifying the types of integer and floating-point literal values.
Answer: suffixes.

2 *(Fill-In)* The following constants have types ________ and ________.

```
1.28f
3.14159L
```

Answer: `float`, `long double`.

15.7 Signal Handling

An external asynchronous **event**, or **signal**, can cause a program to terminate prematurely. Some events include:

- interrupts, such as typing <*Ctrl*> *c* (Linux or Windows) or <*command*> *c* (macOS), and
- termination orders from the operating system.

The **signal-handling library** (`<signal.h>`) enables programs to **trap** unexpected events with function **`signal`**, which receives two arguments:

- an integer signal number, and
- a pointer to a signal-handling function.

A program can generate signals by calling function **`raise`**, which takes an integer signal number as an argument. The following table summarizes the *standard signals* from the `<signal.h>` header:

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.

Demonstrating Signal Handling

Figure 15.4 uses function `signal` to *trap* a `SIGINT`. Line 11 calls `signal` with `SIGINT` and a pointer to the function `signalHandler`. When a `SIGINT` signal occurs, control passes to function `signalHandler`, which prints a message and gives the user the option to continue normal program execution. If the user wishes to continue execution, line 49 reinitializes the signal handler by calling `signal` again, and control returns to the point in the program at which the signal was detected.

```
 1  // fig15_04.c
 2  // Using signal handling
 3  #include <signal.h>
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <time.h>
 7
 8  void signalHandler(int signalValue); // prototype
 9
10  int main(void) {
11     signal(SIGINT, signalHandler); // register signal handler
12     srand(time(NULL));
13
```

Fig. 15.4 | Using signal handling. (Part 1 of 3.)

```c
   // 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);
      }

      printf("%4d", i);

      // output \n when i is a multiple of 10
      if (i % 10 == 0) {
         printf("%s", "\n");
      }
   }
}

// handles signal
void signalHandler(int signalValue) {
   printf("\n%s%d%s\n%s",
      "Interrupt signal (", signalValue, ") received.",
      "Do you wish to continue (1 = yes or 2 = no)? ");
   int response = 0; // user
   scanf("%d", &response);

   // check for invalid responses
   while (response != 1 && response != 2) {
      printf("%s", "(1 = yes or 2 = no)? ");
      scanf("%d", &response);
   }

   // determine whether to continue
   if (response == 1) {
      // reregister signal handler for next SIGINT
      signal(SIGINT, signalHandler);
   }
   else {
      exit(EXIT_SUCCESS);
   }
}
```

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

Fig. 15.4 | Using signal handling. (Part 2 of 3.)

```
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2
```

Fig. 15.4 | Using signal handling. (Part 3 of 3.)

In this program, function `raise` simulates a `SIGINT`. We choose a random number between `1` and `50`. If the number is `25`, line 20 calls `raise` to generate the signal. Normally, `SIGINT`s are initiated outside the program when someone types *<Ctrl> c* (Linux or Windows) or *<command> c* (macOS) to terminate program execution. Signal handling can be used to trap the `SIGINT` and prevent the program from terminating.

✓ Self Check

1 *(Fill-In)* An external asynchronous event, or ________, can cause a program to terminate prematurely.
Answer: signal.

2 *(Multiple Choice)* Which standard signal is described by: "An erroneous arithmetic operation, such as a divide-by-zero or an operation resulting in overflow."

a) `SIGILL`
b) `SIGABRT`
c) `SIGINT`
d) `SIGFPE`

Answer: d.

15.8 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. Arrays are normally static data structures that cannot be resized. The general utilities library (`<stdlib.h>`) provides dynamic memory allocation functions **`calloc`** and **`realloc`** to create dynamic arrays and modify their sizes.

calloc Function

The `calloc` ("contiguous allocation") function

```
void *calloc(size_t nmemb, size_t size);
```

dynamically allocates an array. Its two arguments are

- the array's number of elements (`nmemb`), and
- each element's size (`size`).

Function `calloc` also initializes the array's elements to zero. The function returns a pointer to the allocated memory or a `NULL` pointer if it could not allocate the memory. The primary difference between `malloc` and `calloc` is that `calloc` clears the memory it allocates, whereas `malloc` does not.

`realloc` Function

The `realloc` function

```
void *realloc(void *ptr, size_t size);
```

changes the size of an object allocated by a previous `malloc`, `calloc` or `realloc` call. The original object's contents are not modified as long as the amount of memory allocated is larger than the amount allocated previously. Otherwise, the contents are unchanged up to the new object's size. The functions two arguments are

- a pointer to the original object (`ptr`), and
- the object's new size (`size`).

If `ptr` is `NULL`, `realloc` works identically to `malloc`. 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.

✓ Self Check

1 *(Fill-In)* The general utilities library (`<stdlib.h>`) dynamic memory allocation functions ________ and ________ create and modify dynamic arrays.
Answer: `calloc` and `realloc`.

2 *(Multiple Choice)* Which of the following statements is *false*?
a) Function `calloc` dynamically allocates memory for an array.
b) Function `calloc`'s parameters `size_t nmemb` and `size_t size` represent the new array's number of elements and each element's size.
c) Function `calloc` initializes a dynamically allocated array's elements to zero. The function returns a pointer to the allocated memory, or `NULL` if the memory could not be allocated.
d) Functions `malloc` and `calloc` clear the memory they allocate.
Answer: d) is *false*. Actually, `calloc` clears the memory it allocates, but `malloc` does not.

3 *(True/False)* If `realloc`'s first argument is `NULL`, it works identically to `malloc`. Otherwise, if `realloc`'s size argument is greater than zero, it tries to allocate a new block of memory. If it cannot be allocated, the object pointed to by the function's first argument is unchanged. The function returns either a pointer to the reallocated memory or a `NULL` pointer to indicate that the memory was not reallocated.
Answer: *True.*

15.9 goto: Unconditional Branching

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 an iteration statement's execution before the loop-continuation condition becomes false. This saves unnecessary iterations of the loop if the task is completed *before* loop termination.

Another instance of unstructured programming is the **`goto` statement**—an unconditional branch. The `goto` statement alters the flow of control, continuing execution with the first statement after the **label** specified in the 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. Labels need not be unique among functions.

Demonstrating goto

Figure 15.5 uses `goto` statements to loop ten times and print a counter value each time. Line 6 initializes `count` to `1`. The label `start:` is skipped, because labels do not perform any action. Line 9 tests whether `count` is greater than `10`. If so, line 10 transfers control from the `goto` to the first statement after the label `end:` (line 19). Otherwise, lines 13–14 print and increment `count`, and control transfers from the `goto` (line 16) to the first statement after the label `start:` (line 9).

```
 1  // fig15_05.c
 2  // Using the goto statement
 3  #include <stdio.h>
 4
 5  int main(void) {
 6     int count = 1; // initialize count
 7
 8     start: // label
 9        if (count > 10) {
10           goto end;
11        }
12
13        printf("%d  ", count);
14        ++count;
15
16        goto start; // goto start on line 9
17
18     end: // label
19        putchar('\n');
20  }
```

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

Fig. 15.5 | Using the `goto` statement.

Chapter 3 stated you can write any program in terms of sequence, selection and iteration statements. When following the structured-programming rules, you can create deeply nested control structures within a function 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. This eliminates the need to test multiple conditions to escape from a control structure. There are other 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." Note that the `goto` statement is unstructured and can lead to programs that are more difficult to debug, maintain and modify.

PERF

SE

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) Structured-programming techniques help you to build reliable software that's easy to debug, maintain and modify.
b) In some cases, performance is more important than strict adherence to structured-programming techniques. In these cases, you might choose to use some unstructured-programming techniques.
c) We can use `break` to terminate an iteration statement early. This saves unnecessary iterations of the loop if the task is completed before loop termination.
d) All of the above statements are *true*.

Answer: d.

2 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) An instance of unstructured programming is the `goto` statement—an unconditional branch.
b) The `goto` statement alters the flow of control by continuing execution with the first statement after the label specified in the statement. A label is an identifier followed by a colon.
c) Labels must be unique among all the functions in an application. A label that's the target of a particular `goto` statement may appear in any function in an application.
d) All of the above statements are true.

Answer: c) is *false*. Actually, labels need not be unique among functions. Also, a label that's the target of a `goto` statement in a function must appear in that function.

3 *(True/False)* When you follow the rules of structured programming, it's possible to create deeply nested control structures within a function 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. This eliminates the need to test multiple conditions to escape from a control structure.
Answer: *True*.

Summary

Section 15.2 Variable-Length Argument Lists

- The **header `<stdarg.h>`** (p. 754) provides capabilities for building functions with **variable-length argument lists.**
- An **ellipsis** (`...`; p. 754) in a function prototype indicates a variable number of arguments.
- A **`va_list`** (p. 755) holds information needed by macros `va_start`, `va_arg` and `va_end`.
- Macro **`va_start`** (p. 755) initializes a `va_list` object for use by the `va_arg` and `va_end`.
- Macro **`va_arg`** (p. 755) expands to the value and type of the variable-length argument list's next argument. Each invocation of `va_arg` modifies the object declared with `va_list` to point to the next argument.
- Macro **`va_end`** (p. 755) facilitates a normal return from a function whose variable-length argument list was referred to by the `va_start` macro.

Section 15.3 Using Command-Line Arguments

- To pass arguments to `main` from the command line, include the parameters **`int argc`** and **`char *argv[]`** (p. 756) in `main`'s parameter list. Parameter `argc` receives the number of command-line arguments. Parameter `argv` is an array of strings in which the command-line arguments are stored.

Section 15.4 Compiling Multiple-Source-File Programs

- A function definition must be entirely contained in one file.
- The **storage-class specifier `extern`** (p. 758) indicates that a variable is defined either later in the same file or in a different file of the program.
- **Global variables** must be declared in each file in which they're used.
- A **function prototype** can extend a function's scope beyond the file in which it's defined.
- Applying **storage-class specifier `static`** to a global variable or function prevents it from being used outside the current file. This is called **internal linkage** (p. 759). Global variables and functions not preceded by `static` have **external linkage** (p. 759) and can be accessed in other files if those files contain proper declarations or function prototypes.
- 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, apply `static` to it to enforce the principle of least privilege.

Section 15.5 Program Termination with `exit` and `atexit`

- Function **`exit`** (p. 760) forces a program to terminate.
- Function **`atexit`** (p. 760) registers a function to call when the program terminates by reaching the end of `main` or when `exit` is invoked.
- Function `atexit` takes a pointer to a function as an argument. Functions called at program termination cannot have arguments and cannot return a value.
- Function `exit` takes one argument, normally the symbolic constant **`EXIT_SUCCESS`** (p. 760) or the symbolic constant **`EXIT_FAILURE`** (p. 760).
- When function `exit` is invoked, any functions registered with `atexit` are invoked in the reverse order of their registration.

Section 15.6 Suffixes for Integer and Floating-Point Literals

- **Integer** and **floating-point suffixes** can be used to specify the types of integer and floating-point constants. The integer suffixes are `u` or `U` for an `unsigned` integer, `l` or `L` for a `long` integer, and `ul` or `UL` for an `unsigned long` integer. The type of an integer constant with no suffix is determined by the first type capable of storing a value of that size (`int`, then `long int`, then `unsigned long int`, etc.). The floating-point suffixes are `f` or `F` for a `float`, and `l` or `L` for a `long double`. A floating-point constant with no suffix has type `double`.

Section 15.7 Signal Handling

- The **signal-handling library** (p. 763) enables trapping of unexpected events with function **`signal`** (p. 763).
- Function `signal` receives two arguments—an integer signal number and a pointer to the signal-handling function.
- Signals can also be generated with **function `raise`** (p. 763) and an integer argument.

Section 15.8 Dynamic Memory Allocation: Functions `calloc` and `realloc`

- The **general utilities library** (**`<stdlib.h>`**) provides dynamic memory allocation functions `calloc` and `realloc` for creating dynamic arrays and resizing them.
- Function **`calloc`** (p. 765) allocates memory for an array. It receives the array's number of elements and each element's size and initializes the array's elements to zero. It returns either a pointer to the allocated memory or a `NULL` pointer if the memory is not allocated.
- Function **`realloc`** changes the size of an object allocated by a previous `malloc`, `calloc` or `realloc` call. The original object's contents are not modified as long as the amount of memory allocated is larger than the amount allocated previously.
- Function `realloc` receives a pointer to the original object and the new size of the object. If `ptr` is `NULL`, `realloc` works identically to `malloc`. Otherwise, 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 memory was not reallocated.

Section 15.9 Unconditional Branching with goto

- The **`goto` statement** (p. 767) alters a program's flow of control. Program execution continues at the first statement after the **label** (p. 767) 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.

Self-Review Exercise

15.1 Fill-In the blanks in each of the following:

a) A(n) ________ in the parameter list of a function indicates that the function can receive a variable number of arguments.

b) Macro ________ must be invoked before the arguments in a variable-length argument list can be accessed.

c) Macro ________ accesses the individual arguments of a variable-length argument list.

d) Macro ________ facilitates a normal return from a function whose variable-length argument list was referred to by macro `va_start`.
e) Argument ________ of `main` receives the number of arguments in a command line.
f) Argument ________ of `main` stores command-line arguments as character strings.
g) Function ________ forces a program to terminate execution.
h) Function ________ registers a function to be called upon normal program termination.
i) An integer or floating-point ________ can be appended to an integer or floating-point constant to specify the exact type of the constant.
j) Function ________ can be used to trap unexpected events.
k) Function ________ generates a signal from within a program.
l) Function ________ dynamically allocates memory for an array and initializes the elements to zero.
m) Function ________ changes the size of a block of previously allocated dynamic memory.

Answers to Self-Review Exercise

15.1 a) ellipsis (`...`). b) `va_start`. c) `va_arg`. d) `va_end`. e) `argc`. f) `argv`. g) `exit`. h) `atexit`. i) suffix. j) `signal`. k) `raise`. l) `calloc`. m) `realloc`.

Exercises

15.2 *(Variable-Length Argument List: Calculating Products)* Write a program that calculates the product of a series of integers that are passed to function `product` using a variable-length argument list. Test your function with several calls, each with a different number of arguments.

15.3 *(Printing Command-Line Arguments)* Write a program that prints the command-line arguments of the program.

15.4 *(Sorting Integers)* Write a program that sorts an array of integers into ascending or descending order. Use command-line arguments to pass either argument `-a` for ascending order or `-d` for descending order. [*Note:* This is the standard format for passing options to a program in UNIX.]

15.5 *(Signal Handling)* Read the documentation for your compiler to determine which signals are supported by the signal-handling library (`<signal.h>`). Write a program that contains signal handlers for the standard signals `SIGABRT` and `SIGINT`. The program should trap these signals by calling function `abort` to generate a signal of type `SIGABRT` and by having the user type <*Ctrl*> *c* or <*command*> *C* to generate a signal of type `SIGINT`.

15.6 *(Dynamic Array Allocation)* Write a program that dynamically allocates an array of integers. The size of the array should be input from the keyboard. The elements of the array should be assigned values input from the keyboard. Print the array's values. Next, reallocate the memory for the array to half of the current number of elements. Print the array's remaining values to confirm they match the first half of the original array's values.

15.7 ***(Command-Line Arguments)*** Write a program that takes two filenames as command-line arguments, reads the first file's characters one at a time and writes them to the second file in reverse order.

A

Operator Precedence Chart

Operators are shown in decreasing order of precedence from top to bottom.

Operator	Type	Associativity
`()`	parentheses (function-call operator)	left to right
`[]`	array subscript	
`.`	member selection via object	
`->`	member selection via pointer	
`++`	unary postincrement	
`--`	unary postdecrement	
`++`	unary preincrement	right to left
`--`	unary predecrement	
`+`	unary plus	
`-`	unary minus	
`!`	unary logical negation	
`~`	unary bitwise complement	
(*type*)	C-style unary cast	
`*`	dereference	
`&`	address	
`sizeof`	determine size in bytes	
`*`	multiplication	left to right
`/`	division	
`%`	modulus	
`+`	addition	left to right
`-`	subtraction	
`<<`	bitwise left shift	left to right
`>>`	bitwise right shift	
`<`	relational less than	left to right
`<=`	relational less than or equal to	
`>`	relational greater than	
`>=`	relational greater than or equal to	
`==`	relational is equal to	left to right
`!=`	relational is not equal to	
`&`	bitwise AND	left to right

Operator	Type	Associativity
^	bitwise exclusive OR	left to right
|	bitwise inclusive OR	left to right
&&	logical AND	left to right
||	logical OR	left to right
?:	ternary conditional	right to left
=	assignment	right to left
+=	addition assignment	
-=	subtraction assignment	
*=	multiplication assignment	
/=	division assignment	
%=	modulus assignment	
&=	bitwise AND assignment	
^=	bitwise exclusive-OR assignment	
|=	bitwise inclusive-OR assignment	
<<=	bitwise left-shift assignment	
>>=	bitwise right-shift assignment	
,	comma	left to right

B

ASCII Character Set

In the following table, the digits in the left column are the left digits of the character code's decimal equivalent (0–127), and the digits in the top row are the right digits of the character code's decimal equivalent. For example, the character code for "A" in row number 6 and column number 5 is 65, and the character code for "&" in row number 3 and column number 8 is 38.

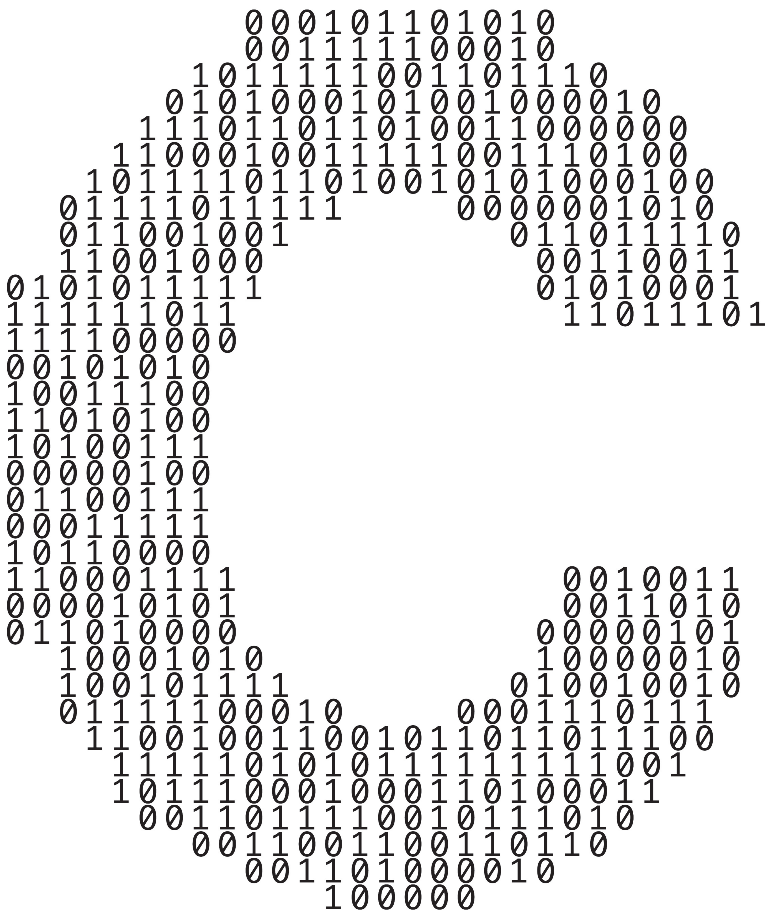
ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

This page is intentionally left blank

C

Multithreading/Multicore and Other C18/C11/C99 Topics



Objectives

In this appendix, you'll:

- Understand the purpose of C18.
- Learn the headers added in C99 and C11/C18.
- Initialize arrays and `struct`s with designated initializers.
- Use data type `bool` to create boolean variables whose data values can be `true` or `false`.
- Perform arithmetic operations on complex variables.
- Learn about preprocessor enhancements.
- Use multithreading to improve performance on today's multicore systems.

Outline

C.1 Introduction

The C99 (1999) and C11 (2011) standards refined and expanded C's capabilities. Since C11, there has been only one new version, C18[1] (2018). It "addressed defects in C11 without introducing new language features."[2] Some features added by the C99 and C11/C18 standards are designated as optional. Before using the features shown in this appendix, check that your compiler supports them. Our goal is to introduce these capabilities and provide resources for further reading.

We explain with complete code examples and code snippets designated initializers, compound literals, type `bool` and complex numbers. We provide brief explanations of additional features, including restricted pointers, reliable integer division, flexible array members, generic math, `inline` functions and `return` without expression.

We discuss C11/C18 capabilities, including improved Unicode® support, the function specifier `_Noreturn`, type-generic expressions, the `quick_exit` function, memory alignment control, static assertions, analyzability and floating-point types.

C11/C18 Multithreading

A key feature of this appendix is Section C.9's introduction to multithreading. In today's *multicore* systems, the hardware can put multiple processors (cores) to work on different parts of your task. This enables the tasks (and the program) to complete faster. To take advantage of multicore architecture from C programs, you need to write multithreaded applications. When a program splits tasks into separate threads,

1. ISO/IEC 9899:2018, Information technology — Programming languages — C, `https://www.iso.org/standard/74528.html`.
2. `https://en.wikipedia.org/wiki/C18_(C_standard_revision)`. Also `http://www.iso-9899.info/wiki/The_Standard`.

a multicore system can run those threads in parallel—that is, simultaneously. Section C.9 first demonstrates two long-running calculations performed in sequence. Then we separate those calculations into two threads to demonstrate the significant performance improvement of running the threads in parallel on multiple cores.

C.2 Headers Added in C99

The following table lists the standard-library headers added in C99—these remain available in C11/C18. We'll discuss the new C11/C18 headers in Section C.8.1.

Header	Explanation
`<complex.h>`	Contains support for complex numbers (see Section C.5).
`<fenv.h>`	Provides information about the C implementation's floating-point environment and capabilities.
`<inttypes.h>`	Defines portable integral types and provides format specifiers for them.
`<stdbool.h>`	Contains macros defining `bool`, `true` and `false`, used for boolean variables (see Section C.4).
`<stdint.h>`	Defines extended integer types and related macros.
`<tgmath.h>`	Provides type-generic macros that allow functions from `<math.h>` to be used with a variety of parameter types (see Section C.7).

C.3 Designated Initializers and Compound Literals

[This section can be read after Section 10.3.]

Designated initializers allow you to initialize array elements by subscript and `union` or `struct` members by name. Figure C.1 shows that we can use designated initializers to initialize specific array elements.

```c
// figC_01.c
// Initializing specific array elements with designated initializers.
#include <stdio.h>

int main(void) {
   int values[5] = {
      [0] = 123, // initialize element 0
      [4] = 456 // initialize element 4
   }; // semicolon is required

   // output array contents
   printf("values: ");

   for (size_t i = 0; i < 5; ++i) {
      printf("%d  ", values[i]);
   }
```

Fig. C.1 | Initializing specific array elements with designated initializers. (Part 1 of 2.)

```
17
18     puts("");
19  }
```

```
values: 123  0  0  0  456
```

Fig. C.1 | Initializing specific array elements with designated initializers. (Part 2 of 2.)

Lines 6–9 define an array and initialize its elements 0 and 4 within the braces. Note the syntax. You separate each initializer from the next by a comma. The initializer list's end brace must be followed by a semicolon. Elements that are not explicitly initialized are implicitly initialized to zero.

Compound Literals

You can use an initializer list to create an unnamed array, struct or union. This is known as a **compound literal**. For example, you could pass Fig. C.1's array to a function without having to declare it beforehand, as in

```
demoFunction((int [5]) {[0] = 1, [4] = 2});
```

Figure C.2 uses compound literals as designated initializers for specific elements in an array of structs. Lines 12 and 13 each use a designated initializer to explicitly initialize a struct element in the array. For example, in line 12, the following expression is a compound literal that creates an anonymous struct object of type struct twoInt:

```
{.x = 1, .y = 2}
```

That object's x and y members are initialized to 1 and 2. Designated initializers for struct and union members list each member's name preceded by a dot (.).

```
1   // figC_02.c
2   // Initializing struct members with designated initializers.
3   #include <stdio.h>
4
5   struct twoInt { // declare a struct of two integers
6      int x;
7      int y;
8   };
9
10  int main(void) {
11     struct twoInt a[5] = {
12        [0] = {.x = 1, .y = 2},
13        [4] = {.x = 10, .y = 20}
14     };
15
16     // output array contents
17     printf("%2s%5s\n", "x", "y");
18
```

Fig. C.2 | Initializing struct members with designated initializers. (Part 1 of 2.)

```
19      for (size_t i = 0; i < 5; ++i) {
20         printf("%2d%5d\n", a[i].x, a[i].y);
21      }
22   }
```

```
 x    y
 1    2
 0    0
 0    0
 0    0
10   20
```

Fig. C.2 | Initializing `struct` members with designated initializers. (Part 2 of 2.)

Lines 11–14 are more straightforward than following executable code, which does not use designated initializers:

```
struct twoInt a[5];

a[0].x = 1;
a[0].y = 2;
a[4].x = 10;
a[4].y = 20;
```

Using initializers rather than runtime assignments improves program startup time. PERF

C.4 Type bool

[This section can be read after Section 3.6.]

The boolean type—`_Bool`—can hold only the values 0 or 1. Recall that in C conditions, zero represents *false*, and any nonzero value represents *true*. Assigning *any* nonzero value to a `_Bool` sets it to 1. The `<stdbool.h>` header defines macros `bool`, `false` and `true`. These macros replace `bool` with the keyword `_Bool`, `false` with 0, and `true` with 1. Figure C.3 uses a function named `isEven` (lines 28–35) that returns the `bool` value `true` if the function's argument is even and `false` if it's odd.

```
 1   // figC_03.c
 2   // Using bool, true and false.
 3   #include <stdio.h>
 4   #include <stdbool.h> // allows the use of bool, true, and false
 5
 6   bool isEven(int number); // function prototype
 7
 8   int main(void) {
 9      // loop for 2 inputs
10      for (int i = 0; i < 2; ++i) {
11         printf("Enter an integer: ");
12         int input = 0; // value entered by user
13         scanf("%d", &input);
14
```

Fig. C.3 | Using `bool`, `true` and `false`. (Part 1 of 2.)

```
15        bool valueIsEven = isEven(input); // determine if input is even
16
17        // determine whether input is even
18        if (valueIsEven) {
19           printf("%d is even\n\n", input);
20        }
21        else {
22           printf("%d is odd\n\n", input);
23        }
24     }
25  }
26
27  // isEven returns true if number is even
28  bool isEven(int number) {
29     if (number % 2 == 0) { // is number divisible by 2?
30        return true;
31     }
32     else {
33        return false;
34     }
35  }
```

```
Enter an integer: 34
34 is even

Enter an integer: 23
23 is odd
```

Fig. C.3 | Using `bool`, `true` and `false`. (Part 2 of 2.)

Line 15 declares the `bool` variable `valueIsEven` and passes the user's input to function `isEven`, which returns a `bool` value. Line 29 determines whether the argument is divisible by 2. If so, line 30 returns `true`; otherwise, line 33 returns `false`. The result is assigned to `bool` variable `valueIsEven` in line 15. If `valueIsEven` is `true`, line 19 displays a string indicating that the value is even. If `valueIsEven` is `false`, line 22 displays a string indicating that the value is odd. Function `isEven`'s body can be written more concisely as

```
return number % 2 == 0;
```

but we wanted to demonstrate the `<stdbool.h>` header's `true` and `false` macros.

C.5 Complex Numbers

[This section can be read after Section 5.3.]
C99 introduced support for complex numbers and complex arithmetic. Figure C.4 shows basic complex-number operations. We compiled and ran this program using Apple's Xcode. Microsoft Visual C++ supports only the complex-number features defined by the C++ standard, not those from C.

```
 1  // figC_04.c
 2  // Complex number operations.
 3  #include <complex.h> // for complex type and math functions
 4  #include <stdio.h>
 5
 6  int main(void) {
 7     double complex a = 3.0 + 2.0 * I;
 8     double complex b = 2.7 + 4.9 * I;
 9
10     printf("a is %.1f + %.1fi\n", creal(a), cimag(a));
11     printf("b is %.1f + %.1fi\n", creal(b), cimag(b));
12
13     double complex sum = a + b; // perform complex addition
14     printf("a + b is: %.1f + %.1fi\n", creal(sum), cimag(sum));
15
16     double complex difference = a - b; // perform complex subtraction
17     printf("a - b is: %.1f + %.1fi\n", creal(difference), cimag(difference));
18
19     double complex product = a * b; // perform complex multiplicaton
20     printf("a * b is: %.1f + %.1fi\n", creal(product), cimag(product));
21
22     double complex quotient = a / b; // perform complex division
23     printf("a / b is: %.1f + %.1fi\n", creal(quotient), cimag(quotient));
24
25     double complex power = cpow(a, 2.0); // perform complex exponentiation
26     printf("a ^ b is: %.1f + %.1fi\n", creal(power), cimag(power));
27  }
```

```
a is 3.0 + 2.0i
b is 2.7 + 4.9i
a + b is: 5.7 + 6.9i
a - b is: 0.3 + -2.9i
a * b is: -1.7 + 20.1i
a / b is: 0.6 + -0.3i
a ^ b is: 5.0 + 12.0i
```

Fig. C.4 | Complex number operations.

To use **`complex`** numbers, include the **`<complex.h>`** header (line 3). This will expand the macro `complex` to the keyword **`_Complex`**—a type that reserves an array of exactly two elements, corresponding to the complex number's *real part* and *imaginary part*. You define `complex` variables as shown in lines 7, 8, 13, 16, 19, 22 and 25. We define each variable as `double complex`, indicating that the `complex` number's real and imaginary parts are stored as `double` values. C also supports `float complex` or `long double complex`.

The arithmetic operators work with complex numbers and the `<complex.h>` header provides additional math functions, such as `cpow` in line 25. You can also use the operators `!`, `++`, `--`, `&&`, `||`, `==`, `!=` and unary `&` with complex numbers.

Lines 13–26 output the results of various arithmetic operations. You access a complex number's *real part* and *imaginary part* via functions **`creal`** and **`cimag`**, respectively, as shown in line 10. In line 26's output, we use the symbol `^` to indicate exponentiation.

C.6 Macros with Variable-Length Argument Lists

Macros may have variable-length argument lists. This allows for macro wrappers around functions like `printf`. For example, to automatically add the name of the current file to a debug statement, you can define a macro as follows:

```
#define DEBUG(...) printf(__FILE__ ": " __VA_ARGS__)
```

This `DEBUG` macro takes a variable number of arguments, as indicated by the `...` in the argument list. As with functions, the `...` must be the last argument. Unlike functions, the `...` can be the macro's *only* argument. The identifier `__VA_ARGS__`, which begins and ends with two underscores, is a placeholder for the variable-length argument list. Assuming this macro appears in the file `file.c`, the preprocessor replaces the following macro call:

```
DEBUG("x = %d, y = %d\n", x, y);
```

with

```
printf("file.c" ": " "x = %d, y = %d\n", x, y);
```

Recall that strings separated by whitespace are concatenated during preprocessing, so the three string literals will be combined to form `printf`'s first argument.

C.7 Other C99 Features

Here we provide brief overviews of some additional C99 features. These include keywords, language capabilities and standard-library additions.

C.7.1 Compiler Minimum Resource Limits

[This section can be read after Section 15.4.]

Before C99, the standard required C implementations to support identifiers of

- no less than 31 characters for identifiers with internal linkage, and
- no less than six characters for identifiers with external linkage (Section 15.4).

C99 increased these limits to 63 characters for identifiers with internal linkage and 31 characters for identifiers with external linkage. These are just lower limits. Compilers are free to support identifiers with more characters than these limits. For more information, see the C18 standard's Section 5.2.4.1.

C also sets minimum limits on many language features. For example, compilers must support at least 1,023 members in a `struct`, `enum` or `union`, and at least 127 parameters to a function. For more information on other limits, see the C18 Standard Section 5.2.4.1.

C.7.2 The `restrict` Keyword

[This section can be read after Section 7.5.]

The keyword `restrict` declares a restricted pointer that should have exclusive access to a region of memory. Objects accessed through a restricted pointer cannot be accessed by other pointers, except when those pointers' values are derived from the restricted pointer's value, e.g., by assigning a `restrict` qualified pointer to a non-`restrict` qualified pointer.

We can declare a restricted pointer to an `int` as:

```
int *restrict ptr;
```

Restricted pointers allow the compiler to optimize the way the program accesses memory. For example, the standard-library function `memcpy` is defined as follows:

```
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

The `memcpy` function's specification states that it should not be used to copy between overlapping regions of memory. Using `restrict`ed pointers allows the compiler to optimize the copy operation by copying multiple bytes simultaneously, which is more efficient. Incorrectly declaring a pointer as restricted when another pointer points to the same region of memory can result in *undefined behavior*. For more information, see C99 Standard Section 6.7.3.1.

PERF

C.7.3 Reliable Integer Division

[This section can be read after Section 2.5.]

In early C compilers, integer division behaviors varied across implementations. Some rounded a negative quotient toward negative infinity, while others rounded toward zero, resulting in different answers. Consider dividing –28 by 5. The exact answer is –5.6. If we round the quotient toward zero, we get –5. If we round –5.6 toward negative infinity, we get –6. Today's C compilers simply discard the fractional part (the equivalent of rounding the quotient toward zero), making integer division results reliable across systems. For more information, see the C Standard Section 6.5.5.

C.7.4 Flexible Array Members

[This section can be read after Section 10.3.]

The *last* member of a `struct` may be an array of unspecified length, as in:

```
struct s {
   int arraySize;
   int array[];
};
```

This is called a flexible array member and is declared by specifying empty square brackets (`[]`) after the array's name. To allocate a `struct` with a flexible array member, use code such as:

```
int desiredSize = 5;
struct s *ptr;
ptr = malloc(sizeof(struct s) + sizeof(int) * desiredSize);
```

The `sizeof` operator ignores flexible array members, so `sizeof(struct s)` returns the size of all the `struct`'s members *except* the flexible array member. The extra space we allocate with `sizeof(int) * desiredSize` is the flexible array's size.

Flexible-Array-Member Restrictions

There are many restrictions on flexible array members:

- A flexible array member may be declared only as a `struct`'s last member, so each `struct` may contain at most one flexible array member.
- A flexible array cannot be a `struct`'s only member—the `struct` must have one or more fixed members.
- A `struct` containing a flexible array may not be a member of another `struct`.
- A `struct` with a flexible array member must be allocated dynamically. You cannot fix the flexible array member's size at compile time.

For more information, see C99 Standard Section 6.7.2.1.

C.7.5 Type-Generic Math

[This section can be read after Section 5.3.]

C99 `<tgmath.h>` header provides type-generic macros for many math functions in `<math.h>`. For example, after including `<tgmath.h>` the expression `sin(x)` will call:

- `sinf` (the `float` version of `sin`) if `x` is a `float`,
- `sin` (which takes a `double` argument) if `x` is a `double`,
- `sinl` (the `long double` version of `sin`) if `x` is a `long double`, or
- one of `csin`, `csinf` or `csinl` (the `sin` functions for `complex` types) if `x` is a `complex`

C11/C18 adds more generics capabilities, which we mention later in this appendix.

C.7.6 Inline Functions

[This section can be read after Section 5.5.]

You can declare inline functions by placing the keyword **`inline`** before the function declaration, as in:

```
inline void someFunction();
```

PERF This can improve performance. Function calls take time. When we declare a function as `inline`, the program might no longer call that function. Instead, the compiler has the option to replace every call to an `inline` function with a copy of that function's code body. This improves the runtime performance, but it may increase the program's size. Declare functions as `inline` only if they are short and called frequently. If you change an `inline` function's definition, you must recompile any code that calls that function. The `inline` declaration is only advice to the compiler, which can decide to ignore it. Compilers also may optimize performance by inlining functions not declared `inline`. For more information, see C99 Standard Section 6.7.4.

C.7.7 `__func__` Predefined Identifier

[This section can be read after Section 14.9.]

The **`__func__`** predefined identifier is similar to the `__FILE__` and `__LINE__` preprocessor macros. When used in a function's body, `__func__` is a string containing the current function's name. Unlike `__FILE__` and `__LINE__`, `__func__` is a real variable, not a string literal visible at preprocessing time. So, `__func__` cannot be concatenated with other literals during preprocessing.

C.7.8 va_copy Macro

[This section can be read after Section 15.2.]
Section 15.2 introduced the `<stdarg.h>` header and functions with variable-length argument lists. The **`va_copy`** macro takes two `va_list`s and copies its second argument into its first argument. This allows for multiple passes over a variable-length argument list without starting from the beginning each time.

C.8 C11/C18 Features

C11/C18 refined and expanded C's capabilities. Some of C11/C18's features are considered optional. Microsoft Visual C++ provides only partial support for features that were added in C99 and C11/C18.

C.8.1 C11/C18 Headers

The following table lists the standard-library headers that were added in C11.

Header	Explanation
`<stdalign.h>`	Provides type-alignment controls.
`<stdatomic.h>`	Provides uninterruptible access to objects used in multithreading.
`<stdnoreturn.h>`	Nonreturning functions.
`<threads.h>`	Thread library (see Section C.9).
`<uchar.h>`	UTF-16 and UTF-32 character utilities.

C.8.2 quick_exit Function

In addition to `exit` (Section 15.5) and `abort`, C11/C18 provide function **`quick_exit`** (header `<stdlib.h>`) to terminate a program. Like `exit`, you call `quick_exit` and pass it an exit status as an argument—typically `EXIT_SUCCESS` or `EXIT_FAILURE`, but other platform-specific values are possible. The program returns the exit status value to the calling environment to indicate whether the program terminated successfully or an error occurred.

When called, `quick_exit` can, in turn, call up to at least 32 other functions to perform cleanup tasks. You register these functions, which must return `void` and have a `void` parameter list, with the **`at_quick_exit`** function (similar to `atexit` in Section 15.5). They're called in the reverse order from which they were registered. The motivation for functions `quick_exit` and `at_quick_exit` is explained at

```
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1327.htm
```

C.8.3 Unicode® Support

Internationalization and localization is the process of creating software that supports multiple spoken languages and locale-specific requirements—such as displaying monetary formats. The **Unicode®** character set contains characters for many of the world's languages and symbols.

C11/C18 includes support for both the *16-bit (UTF-16)* and *32-bit (UTF-32)* Unicode character sets, making it easier for you to internationalize and localize your apps. Section 6.4.5 in the C18 standard discusses Unicode string literals. Section 7.28 in the standard discusses the Unicode utilities header (`<uchar.h>`), which includes the new types `char16_t` and `char32_t` for UTF-16 and UTF-32 characters, respectively.

C.8.4 _Noreturn Function Specifier

The **`_Noreturn` function specifier** indicates that a function will not return to its caller. For example, function `exit` (Section 15.5) terminates a program, so it does not return to its caller. Such C standard-library functions are now declared with `_Noreturn`. For example, the C11/C18 standards show function `exit`'s prototype as:

```
_Noreturn void exit(int status);
```

If the compiler knows a function does not return, it can perform various optimizations and issue error messages if a `_Noreturn` function is inadvertently written to return.

C.8.5 Type-Generic Expressions

C11/C18's **`_Generic` keyword** provides a mechanism that you can use to create a macro (Chapter 14) that can invoke different type-specific versions of functions based on the macro's argument type. In C11/C18, `_Generic` is used to implement the features of the type-generic math header (`<tgmath.h>`). Many math functions provide separate versions that take `float`, `double` or `long double` arguments. In such cases, there is a macro that automatically invokes the corresponding type-specific version. For example, the macro `ceil` invokes the function `ceilf` when the argument is a `float`, `ceil` when the argument is a `double` and `ceill` when the argument is a `long double`. Section 6.5.1.1 of the C18 standard discusses the details of using `_Generic`.

C.8.6 Annex L: Analyzability and Undefined Behavior

The C11/C18 standards documents define the features of the language that compiler vendors must implement. Because of the extraordinary range of hardware and software platforms and other issues, the standard specifies in several places that the result of an operation is undefined behavior. These can raise security and reliability concerns. Whenever there's an undefined behavior, something happens that could leave a system open to attack or failure. The term "undefined behavior" appears approximately 50 times in the C18 standard document.

The people responsible for C11/C18's optional Annex L are from the CERT Division of Carnegie Mellon's Software Engineering Institute:

```
https://www.sei.cmu.edu/about/divisions/cert/index.cfm
```

They scrutinized all undefined behaviors mentioned in the C standard and discovered that these fall into two categories:

- those for which compiler implementers should be able to do something reasonable to avoid serious consequences—known as *bounded undefined behaviors*, and

- those for which implementers would not be able to do anything reasonable—known as *critical undefined behaviors*.

It turned out that most undefined behaviors belong to the first category. David Keaton (a researcher from the CERT Secure Coding Program) explains the categories in the following article:

```
https://insights.sei.cmu.edu/sei_blog/2012/06/improving-security-
  in-the-latest-c-programming-language-standard.html
```

The C11/C18 standard's Annex L identifies the critical undefined behaviors. Including this annex as part of the standard provides an opportunity for compiler implementors. A compiler that's Annex L compliant can be depended upon to do something reasonable for most of the undefined behaviors that might have been ignored in earlier implementations. Annex L still does not guarantee reasonable behavior for critical undefined behaviors. A program can determine whether the implementation is Annex L compliant by using conditional compilation directives (Section 14.5) that test whether the macro `__STDC_ANALYZABLE__` is defined.

C.8.7 Memory Alignment Control

In Chapter 10, we discussed that computer platforms have different boundary alignment requirements, which could lead to `struct` objects requiring more memory than the total of their members' sizes. C11/C18 allows you to specify the boundary alignment requirements of any type using features of the `<stdalign.h>` header. `_Alignas` is used to specify alignment requirements. Operator `alignof` returns its argument's alignment requirement. Function `aligned_alloc` allows you to dynamically allocate memory for an object and specify its alignment requirements. For more details, see Section 6.2.8 of the C18 standard document.

C.8.8 Static Assertions

In Section 14.10, you learned that C's `assert` macro tests an expression's value at execution time. If the condition's value is false, `assert` prints an error message and calls function `abort` to terminate the program. This is useful for debugging purposes. C11/C18 provides **`_Static_assert`** for compile-time assertions that test constant expressions after the preprocessor executes and at a point during compilation when the expressions' types are known. For more details, see Section 6.7.10 of the C18 standard document.

C.8.9 Floating-Point Types

C11/C18 compilers may optionally provide support for the IEC 60559 floating-point arithmetic standard. Among its features, IEC 60559 defines how floating-point arithmetic should be performed to ensure that you always get the same results across implementations, whether the calculations are performed by hardware, software or both. You can learn more about this standard at:

```
http://www.iso.org/iso/iso_catalogue/catalogue_tc/
  catalogue_detail.htm?csnumber=57469
```

C.9 Case Study: Performance with Multithreading and Multicore Systems

It would be nice if we could focus our attention on performing only one task at a time and doing it well. That's usually difficult to do in a complex world where there's so much going on at once. This section presents C's capabilities for creating and managing multiple tasks. As we'll demonstrate, this can significantly improve program performance and responsiveness.

Concurrency vs. Parallelism

When we say that two tasks are operating **concurrently**, we mean that they're both *making progress* at once. Until the early 2000s, most computers had only a single processor. Operating systems on such computers execute tasks concurrently by rapidly switching between them, doing a small portion of each before moving on to the next so that all tasks keep progressing. For example, it's common for your computer to perform many tasks concurrently, such as compiling a program, sending a file to a printer, receiving e-mail messages, posting a tweet, uploading a video to YouTube, uploading a photo to Facebook or Instagram and more.

PERF When we say that two tasks are operating **in parallel**, we mean that they're *executing truly simultaneously*. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. For example, respiration, blood circulation, digestion, thinking and walking can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste.

No one knows exactly how powerful the human brain is, but various articles state that it has the equivalent of 100 billion "processors"[3,4,5] and one article we found says the brain has the equivalent of "five million contemporary 200 million transistor chip cores."[6] Today's multicore computers have multiple processors that can perform tasks in parallel.

C Concurrency

C programs can have multiple **threads of execution**, each with its own function-call stack and program counter (which keeps track of the next instruction to execute), allowing that thread to execute concurrently with other threads. This capability is called **multithreading**.

3. "How Many Supercomputers Would Fit Inside Your Brain?" Accessed December 4, 2020. `https://fountainmagazine.com/2016/issue-111-may-june-2016/how-many-supercomputers-would-fit-inside-your-brain`.
4. "When compared to a computer CPU, is human brain single-core or multi-core?" Accessed December 4, 2020. `https://www.quora.com/When-compared-to-a-computer-CPU-is-human-brain-single-core-or-multi-core/answer/Frank-Heile`.
5. "Which is the equivalent processing of human brain in terms of computer processing?" Accessed December 4, 2020. `https://cs.stackexchange.com/questions/20016/which-is-the-equivalent-processing-of-human-brain-in-terms-of-computer-processin/40075`.
6. "Neural waves of brain." December 4, 2020. `https://biophilic.blogspot.com/2011/05/neural-waves-of-brain.html`.

PERF

A problem with single-threaded applications is that lengthy activities must complete before others can begin—that can lead to poor responsiveness. In a multithreaded application, threads can be distributed across multiple available cores so that multiple tasks execute in parallel, enabling the application to operate more efficiently. Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for an event to occur, such as a timer expiration or the completion of an I/O operation), another can use the processor.

A single-core system with multithreading can have several threads executing concurrently, but not in parallel. A multicore system with multithreading can have some threads executing concurrently and some executing truly in parallel.

Multicore Systems

Though multithreading has been around since the late 1960s,[7] interest in it is rising quickly due to the proliferation of multicore systems. Smartphones and tablets commonly contain multicore processors.

The first multicore CPU was introduced by IBM in 2001.[8] Most new processors today have at least two cores, with three, four and eight cores now common. Apple's recently introduced M1 processor has eight CPU cores and up to eight additional graphics processing unit (GPU) cores.[9] AMD has desktop processors with up to 32 cores.[10] Intel has processors with up to 18 cores[11] for consumers and high-end processors with up to 72 cores for supercomputers, high-end servers and ultra-high-performance desktop systems.[12] To take full advantage of multicore architecture, you need to write multithreaded applications.

Concurrent Programming Is Difficult

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought. To see why multithreaded programs can be challenging to write and understand, try the following experiment: Open three books to page 1 and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you'll appreciate many of multithreading's challenges. You must

7. "Thread (computing)" Accessed December 4, 2020. `https://en.wikipedia.org/wiki/Thread_(computing)`.
8. "Power 4: The First Multi-Core, 1GHz Processor" Accessed December 4, 2020. `https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/`.
9. "Apple unleashes M1." Accessed November 18, 2020. `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1`.
10. "AMD unveils world's most powerful desktop CPUs." Accessed November 18, 2020. `https://www.zdnet.com/article/amd-unveils-worlds-most-powerful-desktop-cpus/`.
11. "Intel Core Processor Family." Accessed November 18, 2020. `https://www.intel.com/content/www/us/en/products/processors/core.html`.
12. "Xeon Phi" Accessed November 18, 2020. `https://en.wikipedia.org/wiki/Xeon_Phi`.

- *switch* between the books,
- *read* briefly,
- *remember your place* in each book,
- *move the book you're reading closer* so that you can see it and
- *push the books you're not reading aside*.

And, amid all this chaos of rapidly repeating these tasks, you must try to *comprehend* the content of the books!

Standard Multithreading Implementation

Previously, C multithreading libraries were nonstandard, platform-specific language extensions. C programmers often want their code to be portable across platforms—this is a key benefit of standardized multithreading. The **`<threads.h>` header** declares the (optional) multithreading capabilities for writing more portable multithreaded code.

Microsoft Visual C++ and Apple's version of the Clang compiler in Xcode do not support `<threads.h>`. So, we tested this section's examples, using:

- GNU `gcc` 10.2 on Ubuntu Linux,
- GNU `gcc` 10.2 in the GNU Compiler Collection Docker container, which can run on Windows, macOS and Linux,
- GNU `gcc` 10.2 on Ubuntu Linux running in the Windows Subsystem for Linux (WSL), and
- Clang 10.0 on Linux.

In this section, we introduce basic features that enable you to create and execute threads and simple multithreaded applications. At the end of the section, we mention several other multithreading features you'll want to explore if you would like to create more sophisticated multithreaded applications.

Running Multithreaded Programs

When you run a program, its tasks compete for the processors' attention with

- the operating system,
- other programs, and
- other activities the operating system runs on your behalf.

When you execute the examples in this section, the time to perform each calculation will vary based on your computer's

- processor speed,
- number of processor cores, and
- what's running on your computer.

It's like driving to a store—the time it takes can vary, based on traffic conditions, weather and other factors. Some days the drive might take 10 minutes, but it could take longer during rush hour or bad weather.

PERF

There's also *overhead* inherent in multithreading itself. Simply dividing a task into two threads and running it on a dual-core system does *not* run it twice as fast, though it will typically run faster than performing the thread's tasks in sequence. Executing a multithreaded application on a single-core processor can actually take longer than simply performing the thread's tasks in sequence.

Overview of This Section's Examples

To provide a convincing demonstration of multithreading's power on a multicore system, this section presents two programs:

- One performs two compute-intensive calculations *sequentially*.
- The other executes the same compute-intensive calculations in *parallel threads*.

The outputs shown were produced using the GNU Compiler Collection Docker container. Docker allows you to specify the number of cores dedicated to the container when you launch it by using the command-line argument:

`--cpus=`*numberOfCores*

We executed each program using the Docker container with one core then with two to show the programs' performance in each scenario. We show the individual calculation times and total calculation time for each program. The outputs show the time improvement when the multithreaded program executes on two cores instead of just one.

C.9.1 Example: Sequential Execution of Two Compute-Intensive Tasks

Lines 35–42 of Fig. C.5 define the recursive `fibonacci` function, originally discussed in Section 5.15. As we saw in that section, for larger Fibonacci values, the recursive implementation can require significant computation time. This example sequentially performs the calculations `fibonacci(50)` (line 14) and `fibonacci(49)` (line 23).

```
 1  // figC_05.c
 2  // Fibonacci calculations performed sequentially
 3  #include <stdio.h>
 4  #include <time.h>
 5
 6  long long int fibonacci(int n); // function prototype
 7
 8  int main(void) {
 9     puts("Sequential calls to fibonacci(50) and fibonacci(49)");
10
11     // calculate fibonacci value for 50
12     time_t startTime1 = time(NULL);
13     puts("Calculating fibonacci(50)");
14     long long int result1 = fibonacci(50);
15     time_t endTime1 = time(NULL);
16
17     printf("fibonacci(50) = %llu\n", result1);
```

Fig. C.5 | Fibonacci calculations performed sequentially. (Part 1 of 2.)

```
   printf("Calculation time = %f minutes\n\n",
      difftime(endTime1, startTime1) / 60.0);

   time_t startTime2 = time(NULL);
   puts("Calculating fibonacci(49)");
   long long int result2 = fibonacci(49);
   time_t endTime2 = time(NULL);

   printf("fibonacci(49) = %llu\n", result2);
   printf("Calculation time = %f minutes\n\n",
      difftime(endTime2, startTime2) / 60.0);

   printf("Total calculation time = %f minutes\n",
      difftime(endTime2, startTime1) / 60.0);
}

// Recursively calculates fibonacci numbers
long long int fibonacci(int n) {
   if (0 == n || 1 == n) { // base case
      return n;
   }
   else { // recursive step
      return fibonacci(n - 1) + fibonacci(n - 2);
   }
}
```

a) Run on a Docker Container with ***One Core***

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci(50)
fibonacci(50) = 12586269025
Calculation time = 1.700000 minutes

Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 1.050000 minutes

Total calculation time = 2.750000 minutes
```

b) Run on a Docker Container with ***Two Cores***

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci(50)
fibonacci(50) = 12586269025
Calculation time = 1.666667 minutes

Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 1.066667 minutes

Total calculation time = 2.733333 minutes
```

Fig. C.5 | Fibonacci calculations performed sequentially. (Part 2 of 2.)

Before and after each `fibonacci` call, we capture the time so we can determine the calculation's total processing time. We also use these times to calculate the total time required for both calculations. Lines 19, 28 and 31 use function `difftime` (from header `<time.h>`) to determine the number of seconds between two times.

The first output shows the program's results in the GNU Compiler Docker Container using one core. The second shows the results of running the program with the Docker container configured to use two cores. Figure C.5 does not use multithreading, so the program can execute only on one core, even on the two-core Docker container. In our testing, running the programs multiple times with one and two cores produced slightly different results each time. Using a single core generally took longer because the processor was being shared between this program and Docker.

C.9.2 Example: Multithreaded Execution of Two Compute-Intensive Tasks

Figure C.6 also uses the recursive `fibonacci` function but executes each call in a *separate thread*. To compile this program with GNU `gcc`—either in Linux or in the GNU Compiler Collection Docker container—use the command:

```
gcc -std=c18 figC_06.c -pthread
```

The linker uses the `-pthread` option to link our program to the Linux operating system's threading library. If you have Clang on Linux, you can compile the program with:

```
clang -std=c18 figC_06.c -pthread
```

```
// figC_06.c
// Fibonacci calculations performed in separate threads
#include <stdio.h>
#include <threads.h>
#include <time.h>

#define NUMBER_OF_THREADS 2

int startFibonacci(void *nPtr);
long long int fibonacci(int n);

typedef struct ThreadData {
   time_t startTime; // time thread starts processing
   time_t endTime; // time thread finishes processing
   int number; // fibonacci number to calculate
} ThreadData; // end struct ThreadData

int main(void) {
   // data passed to the threads; uses designated initializers
   ThreadData data[NUMBER_OF_THREADS] =
      {[0] = {.number = 50},
       [1] = {.number = 49}};

```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 1 of 4.)

```
   // each thread needs a thread identifier of type thrd_t
   thrd_t threads[NUMBER_OF_THREADS];

   puts("fibonacci(50) and fibonacci(49) in separate threads");

   // create and start the threads
   for (size_t i = 0; i < NUMBER_OF_THREADS; ++i) {
      printf("Starting thread to calculate fibonacci(%d)\n",
         data[i].number);

      // create a thread and check whether creation was successful
      if (thrd_create(&threads[i], startFibonacci, &data[i]) !=
         thrd_success) {
         puts("Failed to create thread");
      }
   }

   // wait for each of the calculations to complete
   for (size_t i = 0; i < NUMBER_OF_THREADS; ++i) {
      thrd_join(threads[i], NULL);
   }

   // determine time that first thread started
   time_t startTime = (data[0].startTime < data[1].startTime) ?
      data[0].startTime : data[1].startTime;

   // determine time that last thread terminated
   time_t endTime = (data[0].endTime > data[1].endTime) ?
      data[0].endTime : data[1].endTime;

   // display total time for calculations
   printf("Total calculation time = %f minutes\n",
      difftime(endTime, startTime) / 60.0);
}

// Called by a thread to begin recursive Fibonacci calculation
int startFibonacci(void *ptr) {
   // cast ptr to ThreadData * so we can access arguments
   ThreadData *dataPtr = (ThreadData *) ptr;

   dataPtr->startTime = time(NULL); // time before calculation

   printf("Calculating fibonacci(%d)\n", dataPtr->number);
   printf("fibonacci(%d) = %lld\n",
      dataPtr->number, fibonacci(dataPtr->number));

   dataPtr->endTime = time(NULL); // time after calculation

   printf("Calculation time = %f minutes\n\n",
      difftime(dataPtr->endTime, dataPtr->startTime) / 60.0);
   return thrd_success;
}

```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 2 of 4.)

```
// Recursively calculates fibonacci numbers
long long int fibonacci(int n) {
   if (0 == n || 1 == n) { // base case
      return n;
   }
   else { // recursive step
      return fibonacci(n - 1) + fibonacci(n - 2);
   }
}
```

a) Run on a Docker Container with ***Two Cores***

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci(50)
Starting thread to calculate fibonacci(49)
Calculating fibonacci(50)
Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 1.083333 minutes

fibonacci(50) = 12586269025
Calculation time = 1.733333 minutes

Total calculation time = 1.733333 minutes
```

b) Run on a Docker Container with ***Two Cores***

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci(50)
Starting thread to calculate fibonacci(49)
Calculating fibonacci(50)
Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 1.033333 minutes

fibonacci(50) = 12586269025
Calculation time = 1.600000 minutes

Total calculation time = 1.600000 minutes
```

c) Run on a Docker Container with ***One Core***

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci(50)
Starting thread to calculate fibonacci(49)
Calculating fibonacci(50)
Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 2.150000 minutes

fibonacci(50) = 12586269025
Calculation time = 2.816667 minutes

Total calculation time = 2.816667 minutes
```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 3 of 4.)

*d) Run on a Docker Container with **One** Core*

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci(50)
Starting thread to calculate fibonacci(49)
Calculating fibonacci(50)
Calculating fibonacci(49)
fibonacci(49) = 7778742049
Calculation time = 2.166667 minutes

fibonacci(50) = 12586269025
Calculation time = 2.833333 minutes

Total calculation time = 2.833333 minutes
```

Fig. C.6 | Fibonacci calculations performed in separate threads. (Part 4 of 4.)

The first two outputs show the multithreaded Fibonacci example executing on a two-core Docker container. Though execution times varied, the total time to perform both Fibonacci calculations (in our tests) was less than Fig. C.5's sequential executions—the total execution time was the same as the longer `fibonacci(50)` calculation. Splitting our program into two threads enabled the two Fibonacci calculations to execute simultaneously—one on each core. The last two outputs show the example executing on a one-core Docker container. Again, times varied for each execution, but the total time was *more* than Fig. C.5's sequential executions due to the overhead of sharing *one* processor among the program's threads and Docker.

struct ThreadData

Lines 12–16 define a `ThreadData struct` type containing the `number` we pass to function `fibonacci` and two `time_t` members where we store the time before and after each thread's `fibonacci` call. The function that each thread executes in this example receives a `ThreadData` object as its argument. Lines 20–22 create a `ThreadData` array and use designated initializers (introduced in Section C.3) to set their `number` members to 50 and 49—the Fibonacci numbers we'll calculate.

thrd_t

Line 25 creates an array of **`thrd_t` objects**. When you create a thread, the multithreading library creates a unique *thread ID* and stores it in a `thrd_t` object. The thread's ID can be used with various multithreading functions.

Creating and Executing a Thread

Lines 30–39 create two threads by calling function **`thrd_create`** (line 35). The function's three arguments are:

- A `thrd_t` pointer that `thrd_create` uses to store the thread's ID.
- A pointer to a function (`startFibonacci`) specifying the task to perform in the thread—This function must return an `int` and receive a `void *` pointer repre-

senting the function's argument. The `int` return value represents the thread's state when it terminates. The `void *` pointer enables this function to receive an argument of any type that's appropriate for your application—in our case, a pointer to a `ThreadData` object. Recall that any pointer type can be assigned to a `void *`.

- A `void *` pointer to the argument that `thrd_create` will pass to the function in the second argument.

Function `thrd_create` returns **`thrd_success`** if the thread is created, **`thrd_nomem`** if there was not enough memory to allocate the thread or **`thrd_error`** otherwise. If the thread is created successfully, the function specified as `thrd_create`'s second argument begins executing in the new thread.

Joining the Threads

To ensure that the program does not terminate until the threads terminate, lines 42–44 call **`thrd_join`** for each thread. This causes the program to *wait* until both threads terminate before executing the remaining code in `main`. Function `thrd_join` receives the `thrd_t` ID of the thread to join and an `int` pointer where `thrd_join` stores the status the thread returns when it terminates—if you don't need this status, pass `NULL` for this argument.

Calculating the Execution Times

After the threads terminate, lines 47–56 calculate and display the total execution time by determining the time difference between the time the first thread started and the second thread ended.

Function `startFibonacci`

Function `startFibonacci` (lines 60–75) specifies the tasks to perform. In this case, we:

- call `fibonacci` to perform a calculation recursively,
- time the calculation,
- display the calculation's result, and
- display the time the calculation took (as we did in Fig. C.5).

The thread executes until `startFibonacci` returns the thread's status (`thrd_success`, line 74), at which point the thread terminates. When this function finishes executing, its corresponding thread terminates.

C.9.3 Other Multithreading Features

There are many other multithreading features, including `_Atomic` variables and atomic operations, thread-local storage, conditions and mutexes. For more information on these topics, see the C18 Standard Sections 6.7.2.4, 6.7.3, 7.17 and 7.26 and the following blog post:

```
https://smartbear.com/blog/test-and-monitor/c11-a-new-c-standard-
  aiming-at-safer-programming/
```

and article:

```
http://lwn.net/Articles/508220/
```

For documentation, see the threads page at:

```
https://en.cppreference.com/w/c/thread
```

D

Intro to Object-Oriented Programming Concepts

D.1 Introduction

After you learn C, you'll likely learn one or more C-based or C-influenced object-oriented languages. These include Java, C++, C#, Objective-C, Python, Swift and many more. These languages often support several programming paradigms:

- procedural programming,
- object-oriented programming,
- generic programming, and
- functional-style programming.

This appendix presents a friendly overview of object-oriented programming terminology and concepts.

D.2 Object-Oriented Programming Languages

C spawned a whole new generation of programming languages that go beyond C's procedural-programming model. As demands for new and more powerful software soar, building software quickly, correctly and economically is important. **Objects**, or more precisely, the **classes** objects come from, are essentially **reusable** software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any noun can be reasonably represented as a software object in terms of **attributes** (e.g., name, color and size) and **behaviors** (e.g., calculating, moving and communicating). Software-development groups can use a modular, object-oriented design-and-implementation approach to be more productive than with earlier popular techniques. Object-oriented programs are often easier to understand, correct and modify.

D.3 Automobile as an Object[1]

To help you understand objects and their contents, consider a simple analogy. Suppose you want to drive a car and make it go faster by pressing its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to design it. A car typically begins as engineering drawings, similar to the blueprints that describe a house's design. These drawings include the design for an accelerator pedal. The pedal hides from the driver the complex mechanisms that make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car, and the steering wheel "hides" the mechanisms that turn the car. This enables people to drive a car even if they have little to no knowledge of how engines, braking and steering mechanisms work.

Just as you cannot cook meals in a kitchen's blueprint, you cannot drive a car's engineering drawings. Before you can drive a car, it must be built from the engineering drawings that describe it. A completed car has an actual accelerator pedal to make it go faster, but even that's not enough. The car won't accelerate on its own (hopefully!), so the driver must press the pedal to accelerate the car.

D.4 Methods and Classes

Performing a task in an object-oriented program requires a **method**. Methods house the program statements that perform their tasks. Each method hides these statements from its user, just as a car's accelerator pedal hides from the driver the mechanisms that make the car go faster. In object-oriented programming, a program unit called a **class** houses the set of methods that perform the class's tasks. For example, a class representing a bank account might contain one method to deposit money into an account, another to withdraw money from an account and a third to inquire what the account's balance is. A class is similar in concept to a car's engineering drawings, which house the designs for the accelerator pedal, steering wheel, and so on.

D.5 Instantiation

Just as someone has to build a car from its engineering drawings before you can drive a car, you must build an object of a class before a program can perform the tasks that the class's methods define. The process of doing this is called instantiation. An object is then referred to as an **instance** of its class.

D.6 Reuse

Just as a car's engineering drawings can be reused many times to build many cars, you can reuse a class many times to build many objects. Reusing existing classes when building new classes and programs saves time and effort. Reuse also helps you build

1. As you read the remainder of this appendix, think of how self-driving cars would affect the discussion.

more reliable and effective systems. Existing classes and components often have undergone extensive testing and debugging (finding and removing errors) and performance tuning. Just as the notion of interchangeable parts was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

In object-oriented languages like C++, Java, C#, Python, Swift and many more, you'll typically use a **building-block approach** to create your programs. To avoid reinventing the wheel, you'll use existing high-quality pieces wherever possible. This software reuse is a key benefit of object-oriented programming.

D.7 Messages and Method Calls

When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, to go faster. Similarly, you send messages to an object. Each message is implemented as a **method call** that tells a method of the object to perform its task. For example, a program might call a bank-account object's deposit method to increase the account's balance by a specified amount.

D.8 Attributes and Instance Variables

A car, besides having capabilities to accomplish tasks, also has attributes, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (that is, its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in other cars' tanks.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank-account object has a balance attribute representing the amount of money in the account. Each bank-account object knows the balance in the account it represents, but not the balances of the bank's other accounts. Attributes are specified by the class's **instance variables**. A class's (and its objects') attributes and methods are intimately related, so classes wrap together their attributes and methods.

D.9 Inheritance

A new class of objects can be created conveniently by **inheritance**—the new class (called the **subclass**) starts with the characteristics of an existing class (called the **superclass**), possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class convertible certainly is an object of the more general class "automobile," but more specifically, the roof can be raised or lowered.

D.10 Object-Oriented Analysis and Design (OOAD)

Many programmers create the code (i.e., the program instructions) for their programs without an initial planning phase. This approach may work for small programs like those we presented in the early chapters of this book. But what if you were asked to create a software system to control thousands of automated teller machines for a bank? Or suppose you were asked to work on a team of 1,000 software developers building the next generation of the U.S. air traffic control system?

To create the best solutions for projects so large and complex, you should follow a detailed **analysis** process for determining your project's **requirements**—that is, define *what* the system is supposed to do. You'd then develop a **design** that satisfies those requirements—that is, specify *how* the system should do it. Ideally, before writing any code, you'd go through this process and carefully review the design and have your design reviewed by other software professionals. If this process involves analyzing and designing your system from an object-oriented perspective, it's called an **object-oriented analysis-and-design (OOAD) process**. Programming in an object-oriented language is called **object-oriented programming (OOP)** and allows you to implement an object-oriented design as a working system.