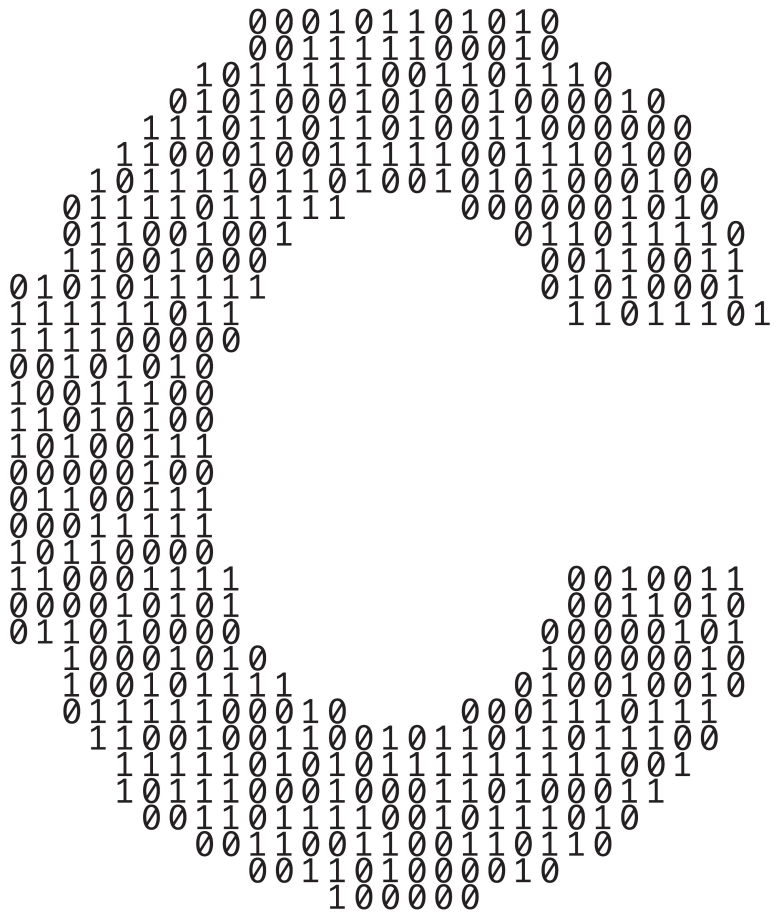
7

Pointers



Objectives

In this chapter, you'll:

- Use pointers and pointer operators.
- Pass arguments to functions by reference using pointers.
- Understand the `const` qualifier's various placements and how they affect what operations you can perform on a variable.
- Use the `sizeof` operator with variables and types.
- Use pointer arithmetic to process array elements.
- Understand the close relationships among pointers, arrays and strings.
- Define and use arrays of strings.
- Use function pointers.
- Learn about secure C programming with pointers.

Outline

7.1 Introduction

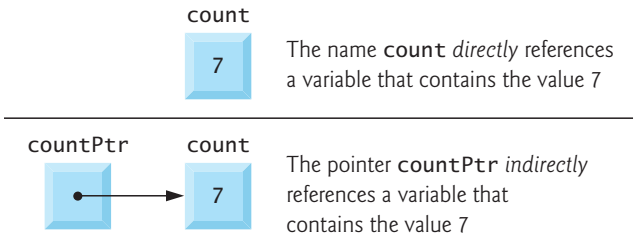
In this chapter, we discuss one of C's most powerful features—the **pointer**. Pointers enable programs to

- accomplish pass-by-reference,
- pass functions between functions,
- manipulate strings and arrays, and
- create and manipulate dynamic data structures that grow and shrink at execution time, such as linked lists, queues, stacks and trees.

This chapter explains basic pointer concepts. In Section 7.13, we discuss various pointer-related security issues. Chapter 10 examines using pointers with structures. Chapter 12 introduces dynamic memory management and shows how to create and use dynamic data structures.

7.2 Pointer Variable Definitions and Initialization

Pointers are variables whose values are memory addresses. Usually, a variable *directly* contains a specific value. A pointer, however, contains the address of another variable that contains a specific value. The pointer *points to* that variable. In this sense, a variable name directly references a value, and a pointer indirectly references a value, as in the following diagram:



Referencing a value through a pointer is called **indirection**.

Declaring Pointers

Pointers, like all variables, must be defined before they can be used. The following statement defines the variable `countPtr` as an `int *`—a pointer to an integer:

```
int *countPtr;
```

This definition is read right-to-left, "`countPtr` is a pointer to `int`" or "`countPtr` points to an object[1] of type `int`." The `*` indicates that the variable is a pointer.

Pointer Variable Naming

Our convention is to end each pointer variable's name with `Ptr` to indicate that the variable is a pointer and should be handled accordingly. Other common naming conventions include starting the variable name with `p` (e.g., `pCount`) or `p_` (e.g., `p_count`).

Define Variables in Separate Statements

The `*` in the following definition does not distribute to each variable:

```
int *countPtr, count;
```

so `countPtr` is a pointer to `int`, but `count` is just an `int`. For this reason, you should always write the preceding declaration as two statements to prevent ambiguity:

```
int *countPtr;
int count;
```

Initializing and Assigning Values to Pointers

Pointers should be initialized when they're defined, or they can be assigned a value. A pointer may be initialized to `NULL`, `0` or an address:

1. In C, an "object" is a region of memory that can hold a value. So objects in C include primitive types such as `int`s, `float`s, `char`s and `double`s, as well as aggregate types such as arrays and `struct`s (which we discuss in Chapter 10).

- A pointer with the value `NULL` points to *nothing*. `NULL` is a *symbolic constant* with the value `0` and is defined in the header `<stddef.h>` (and several other headers, such as `<stdio.h>`).
- Initializing a pointer to `0` is equivalent to initializing it to `NULL`. The constant `NULL` is preferred because it emphasizes that you're initializing a pointer rather than a variable that stores a number. When `0` is assigned, it's first converted to a pointer of the appropriate type. The value `0` is the only integer value that can be assigned directly to a pointer variable.

ERR ⊗

- Assigning a variable's address to a pointer is discussed in Section 7.3. Initialize pointers to prevent unexpected results.

✓ Self Check

1 *(True/False)* The definition:

```
int *countPtr, count;
```

specifies that `countPtr` and `count` are of type `int *`—each is a pointer to an integer.
Answer: *False*. Actually, `count` is an `int`, not a pointer to an `int`. The `*` applies only to `countPtr` and does not distribute to the other variable(s) in the definition.

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

a) A pointer may be initialized to `NULL`, `0` or an address.
b) Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `0` is preferred.
c) The only integer that can be assigned directly to a pointer variable is `0`.
d) Initialize pointers to prevent unexpected results.

Answer: b) is *false*. Actually, `NULL` is preferred because it highlights the fact that the variable is of a pointer type.

7.3 Pointer Operators

Next, let's discuss the address (`&`) and indirection (`*`) operators, and their relationship.

The Address (&) Operator

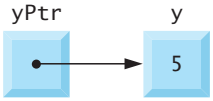
The unary address operator (`&`) returns the *address* of its operand. For example, given the following definition of `y`:

```
int y = 5;
```

the statement

```
int *yPtr = &y;
```

initializes pointer variable `yPtr` with variable `y`'s *address*—`yPtr` is then said to "point to" `y`. The following diagram shows the variables `yPtr` and `y` in memory:



Pointer Representation in Memory

The following diagram shows the preceding pointer's representation in memory, assuming that integer variable `y` is stored at location `600000` and the pointer variable `yPtr` is stored at location `500000`:

The operand of `&` must be a variable; the address operator *cannot* be applied to literal values (like `27` or `41.5`) or expressions.

The Indirection (*) Operator

You apply the unary **indirection operator** (`*`), also called the **dereferencing operator**, to a pointer operand to get the *value* of the object to which the pointer points. For example, the following statement prints `5`, which is the value of variable `y`:

```
printf("%d", *yPtr);
```

Using `*` in this manner is called **dereferencing a pointer**.

Dereferencing a pointer that has not been initialized with or assigned the address of another variable in memory is an error. This could ERR

- cause a fatal execution-time error,
- accidentally modify important data and allow the program to run to completion with incorrect results, or
- lead to a security breach.[2]

Demonstrating the & and * Operators

Figure 7.1 demonstrates the pointer operators `&` and `*`. The `printf` **conversion specification `%p`** outputs a memory location as a hexadecimal integer on most platforms.[3] The output shows that the *address* of `a` and the *value* of `aPtr` are identical, confirming that `a`'s address was indeed assigned to the pointer variable `aPtr` (line 7). The `&` and `*` operators are complements of one another. Applying both consecutively to `aPtr` in either order (line 12) produces the same result. The addresses in the output will vary across systems that use different processor architectures, different compilers and even different compiler settings.

```
1  // fig07_01.c
2  // Using the & and * pointer operators.
3  #include <stdio.h>
4
5  int main(void) {
```

Fig. 7.1 | Using the & and * pointer operators. (Part 1 of 2.)

2. `https://cwe.mitre.org/data/definitions/824.html`.
3. See online Appendix E for more information on hexadecimal integers.

```
6     int a = 7;
7     int *aPtr = &a; // set aPtr to the address of a
8
9     printf("Address of a is %p\nValue of aPtr is %p\n\n", &a, aPtr);
10    printf("Value of a is %d\nValue of *aPtr is %d\n\n", a, *aPtr);
11    printf("Showing that * and & are complements of each other\n");
12    printf("&*aPtr = %p\n*&aPtr = %p\n", &*aPtr, *&aPtr);
13 }
```

```
Address of a is 0x7fffe69386cc
Value of aPtr is 0x7fffe69386cc

Value of a is 7
Value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0x7fffe69386cc
*&aPtr = 0x7fffe69386cc
```

Fig. 7.1 | Using the & and * pointer operators. (Part 2 of 2.)

The following table lists the precedence and grouping of the operators introduced to this point:

Operators	Grouping	Type
`()` `[]` `++` *(postfix)* `--`*(postfix)*	left to right	postfix
`+` `-` `++` `--` `!` `*` `&` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Self Check

1 *(True/False)* Assuming the definitions

```
double d = 98.6;
double *dPtr;
```

the following statement assigns variable `d`'s address to the pointer variable `dPtr`:

```
dPtr = &d;
```

Variable `dPtr` is then said to "point to" `d`.
Answer: *True*.

2 *(Fill-In)* The unary indirection operator (*) returns the value of the object to which its pointer operand points. Using * in this manner is called _____.
Answer: dereferencing a pointer.

7.4 Passing Arguments to Functions by Reference

There are two ways to pass arguments to a function—pass-by-value and pass-by-reference. By default, arguments (other than arrays) are passed by value. As you've seen, arrays are passed by reference. Functions often need to modify variables in the caller or to receive a pointer to a large data object to avoid the overhead of copying the object (as in pass-by-value). As we saw in Chapter 5, a `return` statement can return at most one value from a called function to its caller. Pass-by-reference also can enable a function to "return" multiple values by modifying the caller's variables.

Use & and * to Accomplish Pass-By-Reference

Pointers and the indirection operator enable pass-by-reference. When calling a function with arguments that should be modified in the caller, you use & to pass each variable's address. As we saw in Chapter 6, arrays are *not* passed using operator & because an array's name is equivalent to &*arrayName*`[0]`—the array's starting location in memory. A function that receives the address of a variable in the caller can use the indirection operator (*) to modify the value at that location in the caller's memory, thus effecting pass-by-reference.

Pass-By-Value

The programs in Figs. 7.2 and 7.3 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`. Line 11 of Fig. 7.2 passes the variable `number` by value to function `cubeByValue` (lines 16–18), which cubes its argument and returns the new value. Line 11 assigns the new value to `number` in `main`, replacing `number`'s value.

```
1  // fig07_02.c
2  // Cube a variable using pass-by-value.
3  #include <stdio.h>
4
5  int cubeByValue(int n); // prototype
6
7  int main(void) {
8     int number = 5; // initialize number
9
10    printf("The original value of number is %d", number);
11    number = cubeByValue(number); // pass number by value to cubeByValue
12    printf("\nThe new value of number is %d\n", number);
13 }
14
```

Fig. 7.2 | Cube a variable using pass-by-value. (Part 1 of 2.)

```
15 // calculate and return cube of integer argument
16 int cubeByValue(int n) {
17    return n * n * n; // cube local variable n and return result
18 }
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.2 | Cube a variable using pass-by-value. (Part 2 of 2.)

Pass-By-Reference

Line 12 of Fig. 7.3 passes the variable `number`'s address to function `cubeByReference` (lines 17–19)—passing the address enables pass-by-reference. The function's parameter is a pointer to an `int` called `nPtr` (line 17). The function uses the expression `*nPtr` to dereference the pointer and cube the value to which it points (line 18). It assigns the result to `*nPtr`—which is really the variable `number` in `main`—thus changing `number`'s value in `main`. Use pass-by-value unless the caller explicitly requires the called function to modify the argument variable's value in the caller. This prevents accidental modification of the caller's arguments and is another example of the principle of least privilege.

ERR ⊗

```
1  // fig07_03.c
2  // Cube a variable using pass-by-reference with a pointer argument.
3
4  #include <stdio.h>
5
6  void cubeByReference(int *nPtr); // function prototype
7
8  int main(void) {
9     int number = 5; // initialize number
10
11    printf("The original value of number is %d", number);
12    cubeByReference(&number); // pass address of number to cubeByReference
13    printf("\nThe new value of number is %d\n", number);
14 }
15
16 // calculate cube of *nPtr; actually modifies number in main
17 void cubeByReference(int *nPtr) {
18    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
19 }
```

```
The original value of number is 5
The new value of number is 125
```

Fig. 7.3 | Cube a variable using pass-by-reference with a pointer argument.

Use a Pointer Parameter to Receive an Address

A function receiving an address as an argument must receive it in a pointer parameter. For example, in Fig. 7.3, function `cubeByReference`'s header (line 17) is

```
void cubeByReference(int *nPtr) {
```

which specifies that `cubeByReference` receives the address of an integer variable as an argument, stores the address locally in parameter `nPtr` and does not return a value.

Pointer Parameters in Function Prototypes

The function prototype for `cubeByReference` (Fig. 7.3, line 6) specifies an `int *` parameter. As with other parameters, it's not necessary to include pointer names in function prototypes—they're ignored by the compiler—but it's good practice to include them for documentation purposes.

Functions That Receive One-Dimensional Arrays

For a function that expects a one-dimensional array argument, the function's prototype and header can use the pointer notation shown in the parameter list of function `cubeByReference` (line 17). The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array. So, the function must "know" when it's receiving an array vs. a single variable passed by reference. When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`. The two forms are interchangeable. Similarly, for a parameter of the form `const int b[]` the compiler converts the parameter to `const int *b`.

Pass-By-Value vs. Pass-By-Reference Step-By-Step

Figures 7.4 and 7.5 analyze graphically and step-by-step the programs in Figs. 7.2 and 7.3, respectively.

Step 1: Before `main` calls `cubeByValue`:

```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

number: 5

Step 2: After `cubeByValue` receives the call:

```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

number: 5

```
int cubeByValue( int n ) {

   return n * n * n;

}
```

n: 5

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

number: 5

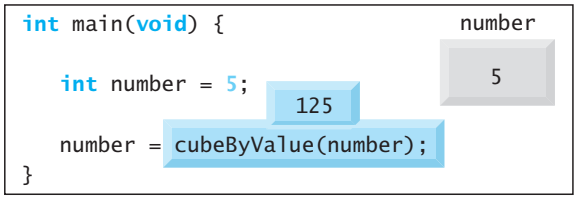
```
int cubeByValue(int n) {

   return n * n * n;

}
```

125

n: 5

Fig. 7.4 | Analysis of a typical pass-by-value. (Part 1 of 2.)

Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:



Step 5: After `main` completes the assignment to `number`:

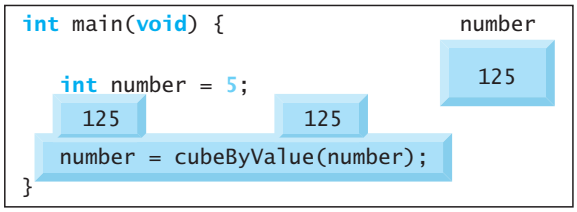
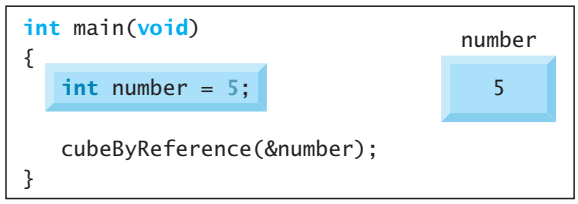
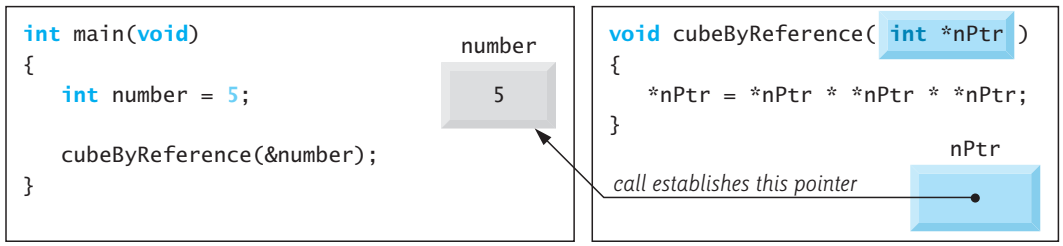


Fig. 7.4 | Analysis of a typical pass-by-value. (Part 2 of 2.)

Step 1: Before `main` calls `cubeByReference`:



Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



Step 3: After `*nPtr` is cubed and before program control returns to `main`:

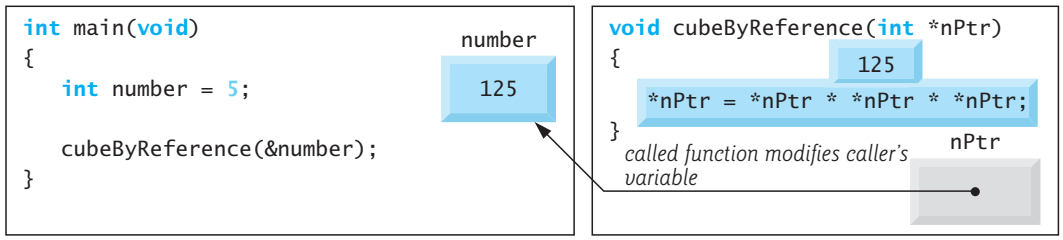


Fig. 7.5 | Analysis of a typical pass-by-reference with a pointer argument.

Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?
   a) By default, arguments (other than arrays) are passed by value. Arrays are passed by reference.
   b) Functions often require the capability to modify variables in the caller or receive a pointer to a large data object to avoid copying the object.
   c) `return` can return one or more values from a called function to a caller.
   d) Pass-by-reference also can enable a function to "return" multiple values by modifying variables in the caller.

Answer: c) is *false*. `return` may be used to return at most one value from a called function to a caller.

2 *(Multiple Choice)* Which of the following statements is *false*?
   a) You use pointers and the indirection operator to accomplish pass-by-reference.
   b) When calling a function with arguments that should be modified, use the address operator (`&`) to pass the argument's addresses.
   c) Arrays are passed by reference using operator `&`.
   d) All of the above statements are *true*.

Answer: c) is *false*. Arrays are not passed by reference using operator `&` because an array's name is equivalent to the address of its first element—`&`*arrayName*`[0]`.

7.5 Using the const Qualifier with Pointers

The **`const` qualifier** enables you to inform the compiler that a particular variable's value should not be modified, thus enforcing the principle of least privilege. This can reduce debugging time and prevent unintentional side effects, making a program more robust, and easier to modify and maintain. If an attempt is made to modify a value that's declared `const`, the compiler catches it and issues an error.

SE

ERR

Over the years, a large base of legacy code was written in early C versions that did not use `const` because it was not available. Even more current code does not use `const` as often as it should. So, there are significant opportunities for improvement by re-engineering existing C code.

There are four ways to pass to a function a pointer to data:

- a **non-constant pointer to non-constant data**.
- a **constant pointer to non-constant data**.
- a **non-constant pointer to constant data**.
- a **constant pointer to constant data**.

Each of the four combinations provides different access privileges and is discussed in the next several examples. How do you choose one of the possibilities? Let the principle of least privilege be your guide. Always award a function enough access to the data in its parameters to accomplish its specified task, but absolutely no more.

7.5.1 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data

The highest level of data access is granted by a **non-constant pointer to non-constant data**. The data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items. A function might use such a pointer to receive a string argument, then process (and possibly modify) each character in the string. Function `convertToUppercase` in Fig. 7.6 declares its parameter, a non-constant pointer to non-constant data called `sPtr` (line 18). The function processes the array `string` (pointed to by `sPtr`) one character at a time. C standard library function `toupper` (line 20) from the `<ctype.h>` header converts each character to its corresponding uppercase letter. If the original character is not a letter or is already uppercase, `toupper` returns the original character. Line 21 increments the pointer to point to the next character in the string. Chapter 8 presents many C standard library character- and string-processing functions.

```
1  // fig07_06.c
2  // Converting a string to uppercase using a
3  // non-constant pointer to non-constant data.
4  #include <ctype.h>
5  #include <stdio.h>
6
7  void convertToUppercase(char *sPtr); // prototype
8
9  int main(void) {
10    char string[] = "cHaRaCters and $32.98"; // initialize char array
11
12    printf("The string before conversion is: %s\n", string);
13    convertToUppercase(string);
14    printf("The string after conversion is: %s\n", string);
15 }
16
17 // convert string to uppercase letters
18 void convertToUppercase(char *sPtr) {
19    while (*sPtr != ) { // current character is not
20       *sPtr = toupper(*sPtr); // convert to uppercase
21       ++sPtr; // make sPtr point to the next character
22    }
23 }
```

```
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

Fig. 7.6 | Converting a string to uppercase using a non-constant pointer to non-constant data.

7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data

A **non-constant pointer to constant data** *can be modified* to point to any data item of the appropriate type, but the data to which it points *cannot be modified*. A function

might receive such a pointer to process an array argument's elements without modifying them. For example, function `printCharacters` (Fig. 7.7) declares parameter `sPtr` to be of type `const char *` (line 20). The declaration is read from *right to left* as "`sPtr` is a pointer to a character constant." The function's `for` statement outputs each character until it encounters a null character. After displaying each character, the loop increments pointer `sPtr` to point to the string's next character.

```
1  // fig07_07.c
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4
5  #include <stdio.h>
6
7  void printCharacters(const char *sPtr);
8
9  int main(void) {
10    // initialize char array
11    char string[] = "print characters of a string";
12
13    puts("The string is:");
14    printCharacters(string);
15    puts("");
16 }
17
18 // sPtr cannot be used to modify the character to which it points,
19 // i.e., sPtr is a "read-only" pointer
20 void printCharacters(const char *sPtr) {
21    // loop through entire string
22    for (; *sPtr != ; ++sPtr) { // no initialization
23       printf("%c", *sPtr);
24    }
25 }
```

```
The string is:
print characters of a string
```

Fig. 7.7 | Printing a string one character at a time using a non-constant pointer to constant data.

Trying to Modify Constant Data

Figure 7.8 shows the errors from compiling a function that receives a non-constant pointer (`xPtr`) to constant data and tries to use it to modify the data. The error shown is from the Visual C++ compiler. The C standard does not specify compiler warning or error messages, and the compiler vendors do not normalize these messages across compilers. So, the actual error message you receive is compiler-specific. For example, Xcode's LLVM compiler reports the error:

```
error: read-only variable is not assignable
```

and the GNU gcc compiler reports the error:

```
error: assignment of read-only location '*xPtr'
```

```c
// fig07_08.c
// Attempting to modify data through a
// non-constant pointer to constant data.
#include <stdio.h>
void f(const int *xPtr); // prototype

int main(void) {
   int y = 7; // define y

   f(&y); // f attempts illegal modification
}

// xPtr cannot be used to modify the
// value of the variable to which it points
void f(const int *xPtr) {
   *xPtr = 100; // error: cannot modify a const object
}
```

Microsoft Visual C++ Error Message

```
fig07_08.c(16,5): error C2166: l-value specifies const object
```

Fig. 7.8 | Attempting to modify data through a non-constant pointer to constant data.

Passing Structures vs. Arrays

As you know, arrays are aggregate types that store related data items of the same type under one name. Chapter 10 discusses another form of aggregate type called a **structure** (sometimes called a **record** or tuple in other languages), which can store related data items of the same or *different* types under one name—e.g., employee information, such as an employee's ID number, name, address and salary.

Unlike arrays, structures are passed by value—a copy of the entire structure is passed. This requires the execution-time overhead of making a copy of each data item in the structure and storing it on the computer's function call stack. Passing large objects such as structures by using pointers to constant data obtains the performance of pass-by-reference and the security of pass-by-value. In this case, the program copies only the *address* at which the structure is stored—typically four or eight bytes.

PERF

If memory is low and execution efficiency is a concern, use pointers. If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege. Some systems do not enforce `const` well, so pass-by-value is still the best way to prevent data from being modified.

7.5.3 Attempting to Modify a Constant Pointer to Non-Constant Data

A **constant pointer to non-constant data** always points to the *same* memory location, but the data at that location *can be modified* through the pointer. This is the default for an array name, which is a constant pointer to the array's first element. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to non-constant data can be used to receive an array as an argu-

ment to a function that accesses array elements using array subscript notation. Pointers that are declared `const` must be initialized when they're defined. If the pointer is a function parameter, it's initialized with a pointer argument as the function is called.

Figure 7.9 attempts to modify a constant pointer. Pointer `ptr` is defined in line 11 to be of type `int * const`, which is read *right-to-left* as "`ptr` is a constant pointer to an integer." The pointer is initialized (line 11) with the address of integer variable `x`. The program attempts to assign `y`'s address to `ptr` (line 14), but the compiler generates an error.

```
1  // fig07_09.c
2  // Attempting to modify a constant pointer to non-constant data.
3  #include <stdio.h>
4
5  int main(void) {
6     int x = 0; // define x
7     int y = 0; // define y
8
9     // ptr is a constant pointer to an integer that can be modified
10    // through ptr, but ptr always points to the same memory location
11    int * const ptr = &x;
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign new address
15 }
```

Microsoft Visual C++ Error Message

```
fig07_09.c(14,4): error C2166: l-value specifies const object
```

Fig. 7.9 | Attempting to modify a constant pointer to non-constant data.

7.5.4 Attempting to Modify a Constant Pointer to Constant Data

The *least* access privilege is granted by a **constant pointer to constant data**. Such a pointer always points to the *same* memory location, and the data at that memory location *cannot be modified*. This is how an array should be passed to a function that only looks at the array's elements using array subscript notation and does *not* modify the elements. Figure 7.10 defines pointer variable `ptr` (line 12) to be of type `const int *const`, which is read *right-to-left* as "`ptr` is a constant pointer to an integer constant." The output shows the error messages generated when we attempt to *modify the data* to which `ptr` points (line 15) and when we attempt to *modify* the *address* stored in the pointer variable (line 16).

```
1  // fig07_10.c
2  // Attempting to modify a constant pointer to constant data.
3  #include <stdio.h>
```

Fig. 7.10 | Attempting to modify a constant pointer to constant data. (Part 1 of 2.)

```
4
5  int main(void) {
6     int x = 5;
7     int y = 0;
8
9     // ptr is a constant pointer to a constant integer. ptr always
10    // points to the same location; the integer at that location
11    // cannot be modified
12    const int *const ptr = &x; // initialization is OK
13
14    printf("%d\n", *ptr);
15    *ptr = 7; // error: *ptr is const; cannot assign new value
16    ptr = &y; // error: ptr is const; cannot assign new address
17 }
```

Microsoft Visual C++ Error Message

```
fig07_10.c(15,5): error C2166: l-value specifies const object
fig07_10.c(16,4): error C2166: l-value specifies const object
```

Fig. 7.10 | Attempting to modify a constant pointer to constant data. (Part 2 of 2.)

Self Check

1 *(Multiple Choice)* What is `sPtr` in the following prototype?

```
void convertToUppercase(char *sPtr);
```

a) A non-constant pointer to constant data.
b) A constant pointer to non-constant data.
c) A non-constant pointer to non-constant data.
d) A constant pointer to constant data.

Answer: c.

2 *(Fill-In)* The least access privilege is granted by a _____ pointer to _____ data. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified.
Answer: constant, constant.

7.6 Bubble Sort Using Pass-By-Reference

Let's improve the bubble-sort[4] program of Fig. 6.12 to use two functions—`bubbleSort` and `swap` (Fig. 7.11). Function `bubbleSort` sorts the array. It calls function `swap` (line 42) to exchange the array elements `array[j]` and `array[j + 1]`.

4. In Chapters 12 and 13, we investigate sorting schemes that yield better performance.

```c
// fig07_11.c
// Putting values into an array, sorting the values into
// ascending order and printing the resulting array.
#include <stdio.h>
#define SIZE 10

void bubbleSort(int * const array, size_t size); // prototype

int main(void) {
   // initialize array a
   int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   puts("Data items in original order");

   // loop through array a
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

   bubbleSort(a, SIZE); // sort the array

   puts("\nData items in ascending order");

   // loop through array a
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

   puts("");
}

// sort an array of integers using bubble sort algorithm
void bubbleSort(int * const array, size_t size) {
   void swap(int *element1Ptr, int *element2Ptr); // prototype

   // loop to control passes
   for (int pass = 0; pass < size - 1; ++pass) {
      // loop to control comparisons during each pass
      for (size_t j = 0; j < size - 1; ++j) {
         // swap adjacent elements if they're out of order
         if (array[j] > array[j + 1]) {
            swap(&array[j], &array[j + 1]);
         }
      }
   }
}
```

Fig. 7.11 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 1 of 2.)

```
48  // swap values at memory locations to which element1Ptr and
49  // element2Ptr point
50  void swap(int *element1Ptr, int *element2Ptr) {
51     int hold = *element1Ptr;
52     *element1Ptr = *element2Ptr;
53     *element2Ptr = hold;
54  }
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 7.11 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 2 of 2.)

Function swap

Remember that C enforces *information hiding* between functions, so `swap` does not have access to individual array elements in `bubbleSort` by default. Because `bubbleSort` wants `swap` to have access to the array elements to swap, `bubbleSort` passes each element's address to `swap`, so the elements are passed by reference. Although entire arrays are automatically passed by reference, individual array elements are *scalars* and are ordinarily passed by value. So, `bubbleSort` uses the address operator (`&`) on each array element:

```
swap(&array[j], &array[j + 1]);
```

Function `swap` receives `&array[j]` in `element1Ptr` (line 50). Function `swap` may use `*element1Ptr` as a synonym for `array[j]`. Similarly, `*element2Ptr` is a synonym for `array[j + 1]`. Even though `swap` is not allowed to say

```
int hold = array[j];
array[j] = array[j + 1];
array[j + 1] = hold;
```

precisely the same effect is achieved by lines 51 through 53:

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

Function bubbleSort's Array Parameter

Note that function `bubbleSort`'s header (line 33) declares `array` as `int * const array` rather than `int array[]` to indicate that `bubbleSort` receives a one-dimensional array argument. Again, these notations are interchangeable; however, array notation generally is preferred for readability.

Function swap's Prototype in Function bubbleSort's Body

The prototype for function `swap` (line 34) is included in `bubbleSort`'s body because only `bubbleSort` calls `swap`. Placing the prototype in `bubbleSort` restricts proper `swap`

calls to those made from `bubbleSort` (or any function that appears after `swap` in the source code). Other functions defined before `swap` that attempt to call `swap` do not have access to a proper function prototype, so the compiler generates one automatically. This normally results in a prototype that does not match the function header (and generates a compilation warning or error) because the compiler assumes `int` for the return and parameter types. Placing function prototypes in the definitions of other functions enforces the principle of least privilege by restricting proper function calls to the functions in which the prototypes appear. SE

Function bubbleSort's `size` Parameter

Function `bubbleSort` receives the array size as a parameter (line 33). When an array is passed to a function, the memory address of the array's first element, of course, does not convey the number of array elements. Therefore, you must pass the array size to the function to know how many elements to sort. Another common practice is to pass a pointer to the array's first element and a pointer to the location just beyond the array's end. As you'll learn in Section 7.8, the difference between these two pointers is the array's length, and the resulting code is simpler.

There are two main benefits to passing the array size to `bubbleSort`—*software reusability* and *proper software engineering*. By defining the function to receive the array size as an argument, we enable the function to be used by any program that sorts one-dimensional integer arrays of any size. SE

We could have stored the array's size in a global variable accessible to the entire program. However, other programs that require an integer array-sorting capability may not have the same global variable, so the function cannot be used in those programs. Global variables usually violate the principle of least privilege and can lead to poor software engineering. Global variables should be used only to represent truly shared resources, such as the time of day. SE

The array size could have been programmed directly into the function. This would restrict the function's use to processing an array of a specific size and significantly reduce its reusability. Only programs processing one-dimensional integer arrays of the specific size coded into the function can use the function.

Self Check

1 *(Code)* Our `bubbleSort` function used the address operator (`&`) on each of the array elements in the `swap` call to effect pass-by-reference as follows:

```
swap(&array[j], &array[j + 1]);
```

Suppose function `swap` receives `&array[j]` and `&array[j + 1]` in `int *` pointers named `firstPtr` and `secondPtr`, respectively. Write the pointer-based code in function `swap` to switch the values in these two elements, using a temporary `int` variable `temp`.

Answer:

```
int temp = *firstPtr;
*firstPtr = *secondPtr;
*secondPtr = temp;
```

2 *(Discussion)* Typically, when we pass an array to a function, we also pass the array size as another argument. Alternatively, we could build the array size directly into the function definition. What's wrong with that approach?
Answer: It would limit the function to processing arrays of a specific size, significantly reducing the function's reusability.

7.7 `sizeof` Operator

C provides the unary operator `sizeof` to determine an object's or type's size in bytes. This operator is applied at compilation time unless its operand is a variable-length array (VLA; Section 6.12). When applied to an array's name as in Fig. 7.12 (line 12), `sizeof` returns as a `size_t` value the array's total number of bytes. Variables of type `float` on our computer are stored in four bytes of memory, and `array` is defined to have 20 elements. Therefore, there are 80 bytes in `array`. `sizeof` is a compile-time operator, so it does not incur any execution-time overhead (except for VLAs).

PERF

```
1  // fig07_12.c
2  // Applying sizeof to an array name returns
3  // the number of bytes in the array.
4  #include <stdio.h>
5  #define SIZE 20
6
7  size_t getSize(const float *ptr); // prototype
8
9  int main(void){
10    float array[SIZE]; // create array
11
12    printf("Number of bytes in the array is %zu\n", sizeof(array));
13    printf("Number of bytes returned by getSize is %zu\n", getSize(array));
14 }
15
16 // return size of ptr
17 size_t getSize(const float *ptr) {
18    return sizeof(ptr);
19 }
```

```
Number of bytes in the array is 80
Number of bytes returned by getSize is 8
```

Fig. 7.12 | Applying `sizeof` to an array name returns the number of bytes in the array.

Even though function `getSize` receives an array of 20 elements as an argument, the function's parameter `ptr` is simply a pointer to the array's first element. When you use `sizeof` with a pointer, it returns the *pointer's size*, not the size of the item to which it points. On our 64-bit Windows, Mac and Linux test systems, a pointer's size is eight bytes, so `getSize` returns 8. On older 32-bit systems, a pointer's size is typically four bytes, so `getSize` would return 4.

The number of elements in an array also can be determined with `sizeof`. For example, consider the following array definition:

```
double real[22];
```

Variables of type `double` normally are stored in eight bytes of memory. Thus, the array `real` contains 176 bytes. The following expression determines the array's number of elements:

```
sizeof(real) / sizeof(real[0])
```

The expression divides the array `real`'s number of bytes by the number of bytes used to store one element of the array (a `double` value). This calculation works *only* when using the actual array's name, *not* when using a pointer to the array.

Determining the Sizes of the Standard Types, an Array and a Pointer

Figure 7.13 calculates the number of bytes used to store each of the standard types. The results of this program are implementation dependent. They often differ across platforms and sometimes across different compilers on the same platform. The output shows the results from our Mac system using the Xcode C++ compiler.

```
1  // fig07_13.c
2  // Using operator sizeof to determine standard type sizes.
3  #include <stdio.h>
4
5  int main(void) {
6     char c = ;
7     short s = 0;
8     int i = 0;
9     long l = 0;
10    long long ll = 0;
11    float f = 0.0F;
12    double d = 0.0;
13    long double ld = 0.0;
14    int array[20] = {0}; // create array of 20 int elements
15    int *ptr = array; // create pointer to array
16
17    printf("    sizeof c = %2zu\t       sizeof(char) = %2zu\n",
18       sizeof c, sizeof(char));
19    printf("    sizeof s = %2zu\t      sizeof(short) = %2zu\n",
20       sizeof s, sizeof(short));
21    printf("    sizeof i = %2zu\t        sizeof(int) = %2zu\n",
22       sizeof i, sizeof(int));
23    printf("    sizeof l = %2zu\t       sizeof(long) = %2zu\n",
24       sizeof l, sizeof(long));
25    printf("   sizeof ll = %2zu\t  sizeof(long long) = %2zu\n",
26       sizeof ll, sizeof(long long));
27    printf("    sizeof f = %2zu\t      sizeof(float) = %2zu\n",
28       sizeof f, sizeof(float));
29    printf("    sizeof d = %2zu\t     sizeof(double) = %2zu\n",
30       sizeof d, sizeof(double));
```

Fig. 7.13 | Using operator `sizeof` to determine standard type sizes. (Part 1 of 2.)

```
31     printf("   sizeof ld = %2zu\tsizeof(long double) = %2zu\n",
32        sizeof ld, sizeof(long double));
33     printf("sizeof array = %2zu\n  sizeof ptr = %2zu\n",
34        sizeof array, sizeof ptr);
35  }
```

```
    sizeof c =  1             sizeof(char) =  1
    sizeof s =  2            sizeof(short) =  2
    sizeof i =  4              sizeof(int) =  4
    sizeof l =  8             sizeof(long) =  8
   sizeof ll =  8        sizeof(long long) =  8
    sizeof f =  4            sizeof(float) =  4
    sizeof d =  8           sizeof(double) =  8
   sizeof ld = 16      sizeof(long double) = 16
sizeof array = 80
  sizeof ptr =  8
```

Fig. 7.13 | Using operator `sizeof` to determine standard type sizes. (Part 2 of 2.)

SE The number of bytes used to store a particular type may vary between systems. When writing programs that depend on type sizes and that will run on several computer systems, use `sizeof` to determine the number of bytes used to store the types.

You can apply `sizeof` to any variable name, type or value (including the value of an expression). When applied to a variable name (that's *not* an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned. The parentheses are required when a type is supplied as `sizeof`'s operand.

Self Check

1 *(Fill-In)* Given the array definition:

```
double temperatures[31];
```

the expression:

```
sizeof(temperatures) / sizeof(temperatures[0])
```

determines what attribute of `temperatures`? _____

Answer: The number of elements in the array (in this case, 31).

2 *(True/False)* When you use `sizeof` with a pointer, it returns the size of the item to which the pointer points.

Answer: *False*. Actually, when you use `sizeof` with a pointer, it returns the pointer's size, not the size of the item to which the pointer points. If you use `sizeof` with an array name, it returns the array's size.

7.8 Pointer Expressions and Pointer Arithmetic

Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions. However, not all arithmetic operators are valid with pointer variables. This section describes the operators that can have pointers as operands, and how these operators are used.

7.8.1 Pointer Arithmetic Operators

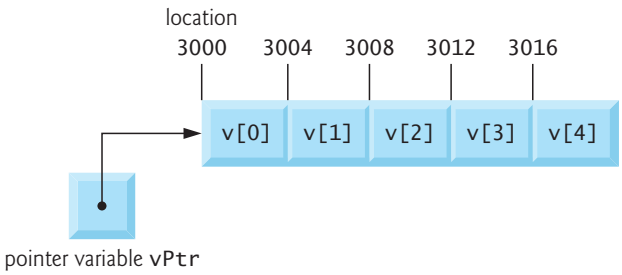
The following arithmetic operations are allowed for pointers:

- incrementing (++) or decrementing (--),
- adding an integer to a pointer (+ or +=),
- subtracting an integer from a pointer (- or -=), and
- subtracting one pointer from another—meaningful only when *both* pointers point into the *same* array.

Pointer arithmetic on pointers that do not refer to array elements is a logic error. ERR

7.8.2 Aiming a Pointer at an Array

Assume the array `int v[5]` is defined, and its first element is at location `3000` in memory. Also, assume the pointer `vPtr` points to `v[0]`—so the value of `vPtr` is `3000`. The following diagram illustrates this scenario for a machine with four-byte integers:



The variable `vPtr` can be initialized to point to array `v` with either of the statements

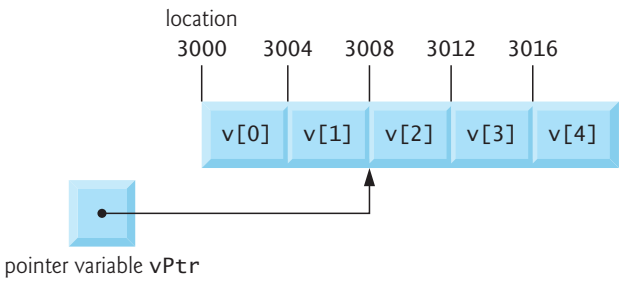
```
vPtr = v;
vPtr = &v[0];
```

7.8.3 Adding an Integer to a Pointer

In conventional arithmetic, `3000 + 2` yields the value `3002`. This is normally not the case with pointer arithmetic. When you add an integer to or subtract one from a pointer, the pointer increments or decrements by that integer *times the size of the object to which the pointer refers*. For example, the statement

```
vPtr += 2;
```

would produce `3008` (`3000 + 2 * 4`), assuming an integer is stored in four bytes of memory. In the array `v`, `vPtr` would now point to `v[2]`, as in the following diagram:



The object's size, depends on its type. When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic because each character is one byte. Type sizes can vary by platform and compiler, so pointer arithmetic is platform- and compiler-dependent.

7.8.4 Subtracting an Integer from a Pointer

If `vPtr` had been incremented to `3016` (`v[4]`), the statement

```
vPtr -= 4;
```

would set `vPtr` back to `3000` (`v[0]`)—the beginning of the array. Using pointer arithmetic to adjust pointers to point outside an array's bounds is a logic error that could lead to security problems.

ERR SEC

7.8.5 Incrementing and Decrementing a Pointer

To increment or decrement a pointer by one, use the increment (`++`) and decrement (`--`) operators. Either of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the next array element. Either of the statements

```
--vPtr;
vPtr--;
```

decrements the pointer to point to the previous array element.

7.8.6 Subtracting One Pointer from Another

If `vPtr` contains the location `3000` and `v2Ptr` contains the address `3008`, the statement

```
x = v2Ptr - vPtr;
```

assigns to `x` the *number of array elements* between `vPtr` and `v2Ptr`, in this case, 2 (not 8). Pointer arithmetic is undefined unless performed on elements of the same array. We cannot assume that two variables of the same type are stored side-by-side in memory unless they're adjacent elements of an array.

ERR

7.8.7 Assigning Pointers to One Another

Pointers of the same type may be assigned to one another. This rule's exception is a **pointer to void** (i.e., **`void *`**)—a generic pointer that can represent *any* pointer type. All pointer types can be assigned to a `void *`, and a `void *` can be assigned a pointer of any type (including another `void *`). In both cases, a cast operation is *not* required.

7.8.8 Pointer to `void`

A pointer to `void` *cannot* be dereferenced. Consider this: The compiler knows on a machine with four-byte integers that an `int *` points to four bytes of memory. However, a `void *` contains a memory location for an *unknown* type—the precise number of bytes to which the pointer refers is *not* known by the compiler. The compiler *must*

know the type to determine the number of bytes that represent the referenced value. *Dereferencing a* `void *` *pointer is a syntax error.* ERR

7.8.9 Comparing Pointers

You can compare pointers using equality and relational operators, but such comparisons are meaningful only if the pointers point to elements of the same array; otherwise, such comparisons are logic errors. Pointer comparisons compare the addresses stored in the pointers. Such a comparison could show, for example, that one pointer points to a higher-numbered array element than the other. A common use of pointer comparison is determining whether a pointer is `NULL`. ERR

Self Check

1 ***(Fill-In)*** When you add an integer to or subtract an integer from a pointer, the pointer increments or decrements by that integer times _____.
Answer: the size of the object to which the pointer points.

2 ***(Fill-In)*** Pointers `v1Ptr` and `v2Ptr` point to elements of the same array of eight-byte double values. If `v1Ptr` contains the address `3000` and `v2Ptr` contains the address `3016`, then the statement

```
size_t x = v2Ptr - v1Ptr;
```

will assign _____ to `x`.
Answer: 2 (not 16)—2 is the number of elements between the pointers.

7.9 Relationship between Pointers and Arrays

Arrays and pointers are intimately related and often may be used interchangeably. You can think of an *array name* as a *constant pointer* to the array's first element. Pointers can be used to do any operation involving array subscripting.

Assume the following definitions:

```
int b[5];
int *bPtr;
```

Because the array name `b` (without a subscript) is a pointer to the array's first element, we can set `bPtr` to the address of the array `b`'s first element with the statement:

```
bPtr = b;
```

This is equivalent to taking the address of array `b`'s first element as follows:

```
bPtr = &b[0];
```

7.9.1 Pointer/Offset Notation

Array element `b[3]` can alternatively be referenced with the pointer expression

```
*(bPtr + 3)
```

The `3` in the expression is the offset to the pointer. When `bPtr` points to the array's first element, the offset indicates which array element to reference—the offset's value

is identical to the array subscript. This notation is referred to as **pointer/offset notation**. The parentheses are required because the precedence of * is *higher* than that of +. Without the parentheses, the above expression would add 3 to the value of the expression *bPtr (i.e., 3 would be added to b[0], assuming bPtr points to the beginning of the array). Just as the array element can be referenced with a pointer expression, the address

```
&b[3]
```

can be written with the pointer expression

```
bPtr + 3
```

An array's name also can be treated as a pointer and used in pointer arithmetic. For example, the expression

```
*(b + 3)
```

refers to element b[3]. In general, all subscripted array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the array's name as a pointer. The preceding statement does not modify the array name in any way; b still points to the first element.

7.9.2 Pointer/Subscript Notation

Pointers can be subscripted like arrays. If bPtr has the value b, the expression

```
bPtr[1]
```

refers to the array element b[1]. This is referred to as **pointer/subscript notation**.

7.9.3 Cannot Modify an Array Name with Pointer Arithmetic

An array name always points to the beginning of the array, so it's like a constant pointer. Thus, the expression

```
b += 3
```

is *invalid* because it attempts to modify the array name's value with pointer arithmetic. Attempting to modify the value of an array name with pointer arithmetic is a compilation error.

ERR

7.9.4 Demonstrating Pointer Subscripting and Offsets

Figure 7.14 uses the four methods we've discussed for referring to array elements—array subscripting, pointer/offset with the array name as a pointer, **pointer subscripting**, and pointer/offset with a pointer—to print the four elements of the integer array b.

```
1 // fig07_14.cpp
2 // Using subscripting and pointer notations with arrays.
3 #include <stdio.h>
4 #define ARRAY_SIZE 4
```

Fig. 7.14 | Using subscripting and pointer notations with arrays. (Part 1 of 3.)

```c

int main(void) {
   int b[] = {10, 20, 30, 40}; // create and initialize array b
   int *bPtr = b; // create bPtr and point it to array b

   // output array b using array subscript notation
   puts("Array b printed with:\nArray subscript notation");

   // loop through array b
   for (size_t i = 0; i < ARRAY_SIZE; ++i) {
      printf("b[%zu] = %d\n", i, b[i]);
   }

   // output array b using array name and pointer/offset notation
   puts("\nPointer/offset notation where the pointer is the array name");

   // loop through array b
   for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
      printf("*(b + %zu) = %d\n", offset, *(b + offset));
   }

   // output array b using bPtr and array subscript notation
   puts("\nPointer subscript notation");

   // loop through array b
   for (size_t i = 0; i < ARRAY_SIZE; ++i) {
      printf("bPtr[%zu] = %d\n", i, bPtr[i]);
   }

   // output array b using bPtr and pointer/offset notation
   puts("\nPointer/offset notation");

   // loop through array b
   for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
      printf("*(bPtr + %zu) = %d\n", offset, *(bPtr + offset));
   }
}
```

```
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40
```

Fig. 7.14 | Using subscripting and pointer notations with arrays. (Part 2 of 3.)

```
Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 7.14 | Using subscripting and pointer notations with arrays. (Part 3 of 3.)

7.9.5 String Copying with Arrays and Pointers

To further illustrate array and pointer interchangeability, let's look at two string-copying functions—`copy1` and `copy2`—in Fig. 7.15. Both functions copy a string into a character array, but they're implemented differently.

```
1  // fig07_15.c
2  // Copying a string using array notation and pointer notation.
3  #include <stdio.h>
4  #define SIZE 10
5
6  void copy1(char * const s1, const char * const s2); // prototype
7  void copy2(char *s1, const char *s2); // prototype
8
9  int main(void) {
10    char string1[SIZE]; // create array string1
11    char *string2 = "Hello"; // create a pointer to a string
12
13    copy1(string1, string2);
14    printf("string1 = %s\n", string1);
15
16    char string3[SIZE]; // create array string3
17    char string4[] = "Good Bye"; // create an array containing a string
18
19    copy2(string3, string4);
20    printf("string3 = %s\n", string3);
21 }
22
23 // copy s2 to s1 using array notation
24 void copy1(char * const s1, const char * const s2) {
25    // loop through strings
26    for (size_t i = 0; (s1[i] = s2[i]) != ; ++i) {
27       ; // do nothing in body
28    }
29 }
```

Fig. 7.15 | Copying a string using array notation and pointer notation. (Part 1 of 2.)

```
30
31  // copy s2 to s1 using pointer notation
32  void copy2(char *s1, const char *s2) {
33     // loop through strings
34     for (; (*s1 = *s2) != ; ++s1, ++s2) {
35        ; // do nothing in body
36     }
37  }
```

```
string1 = Hello
string3 = Good Bye
```

Fig. 7.15 | Copying a string using array notation and pointer notation. (Part 2 of 2.)

Copying with Array Subscript Notation

Function `copy1` uses *array subscript notation* to copy the string in `s2` to the character array `s1`. The function defines counter variable `i` as the array subscript. The `for` statement header (line 26) performs the entire copy operation. The statement's body is the empty statement. The header specifies that `i` is initialized to zero and incremented by one during each iteration. The expression `s1[i] = s2[i]` copies one character from `s2` to `s1`. When the null character is encountered in `s2`, it's assigned to `s1`. Since the assignment's value is what gets assigned to the left operand (`s1`), the loop terminates when an element of `s1` receives the null character, which has the value `0` and therefore is *false*.

Copying with Pointers and Pointer Arithmetic

Function `copy2` uses *pointers and pointer arithmetic* to copy the string in `s2` to the character array `s1`. Again, the `for` statement header (line 34) performs the copy operation. The header does not include any variable initialization. The expression `*s1 = *s2` performs the copy operation by dereferencing `s2` and assigning that character to the current location in `s1`. After the assignment, line 34 increments `s1` and `s2` to point to each string's next character. When the assignment copies the null character into `s1`, the loop terminates.

Notes Regarding Functions copy1 and copy2

The first argument to both `copy1` and `copy2` must be an array large enough to hold the second argument's string. Otherwise, a logic error may occur when an attempt is made to write into a memory location that's not part of the array. In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are *never modified*. Therefore, the second parameter is declared to point to a constant value so that the *principle of least privilege* is enforced. Neither function requires the capability of modifying the string in the second argument, so we simply disallow it.

Self Check

1 *(True/False)* If `bPtr` points to array `b`'s second element (`b[1]`), then element `b[3]` also can be referenced with the pointer/offset notation expression `*(bPtr + 3)`.
Answer: *False*. Since the pointer points to array `b`'s *second element* (`b[1]`), the expression should be `*(bPtr + 2)`.

2 *(Fill-In)* Pointers can be subscripted like arrays. If `bPtr` points to the array `b`'s first element, the expression

```
bPtr[1]
```

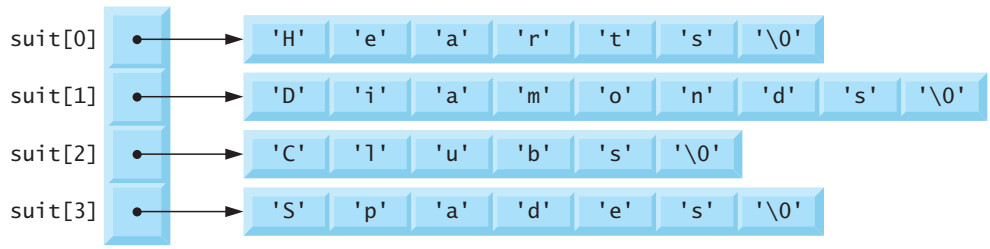
refers to the array element _____.
Answer: `b[1]`.

7.10 Arrays of Pointers

Arrays may contain pointers. A common use of an **array of pointers** is to form an **array of strings**, referred to simply as a **string array**. Each element in a C string is essentially a pointer to its first character. So, each entry in an array of strings is actually a pointer to a string's first character. Consider the definition of the string array `suit`, which might be useful in representing a deck of cards.

```
const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```

The array has four elements. The `char *` indicates that each `suit` element is of type "pointer to `char`." The qualifier `const` indicates that the string each element points to cannot be modified. The strings `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"` are placed into the array. Each is stored in memory as a *null-terminated character string* that's one character longer than the number of characters in the quotes. So, the strings are 7, 9, 6 and 7 characters long. Although it appears these strings are being placed into the array, only pointers are actually stored, as shown in the following diagram:



Each pointer points to the first character of its corresponding string. Thus, even though a `char *` array is *fixed* in size, it can point to character strings of *any length*. This flexibility is one example of C's powerful data-structuring capabilities.

The suits could have been placed in a two-dimensional array, in which each row would represent a suit, and each column would represent a letter from a suit name. Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string. Therefore, considerable memory could be wasted when storing many strings that are shorter than the longest string. We use string arrays to represent a deck of cards in the next section.

Self Check

1 *(Fill-In)* A common use of an array of pointers is to form an array of strings, referred to simply as a _____.
Answer: string array.

2 *(True/False)* The characters of the strings in this section's `suit` array are stored directly in the array's elements.
Answer: *False*. Though the array appears to contain four strings, each element actually contains the address of the corresponding string's first character. The actual letters and terminating null characters are stored elsewhere in memory.

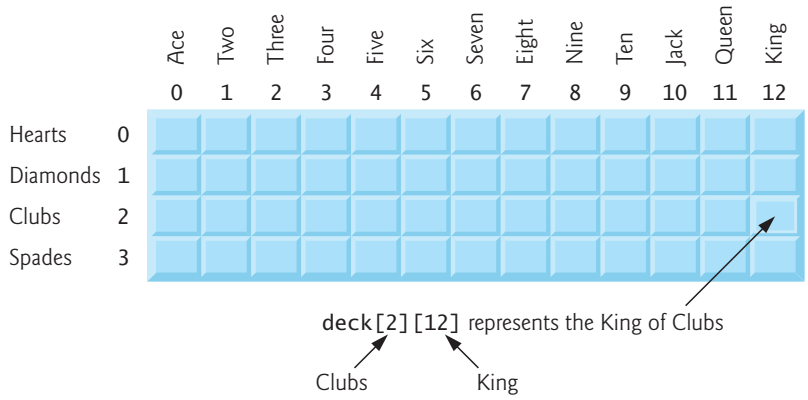
7.11 Random-Number Simulation Case Study: Card Shuffling and Dealing

Let's use random number generation to develop a card shuffling and dealing simulation program, which can then be used to implement programs that play card games. To reveal some subtle performance problems, we've intentionally used suboptimal shuffling and dealing algorithms. In this chapter's exercises and in Chapter 10, we develop more efficient algorithms.

Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards, then deal each card. The top-down approach is particularly useful in attacking more complex problems than you've seen in earlier chapters.

Representing a Deck of Cards as a Two-Dimensional Array

We use a 4-by-13 two-dimensional array `deck` to represent the deck of playing cards:



The rows correspond to the *suits*—row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs and row 3 to spades. The columns correspond to the cards' *face* values. Columns 0 through 9 correspond to ace through ten, and columns 10 through 12 correspond to jack, queen and king. We'll load string array `suit` with character strings representing the four suits, and load string array `face` with character strings representing the 13 face values.

Shuffling the Two-Dimensional Array

This simulated deck of cards may be *shuffled* as follows. First, set all elements of `deck` to `0`. Then, choose a `row` (0–3) and a `column` (0–12) *at random*. Place the number `1` in array element `deck[row][column]` to indicate that this card will be the first one dealt from the shuffled deck. Repeat this process for the numbers 2, 3, …, 52, randomly inserting each in the `deck` array to indicate which cards are to be dealt second, third, …, and fifty-second in the shuffled deck. As the `deck` array begins to fill with card numbers, a card may be selected again—i.e., `deck[row][column]` will be nonzero when it's selected. Ignore this selection and choose other random `row` and `column` values repeatedly until you find an *unselected* card. Eventually, the numbers 1 through 52 will occupy the `deck` array's 52 slots. At that point, the deck of cards is fully shuffled.

Possibility of Indefinite Postponement

This shuffling algorithm can execute *indefinitely* if cards that have already been shuffled are repeatedly selected at random. This phenomenon is known as **indefinite postponement**. In this chapter's exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.

PERF Sometimes an algorithm that emerges in a "natural" way can contain subtle performance problems, such as indefinite postponement. Seek algorithms that avoid indefinite postponement.

Dealing Cards from the Two-Dimensional Array

To deal the first card, we search the array for `deck[row][column]` equal to `1` using nested `for` statements that vary `row` from 0 to 3 and `column` from 0 to 12. What card does that element of the array correspond to? The `suit` array has been preloaded with the four suits, so to get the card's suit, we print the character string `suit[row]`. Similarly, to get the card's face, we print the character string `face[column]`. We also print the character string `" of "`, as in `"King of Clubs"`, `"Ace of Diamonds"` and so on.

Developing the Program's Logic with Top-Down, Stepwise Refinement

Let's proceed with the top-down, stepwise refinement process. The *top* is simply:

```
Shuffle and deal 52 cards
```

Our *first refinement* yields:

```
Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards
```

"Shuffle the deck" may be refined as follows:

```
For each of the 52 cards
    Place card number in a randomly selected unoccupied element of deck
```

"Deal 52 cards" may be refined as follows:

```
For each of the 52 cards
    Find the card number in the deck array and print its face and suit
```

The complete *second refinement* is:

```
Initialize the suit array
Initialize the face array
Initialize the deck array

For each of the 52 cards
    Place card number in a randomly selected unoccupied slot of deck

For each of the 52 cards
    Find the card number in the deck array and print the card's face and suit
```

"Place card number in randomly selected unoccupied slot of deck" may be refined as:

```
Choose slot of deck randomly

While chosen slot of deck has been previously chosen
    Choose slot of deck randomly

Place card number in chosen slot of deck
```

"Find the card number in the deck array and print its face and suit" may be refined as:

```
For each slot of the deck array
    If slot contains card number
        Print the card's face and suit
```

Incorporating these expansions yields our *third refinement*:

```
Initialize the suit array
Initialize the face array
Initialize the deck array

For each of the 52 cards
    Choose slot of deck randomly

    While slot of deck has been previously chosen
        Choose slot of deck randomly

    Place card number in chosen slot of deck

For each of the 52 cards
    For each slot of deck array
        If slot contains desired card number
            Print the card's face and suit
```

This completes the refinement process.

Implementing the Card Shuffling and Dealing Program

The card shuffling and dealing program and a sample execution are shown in Fig. 7.16. When function `printf` uses the conversion specification `%s` to print a

string, the corresponding argument must be a pointer to `char` that points to a string or a `char` array that contains a string. Line 59's format specification displays the card's face *right-aligned* in a field of five characters followed by `" of "` and the card's suit *left-aligned* in a field of eight characters. The *minus sign* in `%-8s` indicates left-alignment.

```
1  // fig07_16.c
2  // Card shuffling and dealing.
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  #define SUITS 4
8  #define FACES 13
9  #define CARDS 52
10
11 // prototypes
12 void shuffle(int deck[][FACES]);
13 void deal(int deck[][FACES], const char *face[], const char *suit[]);
14
15 int main(void) {
16    // initialize deck array
17    int deck[SUITS][FACES] = {0};
18
19    srand(time(NULL)); // seed random-number generator
20    shuffle(deck); // shuffle the deck
21
22    // initialize suit array
23    const char *suit[SUITS] = {"Hearts", "Diamonds", "Clubs", "Spades"};
24
25    // initialize face array
26    const char *face[FACES] = {"Ace", "Deuce", "Three", "Four", "Five",
27       "Six", "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
28
29    deal(deck, face, suit); // deal the deck
30 }
31
32 // shuffle cards in deck
33 void shuffle(int deck[][FACES]) {
34    // for each of the cards, choose slot of deck randomly
35    for (size_t card = 1; card <= CARDS; ++card) {
36       size_t row = 0; // row number
37       size_t column = 0; // column number
38
39       // choose new random location until unoccupied slot found
40       do {
41          row = rand() % SUITS;
42          column = rand() % FACES;
43       } while(deck[row][column] != 0);
44
45       deck[row][column] = card; // place card number in chosen slot
46    }
47 }
```

Fig. 7.16 | Card shuffling and dealing. (Part 1 of 2.)

```
48
49 // deal cards in deck
50 void deal(int deck[][FACES], const char *face[], const char *suit[]) {
51    // deal each of the cards
52    for (size_t card = 1; card <= CARDS; ++card) {
53       // loop through rows of deck
54       for (size_t row = 0; row < SUITS; ++row) {
55          // loop through columns of deck for current row
56          for (size_t column = 0; column < FACES; ++column) {
57             // if slot contains current card, display card
58             if (deck[row][column] == card) {
59                printf("%5s of %-8s  %c", face[column], suit[row],
60                   card % 4 == 0 ?  : ); // 2-column format
61             }
62          }
63       }
64    }
65 }
```

```
  Ace of Hearts     Jack of Hearts     Five of Clubs      King of Clubs
Eight of Diamonds  Three of Clubs     Deuce of Hearts     Four of Hearts
  Ace of Clubs     Deuce of Spades    Queen of Diamonds    Six of Hearts
Seven of Clubs      Five of Hearts    Deuce of Clubs      King of Hearts
 Nine of Spades      Ace of Spades      Ace of Diamonds  Eight of Spades
Eight of Hearts      Ten of Spades      Ten of Hearts    Queen of Clubs
 Jack of Spades     Jack of Diamonds  Three of Spades     Four of Clubs
 Four of Spades      Ten of Clubs      King of Diamonds    Six of Spades
 Nine of Clubs       Six of Diamonds  Queen of Spades     King of Spades
 Four of Diamonds  Eight of Clubs      Jack of Clubs     Seven of Hearts
Seven of Diamonds  Three of Hearts     Five of Spades     Nine of Hearts
 Nine of Diamonds  Three of Diamonds  Deuce of Diamonds  Queen of Hearts
  Six of Clubs     Seven of Spades     Five of Diamonds    Ten of Diamonds
```

Fig. 7.16 | Card shuffling and dealing. (Part 2 of 2.)

Improving the Dealing Algorithm

There's a weakness in the dealing algorithm. Once a match is found, the two inner `for` statements continue searching `deck`'s remaining elements. We correct this deficiency in this chapter's exercises and in a Chapter 10 case study.

Related Exercises

This Card Shuffling and Dealing case study is supported by the following exercises:

- Exercise 7.17 (Card Shuffling and Dealing: Dealing Poker Hands)
- Exercise 7.18 (Project: Card Shuffling and Dealing—Which Poker Hand is Better?)
- Exercise 7.19 (Project: Card Shuffling and Dealing—Simulating the Dealer)
- Exercise 7.20 (Project: Card Shuffling and Dealing—Allowing Players to Draw Cards)

- Exercise 7.21 (Card Shuffling and Dealing Modification: High-Performance Shuffle)

Self Check

1 ***(Fill-In)*** The shuffling algorithm we presented can execute indefinitely if cards that have already been shuffled are repeatedly selected at random. This phenomenon is known as _____.
Answer: indefinite postponement.

2 ***(True/False)*** The format specification `"%5s of%-8s"` prints a string left-aligned in a field of five characters followed by `" of "` and a string right-aligned in a field of eight characters.
Answer: *False*. Actually, this format specification prints a string *right-aligned* in a field of five characters followed by `" of "` and a string *left-aligned* in a field of eight characters.

7.12 Function Pointers

In Chapter 6, we saw that an array name is really the address in memory of the array's first element. Similarly, a function's name is really the starting address in memory of the code that performs the function's task. A **pointer to a function** contains the *address* of the function in memory. Pointers to functions can be passed to functions, returned from functions, stored in arrays, assigned to other function pointers of the same type and compared with one another for equality or inequality.

7.12.1 Sorting in Ascending or Descending Order

To demonstrate pointers to functions, Fig. 7.17 presents a modified version of Fig. 7.11's bubble-sort program. The new version consists of `main` and functions `bubbleSort`, `swap`, `ascending` and `descending`. Function `bubbleSort` receives a pointer to a function as an argument—either function `ascending` or function `descending`—in addition to an `int` array and the array's size. The user chooses whether to sort the array in *ascending* (`1`) or *descending* (`2`) order. If the user enters `1`, `main` passes a pointer to function `ascending` to function `bubbleSort`. If the user enters `2`, `main` passes a pointer to function `descending` to function `bubbleSort`.

```
1  // fig07_17.c
2  // Multipurpose sorting program using function pointers.
3  #include <stdio.h>
4  #define SIZE 10
5
6  // prototypes
7  void bubbleSort(int work[], size_t size, int (*compare)(int a, int b));
8  int ascending(int a, int b);
9  int descending(int a, int b);
10
```

Fig. 7.17 | Multipurpose sorting program using function pointers. (Part 1 of 3.)

```
11 int main(void) {
12    // initialize unordered array a
13    int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
14
15    printf("%s", "Enter 1 to sort in ascending order,\n"
16           "Enter 2 to sort in descending order: ");
17    int order = 0;
18    scanf("%d", &order);
19
20    puts("\nData items in original order");
21
22    // output original array
23    for (size_t counter = 0; counter < SIZE; ++counter) {
24       printf("%5d", a[counter]);
25    }
26
27    // sort array in ascending order; pass function ascending as an
28    // argument to specify ascending sorting order
29    if (order == 1) {
30       bubbleSort(a, SIZE, ascending);
31       puts("\nData items in ascending order");
32    }
33    else { // pass function descending
34       bubbleSort(a, SIZE, descending);
35       puts("\nData items in descending order");
36    }
37
38    // output sorted array
39    for (size_t counter = 0; counter < SIZE; ++counter) {
40       printf("%5d", a[counter]);
41    }
42
43    puts("\n");
44 }
45
46 // multipurpose bubble sort; parameter compare is a pointer to
47 // the comparison function that determines sorting order
48 void bubbleSort(int work[], size_t size, int (*compare)(int a, int b)) {
49    void swap(int *element1Ptr, int *element2ptr); // prototype
50
51    // loop to control passes
52    for (int pass = 1; pass < size; ++pass) {
53       // loop to control number of comparisons per pass
54       for (size_t count = 0; count < size - 1; ++count) {
55          // if adjacent elements are out of order, swap them
56          if ((*compare)(work[count], work[count + 1])) {
57             swap(&work[count], &work[count + 1]);
58          }
59       }
60    }
61 }
62
```

Fig. 7.17 | Multipurpose sorting program using function pointers. (Part 2 of 3.)

```
63 // swap values at memory locations to which element1Ptr and
64 // element2Ptr point
65 void swap(int *element1Ptr, int *element2Ptr) {
66    int hold = *element1Ptr;
67    *element1Ptr = *element2Ptr;
68    *element2Ptr = hold;
69 }
70
71 // determine whether elements are out of order for an ascending order sort
72 int ascending(int a, int b) {
73    return b < a; // should swap if b is less than a
74 }
75
76 // determine whether elements are out of order for a descending order sort
77 int descending(int a, int b) {
78    return b > a; // should swap if b is greater than a
79 }
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

Fig. 7.17 | Multipurpose sorting program using function pointers. (Part 3 of 3.)

Function Pointer Parameter

The following parameter appears in the function header for `bubbleSort` (line 48):

```
int (*compare)(int a, int b)
```

This tells `bubbleSort` to expect a parameter (`compare`) that's a pointer to a function, specifically for a function that receives two `int`s and returns an `int` result. The parentheses around `*compare` are required to group the `*` with `compare` and indicate that `compare` is a *pointer*. Without the parentheses, the declaration would have been

```
int *compare(int a, int b)
```

which declares a function that receives two integers as parameters and returns a pointer to an integer.

To call the function passed to `bubbleSort` via its function pointer, we deference it, as shown in the `if` statement at line 56:

```
if ((*compare)(work[count], work[count + 1]))
```

The call to the function could have been made without dereferencing the pointer as in

```
if (compare(work[count], work[count + 1]))
```

which uses the pointer directly as the function name. The first method of calling a function through a pointer explicitly shows that `compare` is a pointer to a function that's dereferenced to call the function. The second technique makes it appear that `compare` is an *actual* function name. This may confuse someone reading the code who'd like to see `compare`'s function definition and finds that it's never defined.

7.12.2 Using Function Pointers to Create a Menu-Driven System

A common use of function pointers is in *menu-driven systems*. A program prompts a user to select an option from a menu (possibly from 0 to 2) by typing the menu item's number. The program services each option with a different function. It stores pointers to each function in an array of function pointers. The user's choice is used as an array subscript, and the pointer in the array is used to call the function.

Figure 7.18 provides a generic example of the mechanics of defining and using an array of function pointers. We define three functions—`function1`, `function2` and `function3`. Each takes an integer argument and returns nothing. We store pointers to these functions in array `f` (line 13). Beginning at the leftmost set of parentheses, the definition is read, "`f` is an array of `3` pointers to functions that each take an `int` as an argument and return `void`." The array is initialized with the names of the three functions. When the user enters a value between 0 and 2, we use the value as the subscript into the array of pointers to functions. In the function call (line 23), `f[choice]` selects the pointer at location `choice` in the array. We *dereference the pointer to call the function*, passing `choice` as the function's argument. Each function prints its argument's value and its function name to show that the function was called correctly. In this chapter's exercises, you'll develop several menu-driven systems.

```
1  // fig07_18.c
2  // Demonstrating an array of pointers to functions.
3  #include <stdio.h>
4
5  // prototypes
6  void function1(int a);
7  void function2(int b);
8  void function3(int c);
9
10 int main(void) {
11    // initialize array of 3 pointers to functions that each take an
12    // int argument and return void
13    void (*f[3])(int) = {function1, function2, function3};
```

Fig. 7.18 | Demonstrating an array of pointers to functions. (Part 1 of 2.)

```
14
15     printf("%s", "Enter a number between 0 and 2, 3 to end: ");
16     int choice = 0;
17     scanf("%d", &choice);
18
19     // process user
20     while (choice >= 0 && choice < 3) {
21        // invoke function at location choice in array f and pass
22        // choice as an argument
23        (*f[choice])(choice);
24
25        printf("%s", "Enter a number between 0 and 2, 3 to end: ");
26        scanf("%d", &choice);
27     }
28
29     puts("Program execution completed.");
30  }
31
32  void function1(int a) {
33     printf("You entered %d so function1 was called\n\n", a);
34  }
35
36  void function2(int b) {
37     printf("You entered %d so function2 was called\n\n", b);
38  }
39
40  void function3(int c) {
41     printf("You entered %d so function3 was called\n\n", c);
42  }
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.18 | Demonstrating an array of pointers to functions. (Part 2 of 2.)

Self Check

1 *(True/False)* Consider the following parameter which appeared in the function header for our bubble sort function:

```
int (*compare)(int a, int b)
```

This `compare` parameter is a pointer to a function that receives two integer parameters and returns an integer result. The parentheses around `*compare` are optional but we include them for clarity.

Answer: *False*. The parentheses are required to group `*compare` to indicate that `compare` is a pointer. Without the parentheses, the declaration would have been

```
int *compare(int a, int b)
```

which simply is the header of a function that receives two integers as parameters and *returns a pointer to an integer*.

2 ***(Fill-In)*** Just as a pointer to a variable is dereferenced to access the variable's value, a pointer to a function is dereferenced to _____.
Answer: call the function.

7.13 Secure C Programming

`printf_s`, `scanf_s` and Other Secure Functions

Earlier Secure C Programming sections presented `printf_s` and `scanf_s` and mentioned other more secure versions of standard library functions described by Annex K of the C standard. A key feature of functions like `printf_s` and `scanf_s` that makes them more secure is that they have *runtime constraints* requiring their pointer arguments to be non-`NULL`. The functions check these runtime constraints *before* attempting to use the pointers. Any `NULL` pointer argument is a *constraint violation* and causes the function to fail and return a status notification. A call to `scanf_s` returns `EOF` if any of its pointer arguments (including the format-control string) are `NULL`. A call to `printf_s` stops outputting data and returns a negative number if the format-control string is `NULL` or any argument that corresponds to a `%s` is `NULL`. For complete details of the Annex K functions, see the C standard document or your compiler's library documentation.

Other CERT Guidelines Regarding Pointers

Misused pointers are the source of many common security vulnerabilities in systems today. CERT provides various guidelines to help you prevent such problems. If you're building industrial-strength C systems, you should familiarize yourself with the *CERT C Secure Coding Standard* at `https://wiki.sei.cmu.edu/`. The following guidelines apply to pointer programming techniques that we presented in this chapter:

- EXP34-C: Dereferencing `NULL` pointers typically causes programs to crash, but CERT has encountered cases in which dereferencing `NULL` pointers can allow attackers to execute code.
- DCL13-C: Section 7.5 discussed uses of `const` with pointers. If a function parameter points to a value that will not be changed by the function, `const` should be used to indicate that the data is constant. For example, to represent a pointer to a string that will not be modified, use `const char *` as the pointer parameter's type.
- WIN04-C: This guideline discusses techniques for encrypting function pointers on Microsoft Windows to help prevent attackers from overwriting them and executing attack code.

Self Check

1 ***(Fill-In)*** A key feature of functions like `printf_s` and `scanf_s` is that they have runtime constraints requiring their pointer arguments to be _____.
Answer: non-`NULL`.

2 ***(True/False)*** Misused pointers lead to many of the most common security vulnerabilities in systems today.
Answer: *True*.

Summary

Section 7.2 Pointer Variable Definitions and Initialization

- A **pointer** (p. 364) contains an address of another variable that contains a value. In this sense, a variable name *directly* references a value, and a pointer ***indirectly*** **references** a value.
- Referencing a value through a pointer is called **indirection** (p. 365).
- Pointers can be defined to point to objects of any type.
- Pointers should be initialized either when they're defined or in an assignment statement. A pointer may be initialized to **`NULL`**, **`0`** or **an address**. A pointer with the value `NULL` points to nothing. Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred for clarity. The value `0` is the only integer value that can be assigned directly to a pointer variable.
- `NULL` is a symbolic constant defined in the `<stddef.h>` header (and several other headers).

Section 7.3 Pointer Operators

- The **`&`**, or **address operator** (p. 366), is a unary operator that returns its operand's address.
- The operand of the address operator must be a variable.
- The **indirection operator** `*` (p. 367) returns the value of the object to which its operand points.
- The `printf` **conversion specification `%p`** outputs a memory location as a hexadecimal integer on most platforms.

Section 7.4 Passing Arguments to Functions by Reference

- In C, arguments (other than arrays) are **passed by value** (p. 369).
- C programs accomplish **pass-by-reference** (p. 369) by using pointers and the indirection operator. To pass a variable by reference, apply the address operator (`&`) to the variable's name.
- When the address of a variable is passed to a function, the indirection operator (`*`) may be used in the function to read and/or modify the value at that location in the caller's memory.
- A function receiving an address as an argument must define a **pointer parameter** to receive the address.
- The compiler does not differentiate between a function that receives a pointer and one that receives a **one-dimensional array**. A function must "know" when it's receiving an array vs. a single variable passed by reference.
- When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.

Section 7.5 Using the `const` Qualifier with Pointers

- The **`const` qualifier** (p. 373) indicates that a variable's value should not be modified.
- There are four ways to pass a pointer to a function (p. 373): a **non-constant pointer to non-constant data**, a **constant pointer to non-constant data**, a **non-constant pointer to constant data**, and a **constant pointer to constant data**.
- With a non-constant pointer to non-constant data, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items.
- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified.
- A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer. This is the default for an array name.
- A constant pointer to constant data always points to the same memory location, and the data at that memory location cannot be modified.

Section 7.7 `sizeof` Operator

- Unary **operator `sizeof`** (p. 382) determines the size in bytes of a variable or type.
- When applied to an array's name, `sizeof` returns the array's total number of bytes.
- Operator `sizeof` can be applied to any variable name, type or value.
- The parentheses used with `sizeof` are required if a type name is supplied as its operand.

Section 7.8 Pointer Expressions and Pointer Arithmetic

- A limited set of **arithmetic operations** (p. 385) **may be performed on pointers**. You can **increment** (`++`) or **decrement** (`--`) a pointer, **add** an integer to a pointer (`+` or `+=`), **subtract** an integer from a pointer (`-` or `-=`) and **subtract one pointer from another**.
- When you add an integer to or subtract an integer from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.
- Two pointers to elements of the same array may be subtracted from one another to determine the number of elements between them.
- A pointer can be assigned to another pointer if both have the same type. An exception is a `void *` pointer (p. 386), which can represent any pointer type. All pointer types can be assigned a `void *` pointer, and a `void *` pointer can be assigned a pointer of any type.
- A `void *` pointer cannot be dereferenced.
- **Pointers can be compared** using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is **determining whether a pointer is `NULL`**.

Section 7.9 Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An **array name** can be thought of as a **constant pointer**.
- Pointers can be used to do any operation involving array subscripting.
- When a pointer points to the beginning of an array, adding an **offset** (p. 387) to the pointer indicates which element of the array should be referenced. The offset value is identical to the array subscript. This is referred to as pointer/offset notation.

- An **array name can be treated as a pointer** and used in pointer arithmetic expressions that do not attempt to modify the pointer's value.
- **Pointers can be subscripted** (p. 388) like arrays. This is referred to as pointer/subscript notation.
- A parameter of type `const char *` typically represents a constant string.

Section 7.10 Arrays of Pointers

- **Arrays may contain pointers** (p. 392). A common use of an array of pointers is to form an **array of strings** (p. 392). Each element is a string, but a C string is essentially a pointer to its first character. So, each element is actually a pointer to the first character of a string.

Section 7.12 Function Pointers

- A **function pointer** (p. 401) contains the address of a function in memory. A function name is really the starting address in memory of the code that performs the function's task.
- Pointers to functions can be **passed to functions**, **returned from functions**, **stored in arrays** **assigned to other function pointers** and **compared** with one another for equality or inequality.
- A pointer to a function is dereferenced to call the function. A function pointer can be used directly as the function name when calling the function.
- A common use of function pointers is in **menu-driven systems**.

Self-Review Exercises

7.1 Answer each of the following:

a) A pointer variable contains as its value another variable's _____.

b) Three values can be used to initialize a pointer—_____, _____ and _____.

c) The only integer that can be assigned to a pointer is _____.

7.2 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) A pointer that's declared to be `void` can be dereferenced.

b) Pointers of different types may not be assigned to one another without a cast operation.

7.3 Answer each of the following. Assume that single-precision floating-point numbers are stored in four bytes, and that the array's starting address is location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.

a) Define a `float` array called `numbers` with 10 elements, and initialize the elements to the values `0.0`, `1.1`, `2.2`, ..., `9.9`. Assume the symbolic constant `SIZE` has been defined as `10`.

b) Define a pointer, `nPtr`, that points to a `float`.

c) Use a `for` statement and array subscript notation to print array `numbers`' elements. Use one digit of precision to the right of the decimal point.

d) Give two separate statements that assign the starting address of array `numbers` to the pointer variable `nPtr`.

e) Print `numbers`' elements using pointer/offset notation with the pointer `nPtr`.
f) Print `numbers`' elements using pointer/offset notation with the array name as the pointer.
g) Print `numbers`' elements by subscripting pointer `nPtr`.
h) Refer to element 4 of `numbers` using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation with `nPtr` and pointer/offset notation with `nPtr`.
i) Assuming that `nPtr` points to the beginning of array `numbers`, what address is referenced by `nPtr + 8`? What value is stored at that location?
j) Assuming that `nPtr` points to `numbers[5]`, what address is referenced by `nPtr -= 4`? What's the value stored at that location?

7.4 For each of the following, write a statement that performs the specified task. Assume that `float` variables `number1` and `number2` are defined and that `number1` is initialized to `7.3`.

a) Define the variable `fPtr` to be a pointer to an object of type `float`.
b) Assign the address of variable `number1` to pointer variable `fPtr`.
c) Print the value of the object pointed to by `fPtr`.
d) Assign the value of the object pointed to by `fPtr` to variable `number2`.
e) Print the value of `number2`.
f) Print the address of `number1`. Use the `%p` conversion specification.
g) Print the address stored in `fPtr`. Use the `%p` conversion specifier. Is the value printed the same as the address of `number1`?

7.5 Do each of the following:

a) Write the function header for a function `exchange` that takes two pointers to floating-point numbers `x` and `y` as parameters and does not return a value.
b) Write the function prototype for the function in part (a).
c) Write the function header for a function `evaluate` that returns an integer and that takes as parameters integer `x` and a pointer to function `poly`, which represents a function that takes an integer parameter and returns an integer.
d) Write the function prototype for the function in part (c).

7.6 Find the error in each of the following program segments. Assume:

```
int *zPtr; // zPtr will reference array z
int *aPtr = NULL;
void *sPtr = NULL;
int number;
int z[5] = {1, 2, 3, 4, 5};
sPtr = z;
```

a)
```
++zptr;
```
b)
```
// use pointer to get array
number = zPtr;
```
c)
```
// assign array element 2 to number; assume zPtr is initialized
number = *zPtr[2];
```

d)
```
// print entire array z; assume zPtr is initialized
for (size_t i = 0; i <= 5; ++i) {
   printf("%d ", zPtr[i]);
}
```
e)
```
// assign the value pointed to by sPtr to number
number = *sPtr;
```
f)
```
++z;
```

Answers to Self-Review Exercises

7.1 a) address. b) `0`, `NULL`, an address. c) `0`.

7.2 a) *False*. A pointer to `void` cannot be dereferenced, because there's no way to know exactly how many bytes of memory to dereference. b) *False*. Pointers of type `void` can be assigned pointers of other types, and pointers of type `void` can be assigned to pointers of other types.

7.3 See the answers below:

a)
```
float numbers[SIZE] =
   {0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
```
b)
```
float *nPtr;
```
c)
```
for (size_t i = 0; i < SIZE; ++i) {
   printf("%.1f ", numbers[i]);
}
```
d)
```
nPtr = numbers;
nPtr = &numbers[0];
```
e)
```
for (size_t i = 0; i < SIZE; ++i) {
   printf("%.1f ", *(nPtr + i));
}
```
f)
```
for (size_t i = 0; i < SIZE; ++i) {
   printf("%.1f ", *(numbers + i));
}
```
g)
```
for (size_t i = 0; i < SIZE; ++i) {
   printf("%.1f ", nPtr[i]);
}
```
h)
```
numbers[4]
*(numbers + 4)
nPtr[4]
*(nPtr + 4)
```
i) The address is `1002500 + 8 * 4 = 1002532`. The value is `8.8`.

j) The address of `numbers[5]` is `1002500 + 5 * 4 = 1002520`.
The address of `nPtr -= 4` is `1002520 - 4 * 4 = 1002504`.
The value at that location is `1.1`.

7.4 See the answers below:

a)
```
float *fPtr;
```

b) `fPtr = &number1;`
c) `printf("The value of *fPtr is %f\n", *fPtr);`
d) `number2 = *fPtr;`
e) `printf("The value of number2 is %f\n", number2);`
f) `printf("The address of number1 is %p\n", &number1);`
g) `printf("The address stored in fptr is %p\n", fPtr);`
Yes, the value is the same.

7.5 a) `void exchange(float *x, float *y)`
b) `void exchange(float *x, float *y);`
c) `int evaluate(int x, int (*poly)(int))`
d) `int evaluate(int x, int (*poly)(int));`

7.6 a) Error: `zPtr` has not been initialized.
Correction: Initialize `zPtr` with `zPtr = z;` before doing pointer arithmetic.
e) Error: The pointer is not dereferenced.
Correction: Change the statement to `number = *zPtr;`
f) Error: `zPtr[2]` is not a pointer and should not be dereferenced.
Correction: Change `*zPtr[2]` to `zPtr[2]`.
g) Error: Referring to an array element outside the array bounds with pointer subscripting.
Correction: Change the operator <= in the `for` condition to <.
h) Error: Dereferencing a void pointer.
Correction: To dereference the pointer, it must first be cast to an integer pointer. Change the statement to `number = *((int *) sPtr);`
i) Error: Trying to modify an array name with pointer arithmetic.
Correction: Use a pointer variable instead of the array name to accomplish pointer arithmetic, or subscript the array name to refer to a specific element.

Exercises

7.7 Answer each of the following:
a) The _____ pointers are generic pointers that can hold pointer values of any type.
b) An array name can be thought of as a _____ pointer.
c) The _____ operator, is a unary operator that returns the address of its operand and the _____ operator returns the value of the object to which its operand points.

7.8 State whether the following are *true* or *false*. If *false*, explain why.
a) Arithmetic operations cannot be performed on pointers.
b) Pointers to functions can be used to call the functions they point to, passed to functions, returned from functions, stored in arrays, and assigned to other pointers.

7.9 Answer each of the following. Assume that integers are stored in four bytes and that the starting address of the array is at location `2003800` in memory.

a) Define an array of type `int` called `oddNum` with ten elements, and assign the odd integers from `1` to `19` to the elements. Assume the symbolic constant `SIZE` has been defined as `10`.
b) Define a pointer `iPtr` that points to an object of type `int`.
c) Print the elements of array `oddNum` using array index notation. Use a `for` statement, and assume integer control variable `i` has been defined.
d) Give two separate statements that assign the starting address of array `oddNum` to pointer variable `iPtr`.
e) Print the elements of array `oddNum` using pointer/offset notation.
f) Print the elements of array `oddNum` using pointer/offset notation with the array name as the pointer.
g) Print the elements of array `oddNum` by indexing the pointer to the array.
h) Refer to element 3 of array `oddNum` using array index notation, pointer/offset notation with the array name as the pointer, pointer index notation, and pointer/offset notation.
i) What address is referenced by `iPtr + 5`? What value is stored at that location?
j) Assuming `iPtr` points to `oddNum[9]`, what address is referenced by `iPtr -= 3`? What value is stored at that location?

7.10 For each of the following, write a single statement that performs the specified task. Assume that double precision variables `value1` and `value2` have been declared, and `value1` has been initialized to `20.4568`.

a) Declare the variable `dPtr` to be a pointer to an object of type double.
b) Assign the address of variable `value1` to pointer variable `dPtr`.
c) Print the value of the object pointed to by `dPtr`.
d) Assign the value of the object pointed to by `dPtr` to variable `value2`.
e) Print the value of `value2`.
f) Print the address of `value1`.
g) Print the address stored in `dPtr`. Is the value printed the same as `value1`'s address?

7.11 Do each of the following:

a) Write the function header for function `addNumbers` that takes two values, a long integer array parameter `numList`, and size of the array, and returns the sum of the numbers.
b) Write the function prototype for the function in part (a).
c) Write the function header for a function `sort` that takes three arguments: an integer array parameter `n`, a constant integer size and a pointer to a function *f* that takes two integers and returns an integer value; sort does not return anything.
d) Write the function prototype for the function described in part (c).

7.12 Alice and Bob are playing a game. They will throw a fair die (numbered from 1 to 6 inclusive) inside an opaque box. When they open the box, if the number on the die is odd, Alice wins; if the number is even, Bob wins. Create a decision table

that lists all possible outcomes and calculate the winning probabilities of Alice and Bob respectively.

For example:

Number on die	Odd or even	Winner
1	odd	Alice
2	even	Bob
and so on...	...	...

7.13 Let us assume Alice and Bob are playing a game described in Exercise 7.12, but Alice has an additional power. Before they open the box, Alice can choose to subtract the number on the die by 1 or leave it as is.

Create a decision table that lists all possible outcomes and calculate the winning probability of Alice and Bob respectively.

7.14 Let us assume Alice and Bob are playing a game described in Exercise 7.12, but both Alice and Bob have an additional power each. Before they open the box, Alice can choose to subtract the number on the die by 1 or leave it as is, and then Bob can choose to multiply the number by 2 or leave it as is. Alice's subtraction will be executed first before Bob's multiplication.

Create a decision table that lists all possible outcomes and calculate the winning probability of Alice and Bob respectively. What is the optimum strategy for Bob to have a higher chance of winning (if any)? If Bob deploys the optimum strategy, what's his probability of winning against Alice?

7.15 Write a program to simulate the game by Alice and Bob in Exercise 7.12. Your program should output the random number (to simulate the die throw) and then output the winner of the game (e.g., Alice or Bob).

7.16 Write a program to simulate the game by Alice and Bob in Exercise 7.14 extending the program written for Exercise 7.15. The program should prompt Alice and Bob respectively whether they would like to use their special power, before showing the end result and the winner of the game.

Write a separate function to execute Alice's and Bob's special power with the number passed in by reference and modify it directly. For example:

```
void useAliceSpecialPower(int *number) { ... }
```

Note: Exercises 7.17–7.21 are reasonably challenging. Once you have done these problems, you ought to be able to implement most popular card games easily.

7.17 *(Card Shuffling and Dealing: Dealing Poker Hands)* Modify the program in Fig. 7.16 so that the card-dealing function deals a five-card poker hand. Then write the following additional functions:

a) Determine whether the hand contains a pair.

b) Determine whether the hand contains two pairs.

c) Determine whether the hand contains three of a kind (e.g., three jacks).
d) Determine whether the hand contains four of a kind (e.g., four aces).
e) Determine whether the hand contains a flush (i.e., all five cards of the same suit).
f) Determine whether the hand contains a straight (i.e., five cards of consecutive face values).

7.18 ***(Project: Card Shuffling and Dealing—Which Poker Hand is Better?)*** Use the functions developed in Exercise 7.17 to write a program that deals two five-card poker hands, evaluates each, and determines which is the better hand.

7.19 ***(Project: Card Shuffling and Dealing—Simulating the Dealer)*** Modify the program developed in Exercise 7.18 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down" so the player cannot see it. The program should then evaluate the dealer's hand, and based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer's hand. [*Caution:* This is a difficult problem!]

7.20 ***(Project: Card Shuffling and Dealing—Allowing Player's to Draw Cards)*** Modify the program developed in Exercise 7.19 so that it can handle the dealer's hand automatically, but the player is allowed to decide which cards of the player's hand to replace. The program should then evaluate both hands and determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on these games' results, refine your poker-playing program (this, too, is a difficult problem). Play 20 more games. Does your modified program play a better game?

7.21 ***(Card Shuffling and Dealing Modification: High-Performance Shuffle)*** In Fig. 7.16, we intentionally used an inefficient card shuffling algorithm with the possibility of indefinite postponement. In this problem, you'll create a high-performance shuffling algorithm that avoids indefinite postponement.

Modify the program of Fig. 7.16 as follows. Begin by initializing the `deck` array as shown below:

Unshuffled array

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Modify the `shuffle` function to loop row-by-row and column-by-column through the array, touching every element once. Each element should be swapped with a randomly selected element of the array. Print the resulting array to determine whether the deck is satisfactorily shuffled. The following is a sample set of shuffled values:

Sample shuffled array

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

You may want your program to call the `shuffle` function several times to ensure a satisfactory shuffle.

Although the approach in this problem improves the shuffling algorithm, the dealing algorithm still requires searching the `deck` array for card 1, then card 2, then card 3, and so on. Worse yet, even after the dealing algorithm locates and deals the card, the algorithm still searches through the remainder of the deck. Modify the program of Fig. 7.16 so that once a card is dealt, no further attempts are made to match that card number, and the program immediately proceeds with dealing the next card. In Chapter 10, we develop a dealing algorithm that requires only one operation per card.

7.22 What does this program do, assuming the user enters two strings of the same length?

```
1  // ex07_22.c
2  // What does this program do?
3  #include <stdio.h>
4  #define SIZE 80
5
6  void mystery1(char *s1, const char *s2); // prototype
7
8  int main(void) {
9     char string1[SIZE]; // create char array
10    char string2[SIZE]; // create char array
11
12    puts("Enter two strings: ");
13    scanf("%39s%39s" , string1, string2);
14    mystery1(string1, string2);
15    printf("%s", string1);
16 }
17
18 // What does this function do?
19 void mystery1(char *s1, const char *s2) {
20    while (*s1 != ) {
21       ++s1;
22    }
23
24    for (; *s1 = *s2; ++s1, ++s2) {
25       ; // empty statement
26    }
27 }
```

7.23 What does this program do?

```
// ex07_23.c
// what does this program do?
#include <stdio.h>
#define SIZE 80

size_t mystery2(const char *s); // prototype

int main(void) {
   char string[SIZE]; // create char array

   puts("Enter a string: ");
   scanf("%79s", string);
   printf("%d\n", mystery2(string));
}

// What does this function do?
size_t mystery2(const char *s) {
   size_t x;

   // loop through string
   for (x = 0; *s != ; ++s) {
      ++x;
   }

   return x;
}
```

7.24 Find the error in each of the following program segments. If the error can be corrected, explain how.

a)
```
int *number;
printf("%d\n", *number);
```

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```

d)
```
char s[] = "this is a character array";
int count;
for (; *s != ; ++s) {
   printf("%c ", *s);
}
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

f)
```
float x = 19.34;
float xPtr = &x;
printf("%f\n", xPtr);
```

g)
```
char *s;
printf("%s\n", s);
```

7.25 *(Maze Traversal)* The following grid is a two-dimensional array representation of a maze. The # symbols represent the maze's walls, and the periods (.) represent squares in the possible paths through the maze.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

The Wikipedia page `https://en.wikipedia.org/wiki/Maze_solving_algorithm` lists several algorithms for finding a maze's exit. A simple algorithm for walking through a maze guarantees finding the exit (assuming there's an exit). Place your right hand on the wall to your right, and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you'll arrive at the maze's exit. If there's not an exit, you'll eventually arrive back at the starting location. There may be a shorter path than the one you've taken, but you're guaranteed to get out of the maze.

Write recursive function `mazeTraverse` to walk through the maze. The function should receive as arguments a 12-by-12 character array representing the maze and the maze's starting location. As `mazeTraverse` attempts to locate the exit from the maze, it should place the character `X` in each square in the path. The function should display the maze after each move so the user can watch as the maze is solved.

7.26 *(Generating Mazes Randomly)* Write a function `mazeGenerator` that takes as an argument a two-dimensional 12-by-12 character array and randomly produces a maze. The function should also provide the starting and ending locations of the maze. Try your function `mazeTraverse` from Exercise 7.25 using several randomly generated mazes.

7.27 *(Mazes of Any Size)* Generalize functions `mazeTraverse` and `mazeGenerator` of Exercises 7.25–7.26 to process mazes of any width and height.

7.28 What does this program do, assuming that the user enters two strings of the same length?

```
1 // ex07_28.c
2 // What does this program do?
3 #include <stdio.h>
```

```
#define SIZE 80

int mystery3(const char *s1, const char *s2); // prototype

int main(void) {
   char string1[SIZE]; // create char array
   char string2[SIZE]; // create char array

   puts("Enter two strings: ");
   scanf("%79s%79s", string1 , string2);
   printf("The result is %d\n", mystery3(string1, string2));
}

int mystery3(const char *s1, const char *s2) {
   int result = 1;

   for (; *s1 !=  && *s2 != ; ++s1, ++s2) {
      if (*s1 != *s2) {
         result = 0;
      }
   }

   return result;
}
```

Arrays of Function Pointers

7.29 ***(Arrays of Function Pointers)*** Rewrite the program of Fig. 6.17 to use a menu-driven interface. The program should offer the user four options as follows:

```
Enter a choice:
  0  Print the array of grades
  1  Find the minimum grade
  2  Find the maximum grade
  3  Print the average on all tests for each student
  4  End program
```

One restriction on using arrays of pointers to functions is that all the pointers must have the same type. The pointers must be to functions of the same return type that receive arguments of the same type. For this reason, the functions in Fig. 6.17 must be modified so that they each return the same type and take the same parameters. Modify functions `minimum` and `maximum` to print the minimum or maximum value and return nothing. For option 3, modify function `average` of Fig. 6.17 to output the average for each student (not a specific student). Function `average` should return nothing and take the same parameters as `printArray`, `minimum` and `maximum`. Store the pointers to the four functions in array `processGrades` and use the choice made by the user as the subscript into the array for calling each function.

7.30 ***(Calculating Circle Circumference, Circle Area or Sphere Volume Using Function Pointers)*** Using the techniques from Fig. 7.18, create a menu-driven program.

Allow the user to choose whether to calculate a circle's circumference, a circle's area or a sphere's volume. The program should then input a radius from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns `void` and receives a `double` parameter. The corresponding functions should each display messages indicating which calculation was performed, the value of the radius and the result of the calculation.

7.31 ***(Calculator Using Function Pointers)*** Using the techniques you learned in Fig. 7.18, create a menu-driven program that allows the user to choose whether to add, subtract, multiply or divide two numbers. The program should then input two double values from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns `void` and receives two `double` parameters. The corresponding functions should each display messages indicating which calculation was performed, the values of the parameters and the result of the calculation.

7.32 ***(Carbon Footprint Calculator)*** Using arrays of function pointers, as you learned in this chapter, you can specify a set of functions that are called with the same types of arguments and return the same type of data. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three functions that help calculate the carbon footprint of a building, a car and a bicycle, respectively. Each function should input appropriate data from the user, then calculate and display the carbon footprint. (Check out a few websites that explain how to calculate carbon footprints.) Each function should receive no parameters and return `void`. Write a program that prompts the user to enter the type of carbon footprint to calculate, then calls the corresponding function in the array of function pointers. For each type of carbon footprint, display some identifying information and the object's carbon footprint.

Special Section: Building Your Own Computer as a Virtual Machine

In the next several exercises, we take a temporary diversion away from the world of high-level language programming. We "peel open" a fake simple computer and look at its internal structure. We introduce machine-language programming for this computer and write several machine-language programs. To make this an especially valuable experience, we then build a software-based *simulation* of this computer on which you actually can execute your machine-language programs! Such a simulated computer is often called a **virtual machine**.

7.33 ***(Machine-Language Programming)*** Let's create a computer we'll call the Simpletron. As its name implies, it's a simple machine, but as we'll soon see, it's a powerful one as well. The Simpletron runs programs written in the only language it directly understands—that is, **Simpletron Machine Language**, or **SML** for short.

The Simpletron contains an **accumulator**—a "special register" in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of **words**. A word is a signed four-digit decimal number such as `+3364`, `-1293`, `+0007`, `-0001`, etc. The Simpletron is equipped with a `100`-word memory, and these words are referenced by their location numbers `00`, `01`, …, `99`.

Before running an SML program, we must **load** or place the program into memory. The first instruction (or statement) of every SML program is always placed in location `00`.

Each **SML instruction** occupies one word of the Simpletron's memory, so instructions are signed four-digit decimal numbers. We assume an SML instruction's sign is always plus, but a data word's sign may be plus or minus. Each Simpletron memory location may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. Each SML instruction's first two digits are the **operation code** specifying the operation to perform. The SML operation codes are summarized in the following table:

Operation code	Meaning
Input/output operations:	
`#define READ 10`	Read a word from the keyboard into a specific location in memory.
`#define WRITE 11`	Write a word from a specific location in memory to the screen.
Load/store operations:	
`#define LOAD 20`	Load a word from a specific location in memory into the accumulator.
`#define STORE 21`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations:	
`#define ADD 30`	Add a word from a specific location in memory to the word in the accumulator (leave the result in the accumulator).
`#define SUBTRACT 31`	Subtract a word from a specific location in memory from the word in the accumulator (leave the result in the accumulator).
`#define DIVIDE 32`	Divide a word from a specific location in memory into the word in the accumulator (leave the result in the accumulator).
`#define MULTIPLY 33`	Multiply a word from a specific location in memory by the word in the accumulator (leave the result in the accumulator).
Transfer-of-control operations:	
`#define BRANCH 40`	Branch to a specific location in memory.
`#define BRANCHNEG 41`	Branch to a specific location in memory if the accumulator is negative.
`#define BRANCHZERO 42`	Branch to a specific location in memory if the accumulator is zero.
`#define HALT 43`	Halt—i.e., the program has completed its task.

An SML instruction's last two digits are the **operand**—the memory location containing the word to which the operation applies.

Sample SML Program That Adds Two Numbers

Let's consider several simple SML programs. The following SML program reads two numbers from the keyboard, then computes and prints their sum:

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

The instruction `+1007` reads the first number from the keyboard and places it into location `07`. Then `+1008` reads the next number into location `08`. The *load* instruction, `+2007`, copies the first number into the accumulator. The *add* instruction, `+3008`, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator.* The *store* instruction, `+2109`, copies the result from the accumulator into memory location `09`, from which the *write* instruction, `+1109`, then takes the number and prints it as a signed four-digit decimal number to the screen. The *halt* instruction, `+4300`, terminates execution.

Sample SML Program That Determines the Largest of Two Values

The next SML program reads two numbers from the keyboard, then determines and prints the larger value:

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

The instruction +4107 is a conditional transfer of control, like an `if` statement.

Now write SML programs to accomplish each of the following tasks.

a) Use a sentinel-controlled loop to read positive integers, then compute and print their sum.

b) Use a counter-controlled loop to read seven numbers, some positive and some negative. Compute and print their average.

c) Read a series of numbers. Determine and print the largest number. The first number read indicates how many numbers should be processed.

7.34 ***(A Computer Simulator)*** It may at first seem outrageous, but in this exercise you'll build your own computer. No, you won't be soldering components together. Rather, you'll use the powerful technique of **software-based simulation** to create a **software model** of the Simpletron. You will not be disappointed. Your Simpletron simulator will turn the computer you're using into a Simpletron, and you'll actually be able to run, test and debug the SML programs you wrote in Exercise 7.33!

When you run your Simpletron simulator, it should begin by printing:

```
***           Welcome to Simpletron           ***
***                                           ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program.                             ***
```

Simulate the memory of the Simpletron with a 100-element one-dimensional array `memory`. Now assume that the simulator is running, and let's examine the dialog as we enter the program of Example 2 of Exercise 7.33:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins  ***
```

The SML program has now been placed (or loaded) into the array `memory`. Next, the Simpletron executes the SML program. It begins with the instruction in location

00 and continues sequentially, unless directed to some other part of the program by a transfer of control.

Use the variable `accumulator` to represent the accumulator register. Use the variable `instructionCounter` to store the number of the memory location (00 to 99) containing the instruction being performed. Use the variable `operationCode` to store the operation currently being performed (the instruction word's left two digits). Use the variable `operand` to store the number of the memory location on which the current instruction operates. Thus, if an instruction has an `operand`, it's the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then "pick off" the left two digits and place them in the variable `operationCode`, and "pick off" the right two digits and place them in `operand`.

When Simpletron begins execution, the special registers are initialized as follows:

```
accumulator            +0000
instructionCounter        00
instructionRegister    +0000
operationCode             00
operand                   00
```

Now let's "walk through" the execution of the first SML instruction, `+1009` in memory location `00`. This process is called an **instruction execution cycle**.

The `instructionCounter` tells us the location of the next instruction to be performed. We *fetch* the contents of that location from `memory` by using the C statement

```
instructionRegister = memory[instructionCounter];
```

The operation code and the operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, and so on). A `switch` statement differentiates among the twelve operations of SML. The `switch` simulates the behavior of various SML instructions as follows (we leave the others to the reader):

read: `scanf("%d", &memory[operand]);`

load: `accumulator = memory[operand];`

add: `accumulator += memory[operand];`

Various branch instructions: We'll discuss these shortly.

halt: This instruction prints the message

```
*** Simpletron execution terminated ***
```

then prints the name and contents of each register as well as the complete contents of all 100 memory locations. Such a printout is often called a **computer dump**. To help you program your `dump` function, the output below shows a sample dump:

```
REGISTERS:
accumulator              +0000
instructionCounter          00
instructionRegister      +0000
operationCode               00
operand                     00

MEMORY:
        0      1      2      3      4      5      6      7      8      9
 0  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
10  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
20  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
30  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
40  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
50  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
60  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
70  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
80  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
90  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000  +0000
```

A dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. You can print leading 0s in front of an integer that is shorter than its field width by placing the 0 formatting flag before the field width in the format specifier as in `"%02d"`. You can place a + or - sign before a value with the + formatting flag. So to produce a number of the form +0000, you can use the format specifier `"%+05d"`.

Let's proceed with the execution of our program's first instruction, namely the `+1009` in location `00`. As we've indicated, the `switch` statement simulates this by performing the statement

```
scanf("%d", &memory[operand]);
```

A question mark (`?`) should be displayed on the screen before the `scanf` is executed to prompt the user for input. The Simpletron waits for the user to type a value and then press the *Return* (or *Enter*) key. The value is then read into location `09`.

At this point, simulation of the first instruction is completed. All that remains is to prepare the Simpletron to execute the next instruction. Because the instruction just performed was not a transfer of control, we need merely increment the instruction counter register as follows:

```
++instructionCounter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the **instruction execution cycle**) begins anew with the **fetch** of the next instruction to be executed.

Now let's consider how the branching instructions—the transfers of control—are simulated. All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (`40`) is simulated within the `switch` as

```
instructionCounter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if (accumulator == 0) {
   instructionCounter = operand;
}
```

At this point, you should implement your Simpletron simulator and run the SML programs you wrote in Exercise 7.33. You may embellish SML with additional features and provide for these in your simulator. Exercise 7.35 lists several possible embellishments.

Your simulator should check for various types of errors. During the program **loading phase**, for example, each number the user types into the Simpletron's `memory` must be in the range `-9999` to `+9999`. Your simulator should use a `while` loop to test that each number entered is in this range, and, if not, keep prompting the user to reenter the number until a correct number is entered.

During the execution phase, your simulator should check for serious errors, such as attempts to **divide by zero**, attempts to execute an **invalid operation code** and **accumulator overflows** (i.e., arithmetic operations resulting in values larger than `+9999` or smaller than `-9999`). Such serious errors are fatal errors. When a fatal error is detected, print an error message such as:

```
*** Attempt to divide by zero                  ***
*** Simpletron execution abnormally terminated ***
```

and print a full computer dump in the format we've discussed previously. This will help the user locate the error in the program.

Implementation Note: When you implement the Simpletron Simulator, define the `memory` array and all the registers as variables in `main`. The program should contain three other functions—`load`, `execute` and `dump`. Function `load` reads the SML instructions from the user at the keyboard. (Once you study file processing in Chapter 11, you'll be able to read the SML instruction from a file.) Function `execute` executes the SML program currently loaded in the `memory` array. Function `dump` displays the contents of `memory` and all of the registers stored in `main`'s variables. Pass the `memory` array and registers to the other functions as necessary to complete their tasks. Functions `load` and `execute` need to modify variables that are defined in `main`, so you'll need to pass those variables to the functions by reference using pointers. You'll need to modify the statements we showed throughout this problem description to use the appropriate pointer notations.

7.35 ***(Modifications to the Simpletron Simulator)*** In this exercise, we propose several modifications and enhancements to Exercise 7.34's Simpletron Simulator. In Exercises 12.24 and 12.25, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler:

a) Extend the Simpletron Simulator's memory to contain 1000 memory locations (000 to 999) to enable the Simpletron to handle larger programs.

b) Allow the simulator to perform remainder calculations. This requires an additional Simpletron Machine Language instruction.
c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.
d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions. Online Appendix E, Number Systems, discusses hexadecimal.
e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.
f) Modify the simulator to process floating-point values in addition to integer values.
g) Modify the simulator to detect division by 0 logic errors.
h) Modify the simulator to detect arithmetic-overflow errors.
i) Modify the simulator to handle string input. [*Hint:* Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store it beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to either a left or right half word.]
j) Modify the simulator to handle output of strings stored in the format of part (g). [*Hint:* Add a machine-language instruction that prints a string beginning at a specified Simpletron memory location. The first half of the word at that location is the length of the string in characters. Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

Special Section—Embedded Systems Programming Case Study: Robotics with the Webots Simulator

7.36 Webots[5,6] is an open-source, full-color, 3D, robotics simulator with console-game-quality graphics. It enables you to create a virtual reality in which robots interact with simulated real-world environments. It runs on Windows, macOS and Linux. The simulator is widely used in industry and research to test robots' viability and

5. "Webots Open Source Robot Simulator."Accessed December 11, 2020. `https://cyberbotics.com`.
6. The Webots environment screen captures in this case study are Copyright 2020 Cyberbotics Ltd., which is licensed under the Apache License, Version 2.0 (`http://www.apache.org/licenses/LICENSE-2.0`).

develop controller software for those robots. Webots uses an Apache open-source license. The following is from their license webpage:[7]

> *"Webots is released under the terms of the Apache 2.0 license agreement. Apache 2.0 is a [sic] industry friendly, non-contaminating, permissive open source license that grants everyone the right to use a software source code, free of charge, for any purpose, including commercial applications."*

Webots was first developed in 1996 at the Swiss Federal Institute of Technology (EPFL). In 1998, the EPFL spin-off Cyberbotics Ltd. was founded to take over the Webots simulator development. Until 2018, Webots was sold as proprietary licensed software. In 2018, Cyberbotics open-sourced Webots under the Apache 2.0 license.[8,9]

Robotics is not typically discussed in introductory programming textbooks, but Webots makes it easy. Webots comes bundled with simulations for dozens of today's most popular real-world robots that walk, fly, roll, drive and more. For a current list of bundled robots, visit

```
https://cyberbotics.com/doc/guide/robots
```

Self-Contained Development Environment

Webots is a **self-contained robotics-simulation environment** with everything you need to begin developing and experimenting with robotics. It includes:

- an **interactive 3D simulation area** for viewing and interacting with simulations,
- a **code editor** where you can view the bundled simulation code, modify it and write your own, and
- **compilers and interpreters** that enable you to write Webots code in C, C++, Java, Python and MatLab.

You can easily modify existing simulations and re-run their code to see the effects of your changes. You also can develop entirely new simulations, as you'll do in this case study.

Installing Webots

First, check the Webots system requirements at

```
https://cyberbotics.com/doc/guide/system-requirements
```

You can download the Webots installer for Windows, macOS or Linux from the Cyberbotics home page:

```
https://cyberbotics.com/
```

Once downloaded, run the installer and follow the on-screen prompts.

---

7. "Webots User Guide R2020b revision 2—License Agreement." Accessed December 14, 2020. `https://cyberbotics.com/doc/guide/webots-license-agreement`.
8. "Cyberbotics." Accessed December 13, 2020. `https://cyberbotics.com/#cyberbotics`.
9. "Webots." Accessed December 13, 2020. `https://en.wikipedia.org/wiki/Webots`.

Guided Tour

Once the installation completes, run the **Webots application** on your system. You can get a quick overview of many bundled robotics simulations and the robots' capabilities in the **Webots guided tour**. To do so, select **Help > Webots Guided Tour...** (the first time you open the Webots environment, this guided tour will begin automatically). Then, in the window that appears (Fig. 7.19), check the **Auto** checkbox and click **Next**. This will begin the automated guided tour, which will show you each demonstration for a short time, then switch to the next. The **Guided Tour - Webots** window displays a brief description of each simulation. Figure 7.19 shows the description for the third simulation, which appears in Fig. 7.20.

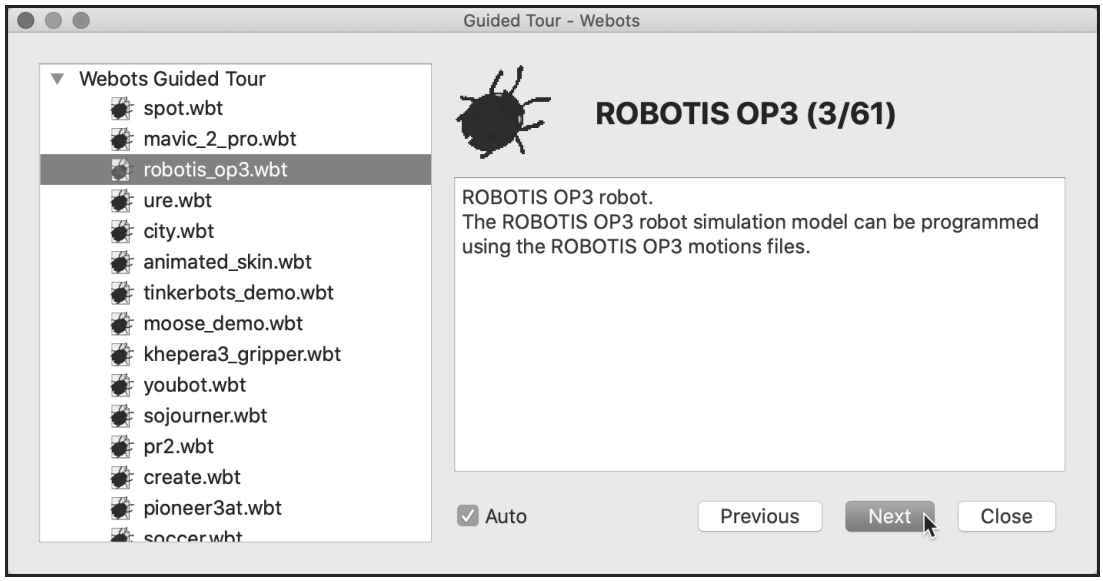


Fig. 7.19 | A brief description of the ROBOTIS OP3 robot shown in Fig. 7.20. [The screen captures in this case study are Copyright 2020 Cyberbotics Ltd., which is licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at `http://www.apache.org/licenses/LICENSE-2.0`.]

As the guided tour overviews each simulation, you'll see it live in the Webots environment's 3D viewing area—shown in the center portion of Fig. 7.20. As you'll soon see:

- the environment's left side enables you to manage and configure the components in your robotics simulations, and
- the environment's right side provides an integrated code editor for writing and compiling the C code that controls a robot in your simulation.

We narrowed the code editor for this screen capture. As in most IDEs, the areas within the window can be resized by dragging the divider bars.

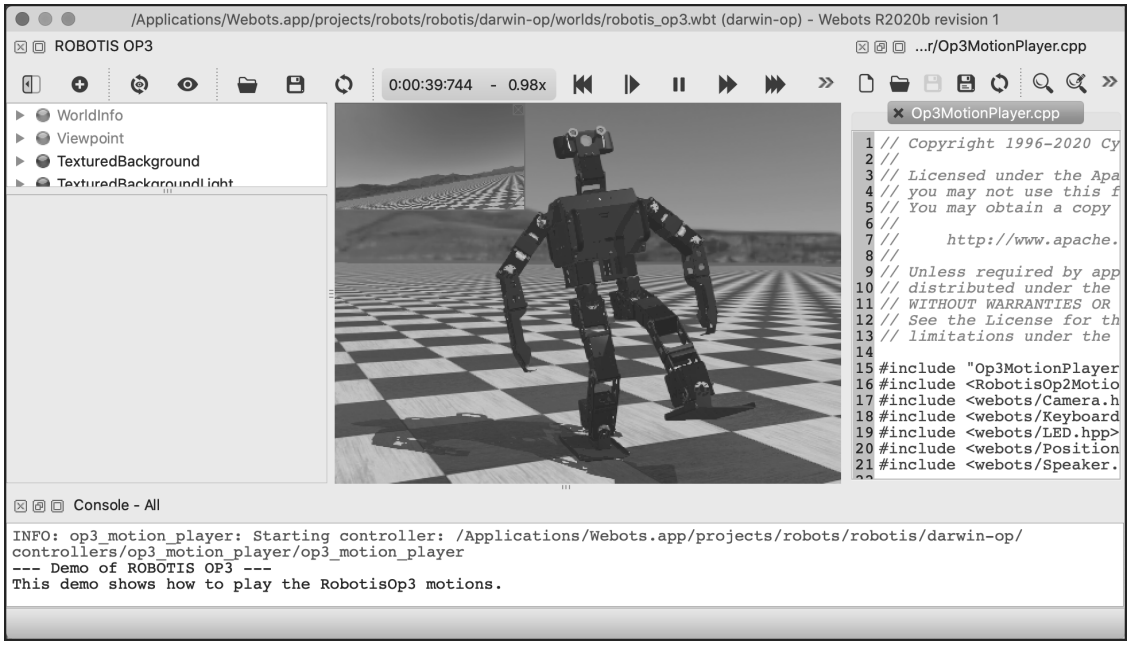


Fig. 7.20 | A ROBOTIS OP3 robotics simulation running in Webots.

User Interface Overview

Familiarize yourself with the Webots environment's user interface by reading the following overview:

```
https://cyberbotics.com/doc/guide/the-user-interface
```

Webots Tutorial 1: Your First Simulation in Webots

The Webots team provides eight tutorials that introduce you to many aspects of the Webots environment and its robotics-simulation capabilities:

- Tutorial 1: Your First Simulation in Webots
- Tutorial 2: Modification of the Environment
- Tutorial 3: Appearance
- Tutorial 4: More about Controllers
- Tutorial 5: Compound Solid and Physics Attributes
- Tutorial 6: 4-Wheels Robot
- Tutorial 7: Your First PROTO
- Tutorial 8: Using ROS

In this case study, you'll follow their first tutorial to create a robotics simulation using several predefined items:

- a `RectangleArena` in which your robot will roam,
- several `WoodenBox` obstacles, and

- an **e-puck robot**—a simple simulated robot that will move around the `RectangleArena` and change directions when it encounters a `WoodenBox` or a wall.

The e-puck simulator corresponds to a real-world, educational robot[10] that has:

- two independently controlled wheels (known as **differential wheels**), so the **robot can change direction** by moving its wheels at different speeds,
- a camera (the upper-left corner of the Webots environment's 3D viewing area shows a small window in which you can **view what a robot "sees"**),
- eight **distance sensors**, and
- **10 LED lights with controllable intensity**—other e-puck robots can "see" these via their camera for visual interactions between multiple e-puck robots.

This tutorial focuses on using the wheels to make the robot move. You can work through Webots Tutorials 2–4 to use additional e-puck features. For more information on the e-puck robot, visit:

```
http://www.e-puck.org/
```

Tutorial Steps

You can find the first Webots tutorial at

```
https://cyberbotics.com/doc/guide/tutorial-1-your-first-simulation-
  in-webots
```

Below, we overview each tutorial step, provide additional insights and clarify some of the tutorial instructions.[11] The step numbers in the rest of this exercise correspond to the "Hands-on" steps in the Webots tutorial. For each step, you should:

1. read the Webots tutorial step,
2. read our additional comments for that step, then
3. perform the step's task(s).

The tutorial shows only one diagram of the robotics simulation you'll create. To help you work through the tutorial, we include additional screen captures[12] to clarify the tutorial instructions. **As you work through the steps, be sure to save your changes after each modification. If you have to reset the simulation, it will revert to the last saved version.**

10. "The e-puck, a Robot Designed for Education in Engineering." Accessed December 13, 2020. `https://infoscience.epfl.ch/record/135236?ln=en`.
11. This discussion was written in December 2020. The software, documentation and tutorials could change. If you run into problems, visit the Webots forum (`https://discord.com/invite/nTWbN9m`) and check the Webots StackOverflow questions (`https://stackoverflow.com/questions/tagged/webots`) or e-mail us at `deitel@deitel.com`.
12. We changed the background color in our version of the simulation so that the screen captures would be more readable in print.

Step 1: Launch the Webots Application

In **Step 1**, you'll simply open the Webots environment. You'll then perform the additional steps discussed below.

Step 2: Create Your Virtual World

A **world** defines the simulation environment and is stored in a **`.wbt` file**. Internally, this file uses **Virtual Reality Modeling Language (VRML)** to describe your world's elements. Each world can have characteristics, such as **gravity**, that affect object interactions. The world specifies the area in which your robot can roam and the objects with which it can interact. The **Wizards** menu's **Create a Webots project directory** wizard (Figs. 7.21–7.24) will guide you through setting up a new world with Webots' required folder structure.

In the first screen of the **Create a Webots project directory** wizard (Fig. 7.21), simply click **Continue** (or **Next**) to move to the next step.

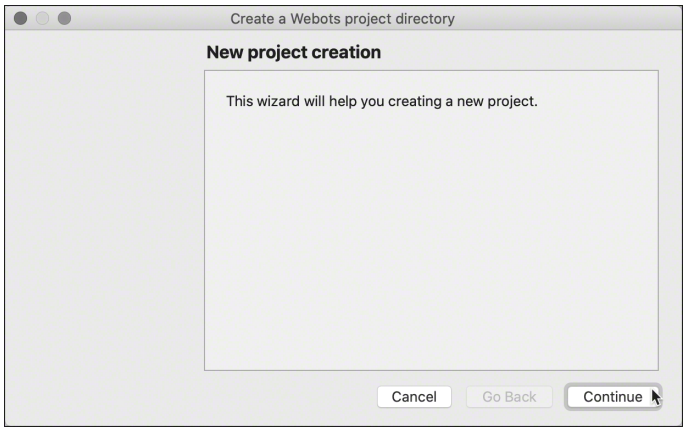


Fig. 7.21 | Initial **Create a Webots project directory** wizard window.

In the wizard's **Directory selection** step (Fig. 7.22), change your project's directory name from `my_project` (the default) to `my_first_simulation`—this folder's default location is your user account's `Documents` folder.[13]

In the wizard's **World settings** step (Fig. 7.23), you'll change the world's filename from `empty.wbt` (the default) to `my_first_simulation.wbt` and ensure that all four checkboxes are checked. This will add several Webots predefined components to your virtual world.

The wizard's **Conclusion** step (Fig. 7.24) shows all the folders and files the wizard will generate for your simulation. In subsequent steps, you'll add obstacles and an e-puck robot, configure various settings and write some C code that controls the robot.

13. You can change the location where your Webots project is stored. Ours is stored in our user account's `/Users/pauldeitel/Documents` folder, which you'll see in several screen captures.

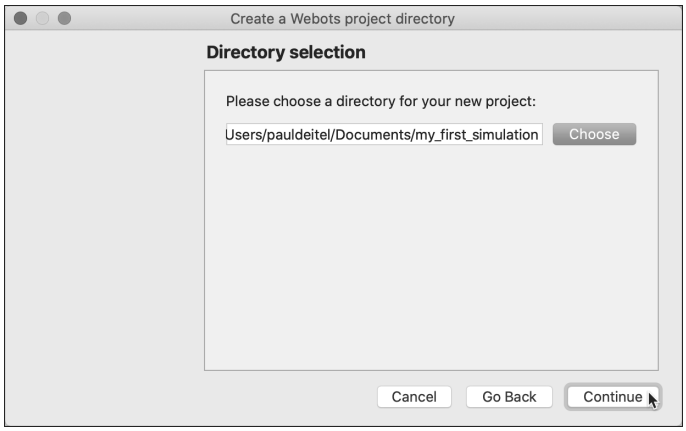


Fig. 7.22 | Changing the default project directory name to `my_first_simulation`.

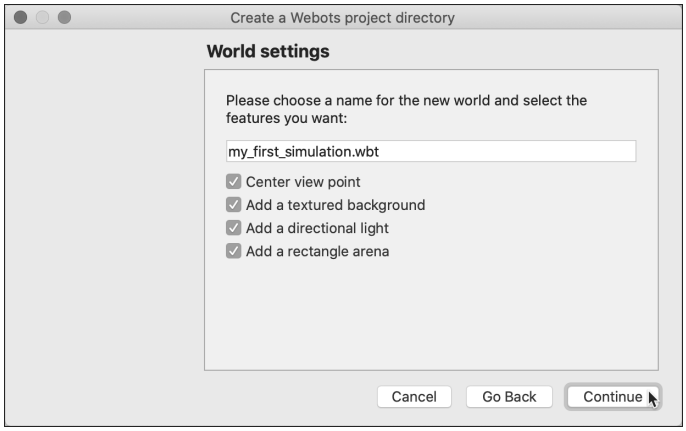


Fig. 7.23 | Changing the default world filename and ensuring all checkboxes are checked.

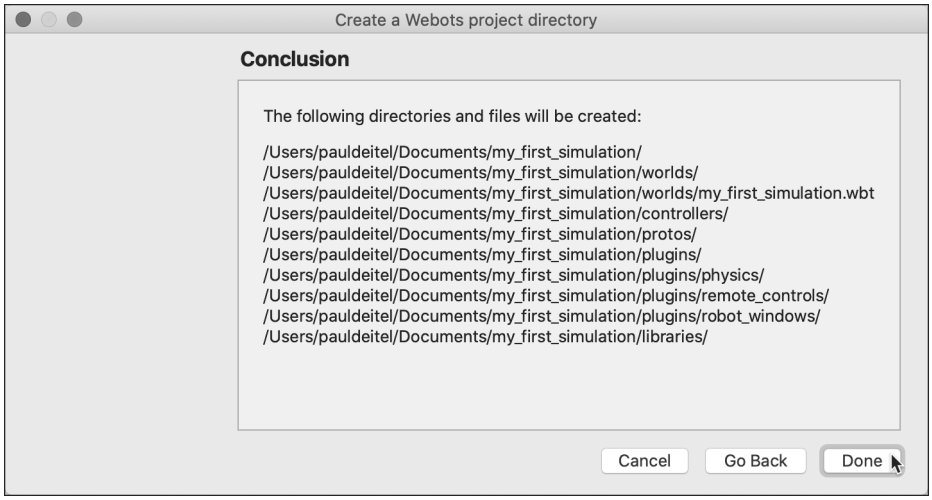


Fig. 7.24 | Summary of the folders and files the wizard will generate for your simulation.

When you click **Done** (macOS) or **Finish** (Windows and Linux) in the **Create a Webots project directory** wizard (Fig. 7.24), the Webots 3D viewing area displays an empty `RectangleArena` with a checkered floor—the default for a `RectangleArena` (Fig. 7.25). There are several floor-style options. You can experiment with these after you learn how to change settings for the elements in your world. In the 3D viewing area, you can zoom in and out using your mouse wheel, and you can click and drag the `RectangleArena` to view it from different angles and rotate it.

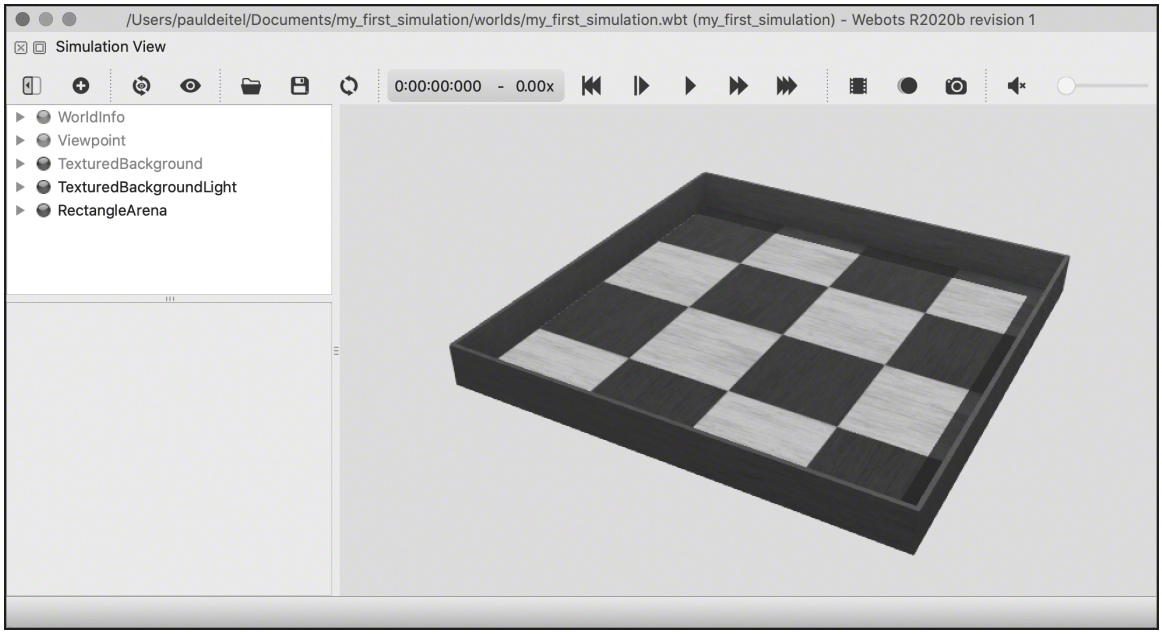


Fig. 7.25 | Initial view of the `RectangleArena` after completing the **Create a Webots project directory** wizard.

Step 3: Modify the `RectangleArena`

Each element in the simulation is a **node** in the world's **scene tree**, which you can see at the left side of the Webots environment. In this step, you'll select a node in the scene tree, then change some of its settings, called **fields**. After you perform this step, the environment should appear as in Fig. 7.26.

The colored arrows over the `RectangleArena` appear for any object you select in the world, either by clicking it in the 3D viewing area or by clicking its node in the scene tree. You can drag these arrowheads to move, rotate and tilt the object.[14] Dragging a straight arrowhead moves the object along that axis. Dragging a circular arrowhead rotates or tilts the object around that axis. Save your environment, then experiment with these arrowheads to see how they affect the `RectangleArena`'s position. You can then reset the environment to restore the original position.

14. "Webots User Guide R2020b revision 2 — The 3D Window — Moving a Solid Object." Accessed December 13, 2020. `https://cyberbotics.com/doc/guide/the-3d-window#moving-a-solid-object`.

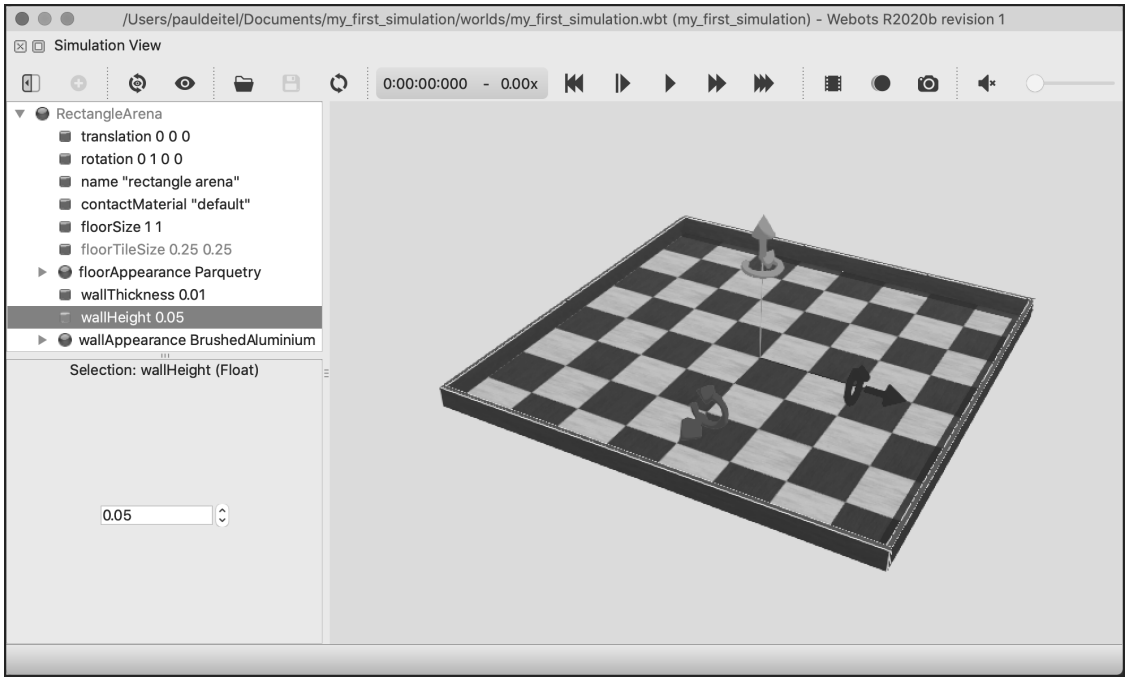


Fig. 7.26 | `RectangleArena` after changing the checkerboard floor pattern's square size and reducing the wall height.

Step 4: Add `WoodenBox` Obstacles

The Webots environment comes with almost **800 predefined objects and robots**—called **PROTO nodes**. A PROTO node describes a complex object or robot that you can add to your simulations. The wide variety of PROTO nodes enables you to create realistic 3D simulations of real-world environments. You also can create your own PROTO nodes.

In this step, you'll add a `WoodenBox` PROTO node using the **Add a node** dialog (Fig. 7.27). When you select a PROTO node in the dialog, it shows a brief node description, provides a link to the node's more detailed online documentation and shows the node's licensing information (with a link for more license information). Browse through the **Add a node** dialog to get a sense of the wide variety of **animate** (robots and vehicles) and **inanimate** (walls, buildings, furniture, plants, etc.) PROTO nodes you can use in your simulations.

Next, you'll **size the `WoodenBox`** and **move it**. Then you'll **make two copies** and move them to other locations within the `RectangleArena`. When this step asks you to copy-and-paste a `WoodenBox`, the new one will have the same size and location as the one you copied. **Hold the *Shift* key** and drag the new one to a different location to see it. We found it easier to copy nodes by selecting them in the scene tree. When you complete this step, your world should appear similar to Fig. 7.28. The last `WoodenBox` you moved will be selected in your world.[15]

15. "Webots User Guide R2020b revision 2 — The 3D Window." Accessed December 13, 2020. `https://cyberbotics.com/doc/guide/the-3d-window#moving-a-solid-object`.

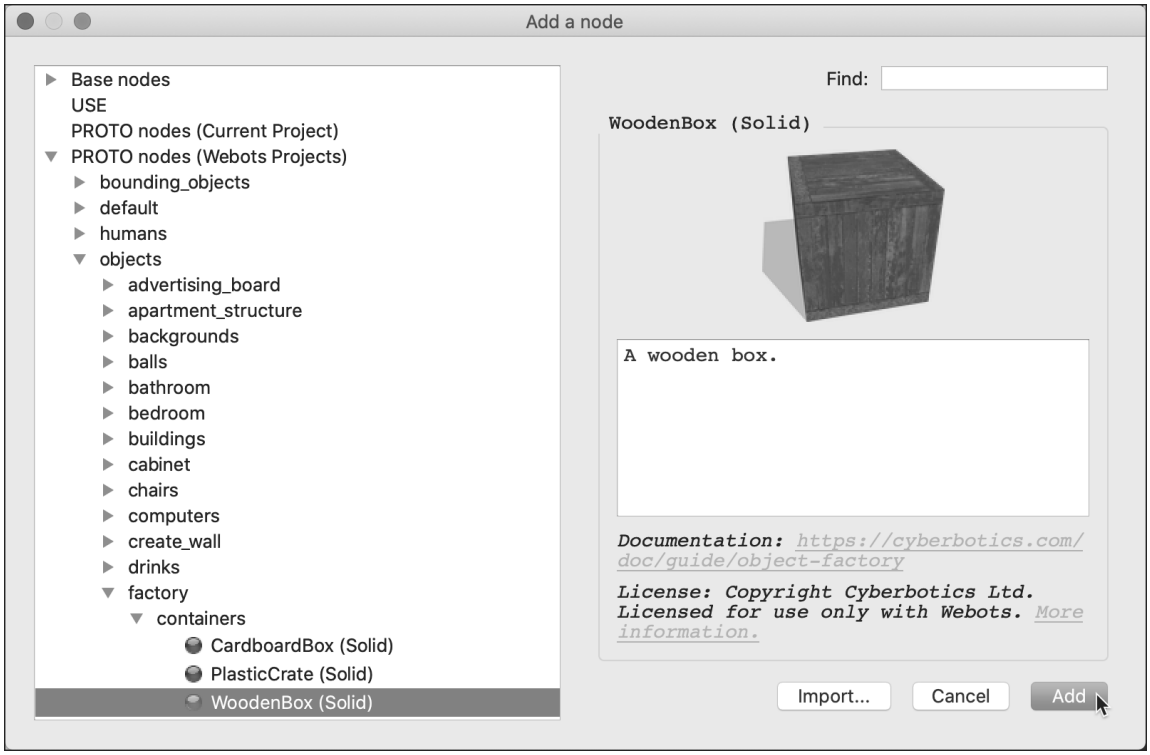


Fig. 7.27 | Selecting a `WoodenBox` PROTO node in the **Add a node** dialog.

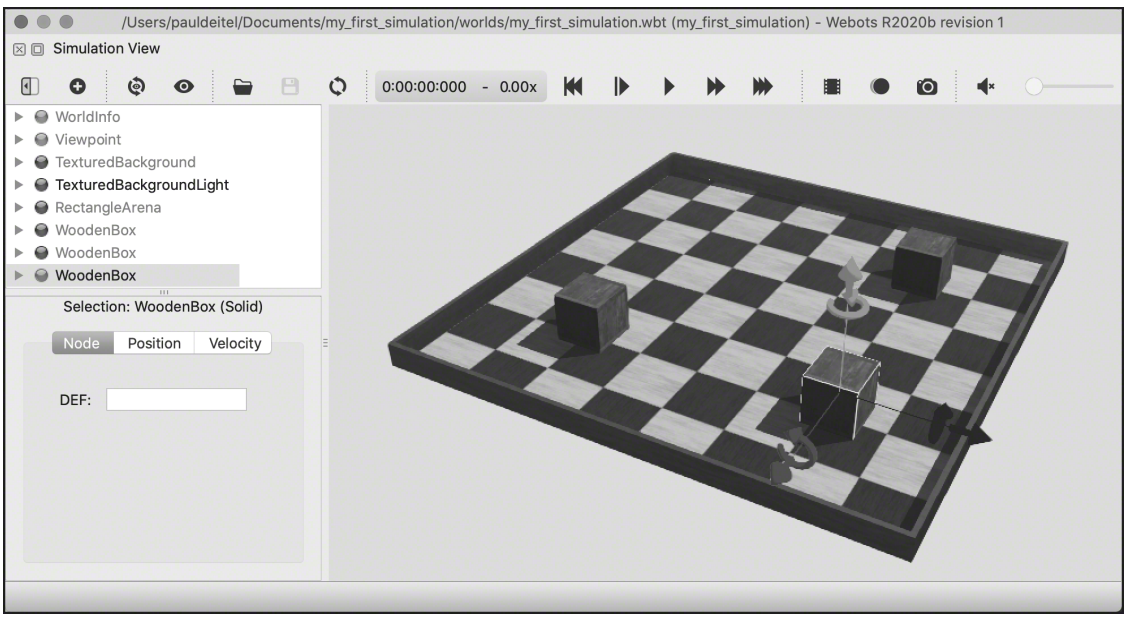


Fig. 7.28 | Virtual world after creating and positioning three `WoodenBox` objects.

Step 5: Add a Robot to Your Virtual World

In this step, you'll use the **Add a node** dialog to add an e-puck robot to the simulation (Fig. 7.29). When you select the E-puck (Robot) node, the dialog displays a description of the robot, the robot's website (`http://www.e-puck.org`), the robot's Webots documentation link and the robot's license information.

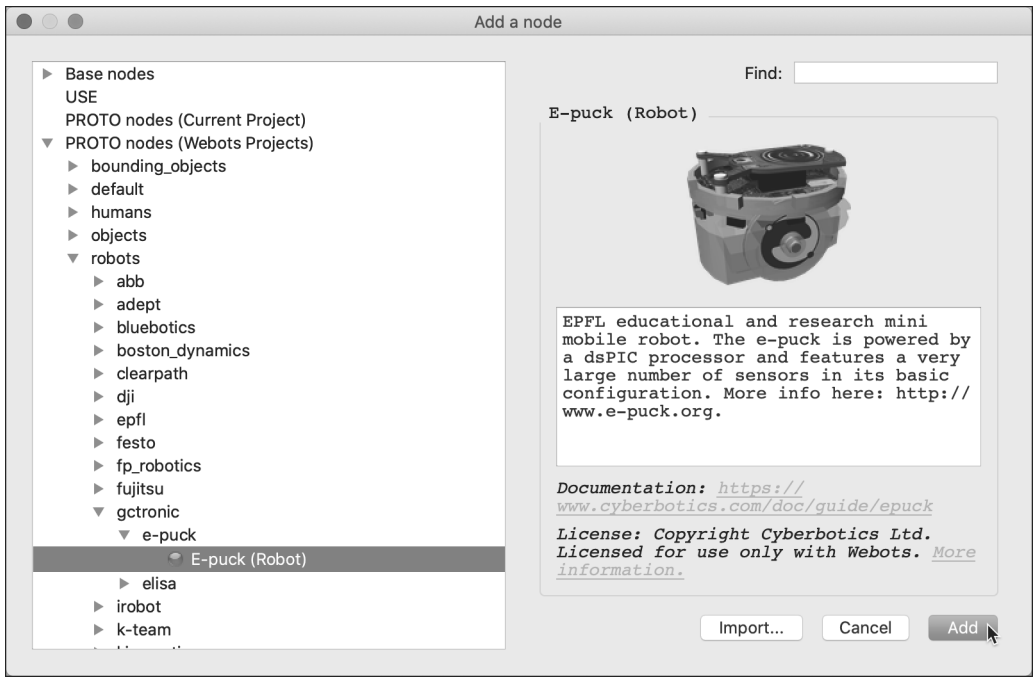


Fig. 7.29 | Adding an e-puck robot PROTO node to the simulation.

The e-puck is **preconfigured to move forward**, **rotating left to change direction** if it **collides with an obstacle**, such as a `WoodenBox` or a wall. Though small, an e-puck robot actually is loaded with technology, including **distance sensors**, which can be used to program it to **avoid collisions** entirely. In Webots Tutorial 4, you'll learn how to avoid obstacles using these distance sensors.

A robot's behavior is specified by its controller. The default e-puck robot controller we just described is named **`e-puck_avoid_obstacles`** (you can view this code in the text editor, by selecting **Tools > Text Editor**). Studying existing bundled controllers is a great way to learn more about controlling Webots robots. To complete this step, you'll run the e-puck robot's default controller (Fig. 7.30). We clicked the world's background area to deselect all the simulation elements.

Step 6: Playing with Physics

The Webots simulator has a physics engine that enables objects to act and interact as they would in the real world. Some physics options you can configure include density, mass, inertia, friction and bounce. For more, visit:

```
https://cyberbotics.com/doc/reference/physics
```

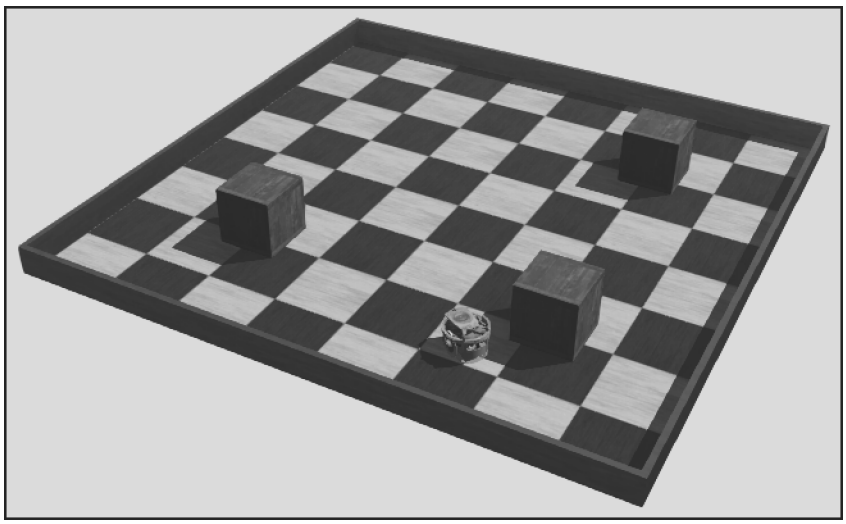


Fig. 7.30 | Simulation with an e-puck robot roaming the `RectangleArena`.

In this step, you'll use your mouse to **apply a force to the e-puck robot**. After you do this, your world should appear similar to Fig. 7.31, with the red arrow indicating the force's direction. When you perform this step, you might accidentally tip over the robot if you apply too much force—as shown in Fig. 7.31. Of course, robots can tip over in the real world, too. To fix this in the simulation, click the **Reset Simulation** button, which will restore the simulation to the point of your most recent save.

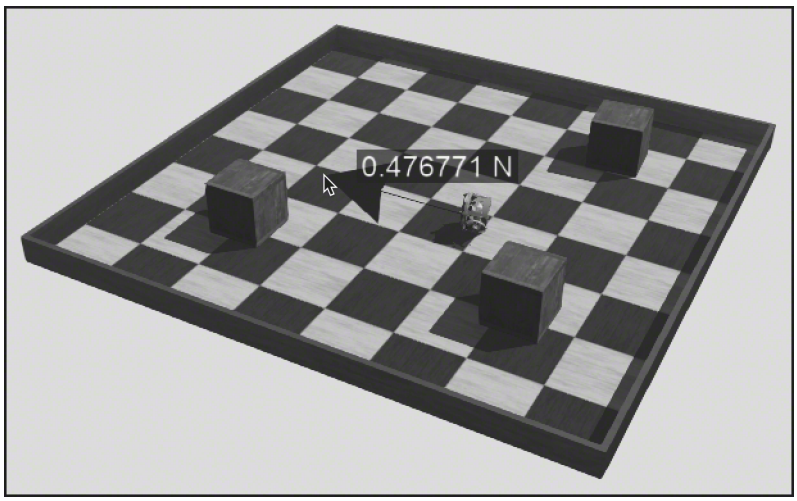


Fig. 7.31 | Manually applying a force to the e-puck robot. In this case, too much force was applied, tipping over the robot.

The `WoodenBox` obstacles you created in **Step 4** are stuck to the floor by default and do not move when the e-puck bumps into them. You'll see in this step that setting the `WoodenBox`es' masses enables them to respond to force. The smaller a `Wooden-`

Box's mass, the more it will move when the e-puck robot collides with it. The tutorial recommends setting the WoodenBoxes' masses to 0.2 kilograms. Try setting each WoodenBox's mass to smaller and larger values to see how these masses affect the physics interactions.

Step 7: Decrease the World's Time Step

Throughout a simulation, Webots keeps track of your simulation's **virtual time**. The **basic time step** is a value in milliseconds. Throughout the simulation, when virtual time increases by the basic time step, Webots performs its physics calculations:[16]

- Larger basic-time-step values decrease the physics-calculation frequency. This enables simulations to run faster because they perform fewer calculations; however, this can make physics interactions, such as collisions between objects, less accurate, and the simulation can feel clunky.
- Smaller values increase the physics-calculation frequency, making physics calculations more accurate. This causes simulations to run slower because they perform more calculations, but movements may appear smoother.

When you created this simulation's files, Webots set the basic time step to 32 milliseconds. In this step, you'll decrease the basic time-step value to 16 milliseconds. For tips on Webots simulation speed and performance, see:

```
https://www.cyberbotics.com/doc/guide/speed-performance
```

Step 8: Create a C Source-Code File for Your Robot's Controller

In this step, you'll replace the default e-puck_avoid_obstacles controller with a new **custom controller**. Many robots can use each controller you create, but each robot may have only one controller at a time. Figures 7.32–7.35 show the steps you'll go through after selecting Wizards > New Robot Controller.... These steps will **create a new C source-code file for your custom controller** and open it in the code editor at the Webots environment's right side:

- In the New controller creation step (Fig. 7.32), you'll simply click Continue.
- In the Language selection step (Fig. 7.33), ensure that C is selected, then click Continue.
- In the Name selection step (Fig. 7.34), change the default custom controller name to epuck_go_forward from my_controller and click Continue.
- The Conclusion step (Fig. 7.35) shows all the folders and files the wizard will generate for your controller.

16. "Webots Reference Manual R2020b revision 2: WorldInfo." Accessed December 12, 2020. https://www.cyberbotics.com/doc/reference/worldinfo.

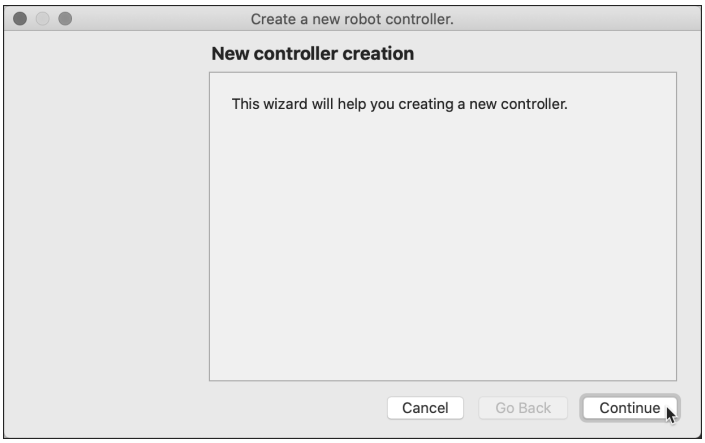


Fig. 7.32 | Initial **Create a new robot controller** wizard window.

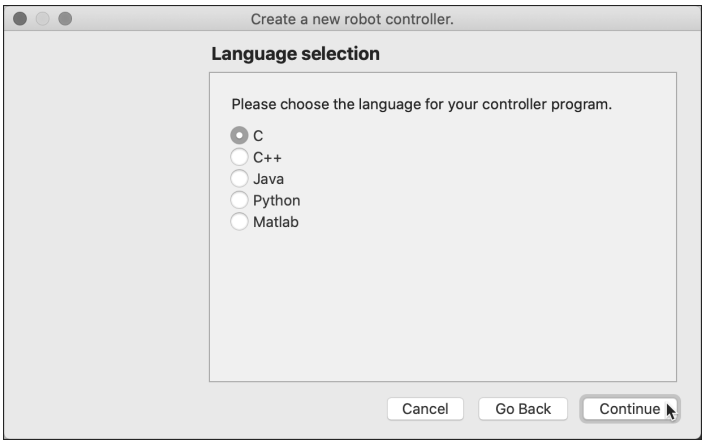


Fig. 7.33 | Selecting the C programming language for your custom robot controller.

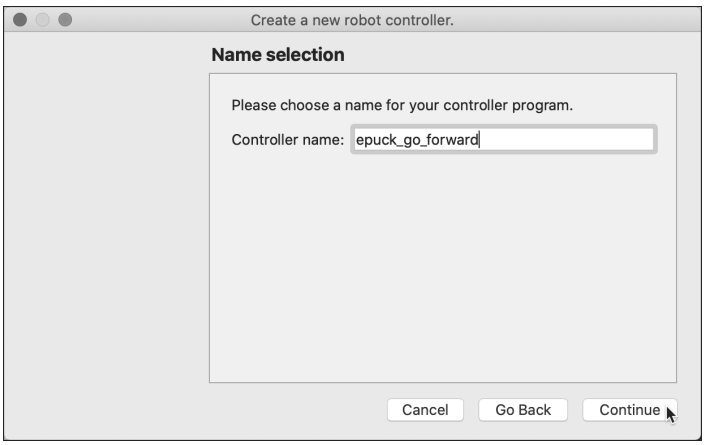


Fig. 7.34 | Changing the default custom controller name to `epuck_go_forward`.

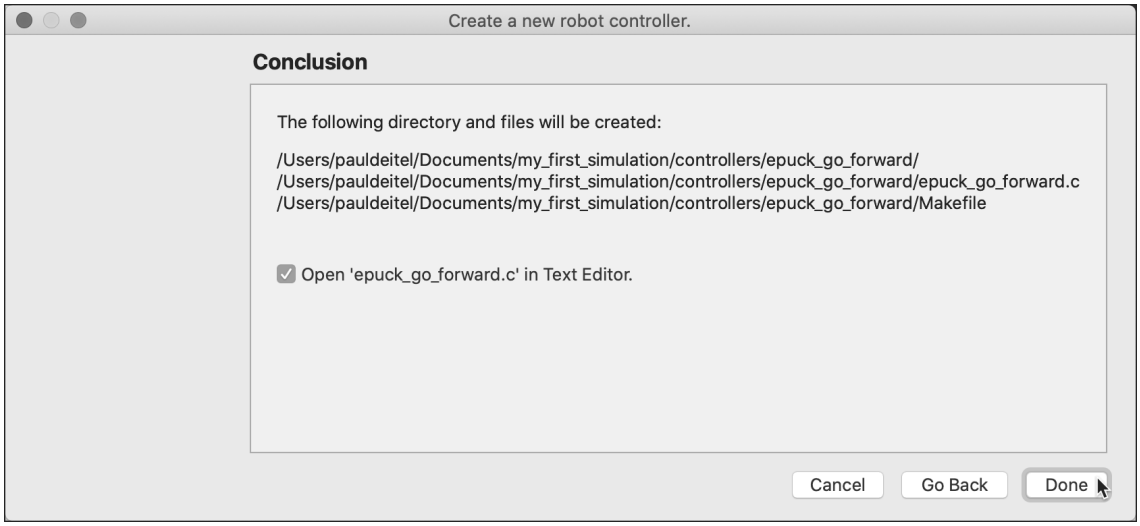


Fig. 7.35 | Summary of the folders and files the wizard will generate for your custom controller.

Step 9: Modify Your Controller's Code to Move the Robot Forward

The new controller you created in **Step 8** contains the basic framework of a simple controller. In this step, you'll add code to this file that will make the e-puck move forward a short distance. The tutorial does not specify precisely where each statement should be added to the controller code, so make the following changes to your controller's source-code file:

1. Add the following `#include` before `main`:

```
#include <webots/motor.h>
```

2. Next, you'll use the Webots `wb_robot_get_device` function[17] to get the devices that represent the e-puck robot's left- and right-wheel motors. Add the following code in `main` after the call to `wb_robot_init`:

```
// get the motor devices
WbDeviceTag left_motor =
   wb_robot_get_device("left wheel motor");
WbDeviceTag right_motor =
   wb_robot_get_device("right wheel motor");
```

3. Finally, you'll use the Webots `wb_motor_set_position` function[18] to move the robot a short distance. Add the following code after the calls to `wb_robot_get_device` and before the `while` loop:

```
wb_motor_set_position(left_motor, 10.0);
wb_motor_set_position(right_motor, 10.0);
```

17. "Webots Reference Manual R2020b revision 2—Robot." Accessed December 13, 2020. https://cyberbotics.com/doc/reference/robot#wb_robot_get_device.
18. "Webots Reference Manual R2020b revision 2—Motor." Accessed December 13, 2020. https://cyberbotics.com/doc/reference/motor#wb_motor_set_position.

Experiment with different values for `wb_motor_set_position`'s second argument to get a sense of how they affect the distance traveled. Also try using different values in the two calls to `wb_motor_set_position` so that the two wheels do not rotate the same amount.

Step 10: Modify Your Controller's Code to Change the Robot's Speed

In this final step, you'll modify your controller code to specify the wheels' speeds. Make the following changes to your controller's source-code file:

1. Speeds in Webots use **radians for rotational motors**, such as those used in wheels; otherwise, speeds use **meters per second**. Add the following `#define` directive after `#include <webots/robot.h>` to define the robot's maximum wheel-rotation **speed in radians** (6.28 is 2π radians):

   ```
   #define MAX_SPEED 6.28
   ```

2. Modify your two calls to the `wb_motor_set_position` function, replacing their second arguments with the Webots constant `INFINITY`, so the **wheels spin continuously** (at the speed you'll set momentarily) throughout the simulation:

   ```
   // set wheels to spin continuously
   wb_motor_set_position(left_motor, INFINITY);
   wb_motor_set_position(right_motor, INFINITY);
   ```

3. Finally, you'll use the Webots function `wb_motor_set_velocity` function[19] to specify the wheel-rotation speeds in radians per second. Add the following code after the calls to `wb_robot_get_device` and before the `while` loop:

   ```
   // set up the motor speeds at 10% of the MAX_SPEED
   wb_motor_set_velocity(left_motor, 0.1 * MAX_SPEED);
   wb_motor_set_velocity(right_motor, 0.1 * MAX_SPEED);
   ```

Experiment with different values for `wb_motor_set_velocity`'s second argument to see how they affect the robot's speed. Try using different values in the two calls so that the two wheels do not rotate in unison. Be careful how you set these values—they could cause the robot to spin around in circles.

Additional Tutorials

Once you complete **Tutorial 1**, you may wish to continue with Webots **Tutorials 2–8**. **Tutorials 2–4** perform various modifications to the world you just created:

- In **Tutorial 2**, you'll **add a ball to your world**. You'll learn more about node types and how to configure physics options that **enable the ball to roll** in the simulation.
- In **Tutorial 3**, you'll learn how to improve your simulation's graphics with **lighting effects** and **textures**.

19. "Webots Reference Manual R2020b revision 2—Motor." Accessed December 13, 2020. `https://cyberbotics.com/doc/reference/motor#wb_motor_set_velocity`.

- In **Tutorial 4**, you'll create a more elaborate controller that enables the e-puck to use **its distance sensors to avoid the obstacles** you created previously.

Challenge: Once you complete **Tutorial 4**, try building a maze in your world and see if you can program the e-puck robot to traverse the maze. The following page lists several algorithms for finding a maze's exit:

```
https://en.wikipedia.org/wiki/Maze_solving_algorithm
```

Tutorials 5–7 dive into more advanced features:

- In **Tutorial 5**, you'll learn more about **physics** in Webots.
- In **Tutorial 6**, you'll work with a **four-wheeled robot** and learn more about **sensors**.
- In **Tutorial 7**, you'll **create your own PROTO node**.

In the advanced **Tutorial 8**, you'll learn about working with Webots nodes from the `webots_ros`. **ROS is the Robot Operating System**[20]—a framework for writing robot software. If you're going to do this tutorial, Webots recommends that you first learn more about ROS in the tutorials at

```
http://wiki.ros.org/ROS/Tutorials
```

Mastering Webots Tutorials 2–8 will be a "resume-worthy" accomplishment.

7.37 ***(Challenge Project: Webots Tortoise-and-Hare-Race Study)*** For this challenging exercise, we recommend that you first complete the Webots tutorials 2–7 mentioned at the end of Exercise 7.36. In Exercise 5.54, you simulated the Tortoise and the Hare race. Now that you're familiar with the Webots 3D robotics simulator, let your imagination run wild with Webots' amazing capabilities. Recreate the race using two robots from the dozens available in Webots. Consider using a small slow one for the tortoise (such as the e-puck from Exercise 7.36) and a larger fast one for the hare (such as the Boston Dynamics Spot[21,22] robot). Recall from the **Webots Guided Tour** that there are many other environments in which your robots may roam. Consider copying an existing terrain environment with objects like grass, flowers, trees and hills in which to race your robots.

20. "About ROS." Accessed December 13, 2020. `https://www.ros.org/about-ros/`.
21. "Webots User Guide—Boston Dynamics' Spot." Accessed December 31, 2020. `https://www.cyberbotics.com/doc/guide/spot`.
22. "Spot." Accessed December 31, 2020. `https://www.bostondynamics.com/spot`.