

# C: From Theory to Practice

Second Edition

George S. Tselikis  
Nikolaos D. Tselikas

CRC Press

CRC Press  
Taylor & Francis Group  
6000 Broken Sound Parkway NW, Suite 300  
Boca Raton, FL 33487-2742

© 2017 by Taylor & Francis Group, LLC  
CRC Press is an imprint of Taylor & Francis Group, an Informa business

International Standard Book Number-13: 978-1-138-63600-2 (Hardback)

**Visit the Taylor & Francis Web site at**  
**<http://www.taylorandfrancis.com>**

**and the CRC Press Web site at**  
**<http://www.crcpress.com>**

# *Contents*

Preface.....	xv
Acknowledgments .....	xvii
About the Authors .....	xxix
<b>1. Introduction to C.....</b>	<b>1</b>
History of C .....	1
The ANSI Standard .....	1
Advantages of C .....	2
Disadvantages of C.....	2
C Program Life Cycle .....	3
Writing a C Program .....	3
Our First C Program.....	3
The <code>#include</code> Directive.....	3
The <code>main()</code> Function .....	4
Comments .....	5
Compilation .....	6
Common Errors.....	7
Linking .....	7
Run the Program.....	8
<b>2. Data Types, Variables, and Data Output .....</b>	<b>9</b>
Variables .....	9
Naming Variables .....	9
Declaring Variables .....	10
Assignment of Values and Constants .....	13
Arithmetic Conversions .....	15
Type Qualifiers.....	18
The <code>#define</code> Directive.....	19
The <code>printf()</code> Function.....	20
Escape Sequences.....	20
Conversion Specifications .....	22
Return Value .....	23
Printing Variables .....	24
Optional Fields .....	25
Precision .....	25
Field Width.....	27
Prefix .....	27
Flags .....	28
Exercises.....	29
Unsolved Exercises .....	31

<b>3. Getting Input with <code>scanf()</code></b> .....	35
The <code>scanf()</code> Function.....	35
Use of Ordinary Characters .....	37
Return Value .....	38
Exercises .....	39
Unsolved Exercises .....	43
<b>4. Operators</b> .....	45
The <code>=</code> Assignment Operator.....	45
Arithmetic Operators .....	46
The <code>++</code> and <code>--</code> Operators.....	47
Relational Operators.....	49
Exercises .....	50
Compound Assignment Operators.....	50
Logical Operators.....	51
The <code>!</code> Operator .....	52
The <code>&amp;&amp;</code> Operator .....	52
The <code>  </code> Operator.....	53
Exercises .....	53
The Comma Operator .....	56
The <code>sizeof</code> Operator .....	57
The <code>enum</code> Type .....	59
Bitwise Operators .....	60
The <code>&amp;</code> Operator .....	61
The <code> </code> Operator .....	61
The <code>^</code> Operator .....	61
The <code>~</code> Operator .....	62
Shift Operators .....	62
Exercises .....	63
Operator Precedence .....	70
Unsolved Exercises .....	71
<b>5. Program Control</b> .....	73
The <code>if</code> Statement .....	73
Common Errors .....	74
The <code>if-else</code> Statement .....	75
Nested <code>if</code> Statements .....	76
Exercises .....	79
The Conditional Operator <code>?:</code> .....	88
Exercises .....	90
The <code>switch</code> Statement .....	92
<code>switch</code> versus <code>if</code> .....	96
Exercises .....	96
Unsolved Exercises .....	100
<b>6. Loops</b> .....	105
The <code>for</code> Statement .....	105
Omitting Expressions .....	108
Exercises .....	109

The <b>break</b> Statement .....	111
The <b>continue</b> Statement .....	113
Exercises .....	114
Nested Loops.....	124
Exercises .....	126
The <b>while</b> Statement .....	129
Exercises .....	132
The <b>do-while</b> Statement .....	140
Exercises .....	141
The <b>goto</b> Statement.....	144
Unsolved Exercises .....	145
<b>7. Arrays</b> .....	151
One-Dimensional Arrays.....	151
Declaring Arrays.....	151
Accessing Array Elements .....	152
Array Initialization .....	154
Exercises .....	155
Two-Dimensional Arrays.....	170
Declaring Arrays.....	170
Accessing Array Elements .....	171
Array Initialization .....	172
Exercises .....	174
Unsolved Exercises .....	184
<b>8. Pointers</b> .....	187
Pointers and Memory .....	187
Pointer Declaration .....	187
Pointer Initialization.....	189
Null Pointers.....	189
Using a Pointer .....	190
Exercises .....	192
The <b>void*</b> Pointer .....	197
Use of <b>const</b> Qualifier .....	198
Pointer Arithmetic .....	199
Pointers and Integers .....	199
Subtracting and Comparing Pointers .....	201
Exercises .....	201
Pointers and Arrays.....	203
Exercises .....	207
Array of Pointers .....	216
Exercises.....	218
Pointer to Pointer .....	219
Exercises.....	220
Pointers and Two-Dimensional Arrays .....	222
Exercises.....	225
Pointer to Function .....	228
Array of Pointers to Functions .....	231
Unsolved Exercises .....	235

<b>9. Characters.....</b>	239
The <code>char</code> Type .....	239
Exercises .....	242
The <code>getchar()</code> and <code>putchar()</code> Functions.....	245
Exercises .....	246
<b>10. Strings.....</b>	253
String Literals .....	253
Storing Strings.....	253
Exercise.....	254
Writing Strings .....	255
Exercises .....	257
Pointers and String Literals .....	258
Exercises.....	260
Read Strings.....	261
Exercises .....	264
String Functions.....	270
The <code>strlen()</code> Function.....	270
Exercises .....	270
The <code>strcpy()</code> and <code>strncpy()</code> Functions .....	278
Exercises .....	280
The <code>strcat()</code> Function.....	282
The <code>strcmp()</code> and <code>strncmp()</code> Functions.....	284
Exercises .....	285
Two-Dimensional Arrays and Strings .....	292
Exercises .....	293
Unsolved Exercises .....	295
<b>11. Functions .....</b>	299
Function Declaration.....	299
Return Value .....	299
Function Parameters.....	300
Function Definition .....	300
Complex Declarations .....	303
The <code>return</code> Statement.....	304
Function Call .....	305
Function Call without Parameters.....	305
Function Call with Parameters .....	307
Exercises .....	311
Storage Classes and Variables Scope.....	321
Local Variables.....	322
Global Variables.....	325
Static Variables.....	327
One-Dimensional Array as Argument .....	330
Exercises .....	336
Two-Dimensional Array as Argument .....	358
Exercises .....	359
Passing Data in <code>main()</code> Function .....	364
Exercises .....	365

Functions with Variable Number of Parameters.....	366
Exercise.....	368
Recursive Functions .....	369
Exercises.....	371
Unsolved Exercises .....	376
<b>12. Searching and Sorting Arrays.....</b>	<b>381</b>
Searching Arrays .....	381
Linear Search Algorithm .....	381
Exercises .....	381
Binary Search Algorithm .....	383
Exercises .....	384
Sorting Arrays .....	386
Selection Sort Algorithm.....	386
Exercises .....	387
Insertion Sort Algorithm .....	392
Exercise.....	393
Bubble Sort Algorithm .....	395
Exercises .....	395
Quick Sort Algorithm.....	398
The <b>bsearch()</b> and <b>qsort()</b> Library Functions .....	400
Exercise.....	401
Unsolved Exercises .....	402
<b>13. Structures and Unions.....</b>	<b>405</b>
Structures.....	405
Declare a Structure .....	405
The <b>typedef</b> Specifier.....	408
Initializing a Structure .....	410
Pointer to a Structure Member.....	412
Structure Operations .....	413
Structure Containing Arrays.....	414
Structure Containing Pointers .....	415
Nested Structures.....	415
Bit Fields .....	417
Pointer to Structure.....	419
Array of Structures .....	420
Exercises .....	422
Structure as Function Argument.....	424
Exercises .....	426
Unions.....	437
Union Declaration.....	437
Access Union Members .....	438
Exercise.....	443
Unsolved Exercises .....	445
<b>14. Memory Management and Data Structures.....</b>	<b>449</b>
Memory Blocks.....	449
Static Memory Allocation .....	450

Dynamic Memory Allocation .....	451
The <b>malloc()</b> Function.....	452
The <b>free()</b> Function .....	455
Memory Management Functions .....	458
Exercises .....	460
Dynamic Data Structures .....	471
Linked List.....	471
Insert a Node .....	472
Delete a Node .....	472
Implementation Examples.....	473
Implementing a Stack.....	473
Exercise.....	473
Implementing a Queue.....	478
Exercise.....	478
Implementing a Linked List.....	481
Exercises .....	481
Binary Search Tree.....	488
Exercises .....	495
Hashing .....	497
Exercise.....	500
Unsolved Exercises .....	501
 15. Files.....	507
Binary Files and Text Files.....	507
Open a File.....	508
Close a File .....	510
Process a File.....	511
Write Data in a Text File.....	511
The <b>fputs()</b> Function .....	511
The <b>fprintf()</b> Function.....	512
Exercise.....	513
The <b>fputc()</b> Function.....	514
Exercises .....	515
Read Data from a Text File .....	518
The <b>fscanf()</b> Function.....	518
Exercises.....	521
The <b>fgets()</b> Function .....	523
Exercise.....	525
The <b>fgetc()</b> Function .....	526
Exercises .....	526
End of File .....	530
The <b>fseek()</b> and <b>ftell()</b> Functions.....	531
Write and Read Data from a Binary File .....	532
The <b>fwrite()</b> Function.....	532
The <b>fread()</b> Function .....	533
Exercises .....	534
The <b>feof()</b> Function .....	544
Exercise.....	545
Unsolved Exercises .....	545

<b>16. The Preprocessor .....</b>	549
Simple Macros .....	549
Macros with Parameters .....	554
# and ## Preprocessor Operators.....	556
Preprocessor Directives and Conditional Compilation .....	557
The <code>#if</code> , <code>#else</code> , <code>#elif</code> , and <code>#endif</code> Directives.....	558
The <code>#ifdef</code> , <code>#ifndef</code> , and <code>#undef</code> Directives.....	560
The <code>defined</code> Operator .....	561
Miscellaneous Directives.....	562
Exercises .....	563
Unsolved Exercises .....	567
<b>17. Building Large Programs.....</b>	571
<b>18. Introduction to C++.....</b>	579
Classes and Objects .....	579
Constructors and Destructors.....	582
Overloading of Functions and Operators .....	584
Inheritance .....	589
Polymorphism .....	594
Templates and Standard Template Library .....	596
Function Templates.....	596
Class Templates .....	599
Standard Template Library.....	600
More Features .....	602
The <code>new</code> and <code>delete</code> Operators .....	602
Reference Variables.....	604
Exceptions .....	606
<b>19. Introduction to Java.....</b>	611
Classes and Objects .....	611
Packages .....	614
Constructors .....	614
Access Modifiers and Nonaccess Modifiers .....	616
Inheritance .....	620
Polymorphism .....	623
Method Overloading .....	624
Method Overriding.....	625
Interfaces .....	626
<b>20. Review Exercises.....</b>	631
<b>Appendix 1: Precedence Table.....</b>	663
<b>Appendix 2: ASCII Table .....</b>	665
<b>Appendix 3: Library Functions .....</b>	667
<b>Appendix 4: Hexadecimal System.....</b>	677
<b>Index .....</b>	679

# Preface

This book is primarily addressed to students who are taking a course on the C language, to those who desire to pursue self-study of the C language, as well as to programmers already familiar with C who want to test their knowledge and skills. It could also prove useful to instructors of a C course who are looking for explanatory programming examples to add in their lectures.

So what exactly differentiates this book from the others in the field? This book tests the skills of both beginners and more experienced developers by providing an easy-to-read compilation of the C theory enriched with tips and advice as well as a large number of difficulty-scaled solved programming exercises.

When we first encountered the C language as students, we needed a book that would introduce us quickly to the secrets of the C language as well as provide in-depth knowledge on the subject—a book with a focus on providing inside information and programming knowledge through practical examples and meaningful advice. That is the spirit that this book aims to capture.

The programming examples are short but concrete, providing programming know-how in a substantial manner. Rest assured that if you are able to understand the examples and solve the exercises, you can safely go on to edit longer programs and start your programming career successfully. The source code of the exercises is available at [www.c4all.gr](http://www.c4all.gr).

For all of you who intend to deal with computer programming, a little bit of advice coming from our long experience may come in handy:

- Programming is a difficult task that, among other things, requires a calm mind, a clear development plan, a methodical approach, patience, self-confidence, and luck as well.
- When coding, try to write in a simple and comprehensive way for your own benefit and for those who are going to read your code. Always remember that the debug, maintenance, and upgrade of a code written in a complex way are a painful process.
- Hands-on! To learn the features of any programming language, you must write your own programs and experiment with them.
- Programming is definitely a creative activity, but do not forget that there are plenty of creative, pleasant, and less stressful activities in life. Do not waste your life in front of a computer screen, searching for *losing* pointers, buffer overflows, memory overruns, and buggy conditions. Program for some time and then have some fun. Keep living and do not keep programming.

Enjoy the C flight; it will be safe, with some turbulence, though.

# Introduction to C

Before getting into the details of C language, this chapter presents, in brief, its history, evolution, strengths, and weaknesses. Then, we'll discuss some basic concepts that we'll need in order to write our first program.

## History of C

The C language was developed at Bell Labs in the early 1970s by Dennis Richie and others. At that time, the *Unix* operating system, also developed at Bell Labs, was written in assembly language. Programs written in assembly are usually hard to debug, maintain, and enhance, and *Unix* was no exception. Richie decided to rewrite *Unix* code in another language that would make the execution of these tasks easier. He named the language C because it was the evolution of an earlier language written by Ken Thompson, called B. The C language continued to evolve during the 1970s, and since then, it has been used by thousands of programmers for a wide variety of applications.

## The ANSI Standard

The rapid expansion of the C language and its increased popularity led many companies to develop their own C compilers. Due to the absence of an official standard, their development relied on the bible of C programmers, the legendary K&R book, written by Brian Kernighan and Dennis Ritchie in 1978.

However, the K&R book was not written in the precise way that a standard requires. Features that were not clearly described could be implemented in different ways. As a result, the same program could be compiled with one C compiler and not with another. In parallel, the C language continued to evolve with the addition of new features and the replacement or obsolescence of existing ones.

The need for the standardization of C language became apparent. In 1983, the American National Standard Institute (ANSI) began the development of the C standard that was completed and formally approved, in 1989, as ANSI C or Standard C. In 1990, the ANSI C standard was adopted by the International Standards Organization (ISO) as ISO/IEC 9899:1990. This standard describes precisely the features, characteristics, and properties of the C language, and every C compiler must support it.

The addition of some new features to the ANSI standard during the late 1990s led to a new standard called C99. C99 was further enriched with new features, leading to the publication of the C11 standard. This book describes the C language based on the ANSI/ISO C standard.

## Advantages of C

Despite the emergence of many programming languages, C still remains competitive and popular in the programming world for several reasons, such as the following:

1. It is a flexible language, which can be used for the development of different kinds of applications, from embedded systems and operating systems to industrial applications. For example, we've used C in the area of communication networks for the development of network protocols and the support of network services.
2. A C program is executed very quickly.
3. It is a small language. Its vocabulary consists of a few words with special meanings.
4. It is portable, meaning that a C program may run under different operating systems.
5. It supports structural programming, meaning that a C program may contain functions to perform several tasks.
6. It is a language very close to the hardware and it can be used for systems programming.
7. Every C compiler comes with a set of ready-to-use functions, called C standard library. The use of these library functions saves considerable programming effort.
8. Thanks to the popularity of the C language, there are many C compilers available; some of them free of charge.
9. Learning C will help you to learn other languages. For example, getting familiar with C is the first step toward object-oriented programming. Most of the C features are supported in several object-oriented languages, like C++, Java, and C#.

## Disadvantages of C

1. Because the C language does not impose many restrictions on the use of its features, it is an error-prone language. When writing a C program, be cautious because you may insert bugs that won't be detected by the compiler.
2. C supports only single-thread flow; it does not support multithread programming, parallel operation, and synchronization.
3. Although C is a small language, it is not an "easy-to-use" language. C code can be very hard to understand even if it consists of a small number of lines. After reading this book, check out the International Obfuscated C Code Contest (<http://www.ioccc.org>) to get a feeling.
4. C is not an object-oriented language; therefore, it does not support object-oriented features.

## C Program Life Cycle

The life cycle of a C program involves several steps: write the code, compile the code, link the object code produced by the compiler with any code needed to produce the executable file, and run the program. Usually, a C compiler provides an integrated development environment (IDE) that allows us to perform this set of operations without leaving the environment.

## Writing a C Program

To write a C program, we can use any available text editor. An editor is often integrated with the compiler. The source code must be saved in a file with extension .c. When the size of the code is very large, it is a common practice to divide the code into several files, in order to facilitate tasks like testing and maintenance. In such cases, each file is compiled separately.

## Our First C Program

Our first program will be a “rock” version of the program that most programmers begin with. Instead of the classical “Hello world” message, our first program displays on the screen the classical song of *Ramones*, *Hey Ho, Let's Go*. Play it loud, and let's go ...

```
#include <stdio.h>
int main(void)
{
    printf("Hey Ho, Let's Go\n");
    return 0;
}
```

The following sections explain the significance of each program line.

### The `#include` Directive

The `#include` directive instructs the compiler to include the contents of the `stdio.h` file into the program before it is compiled. Regarding syntax, directives always begin with a `#` character and do not end with a semicolon ; or some other special marker.

The `stdio.h` file and other files contain information about the standard library functions. The typical extension is .h and they are called header files. These files are supplied with the compiler. For example, as shown in Appendix C, the `stdio.h` (standard input output) file contains declarations of data input and output functions, and because of that, it is almost always included in a program. An included file may also contain `#include` directives and include other files. In general, when you are using a standard library function,

you must include the header file that contains its declaration. The order of their inclusion does not matter. When you get familiar with the C language, you may edit your own header files and include them in your programs.

When the program is compiled, the compiler searches for the included files. The searching rules depend on the implementation. Typically, if the file name is enclosed in brackets <>, the compiler searches in system dependent predefined directories, where system files reside. If it is enclosed in double quotes "", the compiler usually begins with the directory of the source file, then searches the predefined directories. If the file is not found, the compiler will produce an error message and the compilation fails. The file name may include relative or full path information. For example:

```
#include <d:\projects\serial.h> /* DOS/Windows path. */
#include </port/serial.h> /* Linux path. */
#include "../projects\test.h" /* Relative path. */
```

However, it'd be better to avoid including path or drive information, because if your program is transferred to another system its compilation might fail.

## The `main()` Function

Every C program must contain a function named `main()`. Simply put, a function is a series of declarations and statements that have been grouped together and given a name. The code of the program, or else the body of the function, must be enclosed in braces {}. A statement is a command that will be executed when the program runs. Statements are typically written in separate lines, and, almost always, each statement ends with a semicolon. Although the compiler does not care about the layout of the program, proper indentation and spacing make your program easier to read. Braces are used to group declarations and statements into a block or else a compound statement that the compiler treats as one. Besides functions, we'll use braces in control statements and loops.

The `main()` function is called *automatically* when the program runs. The execution of the program ends when the last statement of `main()` is executed, unless an exit statement such as `return` is called earlier. The keyword `int` indicates that `main()` must return an integer to the operating system when it terminates. This value is returned with the `return` statement; the value 0 indicates normal termination. This declaration of `main()` is fairly common. However, you may see other declarations such as:

```
void main()
```

As we'll see in Chapter 11, the keyword `void` indicates that `main()` does not return any value. Although a C compiler may accept this declaration, it is illegal according to the C standard because `main()` must return a value.

Many programmers often omit the word `void` and write `int main()`. It is a habit that might originate either from an older version of C or because of their involvement with C++, where there is no need to use the word `void`. However, if we want to follow the ANSI C standard, we must use the word `void`. The reason is that a pair of empty parentheses indicates that the function takes an unknown number of parameters, not any. In Chapter 11, we'll see another declaration of `main()`, where `main()` takes parameters.

Finally, because the return type is `int` by default, it is allowed to omit the word `int` and write `main(void)`.

As said, we may use functions of the standard library in our programs. For example, `printf()` is a standard library function that is used for data output. The C standard defines the characteristics of the library functions. Appendix C provides a short description. Since `printf()` is declared in `stdio.h`, we have to include that file in our program. As we'll see in Chapter 2, the new line character `\n` moves the cursor to the beginning of the next line. Functions will be discussed in Chapter 11. Until then, it is enough to know that a function is called by using its name; it performs a task and may optionally return a value.

## Comments

Adding comments improves the readability of the program and makes it easier to understand. A comment begins with the `/*` symbol and ends with `*/`. Comments can extend in more than one line. The compiler ignores anything included between the `/* */` symbols, meaning that the comments do not affect the operation of the program. For example, a comment is added to describe the purpose of the program:

```
#include <stdio.h>
/* This program calls printf() to display a message on the screen. */
int main(void)
{
    printf("Hey Ho, Let's Go\n");
    return 0;
}
```

Nested comments are not allowed. For example, the following code is illegal, and the compiler will raise an error message:

```
/*
/* Another comment. */
*/
```

Adding comments is a must when writing programs. An explanatory program saves you time and effort when you need to modify it, and the time of other people who may need to understand and evolve your program.

Based on what you've learned so far, can you tell us what does the following program output? Consider the spaces inside each `printf()`.

```
#include <stdio.h>
int main(void)
{
    printf("That is "); /* A program with multiple printf()
    printf("the first");
    printf(" trap"); /* That's the last one. */
    return 0;
}
```

Have you checked the spaces and answered That is the first trap? Too easy to be the right answer, sorry to disappoint you. Check the program again; where does the first comment end? It ends at the third line, where the \*/ is encountered. Therefore, the program outputs: That is.

---

Be careful when adding comments, because the compiler ignores anything between them.

---

*And yes, that is the first trap. The book contains several traps; we'll ask you questions, give you misleading hints, and wait for you to fall into. Watch out!*

Notice that you may see C programs containing one line comments that begin with // instead of including them within /\* \*/. For example:

```
int main(void) // This is my first C program
```

Although a compiler may support this syntax, it is not according to the C standard. In fact, using // to add a comment is a C++ practice and of the next C versions that may be supported by the compiler you are using, but not by others. Therefore, if you care about portability always use /\* \*/.

---

## Compilation

After writing the program, the next step is to compile it. The compiler translates the program to a form that the machine can interpret and execute. Many companies (e.g., Microsoft and Borland) have developed C compilers for Windows, while one of the most popular free compilers for Unix/Linux systems is gcc (GNU Compiler Collection, <http://gcc.gnu.org>). For example, suppose that the code of the program is saved in a file named *first.c*. In order to compile it with the *gcc* compiler, we write in the command line:

```
$ gcc first.c
```

Many compilers provide an IDE together with an editor and other facilities for the development of C programs. An IDE typically provides a menu command such as *File->New->Source File* to write the program and a command like *Build->Compile* to compile it.

When the program is compiled, the compiler checks if the syntax of the program is according to the language rules. If the compilation fails, the compiler informs the programmer of the fail reason(s). If the program is compiled successfully, the compiler produces an object file that contains the source code translated in machine language. By default, it has the same name as the file that contains the source code and its extension is .obj or .o. For example, if the code is saved in the file *first.c*, the name of the object file would be *first.obj* (Windows) or *first.o* (Unix/Linux).

## Common Errors

The most common errors are syntactic. For example, if you don't add the semicolon at the end of `printf()` or you omit a parenthesis or a double quote, the compilation would fail and the compiler would display error messages.

Spelling errors are very common, as well. For example, if you write `studio.h` instead of `stdio.h` or `print` instead of `printf`, the compilation would fail. C is a case-sensitive language, meaning that it distinguishes between uppercase and lowercase letters. For example, if you write `Print f` instead of `printf`, the compilation would fail.

If the compiler displays many errors, fix the first one and recompile the program. The new compilation may display fewer errors, even none. Also, notice that an error detected by the compiler may not occur in the indicated line, but in some previous line.

It is very important to understand that the compiler detects errors due to the wrong use of the language and not logical errors that may exist within your program. The compiler is not "inside your head" to know what you intend to do. Therefore, a successful compilation does not mean that your program would operate as you expect. For example, if you want to write a program that displays the word `One` if the value of the integer variable `a` is greater than 5, and you write:

```
if(a < 5)
    printf("One");
```

then, although these lines will be compiled successfully, the program won't behave as you desire. This type of error is a logical error (*bug*) not detected by the compiler. The use of the word *bug* as a synonym of a programming error is credited to the mathematician and computer scientist Grace Hopper when she discovered that a hidden bug inside her computer caused its abnormal operation.

Apart from the error messages, the compiler may display warning messages. If there are only warning messages, the program will be compiled. However, don't ignore them; it may warn you against the potential malfunction of your program. As a matter of fact, it'd be safer to set the warning level at the maximum level supported by your compiler.

---

## Linking

In the final step, the linker links the object code produced in the previous step with the code of the library functions that the program uses (e.g., `printf()`) and other necessary information. This code is contained in library files that are supplied with the compiler. To make it clear, both header files and library files are supplied with the compiler; however, the code of the library functions resides in library files, not in header files. For example, if you are using the `gcc` compiler and your program uses a math function, you might need to load the math library by writing: `gcc test.c -lm`. In an integrated environment, a menu command with the name *Build* is typically found, which performs program compilation and linking in one step.

If the linking is successful, an executable file is created. For example, the default name of the executable file produced by the *gcc* compiler is *a.out*, while a *Microsoft* compiler produces an executable file having the same name with the source file and extension *.exe*.

---

## Run the Program

There are several ways to run a program. For example, if you are using the *gcc* compiler, write in a command line:

```
$ a.out
```

If an error message like No such file or directory or command not found is displayed, write:

```
./a.out
```

In an IDE, a menu command like Build->Execute is typically found.

If the program does not operate as you expect, you are in trouble. The depth you are in depends on the size of the source code. If it extends to some hundreds of lines, you'll probably find the logical errors in a short time and the debugging procedure won't take long. But if your program consists of thousands of lines, the debugging may become a time-consuming, painful, and very stressful procedure, particularly if your supervisor hangs over your head demanding immediate results. To avoid such troublesome situations, remember our advice: try to write simple, clear, readable, and maintainable code.

*Before continuing to the next chapter, let's give you some advice on how to read the book's exercises. A program may be written in several ways. Our advice is to hide the answer and write your own version. Then, compare the solutions. Pay extra attention to all questions of type "What is the output of the following program?" Don't rush to see the answer; hide it and give it some time. Each chapter ends with a number of unsolved exercises for more practice. Always try them to test your understanding and skills, before continuing to the next chapter. Remember, to learn any programming language, you must write your own programs and experiment with them, hands on; that is the only way. Any personal references refer to the first author. Oh yes, almost forgot, as you "heard" in our first program, you'll need a pair of speakers...*

# Data Types, Variables, and Data Output

In order to be able to write programs that actually perform useful tasks that save us time and work, this chapter will teach you how to use data types and variables. We'll also go into more detail on the most important function for outputting data, the `printf()` function.

## Variables

The computer's RAM (random access memory) consists of millions of successive memory cells. The size of each cell is one *byte*. For example, an old PC with only 16 MB (megabytes) of RAM consists of  $16 \times 1.024 = 16.384$  kB (kilobytes), or  $16.384 \times 1.024 = 16.777.216$  memory cells. A newer PC with say 8 GB (gigabytes) of RAM would have  $8 \times 1024$  MB =  $8192 \times 1024$  KB =  $8.388.608 \times 1.024 = 8.589.934.592$  memory cells.

A *variable* in C is a memory location with a given name. The value of a variable is the content of its memory location. A program may use the name of a variable to access its value.

## Naming Variables

There are some basic rules for naming variables. These rules also apply for function names. Be sure to follow them or your code won't compile:

1. The name can contain letters, digits, and *underscore characters* \_ .
2. The name must begin with either a letter or the underscore character.
3. C is *case sensitive*, meaning that it distinguishes between uppercase and lowercase letters. For example, the variable `var` is different from the variables `Var` or `vAr`. The standard specifies that at least the first 31 characters are significant. For function names, that number may be less than 31. For names with external linkage (we'll talk about it in Chapter 11), the standard guarantees uniqueness only for the first six characters and the compiler may ignore case distinctions. Of course, a compiler is allowed to make more characters significant.

4. The following keywords cannot be used as variable names because they have special significance to the C compiler.

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Some compilers also treat the words `asm`, `far`, and `near` as keywords.

In short, here are some more rules to follow when choosing names. Don't choose names that begin with the `_` character and a capital letter or two `_` characters, because they are reserved for use in the standard library. In general, to avoid conflict with names used in the library, it'd be better not to choose names that begin with the `_` character. Also, don't use names of predefined macros, and names of variables or functions that are used in the library, e.g., `printf`.

In addition to the rules given earlier, there are a few conventions that are good to follow when naming your variables. While these are not enforced by the C compiler, these "rules of thumb" will tend to make your programs easier for you to understand, as well as for those who have to read your code.

Use descriptive names for variables. It is much easier to read a program when the names of the variables indicate their intended use. For example, if you use a variable that holds the sum of some even numbers, give it a name like `sum_even` rather than an arbitrary name like `i`. Typically, short names such as `i` are used as indexes in arrays or loops.

When necessary, don't be afraid to use long names to describe the role of a variable. If a variable name is several words long, a typical practice is to separate each word with the underscore character `_` for better readability. For example, you might call a variable that holds the number of books `books_number`, instead of `booksnnumber`, or something less readable.

Typically, most C programmers use lowercase letters when naming variables and uppercase letters when defining macros and constants. This is a convention, not a requirement.

---

## Declaring Variables

Variables must be declared before being used in the program, and before any executable statements. To declare a variable, we write:

```
data_type name_of_variable;
```

The `name_of_variable` is the variable's name. The `data_type` should be one of the C supported data types. Each data type specifies the range of values that may be stored in a variable of that type. The actual size of the types may vary from one system to another. Table 2.1 shows the usual ranges on a 32-bit system.

TABLE 2.1

## C Data Types

Data Type	Usual Size (bytes)	Range of Values (min–max)	Precision Digits
<code>char</code>	1	-128...127	
<code>short int</code>	2	-32.768...32.767	
<code>int</code>	4	-2.147.483.648...2.147.483.647	
<code>long int</code>	4	-2.147.483.648...2.147.483.647	
<code>float</code>	4	Lowest positive value: $1.17 \times 10^{-38}$ Highest positive value: $3.4 \times 10^{38}$	6
<code>double</code>	8	Lowest positive value: $2.2 \times 10^{-308}$ Highest positive value: $1.8 \times 10^{308}$	15
<code>long double</code>	8, 10, 12, 16		
<code>unsigned char</code>	1	0...255	
<code>unsigned short int</code>	2	0...65535	
<code>unsigned int</code>	4	0...4.294.967.295	
<code>unsigned long int</code>	4	0...4.294.967.295	

For example, to declare an integer variable named `a`, and a floating point variable named `b`, we write:

```
int a;
float b;
```

Variables of the same type can be declared in the same line, separated with a comma. For example, instead of declaring the variables `a`, `b`, and `c` in three different lines, like so:

```
int a;
int b;
int c;
```

we can declare them in the same line: `int a, b, c;`

The word `int` may be omitted for the integer types, for example, `short` instead of `short int`. The words can be combined in any order. For example, `unsigned long int a;` is the same as `int long unsigned a;`. Once a variable is declared, the compiler reserves the bytes in memory that it needs in order to store its value. Also, it saves the name of the variable and its memory address; thus, whenever a variable is used within the program, the compiler uses the name to retrieve the corresponding address and access the content.

As indicated in the second column of Table 2.1, each type requires specific memory space. For example, the `char` type requires one byte, the `float` and `int` types require four bytes, the `double` type requires eight bytes, and so on.

---

The memory space that a data type reserves may vary from one system to another. For example, the `int` type may reserve two bytes in one system and four bytes in another. To compute the number of bytes a data type reserves on a particular system, use the `sizeof` operator, discussed in Chapter 4.

---

The `char`, `short`, `int`, and `long` types are used to store integer values, which can be either signed or unsigned. If an integer variable is declared as `unsigned`, it has no sign bit and it may store only positive values or zero. The C standard does not specify whether

the `char` type is signed or unsigned. It depends on the compiler; thus, if it is significant for your program, you should explicitly write `signed char` or `unsigned char` instead of `char`. Printable characters are always positive. The other types are signed by default. In signed types, the leftmost bit is reserved for the sign; it is 1 if the number is negative, 0 otherwise. The advantage of using unsigned types is that they have a higher upper limit than their signed counterparts, since they don't need to account for negative values.

The `float`, `double`, and `long double` types are used to store values with a fractional part, that is, floating-point numbers. Unlike integer types, floating-point types are always signed. The C standard does not specify the number of the precision digits since different computers may use different formats to store floating-point numbers. Typically, the default precision of the `float` type is 6 digits after the decimal point, while the precision of the `double` type is 15 digits. The range of the `long double` type isn't shown in Table 2.1 because its length varies, with 10 and 16 bytes being the most common sizes. It may also have the same size as of `double`. Although the `long double` type supports the highest precision, it is rarely used because the precision of the `float` and `double` types is usually sufficient.

Although C types may come in different sizes, the C standard requires that the `long` type must not be smaller than `int`, which must not be smaller than `short`. Similarly, `long double` must not be smaller than `double`, which must not be smaller than `float`. The size of the `short` and `int` types must be 16 bits at least, while the `long` type must be 32 bits at least. If the `short` type meets your needs and the available memory is limited, it'd be better to use it instead of the `int` type, because its size is usually smaller. Information about the integer and floating types in the local implementation is provided in the `limits.h` and `float.h` files, respectively.

*For simplicity, we've made some assumptions throughout this book. We assume that the type characteristics are those shown in Table 2.1, that one byte is eight bits wide, and the system uses the most common method to represent signed numbers, that is, the two's complement. Also, we assume that the underlying character set is the American Standard Code for Information Interchange (ASCII) set.*

---

If you don't care about precision, use the `float` type, because it usually reserves fewer bytes and calculations with `float` numbers tend to be executed faster.

Consider the following program. Although we haven't discussed yet about `if` and `printf()`, we think that you can get the idea. The `==` operator compares the values. So, tell us, what does the following program output?

```
#include <stdio.h>
int main(void)
{
    float a = 3.1;

    if(a == 3.1)
        printf("Yes\n");
    else
        printf("No\n");

    printf("%.9f\n", a-3.1);
    return 0;
}
```

Although the obvious answer is Yes, did the program output No and a nonzero value? Surprise!!!

Remember, when using floating-point variables in comparisons, it is safer to declare them as `double` or `long double` rather than `float`. For example, if `a` had been declared as `double`, the program would most probably have displayed Yes and 0. However, when you test a floating-point value for equality, you can never be sure, due to a potential rounding error. Therefore, if you have to test two floating-point values for equality, don't write `if(a == b)`; it is not safe. You may insert a hard-to-trace bug. A simple, although safer, approach is to check not whether the values are exactly the same but whether their difference is very small. For example, we could write `if(fabs(a-b) <= epsilon)` and choose a value for `epsilon` to specify the acceptable error margin. Notice that there are more efficient solutions, but it is out of scope to give you more details. The `fabs()` function is described in Appendix C.

*When using floating-point values in our programs, we'll use both `float` and `double` variables, in order to get familiar with both types.*

Next, we'll see that we can specify more properties when declaring a variable: the type qualifier and the storage class. Although they may appear in any order, the typical practice is to specify the storage class first, then the qualifier, and last the data type, for example, `static const unsigned int a;`

---

## Assignment of Values and Constants

Until we reach Chapter 11, we are going to declare all our variables within `main()`. Those variables are called *automatic*, and if they are not explicitly initialized, the initial value is undefined (i.e., garbage). For example, the following program uses `printf()`; you'll see how in a short, to output the initial value of `i`:

```
#include <stdio.h>
int main(void)
{
    int i;

    printf("%d\n", i);
    return 0;
}
```

As a matter of style, we use a blank line to separate the declarations of variables with the statements. In general, indentation, alignment, spaces, and blank lines improve the readability of the program. For example, we always indent declarations and statements to make it clear that they are nested inside the function. Also, we use blank lines to divide the program into logical blocks, making it easier for the reader to understand the structure of the program.

A variable can be given a value by using the assignment operator `=`. For example, the following code assigns the value 100 to `a`.

```
int a;
a = 100;
```

Alternatively, a variable can be initialized together with its declaration:

```
int a = 100;
```

We can also initialize more than one variable of the same type when declared. For example, the following statement declares the `a`, `b`, `c`, and `d` variables and initializes the first three with the values 100, 200, and 300, respectively. The variable `d` is not initialized.

```
int a = 100, b = 200, d, c = 300;
```

We could even write:

```
int a = 100, b = a+100, d, c = b+100;
```

The assignments take place from left to right, meaning that first `a` becomes 100, then `b` becomes 200, and then `c` becomes 300. For better readability, our preference is to initialize the variables in separate statements right after their declarations or just before being used within the program. In some examples, we are not consistent; the reason is to save space.

The integer numbers we used in our example (e.g., 100) are *decimal* constants. A decimal constant contains the digits of the decimal system 0-9 but must not begin with 0. An *octal* constant must begin with 0 and contains digits 0-7. For example, the following statement does not assign the value 234 to `a`, but 156, because the number 0234 represents an octal constant.

```
int a = 0234;
```

Since each digit corresponds to a power of 8,  $234_8 = 4 \times 8^0 + 3 \times 8^1 + 2 \times 8^2 = 156_{10}$ .

*Hexadecimal* constants contain digits 0-9 and hex letters and must begin with 0x or 0X. For example, the following statement assigns the value 31 to `a`, since  $1f_{16} = 31_{10}$ .

```
int a = 0x1f;
```

The letters may be either uppercase or lowercase. Appendix D provides a brief introduction to the hex system. Typically, the hex notation is used in applications that communicate with the hardware.

Just remember, an integer constant may be written in decimal, octal, or hex notation; they are nothing more than alternative ways of writing the same number. Whether you write the value eleven as 11, 013, or 0xb, it is stored the same way in the computer, as a binary (base 2) value. Also notice that it is allowed to mix notations in an expression. For example, the following statement assigns the value 180 to `a`.

```
int a = 100+0100+0x10;
```

The type of an unsuffixed decimal constant is the first of the `int`, `long int`, and `unsigned long int` types (plus the `unsigned int` type if it is octal or hexadecimal) that can represent its value. If it is suffixed by l or L (e.g., 5L, 0x123L), the compiler is forced to treat it as `long int` or `unsigned long int`, while if it ends with u or U (e.g., 100U), its type is `unsigned int` or `unsigned long int`. Their combination indicates that the constant is `unsigned long int` (e.g., 100UL). The order of the L and U letters does not matter, nor does their case.

A character constant is a character or a series of characters enclosed in single quotes. The type of the constant is **int**. As we'll see in Chapter 9, the value of a character constant depends on the character set that the system supports. For example, if the ASCII set is used, the value of 'a' is 97. The value of a multicharacter constant is implementation dependent.

The value assigned to a variable should be within the range of its type. For example, the statement:

```
unsigned char ch = 258;
```

does not make ch equal to 258 (100000010 in binary) because the maximum value that can be stored in an **unsigned char** variable is 255. Only the 8 lower bits will be stored, that is, 00000010, and ch becomes 2. This type of bug is very common, so always pay attention to the ranges of your variables!

Another point to pay attention to is that when assigning a floating-point value, we use dot (.) for the fractional part, and not comma (,). For example, we write 1.24, not 1,24.

```
double a = 1.24;
```

A floating constant can be written in scientific notation using the letter E or e and an exponent that indicates the power of 10. For example, we can write a = 124E-2; which is equivalent to  $124 \times 10^{-2}$  or a = 0.124e1;. Scientific notation is usually used when the number is very small or very large to make it easier for the programmer to read and write. By default, the compiler treats a floating constant as **double**. Because of that, if we declare a as **float** in our example, the compiler would most probably issue a warning message such as: “*‘initializing’ : truncation from ‘const double’ to ‘float’*”. To make the compiler store the constant as **float**, add the letter f or F at the end (e.g., 1.24f), or the letter l or L to store it as **long double** (e.g., 1.24L).

As we'll see in Chapter 4, when both operands are integers, the division operator / cuts off the fractional part and produces an integer result. For example:

```
int i = 6;
double j = i/4;
```

The result of i/4 is integer because both operands are integers. Therefore, the value assigned to j is 1, regardless of the type of j. If either of the operands is floating point, the fractional part is not truncated. For example, if i is declared as floating-point variable, j will be 1.5. The same happens, if we write j = i/4.0;, since the divisor is a floating-point constant.

---

## Arithmetic Conversions

C allows different data types to be mixed in the same expression. Therefore, the compiler should follow some rules, in order to perform type conversions if necessary. The arithmetic conversions are applied to binary operators that expect operands of arithmetic type, including arithmetic, relational, and equality operators. The conversion of one type to another is done either *implicitly* or *explicitly* using the cast operator.

When *implicit* conversion is performed, the general rule is that the compiler, without the programmer's intervention, promotes the type of the operand with the "narrower" type into the "wider" type capable of storing its value. The result is of the higher type. We examine two cases:

1. If either operand has a floating type. In this case, the operand whose type is narrower is promoted in that order: **float->double->long double**. That is, if one operand is **long double**, the other is converted to **long double**. Otherwise, if one operand is **double**, the other is converted to **double**. Otherwise, if one operand is **float**, the other is converted to **float**. Notice that these rules also apply in case of an integer operand. That is, if one operand is **float** and the other is **int**, the **int** operand is converted to **float**.
2. If neither operand has a floating type. First, integral promotions are performed in both operands. The integral promotions convert the **char** and **short** types to **int** or to **unsigned int** if the **int** type cannot represent all the values of the original type. Then, the operand whose type is narrower is promoted in that order: **int->unsigned int->long int->unsigned long int**. That is, if one operand is **unsigned long int**, the other is converted to **unsigned long int**. Otherwise, if one operand is **long int**, the other is converted to **long int**, unless its type is **unsigned int**. In this case, if the **long** type can represent all values of the **unsigned int** type, the **unsigned int** operand is converted to **long int**. If not, both operands are converted to **unsigned long int**. Otherwise, if one operand is **unsigned int**, the other is converted to **unsigned int**. Otherwise, both operands have type **int**.

Let's see some conversion examples:

```
char c;
short s;
int i;
unsigned int u;
float f;
double d;
long double ld;

i = i+c; /* c is converted to int. */
i = i+s; /* s is converted to int. */
u = u+i; /* i is converted to unsigned int. */
f = f+u; /* u is converted to float. */
f = f+d; /* d is converted to float. */
d = d+f; /* f is converted to double. */
ld = ld+d; /* d is converted to long double. */
```

In practice, when signed and unsigned types are mixed, the conversion rules may produce unexpected results. For example, apply the right conversion rule and tell us what does the following program output?

```
#include <stdio.h>
int main(void)
{
    int a = -20;
```

```
unsigned int b = 200;

if(a < b)
    printf("Yes\n");
else
    printf("No\n");
return 0;
}
```

Although we'd expect the program to output Yes, it outputs No. Here is why. Since a and b operands have different types, the conversion rule says that the type of a is promoted to `unsigned int` before the comparison. Therefore, the value -20 will be converted to a large positive number greater than b (in a 32-bit system, it is  $2^{32} - 20$ ). As a result, the program outputs No. Be very careful when mixing signed and unsigned operands or, even better, don't mix them at all, in order to avoid unpleasant surprises like this one.

Implicit arithmetic conversions may also occur in assignments when the type of the expression on the right side does not match the type of the variable on the left side. The applied rule says that the type of the expression is converted to the type of the variable. If the type of the variable is at least as wide as the type of the expression, there will be no loss of information. For example:

```
char c;
short s;
float f;
long double ld;

s = c; /* c is converted to short. */
f = s; /* s is converted to float. */
ld = f; /* f is converted to long double. */
```

In a similar way, the statement `double a = 50;` is equivalent to: `double a = 50.0;` because the compiler converts the integer constant 50 to the type of a, that is, `double`.

However, there are cases where information may be lost. For example, when a floating-point value is assigned to an integer variable:

```
int i;
double d = 10.999;
i = d; /* i becomes 10. */
```

The fractional part is truncated; the assigned value is not rounded to 11. When a `double` value is converted to `float` it depends on the implementation whether the original value is rounded up or down.

C allows the programmer to convert *explicitly* the type of an expression to another type using the cast operator. A cast expression has the following form:

`(data_type) expression`

The cast expression converts the type of the expression to the `data_type`. For example, after declaring the following variables:

```
float a, b, c = 2.34;
```

the expression:

```
a = (int)c;
```

casts temporarily the type of c from **float** to **int** and a becomes 2. Notice that the cast does not change the value of c; that is, c remains 2.34. The term “temporarily” means that the compiler treats next c as **float**. In a similar way, the statement:

```
b = (int)(c+4.6);
```

casts the result of the addition to **int** and assigns that result to b. Therefore, b becomes 6.

In the next example, what is performed first, the type cast or the multiplication?

```
int a = (int)1.99 * 10;
```

Let’s see the precedence table in Appendix A. Because the cast operator has higher precedence than the **\*** operator, a becomes 10. If the expression is enclosed in parentheses **(1.99 \* 10)**, a becomes 19. As we’ll see in Chapter 4, if you are not sure about the evaluation order, use parentheses.

---

## Type Qualifiers

Types may also be qualified, to indicate special properties. A variable whose value cannot change during the execution of the program is called *constant*. To declare a constant variable, we use the **const** keyword. A **const** variable must be initialized when it is declared and it is not allowed to change its value thereafter. For example, the following statement declares a as **const** and sets it equal to 10.

```
const int a = 10;
```

An attempt to change its value will produce a compile error.

Unlike **const**, the **volatile** qualifier is used much less. It is used to inform the compiler that the value of a variable might change, even if it does not appear to be modified within the program. This qualifier prevents the compiler from performing undesirable optimizations that would affect the operation of the program. Although we’ll talk about the **while** statement in Chapter 6, we think that you can understand the following code:

```
int flag = 0;
while(flag != 100)
{
    ...
}
```

The **!=** operator tests for inequality. Because the value of flag seems to remain the same, an optimizing compiler might replace the loop with an infinite loop such as **while(true)**. However, if flag can be modified by a “hidden” entity, such as the system hardware, this optimization would alter the behavior of the program. To prevent the compiler from optimizing the code, we use the **volatile** qualifier in its declaration, i.e., **volatile int flag**;

## The `#define` Directive

The `#define` directive is used to define a macro. In most cases, a macro is a name that represents a numerical value. To define such a macro, we write:

```
#define macro_name value
```

For example, the line: `#define NUM 100`

defines a macro named NUM. When a program is compiled, the preprocessor, as we'll see in Chapter 16, replaces each occurrence of NUM with 100. For example, in the following program, the values of a, b, and c are set to -80, 120, and 300, respectively.

```
#include <stdio.h>

#define NUM 100

int main(void)
{
    int a, b, c;

    a = 20 - NUM;
    b = 20 + NUM;
    c = 3 * NUM;
    return 0;
}
```

The convention that most programmers follow is to name macros with all uppercase letters. In our examples, we'll define macros with global scope before `main()`. Notice that there is no semicolon at the end of the `#define` directive. We discuss macros in more detail in Chapter 16. For now, think of them as an alternative to defining constants in your program.

In order to make the program more flexible, a good practice is to use macros to represent values that appear many times within the program.

Once a value is defined as a macro, if you ever need to change it, you only need to change it in one place. For example, to change the value 100 to 300 in the following program where the NUM macro is not used, we would have to replace it three times instead of one.

```
#include <stdio.h>
int main(void)
{
    int a, b, c;

    a = 20 - 100;
    b = 20 + 100;
    c = 3 * 100;
    return 0;
}
```

It is simple enough to make these replacements by hand in a small program like this one, but imagine trying to change every value in a program that is thousands of lines long! And to be precise, not every occurrence of 100, but the right occurrences, because the same value might also represent something else which must not change. Therefore, it is much safer and faster to use a macro to just change a defined value as needed.

Although both `const` and `#define` can be used to create names for constants, there are significant differences between them. We can use `#define` to create a name for a numerical, character, or string constant. On the other hand, we can use `const` with any type, such as pointers, arrays, structures, and unions. The `const` constants are subject to the same scope rules as variables (we'll talk about them in Chapter 11), whereas `#define` constants are not. Unlike `#define` constants, `const` values can be viewed in the debugger.

Concerning our preference, we use `#define` for constants that represent numbers. We mainly use `const` when passing data to functions for protection.

---

## The `printf()` Function

The `printf()` function is used to display data to `stdout` (standard output stream). By default, the `stdout` stream is associated with the screen. The `printf()` function accepts a variable list of parameters. The first argument is a format string, that is, a sequence of characters enclosed in double quotes, which determines the output format. If arguments follow the format string, `printf()` displays their values. The format string may contain *escape sequences* and *conversion specifications*. It may also contain ordinary characters, which are displayed as is to the screen. It is the programmer's responsibility to pass a valid format string to `printf()`; otherwise, the behavior is undefined.

## Escape Sequences

Escape sequences are used to represent nonprintable characters or characters that have a special meaning to the compiler. An escape sequence consists of a backslash (\) followed by a character. Although these sequences look like two characters, they represent only one. A format string may contain any number of escape sequences. Table 2.2 lists the escape sequences and their meaning. If the character following the \ is not supported, the behavior is undefined. The escape sequences are divided into *character escapes* and *numeric escapes*. The character escapes can represent common ASCII control characters, while the numeric escapes, which are the latter two in Table 2.2, can represent any character.

For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    printf("This is\n");
    printf("another C\n");
    printf("program\n");
    return 0;
}
```

**TABLE 2.2**

## Escape Sequences

Escape Sequence	Meaning
\a	Audible alert.
\b	Backspace.
\f	Form feed.
\n	New line.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.
\\\	Backslash.
\'	Single quote.
\"	Double quote.
\?	Question mark.
\ooo	Represent the character that corresponds to the octal number (one to three octal digits). For example, according to the ASCII set in Appendix B, '\a' and '\7' are equivalent. In another example, the value '\142' corresponds to 'b', because $142_8 = 98_{10}$ . The sequence ends if either three digits have been used or a character other than octal digit is met. For example, the string "\141p" contains the characters '\141', that is, 'a', and 'p'. The octal number does not have to begin with 0.
\xhhh	Represent the character that corresponds to the hexadecimal number (h is a hex digit or letter). For example, according to the ASCII set in Appendix B, the value '\x6b' corresponds to 'k', because $6b_{16} = 107_{10}$ . Although the standard sets no limit on the number of h, the resulting value must not exceed that of the largest character (i.e., no more than ff); otherwise, the behavior is undefined. The x must be in lower case, while h digits can be in upper or lower case.

Since the escape sequence \n moves the cursor to the beginning of the next line, the program outputs:

```
This is
another C
program
```

Alternatively, we could use one printf() and write:

```
printf("This is\nanother C\nprogram\n");
```

Here is another example:

```
#include <stdio.h>
int main(void)
{
    printf("\a");
    printf("This\b\b\b is a text\n");
    printf("\75\n"); /* The octal number 75 corresponds to the '='
character. */
```

```
    printf("This\tis\ta\x9text\n"); /* The hexadecimal number 9
corresponds to the horizontal tab. */
    printf("This is a \"text\"\n");
    printf("This is a \\text\\\"\\n");
    printf("Sample\rtext\n");
    return 0;
}
```

The program outputs:

```
(Hear a beep).
T is a text
=
This is a      text
This is a "text"
This is a \text\
textle
```

In the last `printf()`, the `\r` moves the cursor back to the beginning of the line, Samp is overwritten and text is print. As a result, the output is textle.

If you want for better readability to expand the format string of `printf()` to several lines, use a backslash. For example, the following `printf()` is written over three lines, but the output will appear on one line.

```
printf("This printf uses three lines, but the \
text will appear \
on one line ");
```

## Conversion Specifications

A conversion specification begins with the `%` character, and it is followed by one or more characters with special significance. In its simplest form, the `%` is followed by one of the conversion specifiers listed in Table 2.3.

The following program illustrates the use of `printf()` to print characters and numbers in various formats.

```
#include <stdio.h>
int main(void)
{
    int len;

    printf("%c\n", 'w');
    printf("%d\n", -100);
    printf("%f\n", 1.56);
    printf("%s\n", "some text");
    printf("%e\n", 100.25);
    printf("%g\n", 0.0000123);
    printf("%X\n", 15);
    printf("%o\n", 14);
    printf("test%n\n", &len);
    printf("%d%%\n", 100);
    return 0;
}
```

**TABLE 2.3**

## Conversion Specifiers

Conversion Specifier	Meaning
c	Display the character that corresponds to an unsigned integer value.
d, i	Display a signed integer.
u	Display an unsigned integer.
f	Display a floating-point number. The default precision is six digits.
s	Display a sequence of characters.
e, E	Display a floating-point number in scientific notation. The exponent is preceded by the chosen specifier e or E.
g, G	%e or %E form is selected if the exponent is less than -4 or greater than or equal to the precision. Otherwise, the %f form is used.
p	Display the value of a pointer variable.
x, X	Display an unsigned integer in hex form; %x displays lowercase letters (a-f), while %X displays uppercase letters (A-F).
o	Display an unsigned integer in octal.
n	Nothing is displayed. The matching argument must be a pointer to integer; the number of characters printed so far will be stored in that integer.
%	Display the character %.

The program outputs:

w (the character constant must be enclosed in single quotes).

-100

1.560000

some text (the string literal must be enclosed in double quotes).

1.002500e+002 (equivalent to  $1.0025 \times 10^2 = 1.0025 \times 100 = 100.25$ ).

1.23e-005 (because the exponent is less than -4, the number is displayed in scientific form).

F (the number 15 is equivalent to F in hex).

16 (the number 14 is equivalent to 16 in octal).

test (since four characters have been printed before %n is met, the value 4 is stored into len. We'll talk about the & operator in Chapter 8).

100% (to display the % character, we must write it twice).

## Return Value

printf() returns the number of characters printed or a negative value if an error occurs. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    printf("%d\n", printf("Test\n"));
    return 0;
}
```

Although functions are discussed in Chapter 11, you should be able to get a sense of what this program is doing. The inner `printf()` is executed first, it displays Test and returns the number of the characters printed, that is, 5 (as said, `\n` is counted as one character). The outer `printf()` uses the `%d` specifier to display the return value of the inner `printf()`. Therefore, the program displays:

```
Test  
5
```

## Printing Variables

Variable names follow the last double quote of the format string. When printing more than one variable, separate each with a comma. The compiler will associate the conversion specifications with the names of each of the variables from left to right. Each conversion specification should match the type of the respective variable or the output will be meaningless. Take a look at the following program:

```
#include <stdio.h>  
int main(void)  
{  
    int a = 10, b = 20;  
  
    printf("%d + %d = %d\n", a, b, a+b);  
    printf("%f\n", a);  
    return 0;  
}
```

In the first `printf()`, the compiler replaces the first `%d` with the value of `a`, the second `%d` with the value of `b`, and the third one with their sum. Therefore, the program displays:  $10 + 20 = 30$ . The second `printf()` displays a meaningless value, since a wrong conversion specification is used.

The compiler does not check if the number of the conversion specifications equals the number of the variables. If the conversion specifications are more than variables, the program will display meaningless values for any extra specifications. If it is less, the program will not display the values of the extra variables. For example:

```
#include <stdio.h>  
int main(void)  
{  
    int a = 10, b = 20;  
  
    printf("%d and %d and %d\n", a, b);  
    printf("%d\n", a, b);  
    return 0;  
}
```

The first `printf()` uses the `%d` specifier three times, although there are only two output variables. The compiler replaces the first `%d` with the value of `a`, the second with the value of `b`, and the third with a meaningless value. Therefore, the program displays: 10 and 20 and some meaningless value. The second `printf()` uses the `%d` specifier once, while there are two output variables. The compiler replaces the `%d` with the value of `a` and ignores the second variable. Therefore, the program displays 10.

## Optional Fields

As discussed, the conversion specification in its simplest form begins with the % character followed by the conversion specifier. However, a conversion specification may include up to four optional fields between them, as shown in Figure 2.1. We'll review each of these fields in this section.

### Precision

The precision depends on the conversion specifier, as follows:

d, i, o, u, x, X: Specifies the minimum number of the output digits. If the number has fewer digits, leading zeros are added.

e, E, f: Specifies the number of digits after the decimal point.

g, G: Specifies the maximum number of significant digits.

s: Specifies the maximum number of output characters.

The precision is specified with a dot (.), followed by either an integer or an asterisk \* and an integer in the argument list to specify the precision. If only the dot is present, the precision is zero. For example, let's display the value of a floating-point variable using various precisions. The default precision is six digits. The output value is rounded up or down, according to the value of the cutoff digit. If it is less than 5, the output value is rounded down; otherwise, it is rounded up.

```
#include <stdio.h>
int main(void)
{
    float a = 1.2365;

    printf("%f\n", a);
    printf("%.2f\n", a);
    printf("%.1f\n", 3, a);
    printf("%.f\n", a);
    return 0;
}
```

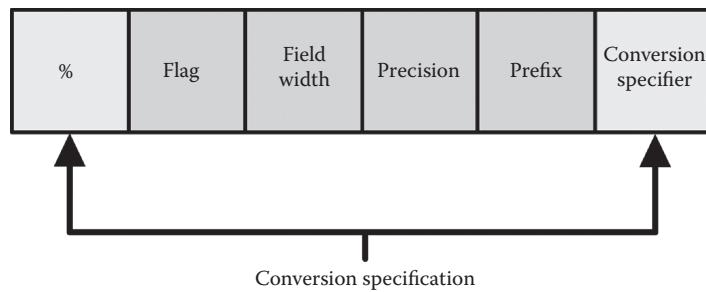


FIGURE 2.1

Conversion specification.

The first `printf()` displays the value of `a` with the default precision of the six digits, that is, 1.236500, while the second displays the value of `a` with two precision digits. This value is rounded up since the cutoff digit is 6. Therefore, the second `printf()` displays 1.24. In the third `printf()` the argument with value 3 specifies the precision. The value of `a` is rounded up since the cutoff digit is 5. Therefore, the third `printf()` displays 1.237. The last `printf()` displays the value of `a` with no precision digits. Since the cutoff digit is 2, it displays 1. If the cutoff digit were greater than or equal to 5, the output would be 2. To sum up, the program displays:

```
1.236500  
1.24  
1.237  
1
```

With this in mind, what is the output of the following program?

```
#include <stdio.h>  
int main(void)  
{  
    printf("%f\n", 5.123456789);  
    return 0;  
}
```

Since the default precision is six digits and the cutoff digit is greater than 5, the program displays 5.123457. If we increase the precision, for example, write `% .9f`, the program would display the exact number.

When displaying a string, we can define how many of its characters will be displayed as with floating-point numbers. If the defined precision exceeds the number of the string's characters, the string is displayed as is and no other character is added. For example:

```
#include <stdio.h>  
int main(void)  
{  
    char msg[] = "This is sample text";  
  
    printf("%s\n", msg);  
    printf("%.6s\n", msg);  
    printf("%.30s\n", msg);  
    return 0;  
}
```

The first `printf()` uses the `%s` to display the characters stored into the array `msg`, while the second one prints the first six characters of the string (including the space character). Since the defined precision, that is, 30, exceeds the length of the string, the third `printf()` displays the entire string as if no precision was specified. As a result, the program displays:

```
This is sample text  
This i  
This is sample text
```

## Field Width

When displaying the value of a numeric variable, we can define the total number of characters to be displayed by adding an integer to specify the width of the output field or an asterisk \* and an integer in the argument list to specify the width. If the value of the variable requires fewer characters than the defined width, leading space characters are added, and the value is right-justified. If the displayed value requires more characters, the field width will automatically expand as needed. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int a = 100;
    float b = 1.2365;

    printf("%10d\n", a);
    printf("%10f\n", b);
    printf("%10.3f\n", b);
    printf("%*.3f\n", 6, b);
    printf("%2d\n", a);
    printf("%6f\n", b);
    return 0;
}
```

Since a is 100, three characters are required to display its value. Therefore, the first printf() first displays seven leading spaces and then the number 100. To display the value of b, eight characters are required: six for the precision digits, plus one for the dot and one for the integer. Therefore, the second printf() first displays two spaces and then 1.236500.

At the third printf(), the minimum field width should be five characters since the precision digits are three digits. Therefore, the program first displays five spaces and then the rounded-up value 1.237. In the fourth printf() the argument with value 6 specifies the width. Therefore, the program first displays one space and then the rounded-up value 1.237.

To display the value of a, three characters are required, while the specified width is 2. Therefore, the width is automatically expanded and the fifth printf() displays 100. To display the value of b, the minimum field width should be eight. Once again, the width is expanded and the last printf() displays 1.236500. To sum up, the program displays:

```
100
1.236500
1.237
1.237
100
1.236500
```

## Prefix

When displaying an integer, the letter h indicates that the integer is **short**, while the letter l indicates that it is **long**. When used with e, E, f, g, or G specifiers, the letter L indicates that the value is **long double**.

## Flags

Flags can be used to control the output, as listed in Table 2.4.

For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int a = 12;

    printf("%-4d\n", a);
    printf("%+4d\n", a);
    printf("% d\n", a);
    printf("%#0x\n", a);
    printf("%#o\n", a);
    printf("%04d\n", a);
    return 0;
}
```

The first `printf()` displays the value 12 left aligned. Since the field width is 4 and the flag `-` is used, the second `printf()` first displays a space and then +12. The third `printf()` prefixes the output value with a space. The fourth `printf()` displays the value of `a` in hex prefixed by `0x`, while the next one displays the value of `a` in octal prefixed by `0`. The last `printf()` pads the number with two leading 0s up to the field width of 4. To sum up, the program displays:

```
12
+12
12
0xc
014
0012
```

*`printf()` is a powerful function, which provides many ways to format data output. A complete description of all its capabilities is beyond the scope of this book. For more details, consult your compiler's documentation.*

**TABLE 2.4**

### Flags

Flag	Meaning
-	Left aligns the output within the defined field width. By default, it is right aligned.
+	Prefixes positive signed numbers with +.
space	Prefixes positive signed numbers with a space character.
#	Prefixes octal numbers with 0 and hex numbers with 0x or 0X.
0	Numbers are padded with leading zeros until the defined width is reached.

## Exercises

C.2.1 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i = 100;

    i = i+i;
    printf("V1:%d V2:%d\n", i+i, i);
    return 0;
}
```

**Answer:** The statement `i = i+i;` makes the value of `i` equal to `i = i+i = 100+100 = 200`. The first `%d` is replaced by the value  $200+200 = 400$ , while the second `%d` by the value of `i`, that is, 200. Therefore, the program displays: V1:400 V2:200.

C.2.2 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i = 30;
    float j = 10.65;

    printf("%f %d\n", i, j);
    return 0;
}
```

**Answer:** Since `i` has been declared as an integer, we should use `%d` to display its value and `%f` to display the value of `j`. Because the specifiers are used in the wrong order, the program displays meaningless values.

C.2.3 Write a program that declares two integers, assigns to them the values 50 and 20, and displays their sum, difference, product, the precise result of their division (i.e., 2.5), and the remainder (i.e., 10). To find the remainder use the `%` operator.

```
#include <stdio.h>
int main(void)
{
    int i = 50, j = 20;

    printf("Sum = %d\n", i+j);
    printf("Diff = %d\n", i-j);
    printf("Product = %d\n", i*j);
    printf("Div = %f\n", (double)i/j);
    printf("Rem = %d\n", i%j);
    return 0;
}
```

**Comments:** As we'll see in Chapter 4, the % operator is used to find the remainder of the division of two integer operands. The other operators are used as in math. Because the cast expression (**double**)*i* converts the type of *i* from **int** to **double**, the result of their division is a float number. If we omit the cast and write *i/j*, we should use %d instead of %f, because the division result is integer. In this case, the program would output the integer result, that is, 2. Notice that we could use one **printf()** to print all values. For example:

```
printf("%d %d %d %f %d\n", i+j, i-j, i*j, (double)i/j, i%j);
```

or even:

```
printf("%d+%d=%d,%d-%d=%d,%d*%d=%d,%d/%d=%f,%d%%%d=%d\n",
i, j, i+j, i, j, i-j, i, j, i*j, i, j, (double)i/j, i, j, i%j);
```

**C.2.4** The following program displays the average of two float numbers. Is there a bug in this code?

```
#include <stdio.h>
int main(void)
{
    double i = 12, j = 5, avg;

    avg = i+j/2;
    printf("Avg = %.2f\n", avg);
    return 0;
}
```

**Answer:** Yes, there is a bug. Because the division operator / has priority over the addition operator +, the division  $5/2 = 2.5$  is performed first and, afterward, the addition  $12+2.5$ , which causes the program to display the incorrect value 14.50 instead of 8.50. This bug is eliminated by enclosing the expression *i+j* in parentheses. As shown in Appendix A, because parentheses have the highest priority among all C operators, the parenthetical expression is executed first. Therefore, the right assignment is  $\text{avg} = (i+j)/2$ .

**C.2.5** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int k;
    float i = 10.9, j = 20.3;

    k = (int)i + j;
    printf("%d %d\n", k, (int)(i + (int)j));
    return 0;
}
```

**Answer:** The cast expression (**int**)*i* converts the type of *i* from **float** to **int**. Since the result of (**int**)*i* is 10, we have  $k = 10+20.3 = 30$  (not 30.3, since *k* is **int**). The result of (**int**)(*i+(int)j*) is (**int**)( $10.9+(\text{int})20.3$ ) = (**int**)( $10.9+20$ ) = (**int**)(30.9) = 30. Therefore, the program outputs: 30 30.

**C.2.6** Write a program that assigns a two-digit positive value to an integer variable and displays the sum of its digits. For example, if the assigned value is 35, the program should display 8.

```
#include <stdio.h>
int main(void)
{
    int i, j, k;

    i = 35;
    j = i/10;
    k = i - (10*j);
    printf("Sum = %d\n", j+k);
    return 0;
}
```

**Comments:** The term `i/10` calculates the tens of `i`. Notice that we could omit the declarations of `j` and `k` and write `printf("Sum = %d\n", i/10+(i-(10*(i/10))));`

**C.2.7** Write a program that declares two floating-point variables (e.g., `i`, `j`), assigns to them two positive values (e.g., 3.45 and 6.78) and swaps their integer parts (that is, `i` becomes 6.45 and `j` becomes 3.78, respectively).

```
#include <stdio.h>
int main(void)
{
    int k;
    float i = 3.45, j = 6.78, tmp;

    tmp = i - (int)i; /* Get the integer part. */
    k = (int)i; /* Save the integer part, before changing its value. */
    i = (int)j + tmp;

    tmp = j - (int)j;
    j = k + tmp;
    printf("%f %f\n", i, j);
    return 0;
}
```

---

## Unsolved Exercises

**U.2.1** Write a program that uses one `printf()` to display the following pattern.

```
*      *
      *
*      *
```

**U.2.2** Fix the errors in order to run the program and display the value of m.

```
include <stdio.h>
int main(void) {
    int m;
    a = 10
    m = 2a+100
    print(%f\n", M);
    return 0;
}
```

**U.2.3** Fix the errors in order to run the program and display the values of a, b, and c.

```
$include <studio.h>
INT main(void)
[
    int b = a/2; a = 10
    float c = 4.5 + a;

    printf("%d %d\n", a, c, b);
    return 0;
]
```

**U.2.4** Fix the errors in order to run the program and display the sum of a and b.

```
#include (stdio.h)

#define NUM 30;

int main{void}
{
    const int a;

    a = 10;
    NUM = a+2;
    a = NUM/2;
    b = a;
    printf('D\n', a+b);
    return 0;
}
```

**U.2.5** Write a program that assigns two negative values into two integer variables and uses those variables to display the corresponding positives.

**U.2.6** Fill in the gaps and complete the program in order to display the following output.

```
21
21
15
25%
A
a
```

10  
77  
077  
63

```
#include <stdio.h>
int main(void)
{
    int x = 21, y = 0xa, z = 077;

    printf("_____\\n", x);
    printf("_____\\n", x);
    printf("_____\\n", x);
    printf("_____\\n", x);
    printf("_____\\n", y);
    printf("_____\\n", y);
    printf("_____\\n", y);
    printf("_____\\n", z);
    printf("_____\\n", z);
    printf("_____\\n", z);

    return 0;
}
```

**U.2.7** Fill in the gaps and complete the program in order to display the following output.

-12.123  
-12.123456789  
-12.123456789  
-12.123457  
-12.12346  
-12

```
#include <stdio.h>
int main(void)
{
    double x = -12.123456789;

    printf("_____\\n", x);
    printf("_____\\n", x);
    printf("_____\\n", x);
    printf("_____\\n", x);
    printf("_____\\n", x);
    printf("_____\\n", x);

    return 0;
}
```

**U.2.8** Use the flags of printf() to fill in the gaps and complete the program in order to display the following output.

x + yj = 2-3j  
x - yj = 2+3j  
y + xj = -3+2j  
y - xj = -3-2j

```
#include <stdio.h>
int main(void)
{
    int x = 2, y = -3;

    printf("x + yj = _____\n", x, y);
    printf("x - yj = _____\n", x, -y);
    printf("y + xj = _____\n", y, x);
    printf("y - xj = _____\n", y, -x);
    return 0;
}
```

**U.2.9** Write a program that assigns two positive values into two integer variables and displays the remainder of their division. Use only two variables and don't use the % operator.

**U.2.10** Write a program that assigns two positive values into two float variables and displays the integer part of their division and the fractional part. For example, if they are assigned the values 7.2 and 5.4, the program should display 1 and 1.8, since  $7.2 = (1 \times 5.4) + 1.8$ .

**U.2.11** Write a program similar to C.2.6 for a three-digit positive value.

## Getting Input with scanf()

This chapter focuses on using the `scanf()` function to get input from the user, store that input in variables, and use it in your programs. `scanf()` is a powerful function that offers many ways to read and store data in program variables. We won't cover every aspect of using `scanf()`, but we'll give you enough of what you need, in order to use it to read data. In this chapter, we'll mainly use `scanf()` to read numeric data and store that data in numeric variables. You'll see other uses of `scanf()` over the next chapters. Just remember, as you read, that `scanf()` is not an easy function to use; it contains several traps, so use it with great care.

### The `scanf()` Function

The `scanf()` function is used to read data from `stdin` (*standard input stream*) and store that data in program variables. By default, the `stdin` stream is associated with the keyboard. The input data are read according to a particular format, very similar to the way output is handled with `printf()`.

The `scanf()` function accepts a variable list of parameters. The first is a format string similar to that of `printf()`, followed by the memory addresses of the variables in which the input data will be stored. Typically, the format string contains only conversion specifiers. The conversion characters used in `scanf()` are the same as those used in `printf()`. For example, the `%d` specifier is used to read an integer. As with `printf()`, the number of conversion specifiers included in the format string should equal the number of variables, and the type of each specifier should match the data type of the respective variable.

In the following example, we use `scanf()` to read an integer from the keyboard and store it in variable `i`. The first argument is the `%d`, and the second is the memory address of `i`. Once the user enters an integer and presses *Enter*, this value will be stored into `i`.

```
int i;  
scanf("%d", &i);
```

The `&` character in front of the variable represents the *address operator*, and it indicates the memory location, in which the input number will be stored. We'll talk much more about the `&` operator when discussing pointers in Chapter 8. The new line character ('`\n`') that is generated when the *Enter* key is pressed is left in `stdin`, and it will be the first character read in next call to `scanf()`. In the following example, we use `scanf()` to read an integer and a float number from the keyboard and store them in variables `i` and `j`, respectively.

```
int i;  
float j;  
scanf("%d%f", &i, &j);
```

The first argument is the format string `%d%f`, while the next arguments are the memory addresses of `i` and `j`. The `%d` specifier corresponds to the address of `i`, while the `%f` specifier corresponds to the address of `j`.

When `scanf()` is used to read more than one value, a *white-space* character should be used to separate them. So, in the previous example, if the user enters the values 10 and 4.65 separated by one or more spaces and presses *Enter*, `i` will become 10 and `j` will become 4.65.

---

When `scanf()` is used to read numeric values, it skips white-space characters (i.e., spaces, tabs, or new line characters) before the input values.

If precision is critical in your program, use a `double` variable instead of `float` and the `%lf` conversion to read it, not `%f`. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    float a;

    printf("Enter number: ");
    scanf("%f", &a);
    printf("%f\n", a);
    return 0;
}
```

Because the `float` type may not always represent float numbers precisely, the program may not display the input value, but a value close to it. It is much safer to declare `a` as `double` and use the `%lf` conversion to read it.

---

`printf()` uses `%f` to display the values of either `float` or `double` variables, while `scanf()` uses `%lf` to store the input value in a `double` variable and `%f` to store it in a `float` variable.

The letter `l` in front of the conversion characters `e`, `E`, `f`, `g`, and `G` indicates that the input value will be stored into a `double` variable, while the letter `L` indicates a `long double` variable. The letter `l` in front of `d`, `i`, `o`, `u`, `x`, and `X` indicates that the value will be stored into a `long` variable, while the letter `h` indicates a `short` variable. The following example uses `scanf()` to read a character and store it into `ch`.

```
char ch;
scanf("%c", &ch);
```

In the following example, we use `scanf()` to read a series of characters and store them in the `str` array. `scanf()` appends the null character at the end. Notice that the `&` operator does not precede `str` because, as we'll see in Chapters 8 and 11, the passed argument is a pointer to the first element of the array.

```
char str[100];
scanf("%s", str);
```

For example, if the user enters sample and presses *Enter*, its characters will be stored into the respective elements of the `str` array. That is, the value of `str[0]` will be 's', `str[1]` will be 'a', and so on. When `%s` is used, `scanf()` stops reading once it encounters a white-space character. For example, if the user enters many words only the word many will be stored into `str`. As it will be discussed in Chapter 10, in order to avoid memory overflow it is much safer to use `%ns` rather than `%s`, where `n` specifies the maximum number of characters to be read.

For greater flexibility, `scanf()` supports the [ specifier in the form `%[set]` or `%[^set]`, where `set` can be any set of characters. The first form stores into the array only input characters that are contained in `set`, while the second form stores characters that are not contained. For example, if we write `scanf(%[tpe], str);` `scanf()` stops reading once it encounters an input character other than t, p or e. If the user enters test, `scanf()` will store te into `str`. In a similar way, if we write `%[^tpe]` and the user enters another, `scanf()` will store ano into `str`.

---

Omitting the & operator is an extremely popular error. Remember, `scanf()` needs it in front of each numeric variable (i.e., `int`, `double`, `char`, `float`, ...). If you forget it, the program won't execute correctly. On the other hand, if the variable is pointer, don't use the & operator.

---

Another common error is that the use of the & operator in `scanf()` makes novice programmers to do the same in `printf()`. For example, `printf("%d\n", &a);`. The program won't display the value `a`, but the memory address of `a`, as will be further discussed in Chapter 8.

## Use of Ordinary Characters

In most cases, the format string does not contain other characters than the conversion characters (e.g., `%d%d`). If you add any characters, the user must also enter them in the same place. If the characters don't match, `scanf()` aborts reading. For example, notice the character m between the %d specifiers in the following program:

```
#include <stdio.h>
int main(void)
{
    int a, b;

    scanf("%dm%d", &a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

The user should also add the character m between the values (e.g., 12m43) or `scanf()` will fail and the program won't display the input values.

In another example, if a program asks from the user to enter a date using / to separate the values, the format string should be `%d/%d/%d` and the user must enter the date in that form (e.g., 3/8/2020).

Novice programmers often add a \n at the end of the format string. However, the \n character advances `scanf()` to read the next non white-space character. For example, if we

`write scanf("%d\n", &i);` `scanf()` reads an integer and then the program hangs until the user enters a non white-space character.

## Return Value

`scanf()` returns the number of data items that were successfully read and assigned to program variables. `scanf()` returns prematurely if the input values do not match the types or order of the conversion specifiers. The unread values are left in `stdin`. In the following example, if the user enters an integer first and then a float number, `scanf()` returns 2.

```
int i;
float j;
scanf ("%d%f", &i, &j);
```

If the user enters other values than expected, such as two float numbers or a float number and then an integer, `scanf()` won't be executed correctly.

The following program checks the return value of `scanf()`, to verify that the input integer is successfully read and stored into `num`. A return value other than 1 indicates a read error. You'll learn more about how this program works after reading Chapters 6 and 11.

```
#include <stdio.h>
int main(void)
{
    int ch, num;

    printf("Enter number: ");
    while (scanf ("%d", &num) != 1)
    {
        printf("Enter number: ");
        while ((ch = getchar()) != '\n' && ch != EOF)
            ; /* Consume the unread characters. */
    }
    printf("Input value: %d\n", num);
    return 0;
}
```

For example, if the user enters a character and not an integer, `scanf()` won't return 1 and the outer loop will be executed again. The inner loop gets the characters left in `stdin` in order to read the new input value. Instead, many programmers write `fflush(stdin);`

---

Be careful, according to the C standard, the behavior of `fflush()` when applied on an input stream is undefined. Simply put, undefined means don't do it.

For the sake of brevity and simplicity, we won't check the return value of `scanf()` throughout this book. Instead, we'll assume that the user's input is valid. Nevertheless, remember that a robust program should always check its return value to verify that no read failure occurred. As a matter of fact, a robust program should check the return value of any called function that returns a value.

*`scanf()` is a powerful function, which provides many ways to format data input. A complete description of all its capabilities is beyond the scope of this book. For more details, consult your compiler's documentation. Just remember to be very careful when using it.*

## Exercises

C.3.1 Write a program that reads a character, an integer, and a float number and displays them. Use one `scanf()`.

```
#include <stdio.h>
int main(void)
{
    char ch;
    int i;
    float f;

    printf("Enter character, int and float: ");
    scanf("%c%d%f", &ch, &i, &f);
    printf("\nC:%c\tI:%d\tF:%f\n", ch, i, f);
    return 0;
}
```

C.3.2 Write a program that reads two integers and displays their sum, difference, product, and the result of their division.

```
#include <stdio.h>
int main(void)
{
    int i, j;

    printf("Enter 2 integers (the second should not be 0):");
    scanf("%d%d", &i, &j);
    printf("Sum:%d Diff:%d Prod:%d Div:%f\n", i+j, i-j, i*j, (double)
i/j); /* Cast i to double, in order to display the decimal part of the
division. */
    return 0;
}
```

**Comments:** If the user enters 0 as the second integer, the program won't perform the division, since division by 0 is impossible. A typical response is the abnormal program termination. This can be avoided with the use of the `if` statement, as we'll see in Chapter 5.

C.3.3 Write a program that reads the prices of three products and displays their average.

```
#include <stdio.h>
int main(void)
{
    float i, j, k, avg;

    printf("Enter 3 prices: ");
    scanf("%f%f%f", &i, &j, &k);

    avg = (i+j+k)/3; /* The parentheses are necessary to perform the
addition first and then the division. */
    printf("Average = %f\n", avg);
    return 0;
}
```

**Comments:** Without using avg we could write `printf("Average = %f\n", (i+j+k)/3);`

**C.3.4** Write a program that reads the radius of a circle and displays its area and perimeter.

```
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    double radius;

    printf("Enter radius: ");
    scanf("%lf", &radius);
    printf("%f %f\n", PI*radius*radius, 2*PI*radius);
    return 0;
}
```

**C.3.5** Write a program that reads a positive float number and displays the previous and next integers.

```
#include <stdio.h>
int main(void)
{
    float i;

    printf("Enter positive number: ");
    scanf("%f", &i);
    printf("%f is between %d and %d\n", i, (int)i, (int)(i+1));
    return 0;
}
```

**C.3.6** Write a program that reads two integers and swaps their values.

```
#include <stdio.h>
int main(void)
{
    int i, j, temp;

    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);

    temp = i;
    i = j;
    j = temp;
    printf("%d %d\n", i, j);
    return 0;
}
```

**C.3.7** Continuing the previous exercise, write a program that reads three floats, stores them in three variables, and rotates them one place right. For example, if the user enters the numbers 1.2, 3.4, and 5.6 and they are stored in variables d1, d2, and d3, the program

should rotate their values one place right, so that d1, d2, and d3 become 5.6, 1.2, and 3.4, respectively.

```
#include <stdio.h>
int main(void)
{
    double d1, d2, d3, temp;

    printf("Enter numbers: ");
    scanf("%lf%lf%lf", &d1, &d2, &d3);

    temp = d1;
    d1 = d2;
    d2 = temp;

    temp = d3;
    d3 = d1;
    d1 = temp;
    printf("%f %f %f\n", d1, d2, d3);
    return 0;
}
```

**C.3.8** Write a program that reads two positive numbers, a float and an integer, and displays the remainder of their division. For example, if the user enters 7.24 and 4, the program should display 3.24.

```
#include <stdio.h>
int main(void)
{
    int num2, div;
    double num1;

    printf("Enter positives float and int: ");
    scanf("%lf%d", &num1, &num2);
    div = num1/num2; /* Suppose that the input values are 7.24 and 4.
Since div is declared as integer it becomes 1. */
    printf("%f\n", num1 - (div*num2));
    return 0;
}
```

**Comments:** Without using the variable div, we could write: `printf("%f\n", num1 - (int)num1/num2*num2);`

**C.3.9** Write a program that reads two positive float numbers and displays the sum of their integer and decimal parts. Use only two `double` variables. For example, if the user enters 1.23 and 9.56, the program should display 10 and 0.79.

```
#include <stdio.h>
int main(void)
{
    double i, j;
```

```

printf("Enter two positives: ");
scanf("%lf%lf", &i, &j);
printf("%d %f\n", (int)i+(int)j, (i-(int)i) + (j-(int)j));
return 0;
}

```

**C.3.10** Write a program that reads a positive integer and converts it to multiples of 50, 20, 10, and 1. For example, if the user enters 285, the program should display 5\*50, 1\*20, 1\*10, 5\*1.

```

#include <stdio.h>
int main(void)
{
    int i, n_50, n_20, n_10, n_1, rem;

    printf("Enter positive number: ");
    scanf("%d", &i); /* Suppose that the user enters 285,
/* */
    n_50 = i/50; /* n_50 = 285/50 = 5. */
    rem = i%50; /* The rem = 285%50 = 35 should be analysed in 20s,
10s, and units. */

    n_20 = rem/20; /* n_20 = 35/20 = 1. */
    rem = rem%20; /* The rem = 35%20 = 15 should be analysed in 10s
and units. */

    n_10 = rem/10; /* n_10 = 15/10 = 1. */
    n_1 = rem%10; /* n_1 = 15%10 = 5. */
    printf("%d*50,%d*20,%d*10,%d*1\n", n_50, n_20, n_10, n_1);
    return 0;
}

```

**Comments:** The % operator is used to find the remainder of the division. We could use only i, and write: printf("%d\*50, %d\*20, %d\*10, %d\*1\n", i/50, i%50/20, i%50%20/10, i%50%20%10);

**C.3.11** Write a program that reads a two-digit positive integer and displays the reversed number. For example, if the user enters 72, the program should display 27. Use one integer variable.

```

#include <stdio.h>
int main(void)
{
    int i;

    printf("Enter number between 10 and 99: ");
    scanf("%d", &i);

    i = 10*(i%10) + i/10;
    printf("%d\n", i);
    return 0;
}

```

**Comments:** The term  $(i/10)$  calculates the tens of  $i$ , while the term  $(i \% 10)$  calculates its units. The reversed number is produced by multiplying the units by 10 and adding the tens.

C.3.12 Write a program similar to the previous one for a three-digit positive integer.

```
#include <stdio.h>
int main(void)
{
    int i;

    printf("Enter number between 100 and 999: ");
    scanf("%d", &i);
    i = 100*(i%10) + 10*(i%100/10) + i/100;
    printf("%d\n", i);
    return 0;
}
```

**Comments:** The term  $(i \% 10)$  calculates the units of  $i$ , the term  $(i \% 100 / 10)$  its tens, and the term  $i / 100$  its hundreds.

C.3.13 Write a program that reads a positive integer and rounds it up or down depending on the last digit. For example, if the user enters 12, the program should display 10. If the input number is 3, the program should display 0, while if it is 455, the program should display 460.

```
#include <stdio.h>
int main(void)
{
    int i;
    float j;

    printf("Enter positive number: ");
    scanf("%d", &i);

    i = i+5;
    j = (float)i/10;
    i = (int)j * 10; /* Cut off the last digit. */
    printf("%d\n", i);
    return 0;
}
```

---

## Unsolved Exercises

U.3.1 Write a program that reads an integer and a float number and displays the triple of their sum.

U.3.2 Write a program that reads a bank deposit and the annual interest rate as percentage and displays the total amount one year after.

**U.3.3** Write a program that reads the ages of a father and his son and displays in how many years the father will have the double age of his son and their ages at that time, as well.

**U.3.4** Write a program that reads the number of students who passed and failed in the exams and displays the percentages. For example, if the user enters 12 and 8, the program should display

Success Ratio: 60%

Fail Ratio: 40%

**U.3.5** Suppose that a customer in a store bought some plates and cups. Write a program that reads the number of the plates and the price of one plate, the number of the cups and the price of one cup, and the amount the customer paid. The program should calculate and display the change.

**U.3.6** Write a program that reads an octal number, a hexadecimal number, and a decimal integer and displays their sum in decimal. For example, if the user enters 20, 3f, and 5, the program should display 84.

**U.3.7** Write a program that reads an integer that represents a number of seconds and splits it in hours, minutes, and seconds. For example, if the user enters 8140, the program should display 2h 15m 40s. Use only one variable.

**U.3.8** Modify the exercise C.3.13, so that the program reads a positive integer and rounds it up or down depending on its two last digits. For example, if the user enters 40, the program should display 0. If the input number is 130, the program should display 100, while if it is 4550, the program should display 4600.

**U.3.9** Write a program that reads a two-digit positive integer and duplicates its digits. For example, if the user enters 12, the program should display 1122.

**U.3.10** Write a program that reads two three-digit positive integers and swaps the hundreds of the first one with the units of the second one, and displays the new values. For example, if the user enters 123 and 456, the program should display 623 and 451. Use up to three variables.

**U.3.11** Write a program that reads a four-digit positive integer and a one-digit positive integer. The program should display the value of a five-digit integer, which is produced by inserting the one-digit integer into the middle of the four-digit integer. For example, if the user enters 1234 and 5, the program should display 12534.

## Operators

Now that you've been introduced to the concepts of C data types, variables, and constants, it is time to learn how to manipulate their values. C provides a rich set of operators that are used to form expressions and perform operations on them. In this chapter, we'll present some of Cs most important operators. We'll introduce the rest gradually over the next several chapters, and you'll see how to apply them when writing more complex programs.

### The = Assignment Operator

The = operator is used to assign a value to a variable. For example, the statement `a = 10;` assigns the value 10 to the variable a, while the statement `a = k;` assigns the value of the variable k to a. When used in a chained assignment, the assigned value is stored in all variables in the chain. For example:

```
int a, b, c;  
a = b = c = 10;
```

the values of a, b, and c become 10. In particular, because the = operator is right associative, the rightmost expression is evaluated first and evaluation proceeds from right to left. Therefore, first c becomes 10, then c is assigned to b, and, finally, b is assigned to a.

An assignment such as `a = 10;` produces a result, which is the value of the left operand (i.e., a) after the assignment. For example:

```
a = 10;  
c = a + (b = a+10);
```

The result of the assignment `b = a+10` is 20, so c becomes 30. Use separate assignments rather than mixing them together; it is much easier to control the code.

The right operand (*rvalue*) is an expression that has a value, such as a constant or a variable. The left operand must be an *lvalue*. An *lvalue* refers to an object stored in memory, such as a variable. It is not allowed to use a constant or the result of a computation. For example:

```
10 = a; /* Illegal. */  
a+b = 20; /* Illegal. */
```

The compiler will produce an error message similar to "error '=' : left operand must be lvalue." Also, the *lvalue* must be modifiable. For example, we'll see in Chapter 8 that an array variable is not a modifiable *lvalue*.

If the variable and the assigned value are not of the same type, the value is first converted to the variable's type, if possible, and then assigned. For example, what would the value of b be in the following code?

```
int a;  
float b;  
b = a = 10.9;
```

Since a is an integer, its value becomes 10, and this value is stored in b. Therefore, b becomes 10, not 10.9.

If an overflow occurs in a computation with signed numbers, for example if the outcome of an addition is too big to be stored into a variable, the result is undefined. If the numbers are unsigned, the result is defined as modulo  $2^n$ , where n is the number of bits of the unsigned type. For example, if the size of the `unsigned int` type is 32 bits the result of `4000000000u + 2000000000u` is 1705032704.

---

When performing arithmetic operations, be sure that overflow won't occur in order to avoid unpleasant situations.

---

For example, if we use `malloc()` to allocate memory, we'll see how in Chapter 14, and an overflow occurs in the computation of its size, the size of the allocated memory won't be the one asked with unpredictable consequences on the program execution. We won't go into details; however, we'll give you a hint that the basic idea in order to avoid overflows is to check the possibility of an overflow before it occurs. For example, suppose that all a, b, and c are `int` variables and b is positive; before assigning the result of a+b into c, we could write:

```
if(a > INT_MAX - b) /* INT_MAX is defined in <limits.h>. */  
{  
    printf("Error: overflow case\n");  
    return;  
}  
c = a+b;
```

For simplicity, we assume that the values used throughout the book's programs are within valid ranges and no overflow occurs.

---

## Arithmetic Operators

The arithmetic operators `+`, `-`, `*`, and `/` are used to perform addition, subtraction, multiplication, and division, respectively. If the operands of the additive operators `+` and `-` have arithmetic type, the arithmetic conversions discussed in Chapter 2 are applied, if necessary.

When both operands are integers, the `/` operator cuts off the decimal part. For example, suppose that a and b are `int` variables with values 3 and 2, respectively. Then, the result of `a/b` is 1, not 1.5. If either of the operands is a floating point variable or constant, the

decimal part is not truncated. For example, the result of `a/5.0` is `0.6`, since the decimal point makes the constant floating-point.

The `%` operator is used to find the remainder of the division of two integer operands. Both operands must be integers or the program won't compile. In our example, the result of `a%b` is `1`. If the right operand of the `/` or `%` operator is `0`, the result is undefined (e.g., the program may terminate). If the operands are not positives, the result depends on the implementation. For example, the result of `-8/6` could be either `-1` or `-2`; it depends whether the implementation rounds up or down the result. Similarly, the result of `-8%6` could be either `2` or `-2`; it depends on the implementation. Therefore, to check if `n` is odd, don't write something like this:

```
if((n % 2) == 1)
```

If the number is odd and negative, the result in some implementation could be `-1`, so the test would fail. The safe test is to write:

```
if((n % 2) != 0)
```

As we'll see next, an alternative way is to use bit operators.

These five operators are called *binary* because they require two operands. Also, C provides the `+` and `-` *unary* operators, which require one operand. The unary `+` operator has no effect. The result with the unary `-` is the negative of its operand. When both unary and binary `+` and `-` operators are mixed, the compiler figures out how to apply them. For example:

```
int a = +20; /* + has no effect. */
a = -10; /* - corresponds to the unary operator. */
int b = -(a-5); /* The inner - corresponds to the binary operator and the
outer - to the unary operator. */
-a; /* This statement does nothing; the value of a remains the same. */
```

The variable `b` is initialized to `15` and `a` is `-10`.

By convention, most programmers add a space before and after the operators. Most times, we do the same, less often with the arithmetic operators though.

---

## The `++` and `--` Operators

The `++` increment operator is used to increment the value of a variable by `1`. It can be added before (*prefix*) or after (*postfix*) the variable's name. The operand must be a modifiable lvalue, such as an ordinary variable. For example:

```
int a = 4;
a++; /* Equivalent to a = a+1; */
+a; /* Equivalent to a = a+1; */
(a+10)++; /* Illegal. */
```

The value of `a` is incremented twice and becomes `6`.

If the `++` operator is used in postfix form, the increment is performed after the current value of the variable is used in an expression. For example:

```
int a = 4, b;  
b = a++;
```

The current value of `a` is first stored into `b`, and then `a` is incremented. Therefore, `a` becomes 5, and `b` becomes 4.

If used in the prefix form, the variable's value is first incremented, and the new value is used in the evaluation of the expression. For example:

```
int a = 4, b;  
b = ++a;
```

`a` is first incremented, and then that value is stored into `b`. Therefore, both variables become 5.

In a similar way, the `--` decrement operator is used to decrement the value of a variable by 1. For example:

```
int a = 4;  
a--; /* Equivalent to a = a-1; */  
--a; /* Equivalent to a = a-1; */
```

The value of `a` is decremented twice and becomes 2. When the `--` operator is used in an expression, it behaves like the `++` operator. For example:

```
#include <stdio.h>  
int main(void)  
{  
    int a = 4, b;  
  
    b = a--;  
    printf("a = %d b = %d\n", a, b);  
  
    b = --a;  
    printf("a = %d b = %d\n", a, b);  
    return 0;  
}
```

With the statement `b = a--`; the current value of `a` is first stored into `b` and then its value is decremented. Therefore, the first `printf()` displays: `a = 3 b = 4`. Next, with the statement `b = --a`; the value of `a` is first decremented and then its value is stored into `b`. Therefore, the second `printf()` displays: `a = 2 b = 2`. Let's see another example:

```
#include <stdio.h>  
int main(void)  
{  
    double a = 3.45;  
  
    printf("%.2f\n", a++);  
    printf("%.2f\n", ++a);  
    return 0;  
}
```

The program first displays 3.45, and then the value of a is incremented. Next, a is first incremented, so the program displays 5.45. Let's see another example where the ++ and -- operators are combined in the same expression.

```
#include <stdio.h>
int main(void)
{
    int a = 1, b = 2, c = 3;
    printf("%d\n", (++a) - (b--) + (--c));
    return 0;
}
```

The operation `++a` first increases the current value of a, and then its new value is used. With the operation `b--` the current value of b is first used and then its value is decremented. Finally, with the operation `--c` the value of c is first decremented and then used in the expression. Therefore, the result of the expression is  $2-2+2 = 2$  and the values of a, b, and c become 2, 1, and 2, respectively.

---

Don't apply the ++ and -- operators in the same variable and create expressions whose values depend on the evaluation order.

For example, the result of: `a * a++;` is implementation dependent. It depends on the compiler whether a is first incremented and then multiplied, or not.

---

## Relational Operators

The relational operators `>`, `>=`, `<`, `<=`, `!=`, and `==` are used to compare two operands and determine their relationship. As we'll see in next chapters, the relational operators are mostly used in `if` statements and iteration loops. Consider the following statements:

- `if(a > 10)` the `>` operator is used to check whether a is greater than 10.
- `if(a >= 10)` the `>=` operator is used to check whether a is greater than or equal to 10.
- `if(a < 10)` the `<` operator is used to check whether a is less than 10.
- `if(a <= 10)` the `<=` operator is used to check whether a is less than or equal to 10.
- `if(a != 10)` the `!=` operator is used to check whether a is not equal to 10.
- `if(a == 10)` the `==` operator is used to check whether a is equal to 10.

---

An expression is considered true when it has a nonzero value; otherwise, it is considered false.

When used in an expression, the value of the expression is either 0 or 1. For example, the value of `(a > 10)` is 1 only if a is greater than 10; otherwise, it is 0. How about this one: `-5 < a < 2?` Does the value of that expression depends on the value of a?

Because the `<` operator is left associative (see Appendix A), first the value of `-5 < a` is evaluated, which is either 0 or 1, and then that value is compared with 2, which is always true. Therefore, the value of the expression is 1, no matter what the value of `a` is. As we'll see next, to test whether a value is within a given set, we use the `&&` operator.

---

Don't confuse the `==` operator with the `=` operator. The `==` operator is used to check whether two expressions have the same value, while the `=` operator is used to assign a value.

For example, the value of the expression `a == 10` is 1, only if `a` is equal to 10, otherwise, it is 0. On the other hand, the expression `a = 10` assigns the value 10 into `a`. We'll see more examples like this when presenting the `if` statement in the next chapter.

---

## Exercises

C.4.1 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = 3, b = 5, c;

    a = (a > 3) + (b <= 5);
    b = (a == 3) + ((b-2) >= 3);
    c = (b != 1);
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

**Answer:** Since `a` is 3, the value of the expression `(a > 3)` is 0. The value of `(b <= 5)` is 1 because `b` is 5. Therefore, `a = 0+1 = 1`.

Since `a` is 1, the value of `(a == 3)` is 0. The value of `((b-2) >= 3)` is 1 because `b-2 = 5-2 = 3`. Therefore, `b = 0+1 = 1`.

Since `b` is 1, the value of `(b != 1)` is 0. Therefore, `c` becomes 0. As a result, the program displays: 1 1 0.

---

## Compound Assignment Operators

The compound assignment operators are used to shorten statements using the form: `exp1 op= exp2`; This statement is equivalent to `exp1 = exp1 op (exp2)`. Notice that the compiler encloses the expression `exp2` in parentheses for reasons of priority. Typically, the `op` operator is one of the arithmetic operators: `+`, `-`, `*`, `%`, `/`, although it can also be any of

the bit operators: `&`, `^`, `|`, `<<`, `>>`, as we'll describe later in this chapter. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int a = 4, b = 2;

    a += 6;
    a *= b+3;
    a -= b+8;
    a /= b;
    a %= b+1;
    printf("Num = %d\n", a);
    return 0;
}
```

The statement `a += 6`; is equivalent to: `a = a+6 = 4+6 = 10`.

The next statement `a *= b+3`; is equivalent to: `a = a*(b+3) = 10*(2+3) = 50`. Notice that the compiler encloses the expression `b+3` in parentheses rather than `a = a*b+3`.

The statement `a -= b+8`; is equivalent to: `a = a-(b+8) = 50-(2+8) = 40`.

The statement `a /= b`; is equivalent to: `a = a/b = 40/2 = 20`.

Finally, the statement `a %= b+1`; is equivalent to: `a = a%(b+1) = 20%(2+1) = 2`. Therefore, the program outputs: Num = 2.

When the expression is a long one, the compound operator can make the code easier to read and safer to write as well. For example, it is more convenient and safer to write:

```
publisher[i].book[j].prc += 10;
```

rather than the two long expressions. Yes, it is more convenient, but why is it safer? Because it is too easy to make a typing error and write:

```
publisher[i].book[j].prc = publisher[i].book[i].prc + 10;
```

and use `i` instead of `j`, in the right expression. As you'll learn in Chapter 7, a crash might happen because of that error!

---

When using the compound assignment operators, be careful first to write the op operator and then the `=`. If you switch the characters, you most likely insert a bug.

For example, if you write `a == 3`; the value of `a` becomes `-3`. In a later chapter, there is an exercise that contains such an oversight. Watch out!

## Logical Operators

The logical operators `!`, `&&`, and `||` are used to form logical expressions. The `!` operator is *unary*, while `&&` and `||` are both binary. The logical operators produce either `0` or `1`, just like

the relational operators. The logical operators are left associative, so the compiler evaluates the operands from left to right.

## The ! Operator

The `!` operator is applied on a single operand. If an expression `exp` has a nonzero value, then the value of `!exp` is 0. Otherwise, it is 1. For example:

```
#include <stdio.h>
int main(void)
{
    int a = 4;
    printf("%d\n", !a);
    return 0;
}
```

Since `a` has a nonzero value, the program outputs 0. If `a` were 0, the program would output 1.

As we'll see in the next chapter, the `!` operator is mostly used in `if` statements to test whether the value of an expression is true or false. For example:

- the statement `if(!a)` is equivalent to `if(a == 0)`
- the statement `if(a)` is equivalent to `if(a != 0)`

## The `&&` Operator

The `&&` operator represents the logical *AND* operator. It works like the AND gate in electronic circuit designs. Specifically, the value of the expression is 1, if both operands are true. Otherwise, the result is 0. For example, if we write: `a = (10 == 10) && (5 > 3);` `a` becomes 1, since both operands are true. If we write: `a = (10 == 10) && (5 > 3) && (13 < 8);` `a` becomes 0, since the last operand is false.

---

The `&&` operator performs “short-circuit” evaluation of its operands. That is, if an operand is false, the compiler stops evaluating the other operands and the value of the entire expression becomes 0.

For example, in the following program `b` is not incremented because the first operand (`a > 15`) is false.

```
#include <stdio.h>
int main(void)
{
    int a = 10, b = 20, c;

    c = (a > 15) && (++b > 15);
    printf("%d %d\n", c, b);
    return 0;
}
```

Therefore, the program outputs: 0 20.

## The || Operator

The `||` operator represents the logical *OR* operator. It works like the *OR* gate in electronic circuit designs. Specifically, the value of the expression is 1 if either of its operands is true. If both operands are false, the result is 0. For example, if we write: `a = (10 == 10) || (3 > 5);` `a` becomes 1, since the first operand is true. If we write: `a = (10 != 10) || (3 > 5) || (4+5 > 40-10);` `a` becomes 0 since all operands are false.

---

Similar to the `&&` operator, the `||` operator performs “short-circuit” evaluation of its operands. If an operand is true, the compiler stops evaluating the other operands and the value of the entire expression becomes 1.

For example, in the following program, `b` is not incremented because the first operand (`a > 5`) is true.

```
#include <stdio.h>
int main(void)
{
    int a = 10, b = 20, c;

    c = (a > 5) || (++b > 15);
    printf("%d %d\n", c, b);
    return 0;
}
```

Therefore, the program displays: 1 20.

*Although we have not discussed about the `if` statement, we'll use it in some of the following programs in order to show you how to use logical operators.*

---

## Exercises

C.4.2 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = 5;

    printf("%d %d %d\n", !!a, !!!a, a);
    return 0;
}
```

**Answer:** Since `a` is 5, the value of `!a` is 0. Therefore, the expression `!!a` is equivalent to `!0` and the expression `!!!a` is equivalent to `!1`. The value of `a` does not change. As a result, the program displays 1, 0, and 5.

#### C.4.3 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = 4, b = 3, c = 5;

    printf("%d\n", (a < b) == !(c > b));
    return 0;
}
```

**Answer:** The value of the expression  $(a < b)$  is 0 because  $a$  is 4 and  $b$  is 3. Since the value of  $(c > b)$  is 1, the value of  $!(c > b)$  is 0. Since both expressions have the same value, the program displays 1.

#### C.4.4 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = 10, b = -10;

    if(a > 0 && b < -10)
        printf("One ");
    else
        printf("Two ");

    if(a > 10 || b == -10)
        printf("One ");
    else
        printf("Two ");
    return 0;
}
```

**Answer:** Since  $b$  is -10, the second operand is false and the program outputs Two. Since the second operand in the second **if** statement is true, the program outputs One.

#### C.4.5 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = 1, b = 4, c = 4;

    printf("%d\n", (a-1) || (b-- == c));
    printf("%d", !(a > b) && (a-1 == b-3) && (c%2 || (a+b-c)));
    return 0;
}
```

**Answer:** Since a is 1, the value of the expression (a-1) is 0. The value of (b-- == c) is 1 because b, that is, 4, it is first compared with c, that is, 4, and then it becomes 3. Therefore, the program outputs 1.

Since a is 1 and b is 3, the value of the expression (a > b) is 0. Therefore, the value of !(a > b) is 1.

The value of (a-1 == b-3) is 1 because 0 equals 0.

The value of c%2 is 0, because c is 4. The value of (a+b-c) is  $1+3-4 = 0$ . Since both operands are false, the value of the third operand is 0.

Since the third operand is false, the value of the entire expression is false and the program outputs 0.

**C.4.6** Write a program that reads an annual income and calculates the tax according to Table 4.1.

For example, if the user enters 23000, the tax is calculated as:  $\text{tax} = (20000-5000)*0.15 + (23000-20000)*0.3$ .

```
#include <stdio.h>
int main(void)
{
    float tax, a;

    printf("Enter income: ");
    scanf("%f", &a);

    tax = (a > 5000 && a <= 20000)*(a-5000)*0.15 + (a > 20000)*((a-
20000)*0.3 + 15.000*0.15);
    printf("Tax = %.2f\n", tax);
    return 0;
}
```

**Comments:** If the income is between 5000 and 20000, the value of the expression (a > 5000 && a <= 20000) is 1, while the value of (a > 20000) is 0. The tax is calculated for the part of the amount over than 5000. If the income is greater than 20000, the reverse is true. The tax is calculated for the part of the amount over than 20000 plus the tax for the first 20000. If the income is less than 5000, both expressions are false, so tax becomes 0.

Isn't it a complicated solution? Yes, it is. However, there's no need to worry. In the next chapter, you'll see how the **if** statement makes easy to solve this type of exercise.

**TABLE 4.1**

Tax Rates

Income	Tax Rate (%)
0–5.000	0
5.001–20.000	15
>20.000	30

## The Comma Operator

The comma (,) operator can be used to merge several expressions to form a single expression. It is used like this:

```
exp1, exp2, exp3, ...
```

Because the comma operator is left associative, the expressions are evaluated from left to right. Therefore, `exp1` is evaluated first, then `exp2`, then `exp3`, through the last expression. The type and the value of the entire expression are those of the last expression evaluated. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int b;
    b = 20, b = b+30, printf("%d\n", b);
    return 0;
}
```

Since the expressions are evaluated from left to right, `b` becomes 50 and the program displays 50. Let's see another example:

```
#include <stdio.h>
int main(void)
{
    int a, b, c = 0;

    a = (b = 20, b = b+30, b++);
    printf("%d %d\n", a, b);
    if(a, b, c)
        printf("One\n");
    return 0;
}
```

The expression `b = b+30` makes `b` equal to 50. The expression `b++` first stores the current value of `b` into `a`, and then `b` is incremented. Therefore, the program displays: 50 51. Next, since the value of the entire expression is the value of `c`, and because `c` is 0, the `if` condition is false and the program displays nothing.

Because the precedence of the comma operator is the lowest, the parentheses in an assignment are necessary.

For example, if we write: `a = b, b+30;` the value of `a` will be equal to `b` because the `=` operator has higher precedence than comma. The expression `b+30` is evaluated, but its value is thrown away, not assigned.

Because the use of the comma operator may produce code that is more complex and harder to evaluate, it is not often used. The place that it is primarily used is in the **for** statement. For example:

```
int a, b;
for(a = 1, b = 2; b < 10; a++, b++)
```

The first expression first assigns 1 to a, then assigns 2 to b. The third expression first executes a++, then b++.

Notice that the commas we use to separate function arguments, initializers, and variables in declarations are not comma operators, but just separators.

---

## The **sizeof** Operator

The **sizeof** operator is used to compute the number of bytes required to store the value of a particular type, variable, constant, or expression. Notice that the type of the result is **size\_t**, which is defined in C library as unsigned integer (usually as **unsigned int**). Although many programmers use the %d conversion to display the **sizeof** value, the safest practice is to cast the value to the largest of the unsigned types, that is, **unsigned long**, and use the %lu conversion. Here is an example how we'd use cast to display the size of the **float** type:

```
printf("Float:%lu bytes\n", (unsigned long)sizeof(float));
```

For simplicity, when printing **sizeof** values in the next programs, we omit the cast and use the %u conversion. The following program uses the **sizeof** operator to display how many bytes allocate in memory the variables of the program.

```
#include <stdio.h>
int main(void)
{
    char c;
    int i;
    double d;

    printf("%u %u %u\n", sizeof(c), sizeof(i), sizeof(d));
    return 0;
}
```

The program displays: 1 4 8. Notice that if we declare another variable, e.g., **int j**, and write **sizeof(i+j)**, the program will display 4 again because the type of the expression is integer. However, if we write **sizeof(i+d)**, the program will display 8 because, as we learned in Chapter 2, the type of the expression is **double**.

Notice that if the operand is a type (e.g., `int`), parentheses are required, but for a variable, this is not required. However, if we are not using parentheses, the result of an expression such as `sizeof i*j` is not clear. `sizeof` applies first and then the multiplication, or is it the other way around? Because you'll learn always to use parentheses in `sizeof` expressions, as we do to avoid problems, we won't give you the answer. If you insist on finding the answer, consult the precedence table in Appendix A. Nevertheless, parentheses may give the false impression that using `sizeof` is a function call. No, `sizeof` is not a function; it is an operator.

The `sizeof` operator is particularly useful in writing portable code. For example, as we'll see in Chapter 14, in order to allocate memory for, let's say, 100 integers, we write:

```
malloc(100*sizeof(int));
```

In this way, the allocated memory size does not depend on the system. In the next chapters, we'll see its usefulness with other data types, such as arrays and structures.

Here is a quite easy question to introduce you in the `if-else` statement. What does the following program output?

```
#include <stdio.h>
int main(void)
{
    if(sizeof(int) > -1)
        printf("Yes\n");
    else
        printf("No\n");
    return 0;
}
```

Who cares about the **if-else** statement? It is just put there to mislead you. Although the obvious answer is Yes, the program outputs No. We just told you that the type of the **sizeof** value is unsigned, didn't we? Therefore, when the **if** condition is evaluated, -1 is promoted to an unsigned value greater than 4 and the condition becomes false.

---

## The **enum** Type

The **enum** type is used to define an enumeration type, which is a set of named integer constant values. The names must be different from ordinary variables in the same scope, but values need not be distinct. Typically, it is declared as follows:

```
enum tag {enumeration_list};
```

The **tag** is an optional identifier that identifies the enumeration list. For example, the statement:

```
enum seasons {AUTUMN, WINTER, SPRING, SUMMER};
```

defines the **seasons** enumeration type and the enumeration constants **AUTUMN**, **WINTER**, **SPRING**, and **SUMMER**. The names of the constants are subject to C scope rules. For example, if an enumeration is declared inside a function, its constants won't be visible outside the function. We'll talk about scope rules in Chapter 11.

The value of the first constant is 0 by default, although we can set specific values when declaring the enumeration. For example:

```
enum seasons {AUTUMN=10, WINTER, SPRING=30, SUMMER};
```

If a constant is not explicitly assigned a value, it is initialized to the value of the last initialized constant plus one. As a result, the values of **AUTUMN**, **WINTER**, **SPRING**, and **SUMMER** constants in the first example become 0, 1, 2, and 3, while in the second one, they become 10, 11, 30, and 31, respectively.

Recall from Chapter 2 that the **#define** directive is an alternative way to define symbolic names for values. Their main difference is that the **enum** type groups the constants, in order to form a set of values.

In order to declare enumeration variables of a named enumeration type, we write:

```
enum tag variable_list;
```

For example, with the statement: **enum seasons s1, s2;** **s1** and **s2** are declared as enumeration variables of the type **seasons**. Alternatively, we may declare variables when the type is defined. For example:

```
enum seasons {AUTUMN, WINTER, SPRING, SUMMER} s1, s2;
```

In this example, if the tag **seasons** won't be used later to declare extra variables, it can be omitted. However, even if it won't be used, our preference is always to use an enumeration tag, in order to make it easier for the reader to realize the purpose of the enumeration.

The important thing to remember is that C treats enumeration variables and constants as integers.

For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int next_seas;
    enum seasons {AUTUMN = 1, WINTER, SPRING, SUMMER} s;

    printf("Enter season [1-4]: ");
    scanf("%d", &s);

    if(s == SUMMER)
        next_seas = AUTUMN;
    else
        next_seas = s+1;
    printf("Next season = %d\n", next_seas);
    return 0;
}
```

The program reads an integer, stores it into the enumeration variable `s`, and displays the number that corresponds to the next season. Notice that the compiler need not check whether the value stored in an enumeration variable is a valid value for the enumeration. If a value other than its constants is assigned, it is common for a compiler to issue a warning.

Concerning our preference among `enum` and `#define`, we'd say that when we need a set of values to characterize an entity (e.g., seasons), the `enum` type is more representative. And if a function needs that set, it is preferable to pass an enumeration variable than a plain integer. For example:

```
void test(enum codes c); instead of void test(int c);
```

The first prototype tells the reader that the passed value should belong within a set. Therefore, it becomes easier for the reader to read and verify the code. On the other hand, the second prototype does not provide any information about the passed value.

Moreover, the preprocessor removes the names defined with the `#define` directive, so they won't be available during debugging. In general, we use `#define` when the constants we want to define are unrelated, or few, or they don't represent integers.

## Bitwise Operators

Bitwise operators are used to access the individual bits of an integer variable or constant. A bit value is either 0 or 1. Bitwise operators are particularly useful in several applications; in our experience, we've used them in communication protocols, in data encryption algorithms, and in low-level applications that communicate with the hardware. We'll show you some examples later.

When using bitwise operators, it is safer to apply them on **`unsigned`** variables. If you choose not to, make sure you account for the sign bit when performing your calculations.

## The & Operator

The & operator performs the Boolean AND operation on all bits of its two operands. If both corresponding bits are 1, the result bit is 1. Otherwise, the bit is set to 0. For example, the result of `19 & 2` is 2 (we use 8 bits for simplicity):

```
&      00010011 (19)
      00000010 (2)
      -----
      00000010 (2)
```

The & operator is often used to make zero a number of bits. For example, the statement: `a = a & 3;` sets to zero all but the two lower bits of `a`.

Don't confuse the `&&` and `&` operators. For example, if `a=1` and `b=2`, the result of `a && b` is 1, while the result of `a & b` is 0.

## The | Operator

The | operator performs the Boolean OR operation on all bits of its two operands. If both corresponding bits are 0, the result bit is 0. Otherwise, the bit is set to 1. For example, the result of `19 | 6` is 23:

```
|      00010011 (19)
      00000110 (6)
      -----
      00010111 (23)
```

The | operator is often used to set to one a number of bits. For example, the statement: `a = a | 3;` sets to one the two lower bits of `a`.

Don't confuse the `||` and `|` operators. For example, if `a=1` and `b=2`, the result of `a || b` is 1, while the result of `a | b` is 3.

## The ^ Operator

The ^ operator performs the Boolean XOR operation on all bits of its two operands. If the corresponding bits are different, the result bit is 1. Otherwise, the bit is set to 0. For example, the result of `19 ^ 6` is 21:

```
^      00010011 (19)
      00000110 (6)
      -----
      00010101 (21)
```

## The ~ Operator

The `~` complement operator is unary and performs the Boolean NOT operation on all bits of its operand. In particular, it reverses the 1s to 0s and vice versa. For example, the result of `~19` in a 32-bit system is:

```
~ 00000000 00000000 00000000 00010011 (19) =  
11111111 11111111 11111111 11101100
```

## Shift Operators

The shift operators shift the bits of an integer variable or constant left or right. The `>>` operator shifts the bits to the right, while the `<<` operator shifts the bits to the left. Both operators take a right operand, which indicates how many places the bits will be shifted. If it is negative or it has a value equal or greater than the number of the bits of the type of the left operand, the result is undefined. It is the programmer's responsibility to make sure that the right operand has a valid value. For example, if the `int` type has 32 bits and `a` is `int`, it is valid to write `a << 31`, but not `a << 32` or `a << -1`.

The expression `i >> n` shifts the bits in `i` by `n` places to the right. If `i` is positive or its type is `unsigned`, then the `n` high-order bits inserted on the left are set to 0. If `i` is negative, it depends on the implementation whether 1s or 0s are added. For example, what value would `a` have in the following code?

```
unsigned int a, b = 35;  
a = b >> 2;
```

Let's write the number in binary to explain the shifting operation. For simplicity, we'll use 8 bits. Since 35 is `00100011` in binary, the result of `00100011 >> 2` is `00001000`, and that will be the value of `a`. In particular, the two rightmost bits 1 and 1 of the original number are "shifted off" and two zero bits are entered in positions 7 and 8. Notice that the value of `b` remains 35.

---

Since a bit position corresponds to a power of 2, shifting a positive integer `n` places to the right divides its value by  $2^n$ .

---

The expression `i << n` shifts the bits in `i` by `n` places to the left. The `n` low-order bits inserted on the right are set to 0. For example, what value would `a` have in the following code?

```
unsigned int a, b = 35;  
a = b << 3;
```

Since the number 35 is `00100011` in binary, the shifting result of `00100011 << 3` is `00100011000`, and that will be the value of `a`. In particular, the bits of the original number are shifted three places to the left and three zero bits are added in positions 1, 2, and 3. Therefore, `a` becomes 280, while `b` remains 35.

---

If no overflow occurs, shifting a positive number `n` places to the left multiplies its value by  $2^n$ .

---

When you use the `<<` operator to store the shifting result into a variable, make sure the type of the variable is large enough to hold the value. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    unsigned char a = 1;

    a <<= 8; /* Equivalent to a = a << 8; */
    printf("%d\n", a);
    return 0;
}
```

The statement `a <<= 8;` shifts the value of `a` eight places to the left and inserts eight zero bits at the right. Therefore, the shifting result is `100000000`. However, since `a` is declared as `unsigned char`, it can hold only the value of the eight low-order bits. Therefore, the program displays `0` and not `256`, as you might expect. If it were declared as `int`, the program would display `256`.

Let's see another example. If we write:

```
char a = 8;
a <<= 4;
```

the value of `a` is shifted four places to the left and becomes `10000000`. If the `char` type is signed, the leftmost bit is reserved for the sign of the number and `a` becomes `-128`. If `a` were declared as `unsigned char`, its value would be `128`.

*Remember, it is safer to perform bitwise operations on `unsigned` types.*

---

## Exercises

C.4.7 The `^` operator is often used in data encryption. For example, write a program that reads an integer that corresponds to the cipher key and another integer that will be encrypted with that key. The encryption is performed by applying the `^` operator on them. Then, the program should use once more the `^` operator to decrypt the encrypted result.

```
#include <stdio.h>
int main(void)
{
    int num, key;

    printf("Enter key: ");
    scanf("%d", &key);

    printf("Enter number: ");
    scanf("%d", &num);
```

```
    num = num ^ key;
    printf("Encrypted : %d\n", num);

    num = num ^ key;
    printf("Original: %d\n", num);
    return 0;
}
```

**Comments:** Since  $(a \wedge b) \wedge b = a$ , the result of the decryption is the input number.

**C.4.8** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    printf("%d\n", ~(~0 << 4));
    return 0;
}
```

**Answer:** According to the precedence table in Appendix A, the `~` operator has higher precedence than the `<<` operator; therefore, the expression `~0` is evaluated first, and then the result is shifted four places to the left. In a 32-bit system, the result of `~0` is the number with all 32 bits set to 1s. Therefore, the value of `(~0 << 4)` is the number:

```
11111111 11111111 11111111 11110000
```

Since the `~` operator reverses its bits, the value of the entire expression is:

```
00000000 00000000 00000000 00001111 and the program displays 15.
```

**C.4.9** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    unsigned int i = 10;

    if((i >> 4) != 0)
        printf("One\n");
    else
        printf("Two\n");

    printf("%d\n", i);
    i = 1;
    if((i << 3) == 8)
        printf("One\n");
    else
        printf("Two\n");

    printf("%d\n", i);
    return 0;
}
```

**Answer:** Since the value of the expression ( $i >> 4$ ) is 0, the first **if** statement displays Two. Notice that the **if** statement does not change the value of  $i$  to 0; it just tests whether  $i$  shifted four places to the right is 0 or not. Because  $i$  remains 10, the program displays 10.

The second **if** statement tests whether  $i$  shifted three places to the left is 8. Since it is true, the program displays One. As previously,  $i$  does not change, and the program displays 1.

**C.4.10** Write a program that reads an integer and displays a message to indicate whether it is even or odd.

```
#include <stdio.h>
int main(void)
{
    int num;

    printf("Enter number: ");
    scanf("%d", &num);

    if((num & 1) == 1) /* The inner parentheses are necessary for
reasons of priority. We could also write if(num & 1) */
        printf("The number %d is odd\n", num);
    else
        printf("The number %d is even\n", num);
    return 0;
}
```

**Comments:** To determine whether a number is even or odd we check its last digit. If it is 0, the number is even; otherwise, it is odd. Suppose that the variable  $num$  is coded in binary as:

xxxxxxxxx xxxxxxxxx xxxxxxxxx xxxxxxxxx

where each 'x' represents a bit of either 0 or 1. Then, the result of  $num \& 1$  is:

&	xxxxxxxxx	xxxxxxxxx	xxxxxxxxx	xxxxxxxxx
	00000000	00000000	00000000	00000001
	-----			
	00000000	00000000	00000000	0000000x

Therefore, the result is equal to the value of the last bit. If it is 1, the condition in the **if** statement is true and the program displays a message that the number is odd. If it is 0, the condition is false and the program displays a message that the number is even.

Notice that this solution is based on the assumption we've made in Chapter 2 that the system uses the most common method to represent negative numbers, that is, the two's complement. For a more generic and portable solution, you should use the `%` operator.

**C.4.11** Applications that get data from the hardware often need to retrieve the values of certain bits. For example, write a program that reads an integer and a bit position and displays the value of the respective bit. Assume that the user enters a bit position within [1, 32].

```

#include <stdio.h>
int main(void)
{
    unsigned int num, pos;

    printf("Enter number: ");
    scanf("%d", &num);

    printf("Enter bit position [1-32]: ");
    scanf("%d", &pos);

    printf("bit%d is %d\n", pos, (num >> (pos-1)) & 1);
    return 0;
}

```

**C.4.12** Write a program that reads an integer in [0, 255], then it swaps the two quads of its binary digits and displays the new number. For example, if the user enters 10 ( $00001010_2$ ), the program should display 160 ( $10100000_2$ ).

```

#include <stdio.h>
int main(void)
{
    unsigned char num, tmp;

    printf("Enter number [0-255]: ");
    scanf("%d", &num);

    tmp = num & 0xF; /* The value of the four lower bits is stored
into tmp. 0xF is 1111 in binary. Of course, we could use 15 instead of
0xF. */
    tmp <= 4; /* Shift the value four places to the left. */
    tmp += num >> 4; /* Add to tmp the value of num shifted four
places to the right. */
    printf("%d\n", tmp);
    return 0;
}

```

**Comments:** In case you wonder about the value of num, its value does not change. Also, we could omit tmp and write: `printf("%d\n", ((num & 0xF) << 4) + (num >> 4));` or `printf("%d\n", (unsigned char)(num << 4) + (num >> 4));`

**C.4.13** Bitwise operators are often used in data coding in network communications. For example, some specific bits in the header of a Transport Control Protocol (TCP) packet are coded as depicted in Figure 4.1.

Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1
		URG	ACK			SYN	FIN

**FIGURE 4.1**

TCP header information.

FIN (bit1): If it is 1, it indicates the release of the TCP connection.

SYN (bit2): If it is 1, it indicates the establishment of the TCP connection. That's the famous bit responsible for Denial-of-Service (DoS) flooding attacks, but this is another story ...

ACK (bit5): If it is 1, it indicates the acknowledgment of data reception.

URG (bit6): If it is 1, it indicates the transfer of urgent data.

Write a program that reads the values of URG, ACK, SYN, and FIN bits and encodes this information in a program variable. Then, the program should decode the value of this variable and display the values of the respective bits.

```
#include <stdio.h>
int main(void)
{
    unsigned char temp, urg, ack, syn, fin;

    printf("Enter FIN bit: ");
    scanf("%d", &fin);

    printf("Enter SYN bit: ");
    scanf("%d", &syn);

    printf("Enter ACK bit: ");
    scanf("%d", &ack);

    printf("Enter URG bit: ");
    scanf("%d", &urg);

    temp = fin + (syn << 1) + (ack << 4) + (urg << 5);
    printf("\nEncoding: %d\n", temp);

    fin = temp & 1;
    syn = (temp >> 1) & 1;
    ack = (temp >> 4) & 1;
    urg = (temp >> 5) & 1;
    printf("FIN = %d, SYN = %d, ACK = %d, URG = %d\n", fin, syn, ack,
    urg);
    return 0;
}
```

**C.4.14** What does the following program do?

```
#include <stdio.h>
int main(void)
{
    unsigned char ch = 3;

    ch = ((ch&1) << 7) | ((ch&2) << 5) | ((ch&4) << 3) | ((ch&8) << 1)
    | ((ch&16) >> 1) | ((ch&32) >> 3) | ((ch&64) >> 5) | ((ch&128) >> 7);
    printf("%d\n", ch);
    return 0;
}
```

**Answer:** Suppose that ch is coded in binary as  $x_8x_7x_6x_5x_4x_3x_2x_1$ , where each  $x_i$  is either 1 or 0. Therefore, the value of the expression  $(ch \& 1) \ll 7$  is:

$x_8x_7x_6x_5x_4x_3x_2x_1$   
& 0 0 0 0 0 0 0 1  
-----  
0 0 0 0 0 0 0  $x_1 \ll 7 = x_1 0 0 0 0 0 0 0$

Similarly, the value of  $(ch \& 2) \ll 5$  is:

$x_8x_7x_6x_5x_4x_3x_2x_1$   
& 0 0 0 0 0 0 1 0  
-----  
0 0 0 0 0 0  $x_2 0 \ll 5 = 0 x_2 0 0 0 0 0 0$

In a similar way:

The value of  $(ch \& 4) \ll 3$  is: 0 0  $x_3 0 0 0 0 0 0$

The value of  $(ch \& 8) \ll 1$  is: 0 0 0  $x_4 0 0 0 0 0$

The value of  $(ch \& 16) \gg 1$  is: 0 0 0 0  $x_5 0 0 0 0$

The value of  $(ch \& 32) \gg 3$  is: 0 0 0 0 0  $x_6 0 0$

The value of  $(ch \& 64) \gg 5$  is: 0 0 0 0 0 0  $x_7 0$

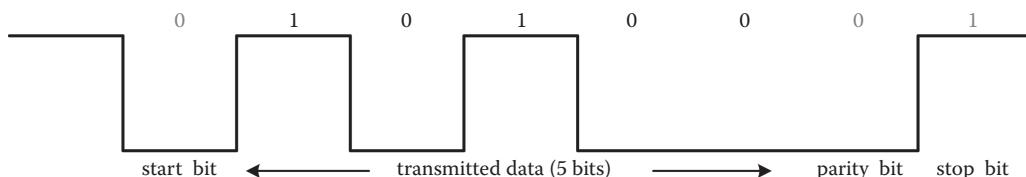
The value of  $(ch \& 128) \gg 7$  is: 0 0 0 0 0 0 0  $x_8$

Therefore, the entire expression is evaluated as:

$(x_1 0 0 0 0 0 0 0) | (0 x_2 0 0 0 0 0 0) | (0 0 x_3 0 0 0 0 0) | (0 0 0 x_4 0 0 0 0) | (0 0 0 0 x_5 0 0 0 0) | (0 0 0 0 0 x_6 0 0 0 0) | (0 0 0 0 0 0 x_7 0) | (0 0 0 0 0 0 0 x_8) = x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8$

In fact, the program reverses the bits of ch, without using a second variable. Therefore, ch ( $00000011_2$ ) becomes 192 ( $11000000_2$ ), and the program displays 192.

**C.4.15** Bitwise operators are often used in applications that communicate with the hardware. For example, a common configuration method used in asynchronous serial communication is eight bit characters, with one *start\_bit*, one *stop\_bit*, and one *parity\_bit*. The transmission of data is signaled with the *start\_bit* set to 0 to inform the receiver for arrival of data. The *stop\_bit* indicates the end of the transmission. For example, the data bits 10100 are encoded as depicted in Figure 4.2.



**FIGURE 4.2**

Communication example.

The sender may optionally transmit a *parity\_bit*, which is used by the receiver to detect transmission errors. There are two variants of parity: even parity and odd parity. In even parity, the value of the *parity\_bit* is calculated in a way that the total number of 1s in data plus the parity bit is even. In case of odd parity, the total number should be odd. In our example, we choose even parity. Therefore, since the number of 1s in data is two, that is, even, the *parity\_bit* is set to 0.

Write a program that simulates the sender. The program should read the values of five data bits and use even parity to generate an 8-bit value.

```
#include <stdio.h>
int main(void)
{
    unsigned char data, ones, bit1, bit2, bit3, bit4, bit5,
    parity_bit;

    ones = 0;

    printf("Enter bit1: ");
    scanf("%d", &bit1);
    if(bit1 == 1)
        ones++;

    printf("Enter bit2: ");
    scanf("%d", &bit2);
    if(bit2 == 1)
        ones++;

    printf("Enter bit3: ");
    scanf("%d", &bit3);
    if(bit3 == 1)
        ones++;

    printf("Enter bit4: ");
    scanf("%d", &bit4);
    if(bit4 == 1)
        ones++;

    printf("Enter bit5: ");
    scanf("%d", &bit5);
    if(bit5 == 1)
        ones++;

    if(ones & 1) /* It means that the number of 1s is odd. */
        parity_bit = 1;
    else
        parity_bit = 0;

    data = (bit1 << 6) + (bit2 << 5) + (bit3 << 4) + (bit4 << 3) +
    (bit5 << 2) + (parity_bit << 1) + 1; /* The last 1 represents the
stop_bit. */

    printf("%d\n", data);
    return 0;
}
```

## Operator Precedence

Each operator has a *precedence* that determines the order in which operators are evaluated. In an expression with several operators, the operators with higher precedence are evaluated first. For example, since the \* operator has higher precedence than the + and - operators, the result of:  $7+5*3-1$  is 21.

When operators of equal precedence appear in the same expression, the operators are evaluated according to their *associativity*. For example, since the \* and / operators have the same precedence and are *left associative*, the result of:  $7*4/2*5$  is 70 because the multiplication  $7*4 = 28$  is executed first, then the division  $28/2 = 14$ , and last the multiplication  $14*5 = 70$ . Let's see another example:

```
#include <stdio.h>
int main(void)
{
    int a = 4, b = 5, c = 3;

    printf("%d\n", a < b > c);

    a = b = c = 2;
    printf("%d\n", a == b == c);

    a += b -= ++c;
    printf("%d %d %d\n", a, b, c);
    return 0;
}
```

According to the precedence table in Appendix A, the < and > operators have the same precedence and are left associative. Therefore, the expression  $(a < b)$  is first evaluated. Since a is 4 and b is 5, its value is 1. Therefore, the expression  $a < b > c$  is equivalent to  $1 > c$ , which is false since c is 3. As a result, the program outputs 0.

Since the == operator is left associative, the expression  $a == b$  is first evaluated. Since a equals b, its value is 1. Therefore, the expression  $a == b == c$  is equivalent to  $1 == c$ . Since c is 2, this expression evaluates to 0 and the program outputs 0. If c were 1, the program would output 1.

The compound operators are executed from right to left. Therefore, the expression  $a += b -= ++c;$  is equivalent to  $a = a + (b -= ++c);$ . By itself, the expression  $(b -= ++c)$  is equivalent to  $b = b - (++c)$ . The value of c is first incremented and becomes 3; therefore,  $b = b - 3 = 2 - 3 = -1$ . As a result, the expression  $a = a + (b -= ++c)$  is equivalent to  $a = a - 1 = 2 - 1 = 1$  and the program outputs: 1 -1 3.

In another example, do the following statements produce the same result? Give it some thought.

```
float a, b = 3, c = 4;
a = 1/2*b*c;
a = b*c*1/2;
```

Since the result of the division  $1/2$  is 0, the first statement assigns 0 to a. Since multiplications are performed first and the numerator is **float**, the second statement assigns 6 to a.

Some operators have precedence other than the expected one or the one we'd like to have. For example, suppose that we want to shift a variable three places to the right and add 10. It'd be plausible to write: `a = b>>3 + 10`; assuming that the evaluation order is from left to right. Unfortunately, that would be wrong; the addition is performed first because the `+` operator has higher precedence. To get the desired result, the expression `b>>3` must be enclosed in parentheses. In another example:

```
if(a & 15 == 0)
    printf("One");
```

Since the `==` operator has higher precedence than the `&` operator, the expression `15 == 0` is first evaluated, which is false, and then the value of `a & 0`, which is false for any value of `a`. As a result, this `if` condition is always false. To get the desired result, the expression `a & 15` must be enclosed in parentheses.

If you are not certain about the evaluation order, use parentheses. The use of parentheses, even when it is not necessary to do so, increases the readability of code and clarifies the way that expressions are evaluated. For example, it is clearer to write: `a = 7+(5*3)-1`; rather than `a = 7+5*3-1`;

The `?:`, `[]`, `->`, `.`, `&`, `*` operators are yet to be discussed in the following chapters.

Hold on one second, didn't we describe the `&` and `*` operators? Are we going to describe them once more? Yes, we like to repeat the same things. Besides, repetition is the mother of learning.

Just kidding, we'll see in Chapter 8 that these two operators can have different meanings. That's why they appear twice in the precedence table. That is, the authors of the standard decided to use the same symbol for different operations. Why did they do that? We don't know; one reason could be the lack of available characters.

## Unsolved Exercises

Don't use the `if` statement in any of the following exercises.

**U.4.1** Write a program that reads two integers and displays the smallest.

**U.4.2** Suppose that a customer in a store buys some things. If the total cost is less than \$100, no discount is due; otherwise, a 5% is offered. Write a program that reads the total cost and displays the amount to be paid.

**U.4.3** Write a program that reads the minimum score required to pass the exams and the grades of three students and displays how many of them succeeded. A grade to be valid should be `<= 10`.

**U.4.4** Continue the previous exercise and find the average grade of those who passed the exams. Assume that one student succeeded, at least.

**U.4.5** Write a program that reads two integers, stores them into two variables, and uses the `^` operator and the equations  $(x^y)^x = y$  and  $(x^y)^y = x$  to swap their values. Use only two variables.

**U.4.6** Write a program that reads an integer and displays the sum of the digits in the positions 2, 4, 6, and 8. For example, if the user enters 170 (binary: 10101010), the program should display 4. Use only one variable.

**U.4.7** Write a program that reads a positive integer and another integer that represents a number of bytes (e.g., n). The program should rotate the input number, n bytes to the right, and display the new value. Assume that the size of the int type is 32 bits and the user enters a valid value for n within [1, 3]. For example, if the user enters 553799974 (binary: 00100001|00000010|01010001|00100110) and 1, the program should display: 639697489 (binary: 00100110|00100001|00000010|01010001).

**U.4.8** Write a program that reads an integer in [0, 255] and the number of shifting bits (e.g., n). The program should display the sum of:

- a. Shifting the input number n places to the left and add the “shifted-off” bits to the right of the number.
- b. Shifting the input number n places to the right and add the “shifted-off” bits to the left of the number.

For example, if the user enters 42 (binary: 00101010) and 3, the program should display the sum of:

**00101010 << 3 = 01010001 = 81** (the shifted-off bits are: 001) and: **00101010 >> 3 = 01000101 = 69** (the shifted-off bits are: 010). The sum is 150.

**U.4.9** Write a program that reads an integer and the positions of two bits and displays the value of the bits between them. The program should find the size of the int type; don't assume that it is 32 bits. For example, if the user enters 176 (binary: 10110000) and the values 6 and 4, the program should display the value of the bits between the positions 6 and 4, that is, the decimal value of the bits 110, which is 6.

**U.4.10** Write a program that reads a positive integer and rounds it up to the next highest power of 2. For example, if the user enters 35, the program should display 64, since  $35 < 64 = 2^6$ . If the user enters 16, the program should display 16.

# Program Control

Up to this point, we have seen that program's statements are executed from top to bottom, in the order that they appear inside the source code. However, in real programming, certain statements should be executed only if specific conditions are met. This chapter will teach you how to use the `if` and `switch` selection statements to control the flow of your program and make it select a particular execution path from a set of alternatives. It also introduces the conditional operator `?:`, which can be used to form conditional expressions.

## The `if` Statement

The `if` statement controls program flow based on whether a condition evaluates to true or false. The simplest form of the `if` statement is:

```
if (condition)
{
    ... /* block of statements */
}
```

If the value of the condition is true, the block of statements between the braces will be executed. For example:

```
int x = 3;
if(x != 0)
{
    printf("Yes\n");
}
```

Since `x` is not 0, the `if` condition is true and the program displays `Yes`.

`if(x)` is equivalent to `if(x != 0)`

If the value of the condition is false, the block of statements won't be executed. For example, the following code displays nothing:

```
int x = -3;
if(x == 0)
{
    printf("Yes\n");
}
```

The position of the braces, not only in case of `if`, but also in the loop statements of Chapter 6, is a matter of personal preference. For example, many programmers prefer the K&R style:

```
if(x == 0){  
    printf("Yes\n");  
}
```

The reason we don't prefer this style is because the left brace is not clearly shown. Our preference is to put each brace in a separate line, so that we can easily check that they match. You may choose the style you like more. However, the common practice, and this is what we strongly recommend, is to indent the inner statements in order to make clearer their grouping and the code easier to read. We use the same style for the loop statements in Chapter 6.

If the block of statements consists of a single statement, the braces can be omitted. For example, the previous code could be written like this:

```
if(x == 0)  
    printf("Yes\n");
```

In general, the braces are used to group declarations and statements into a compound statement, or block, which is treated by the compiler as a single statement. As we'll see in Chapter 6, compound statements are also very common in loops.

## Common Errors

A common error is to add a semicolon (;) at the end of the `if` statement, as you usually do with the most statements. The semicolon is handled as a statement that does nothing, that is, a *null statement*, and the compiler terminates the `if`. For example, in the following code:

```
int x = -3;  
if(x > 0);  
    printf("Yes\n");
```

the ; terminates the `if` statement, so the `printf()` call is not inside `if`. Therefore, this code outputs Yes regardless of the value of x.

A very popular error is to confuse the == equality operator with the = assignment operator. Remember that the == operator checks whether two expressions have the same value, while the = operator assigns a value to a variable. Also remember that the assignment of a nonzero value to a variable corresponds to a true expression, while the assignment of zero value corresponds to a false expression. For example, the following code:

```
int x = -10;  
if(x = -20)  
    printf("Yes\n");
```

outputs Yes although x is -10. Had we written `if(x == -20)`, nothing would have been displayed. In a similar way, the following code:

```
int x = 0;
if(x = 0)
    printf("Yes\n");
```

displays nothing since the assignment of 0 into x makes the condition false.

Although they are not protected when two variables are compared, some programmers prefer to write the constant first when a variable is compared against a constant. For example:

```
if(-20 == x) /* Instead of if(x == -20) */
```

The reason they prefer this syntax is that if they forget one =, the compiler would display an error message for illegal action; it is not allowed to assign a value to a constant. How do we feel about this syntax? Well, we prefer to forget one = and insert a bug than using this style. Besides, bugs are inserted to make the programmer happy when finding them.

Another common error is to write in reverse order the characters of the != operator in a comparison condition. For example:

```
#include <stdio.h>
int main(void)
{
    int i = 30;

    if(i =! 20) /* Instead of != */
        printf("One\n");
    else
        printf("Two\n");
    printf("%d\n", i);
    return 0;
}
```

Because the ! operator is applied to the constant value 20, i becomes 0, so the condition becomes false. Therefore, the program outputs Two and 0.

---

## The if-else Statement

An `if` statement may have an `else` clause:

```
if(condition)
{
    ... /* block of statements A */
}
else
{
    ... /* block of statements B */
}
```

If the condition is true, the first block of statements is executed, otherwise the second one. For example, the following code outputs Two, because x is less than 0.

```
int x = -3;
if(x > 0)
{
    printf("One\n");
}
else
{
    printf("Two\n");
}
```

And, since both blocks consist of a single statement, we could omit the braces:

```
int x = -3;
if(x > 0)
    printf("One\n");
else
    printf("Two\n");
```

---

## Nested if Statements

An **if** statement may contain other **if** and **else** statements, which in turn may contain others, and so on. For example, the following program contains two nested **if** statements:

```
#include <stdio.h>
int main(void)
{
    int a = 10, b = 20, c = 30;

    if(a > 5)
    {
        if(b == 20)
            printf("1 ");

        if(c == 40)
            printf("2 ");
        else
            printf("3 ");
    }
    else
        printf("4\n");

    return 0;
}
```

Since a is greater than 5, the program outputs: 1 3

Because the `else` part is optional, there is an ambiguity when an `else` is omitted from a nested `if` sequence. To match `if` and `else` statements, the rule is that an `else` is associated with the closest unmatched `if` inside the same group of statements. For example, can you apply the rule and tell us what does the following program output?

```
#include <stdio.h>
int main(void)
{
    int a = 5;

    if(a != 5)
        if(a-2 > 5)
            printf("One\n");
    else
        printf("Two\n");
    return 0;
}
```

According to this rule, the `else` statement is associated with the inner `if`. As a result, the program displays nothing because the first `if` condition is false and it has no `else` associated with.

Notice how the indenting can trick you into thinking that the `else` statement is associated with the first `if` statement. By aligning each `else` statement with the matching `if`, indenting the statements, and adding braces to `if-else` statements even if you don't have to, the program becomes easier to read. For example, here is how we could make the previous program more readable by aligning the correct `if` with the matching `else`:

```
#include <stdio.h>
int main(void)
{
    int a = 5;

    if(a != 5)
    {
        if(a-2 > 5)
            printf("One\n");
        else
            printf("Two\n");
    }
    return 0;
}
```

Although the braces of the first `if` are not needed, we add them to make the code more readable. If something else is desired, braces must be used to force the proper association. For example:

```
if(a != 5)
{
    if(a-2 > 5)
        printf("One\n");
}
else
    printf("Two\n");
```

To test a series of conditions, the typical choice is to use the following syntax:

```
if(condition_A)
{
    ... /* block of statements A */
}
else if(condition_B)
{
    ... /* block of statements B */
}
else if(condition_C)
{
    ... /* block of statements C */
}
.
.
.
else /* optional */
{
    ... /* block of statements N */
}
```

There can be any number of **else if** clauses between the first **if** and the last **else**. The conditions are evaluated in order starting from the top. Once a true condition is found, the corresponding block of statements is executed and the remaining **else if** conditions are not evaluated. The program continues with the execution of the next statement after the last **else**. For example, if `condition_A` is true, the first block of statements is executed; if not, `condition_B` is checked. If it is true, the second block of statements is executed; if not, `condition_C` is checked, and so on. If none condition is true and the last **else** is present, its block of statements will be executed. The following program reads an integer and displays a message according to the input value:

```
#include <stdio.h>
int main(void)
{
    int a;

    printf("Enter number: ");
    scanf("%d", &a);

    if(a == 1)
        printf("One\n");
    else if(a == 2)
        printf("Two\n");
    else
        printf("Other\n");

    printf("End\n");
    return 0;
}
```

For example, if the user enters 1, the program will display One; if the number 2 is entered, the program will display Two. If an integer other than 1 or 2 is entered, the program will display Other. In any case, the program continues with the last `printf()` and displays End.

The block of statements in the last `else` is executed only if all previous conditions are false. However, the last `else` is optional. If it is missing and all conditions are false, no action takes place; the program continues with the execution of the next statement after the last `else if`.

For example, in the previous program, if the number 4 is entered and the last `else` was missing, the program would display just End.

## Exercises

C.5.1 Write a program that reads a number and displays its absolute value.

```
#include <stdio.h>
int main(void)
{
    int num;

    printf("Enter number: ");
    scanf("%d", &num);

    if(num >= 0)
        printf("The absolute value is %d\n", num);
    else
        printf("The absolute value is %d\n", -num);
    return 0;
}
```

C.5.2 Write a program that reads two integers and displays them in ascending order.

```
#include <stdio.h>
int main(void)
{
    float i, j;

    printf("Enter grades: ");
    scanf("%f%f", &i, &j);

    if(i < j)
        printf("%f %f\n", i, j);
    else
        printf("%f %f\n", j, i);
    return 0;
}
```

**C.5.3** Write a program that reads two integers and displays their relationship without using an **else** statement.

```
#include <stdio.h>
int main(void)
{
    int i, j;

    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);

    if(i < j)
        printf("%d < %d\n", i, j);
    if(i > j)
        printf("%d > %d\n", i, j);
    if(i == j)
        printf("%d = %d\n", i, j);
    return 0;
}
```

**C.5.4** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = 4, b = 5, c = 3;

    if(a && a/b)
        printf("One\n");
    else
        printf("Two\n");

    if(++a == b++)
        printf("One\n");
    else
        printf("Two\n");

    if(a < b < c)
        printf("One\n");
    else
        printf("Two\n");
    return 0;
}
```

**Answer:** Since the / operator has higher precedence than the && operator, the division is performed first. The result of a/b is 0, not 0.8, because a and b are both integer variables. Therefore, the condition is false and the program displays Two.

The == operator has lower precedence than the ++ operator applied in both forms. The prefix form first increases a and makes it 5. The postfix form increases b, after b is used in the comparison. Therefore, since b is 5, the program displays One and b becomes 6.

According to the precedence rules, the < operator is left associative. Since a is 5 and b is 6, the expression a < b is true, so its value is 1. Therefore, the expression (a < b < c) is

equivalent to  $(1 < c)$ , which is true because  $c$  is 3. Therefore, the program displays One. Remember, if we want to compare  $b$  against  $a$  and  $c$  we would use the `&&` operator and write: `if(a < b && b < c)`

#### C.5.5 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i = 10, j = 20, k = 0;

    if(i = 40)
        printf("One ");
    if(j = 50)
        printf("Two ");
    if(k = 60)
        printf("Three ");
    if(k = 0)
        printf("Four ");
    printf("%d %d %d\n", i, j, k);
    return 0;
}
```

**Answer:** This is an example of how the wrong use of the `=` operator instead of the `==` operator can produce unexpected results. In particular, the `if` conditions don't test the variables for equality, but the variables are assigned with the respective values instead. As a result,  $i$  becomes 40,  $j$  becomes 50, while  $k$  first becomes 60, and then 0. Since the first three `if` conditions are true and the fourth is false (since 0 is assigned to  $k$ ), the program displays One Two Three 40 50 0

#### C.5.6 Write a program that reads two floats (e.g., $a$ and $b$ ) and displays the solution of the equation $a*x + b = 0$ , if any.

```
#include <stdio.h>
int main(void)
{
    double a, b;

    printf("Enter numbers: ");
    scanf("%lf%lf", &a, &b);

    if(a == 0)
    {
        if(b == 0)
            printf("Infinite solutions\n");
        else
            printf("There is no solution !!!\n");
    }
    else
        printf("The solution is %f\n", -b/a);
    return 0;
}
```

**TABLE 5.1**

## Weight Categories

Mass Index	Result
BMI < 20	Lower than normal weight
20 <= BMI <= 25	Normal weight
25 < BMI <= 30	Overweight
30 < BMI <= 40	Obese
40 < BMI	Extremely obese

**C.5.7** Write a program that reads a man's height (in meters) and weight (in kilograms) and calculates his body mass index (BMI) using the formula  $BMI = \text{weight}/\text{height}^2$ . The program should display the BMI and a corresponding message according to Table 5.1, and the lower and upper limit of the normal weight for the given height.

```
#include <stdio.h>
int main(void)
{
    float bmi, height, weight;

    printf("Enter height (in meters): ");
    scanf("%f", &height);

    printf("Enter weight (in kgrs): ");
    scanf("%f", &weight);

    bmi = weight/(height*height);
    printf("***** BMI: %.2f\n", bmi);

    if(bmi < 20)
        printf("Under normal weight\n");
    else if(bmi <= 25) /* Since the previous if checks values up to
20, this condition is equivalent to: else if(bmi >= 20 && bmi <= 25). */
        printf("Normal weight\n");
    else if(bmi <= 30)
        printf("Overweight\n");
    else if(bmi <= 40)
        printf("Obese\n");
    else
        printf("Serious obesity\n");

    printf("According to your height the bounds of normal weight are
[%.2f-%.2f]\n", 20*height*height, 25*height*height); /* The lower and the
upper bmi limits for normal weight are 20 and 25, respectively. */
    return 0;
}
```

**C.5.8** To change the PIN code of a mobile phone, the user is asked to enter the current PIN code and the device checks if that code is equal to the one stored in the SIM card. If they are the same, the user is asked to enter the new PIN code once more for verification, and if it is entered correctly, it is stored in the SIM card. Write a program that simulates this process. Assume that the current code stored in SIM is 1234.

```

#include <stdio.h>
int main(void)
{
    unsigned int tmp, new_code;

    printf("Enter code: ");
    scanf("%d", &tmp);

    if(tmp == 1234)
    {
        printf("Enter new code: ");
        scanf("%d", &tmp);

        printf("Enter new code once more: ");
        scanf("%d", &new_code);

        if(new_code == tmp)
            printf("New code is stored\n");
        else
            printf("Different codes are entered\n");
    }
    else
        printf("Wrong code\n");
    return 0;
}

```

**C.5.9** Write a program that reads the prices of four products and displays the highest one.

```

#include <stdio.h>
int main(void)
{
    double i, j, k, m, max;

    printf("Enter prices: ");
    scanf("%lf%lf%lf%lf", &i, &j, &k, &m);

    if(i > j)
        max = i;
    else
        max = j;

    if(k > max)
        max = k;
    if(m > max)
        max = m;
    printf("Max = %f\n", max);
    return 0;
}

```

**Comments:** To display the lowest price change > to <.

**C.5.10** Write a program that reads the annual income and calculates the due tax according to Table 5.2.

**TABLE 5.2**

## Tax Rates

Income	Tax Rate (%)
0–12,000	0
12,001–14,000	15
14,001–30,000	30
>30,000	40

For example, if the user enters 18000, the tax due is tax: =  $(14000 - 12000) * 0.15 + (18000 - 14000) * 0.3$ .

```
#include <stdio.h>
int main(void)
{
    float income, tax;

    printf("Enter income: ");
    scanf("%f", &income);

    if(income <= 12000)
        tax = 0;
    else if(income <= 14000)
        tax = (income - 12000)*0.15;
    else if(income <= 30000)
        tax = (2000*0.15) + (income - 14000)*0.3; /* 2000 is the
difference of 14000-12000. */
    else
        tax = (2000*0.15) + (16000*0.3) + (income - 30000)*0.4; /* 16000 is the difference of 30000-14000. */
    printf("Tax = %.2f\n", tax);
    return 0;
}
```

**C.5.11** Write a program that reads the grades of three students and displays them in ascending order.

```
#include <stdio.h>
int main(void)
{
    float i, j, k;

    printf("Enter grades: ");
    scanf("%f%f%f", &i, &j, &k);

    if(i <= j && i <= k)
    {
        printf("%f ", i);
        if(j < k)
            printf("%f %f\n", j, k);
        else
            printf("%f %f\n", k, j);
    }
}
```

```

    else if(j < i && j < k)
    {
        printf("%f ", j);
        if(i < k)
            printf("%f %f\n", i, k);
        else
            printf("%f %f\n", k, i);
    }
else
{
    printf("%f ", k);
    if(j < i)
        printf("%f %f\n", j, i);
    else
        printf("%f %f\n", i, j);
}
return 0;
}

```

**Comments:** At first, we compare each grade with the other two, in order to find and display the lowest. Next, we compare the two remaining grades and display them in ascending order.

**C.5.12** The 13-digit International Standard Book Number (ISBN) is a unique code that identifies a book commercially. The last digit is a check digit used for error detection. To calculate its value, each digit of the first 12 digits is alternately multiplied, from left to right, by 1 or 3. The products are summed up and divided by 10. The check digit is the remainder of the division subtracted from 10. If it is 10, it becomes 0. For example, assume that the first 12 digits are: 978960931961.

$$\begin{aligned}
 \text{a. } & (9*1 + 7*3 + 8*1 + 9*3 + 6*1 + 0*3 + 9*1 + 3*3 + 1*1 + 9*3 + 6*1 + \\
 & 1*3) = 126 \\
 \text{b. } & \text{check\_digit} = 10 - (126 \% 10) = 10 - 6 = 4
 \end{aligned}$$

Write a program that reads a 13-digit ISBN code and checks the last digit to verify if it is valid or not.

```

#include <stdio.h>
int main(void)
{
    int dig1, dig2, dig3, dig4, dig5, dig6, dig7, dig8, dig9, dig10,
dig11, dig12, dig13, chk_dig, sum;

    printf("Enter ISBN's digits: ");
    scanf("%d%d%d%d%d%d%d%d%d", &dig1, &dig2, &dig3, &dig4,
&dig5, &dig6, &dig7, &dig8, &dig9, &dig10, &dig11, &dig12, &dig13); /* Since a 13-digit number exceeds the size of the int type, we use a variable for each digit. */

    sum = dig1 + dig2*3 + dig3 + dig4*3 + dig5 + dig6*3 + dig7 +
dig8*3 + dig9 + dig10*3 + dig11 + dig12*3;
    chk_dig = 10 - (sum%10);
}

```

```

    if(chk_dig == 10)
        chk_dig = 0;

    if(chk_dig == dig13)
        printf("Valid ISBN\n");
    else
        printf("Invalid ISBN\n");
    return 0;
}

```

**C.5.13** Suppose that two PCs reside in the same Internet Protocol (IP) network. Write a program that reads their IP addresses (version 4) and displays if they are configured correctly in order to communicate. The form of the IP address is  $x.x.x.x$ , where each  $x$  is an integer within  $[0, 255]$ . The value of the first octet of an IP address defines its class, as follows:

- a. Class A:  $[0, 127]$
- b. Class B:  $[128, 191]$
- c. Class C:  $[192, 223]$

If the two IP addresses indicate different classes, the PCs cannot communicate. If they belong in the same class, we compare their network octet(s). The octet(s) to be compared are defined according to their class, as follows:

- a. Class A: first octet
- b. Class B: first two octets
- c. Class C: first three octets

If they are the same, the PCs may communicate. For example, the PCs with IP addresses 192.168.1.1 and 192.168.1.2 may communicate because they belong in the same class C and the first three octets that specify the network, that is, 192.168.1, are the same.

```

#include <stdio.h>
int main(void)
{
    int a1, a2, a3, a4, b1, b2, b3, b4;

    printf("Enter first IP address: ");
    scanf("%d.%d.%d.%d", &a1, &a2, &a3, &a4);

    printf("Enter second IP address: ");
    scanf("%d.%d.%d.%d", &b1, &b2, &b3, &b4);

    if(a1 < 128)
    {
        if(a1 == b1)
            printf("Class A: Correct Configuration\n");
        else
            printf("Class A: Wrong Configuration\n");
    }
    else if(a1 < 192)

```

```

    {
        if(a1 == b1 && a2 == b2)
            printf("Class B: Correct Configuration\n");
        else
            printf("Class B: Wrong Configuration\n");
    }
    else if(a1 < 224)
    {
        if(a1 == b1 && a2 == b2 && a3 == b3)
            printf("Class C: Correct Configuration\n");
        else
            printf("Class C: Wrong Configuration\n");
    }
    else
        printf("Error: Wrong class\n");
    return 0;
}

```

**Comments:** Notice that we put on purpose the dot in `scanf()` so that the user inputs the IP address in its familiar form.

**C.5.14** In a course exam, each test is graded by two graders. If the difference of their grades is less than  $x$ , the final grade is their average. Otherwise, the test is reviewed by a third grader, as follows:

- If the grade of the third reviewer is equal to the average of the first two grades, that is the final grade.
- If it is less than the minimum (e.g., `min`) of the first two grades, the final grade is `min`.
- Otherwise, the final grade is the average of the grade of the third reviewer and the one of the first two grades closest to it.

Write a program that reads the two grades and the difference  $x$  and displays the final grade according to that procedure.

```
#include <stdio.h>
int main(void)
{
    double g1, g2, g3, fin_grd, avg, min, max, x;

    printf("Enter grades: ");
    scanf("%lf%lf", &g1, &g2);

    printf("Enter difference: ");
    scanf("%lf", &x);
    if(g1 < g2)
    {
        min = g1;
        max = g2;
    }
    else
```

```

    {
        min = g2;
        max = g1;
    }
    avg = (g1+g2)/2;
    if((max-min) < x)
        fin_grd = avg;
    else
    {
        printf("Enter third grade: ");
        scanf("%lf", &g3);
        if(g3 == avg) /* For simplicity, we make a simple comparison.
However, this comparison is not safe as mentioned in Chapter 2. */
            fin_grd = avg;
        else if(g3 < min)
            fin_grd = min;
        else
        {
            if(g3 > avg) /* The max value is closer. */
                fin_grd = (g3+max)/2;
            else
                fin_grd = (g3+min)/2;
        }
    }
    printf("%f\n", fin_grd);
    return 0;
}

```

---

## The Conditional Operator ?:

The conditional operator ?: allows a program to perform one of two actions depending on the value of an expression. The form of a conditional expression using the ?: operator is:

```
exp1 ? exp2 : exp3;
```

Because the conditional operator takes three operands, it is often referred to as a *ternary* operator. If the value of exp1 is true, exp2 will be evaluated, otherwise exp3. The value of the entire conditional will be the value of the evaluated expression, that is, exp2 or exp3. The result type depends on the types of exp2 and exp3. If they are arithmetic of different types, the arithmetic conversions discussed in Chapter 2 are applied and the produced type is the type of the result. For example, if d is **double** and i is **int**, the type of:

```
(j > 0) ? d : i;
```

is **double** regardless of the value of j. Simply put, a conditional expression is a sort of an **if-else** statement:

```
if(exp1)
    exp2;
```

```
else  
    exp3;
```

For example, the following program reads an integer and if it is greater than 10 the program displays One, otherwise Two:

```
#include <stdio.h>  
int main(void)  
{  
    int a;  
  
    printf("Enter number: ");  
    scanf("%d", &a);  
    (a > 10) ? printf("One\n") : printf("Two\n");  
    return 0;  
}
```

The value of a conditional expression can be stored to a variable. For example:

```
int a = 5, k;  
k = (a > 0) ? 100 : -1;
```

Since a is greater than 0, k becomes equal to 100. The parentheses are not necessary because the precedence of ?: is very low. However, it is advisable to use them to make clearer the tested condition.

Typically, the conditional operator is used in place of an **if-else** statement, when it has a simple form. For example, the next **if-else** statement:

```
if(a > b)  
    max = a;  
else  
    max = b;
```

can be replaced with: `max = (a > b) ? a : b;`

Several conditional expressions may be merged by replacing the expression that follows the colon : with another conditional expression. In the following example, `exp3` is replaced with the `add1 ? add2 : add3`; expression:

```
k = exp1 ? exp2 : add1 ? add2 : add3;
```

If `exp1` is true, `k` will be equal to the value of `exp2`. If `exp1` is false, the value of `add1` is evaluated. If `add1` is true, `k` will be equal to the value of `add2`. Otherwise, it will be equal to the value of `add3`. The equivalent chain of **if-else** statements is:

```
if(exp1)  
    k = exp2;  
else if(add1)  
    k = add2;  
else  
    k = add3;
```

## Exercises

C.5.15 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = 10, b = !a;

    a = b ? 50 : 200;
    printf("Val = %d\n", a);
    return 0;
}
```

**Answer:** The statement `a = b ? 50 : 200;` is equivalent to:

`a = (b != 0) ? 50 : 200;` Since `b` is 0, `a` becomes 200 and the program displays:  
Val = 200.

C.5.16 Write a program that reads a student's grade and stores it in a variable only if the input grade is within [5,10]. Otherwise, the variable should be set to -1. Use the conditional operator.

```
#include <stdio.h>
int main(void)
{
    float i, grd;

    printf("Enter grade: ");
    scanf("%f", &grd);

    i = (grd >= 5 && grd <= 10) ? grd : -1;
    printf("%f\n", i);
    return 0;
}
```

C.5.17 Write a program that reads an integer and displays a message to indicate whether it is positive or negative. If it is 0, the program should display Zero. Use the conditional operator.

```
#include <stdio.h>
int main(void)
{
    int num;

    printf("Enter number: ");
    scanf("%d", &num);

    num > 0 ? printf("Pos\n") : (num < 0) ? printf("Neg\n") :
printf("Zero\n");
    return 0;
}
```

**Comments:** Alternatively, we could write:

```
printf("%s\n", num > 0 ? "Pos" : num < 0 ? "Neg" : "Zero");
```

C.5.18 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int a = -10;

    if(((a < 0) ? -a : a) <= 10)
        printf("One %d\n", -a);
    else
        printf("Two %d\n", -a);
    return 0;
}
```

**Answer:** Since a is -10, the result of the conditional expression is -a, that is, 10. Notice that the value of a does not change, it remains equal to -10. Therefore, since the **if** condition is true, the program displays: One 10

C.5.19 The following program reads the prices of three products and displays the highest price. Use the conditional operator to replace the **if-else** statements and a single **printf()** to display the highest price.

```
#include <stdio.h>
int main(void)
{
    float i, j, k;

    printf("Enter prices: ");
    scanf("%f%f%f", &i, &j, &k);

    if(i >= j && i >= k)
        printf("Max = %f\n", i);
    else if(j > i && j > k)
        printf("Max = %f\n", j);
    else
        printf("Max = %f\n", k);
    return 0;
}
```

**Answer:** **printf("Max = %f\n", (i >= j && i >= k) ? i : (j > i && j > k) ? j : k);** To display the lowest price change > to <.

C.5.20 Write a program that reads a student's grade and uses the conditional operator to display a corresponding message according to the next scale:

- If grade is within [7.5–10], the program outputs: A
- If grade is within [5–7.5), the program outputs: B

- c. If grade is within [0–5), the program outputs: Next time
- d. If grade is out of [0, 10], the program outputs: Wrong input

NOTE: The right parenthesis ")" means that the right number is not included in the indicated set.

```
#include <stdio.h>
int main(void)
{
    float grd;

    printf("Enter grade: ");
    scanf("%f", &grd);

    printf("%s\n", (grd >= 7.5 && grd <= 10) ? "A" : (grd >= 5 && grd
< 7.5) ? "B" : (grd >= 0 && grd < 5) ? "Next time" : "Wrong input");
    return 0;
}
```

---

## The **switch** Statement

The **switch** statement can be used as an alternative to an **if-else-if** series, when we want to test the value of an expression against a series of values and handle each case differently. In the most common form, or at least the one we prefer, the syntax of the **switch** statement is:

```
switch(expression)
{
    case constant_1:
        /* block of statements that will be executed if the value of the
expression is equal to constant_1. */
        break;

    case constant_2:
        /* block of statements that will be executed if the value of the
expression is equal to constant_2. */
        break;
    ...
    case constant_n:
        /* block of statements that will be executed if the value of the
expression is equal to constant_n. */
        break;

    default:
        /* block of statements that will be executed if the value of the
expression is other than the previous constants. */
        break;
}
```

As a matter of style, there are several ways to write the **switch** statement. The most popular is to put the statements under the **case** label and the **break** statement some space(s) inner or aligned with the statements. Our preference is to align each **break** with the **case** label and leave a space between each **case** and the preceding **break**.

The expression must be an integer variable or expression in parentheses, and the values of all **constant\_1**, **constant\_2**, ..., **constant\_n** must be integer constant expressions (e.g., 30 is a constant expression, and 30+40 as well) with different values. The order of the cases does not matter. When the **switch** statement is executed, the value of the expression is compared with each **case** constant. If a match is found, the group of statements of the respective **case** will be executed and the remaining cases will not be tested. If no match is found and a **default** case is present, its group of statements will be executed. The **break** statement terminates the execution of the **switch** and the program continues with the next statement after the **switch**. We'll discuss more about the **break** statement in Chapter 6. **switch** statements may be nested, and a **case** or **default** clause is associated with the enclosing **switch**.

For example, the following program reads an integer, and if it is 1, it outputs One; if it is 2, it outputs Two; otherwise, it outputs Other. In any case, the **break** statement terminates the **switch** and the program outputs End.

```
#include <stdio.h>
int main(void)
{
    int a;

    printf("Enter number: ");
    scanf("%d", &a);

    switch(a)
    {
        case 1:
            printf("One\n");
            break;

        case 2:
            printf("Two\n");
            break;

        default:
            printf("Other\n");
            break;
    }
    printf("End\n");
    return 0;
}
```

Notice that the **default** case is optional; it may also appear anywhere in the list of cases, and will be executed if there is no match. If it is present, our suggestion is to put it always at the end to make it clear. If it is missing and there is no matching case, no action takes place.

Let's make a parenthesis for the **default** case. Typically, it is used to report an "impossible" condition. One time, I've chosen to output the message *Panic: Wish you weren't here* paraphrasing the classical song of *Pink Floyd*. As you may guess, this wrong situation had never occurred during the testing phase. Can you imagine when it occurred? After

the delivery of the application to the client, of course. As a matter of fact, I should have expected that I'd be a victim of *Murphy's Law*. And believe me, it is not one of my favorite memories the moment when the client called me in "panic" to ask me about this message. I think I told him to listen to the song and relax. I should have known better; am sure that if I had chosen a happier song the bug wouldn't occur... Something similar had happened to the second author another time, when a message with "inappropriate" content popped up on the screen during a live demonstration. Besides some light relief, the reason we mentioned these incidents is to warn you about the messages you choose to display, when others are going to use your program.

Besides the **break** statement, the **return** statement is often used to exit from the **switch** statement. For example:

```
#include <stdio.h>
int main(void)
{
    int a = 1;

    switch(a)
    {
        case 1:
            printf("One\n");
            return 0;

        case 2:
            printf("Two\n");
            break;
    }
    printf("End\n");
    return 0;
}
```

As we'll see in Chapter 11, the **return** statement not only terminates the **switch** but also the function in which it is contained. Since **a** is 1, the program displays One, and then the **return** statement terminates the program. Therefore, the message End won't be displayed.

The presence of the **break** statement in each **case** is not mandatory. If it is missing from the matching **case**, the program will continue with the execution of the next **case** statements. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int a = 1;

    switch(a)
    {
        case 1:
            printf("One\n");
        case 2:
            printf("Two\n");
    }
}
```

```

        case 3:
            printf("Three\n");
        break;
        default:
            printf("Other\n");
        break;
    }
    printf("End\n");
    return 0;
}

```

Since `a` is 1, the first `case` outputs One. However, since there is no `break`, the `switch` statement is not terminated and the execution falls through to the next cases, until the `break` is met. Therefore, the program outputs Two and Three. The `break` statement terminates the `switch` and the program displays End. If that `break` was missing, the program would also output Other. The last `break` can be omitted, since the right brace } terminates the `switch` anyway. Our preference is to put it, not only to keep a uniform style, but to have it as a safe guard if some day another `case` is added there.

---

Typically, the absence of `break` is a cause of a potential bug. If you deliberately omit it, add a comment to state it explicitly, so that someone who reads your code or even you after a while won't wonder.

If we want to execute the same block of statements in more than one case, we can merge them together. For example:

```

case constant_1:
case constant_2:
case constant_3:
/* block of statements that will be executed if the value of the
expression matches any of constant_1 or constant_2 or constant_3. */
break;

```

In another example, the following program reads an integer that represents a month (1 for January, 12 for December) and displays the season in which the month belongs to.

```

#include <stdio.h>
int main(void)
{
    int month;

    printf("Enter month [1-12]: ");
    scanf("%d", &month);

    switch(month)
    {
        case 12:
        case 1:
        case 2:
            printf("Winter\n"); /* If the input value is 1, 2 or
12, the program outputs Winter. */
            break;
    }
}

```

```
case 3:  
case 4:  
case 5:  
    printf("Spring\n");  
break;  
  
case 6:  
case 7:  
case 8:  
    printf("Summer\n");  
break;  
  
case 9:  
case 10:  
case 11:  
    printf("Autumn\n");  
break;  
  
default:  
    printf("Error: Wrong month\n");  
break;  
}  
return 0;  
}
```

---

## switch versus if

The main disadvantage of the `switch` statement is that we can make tests only for equality, that is, to test whether the value of an expression matches one of the `case` constants. In contrast, we can use `if` to test any kind of condition. Moreover, the `case` constants are restricted to integers and the expression must be an integer variable or expression only. Because characters are treated as small integers they may be used, but floating-point numbers and strings are not permitted. On the other hand, the use of the `switch` statement in place of cascaded `if-else-if` statements may make the program easier to read in some situations. For example, a typical use of the `switch` statement is to handle user choices in a menu of commands.

---

## Exercises

C.5.21 Write a program that simulates a physical calculator. The program should read the symbol of an arithmetic operation and two integers and display the result of the arithmetic operation.

```
#include <stdio.h>  
int main(void)
```

```

{
    char sign;
    int i, j;

    printf("Enter math sign and two integers: ");
    scanf("%c%d%d", &sign, &i, &j);

    switch(sign)
    {
        case '+':
            printf("Sum = %d\n", i+j);
            break;

        case '-':
            printf("Diff = %d\n", i-j);
            break;

        case '*':
            printf("Product = %d\n", i*j);
            break;

        case '/':
            if(j != 0)
                printf("Div = %.3f\n", (float)i/j);
            else
                printf("Second num should not be 0\n");
            break;

        default:
            printf("Unacceptable operation\n");
            break;
    }
    return 0;
}

```

**Comments:** As we'll see in Chapter 9, the character constants are treated as integers and declared within single quotes ' '.

**C.5.22** Write a program that displays the area of a square or a circle based on the user's choice. If the user enters 0, the program should read the side of the square and display its area. If the user enters 1, the program should read the radius of the circle and display its area.

```

#include <stdio.h>
int main(void)
{
    int sel;
    double len;

    printf("Enter choice (0:square 1:circle) : ");
    scanf("%d", &sel);

    switch(sel)

```

```

{
    case 0:
        printf("Enter side length: ");
        scanf("%lf", &len);
        if(len <= 0)
        {
            printf("Error: Wrong length\n");
            return 0;
        }
        printf("Square area is %f\n", len*len);
        break;

    case 1:
        printf("Enter radius: ");
        scanf("%lf", &len);
        if(len <= 0)
        {
            printf("Error: Wrong length\n");
            return 0;
        }
        printf("Circle area is %f\n", 3.14*len*len);
        break;

    default:
        printf("Error: Wrong choice\n");
        break;
}
return 0;
}

```

C.5.23 Write a program that reads a person's sex and height and displays the corresponding description for the height, according to Table 5.3.

```

#include <stdio.h>
int main(void)
{
    int sex;
    float height;

    printf("Enter sex (0:man - 1:woman): ");
    scanf("%d", &sex);
}

```

**TABLE 5.3**

Height Categories

Sex	Height (m)	Result
Male	<1.70	Short
Male	≥1.70 and <1.85	Normal
Male	≥1.85	Tall
Female	<1.60	Short
Female	≥1.60 and <1.75	Normal
Female	≥1.75	Tall

```

printf("Enter height in meters: ");
scanf("%f", &height);

switch(sex)
{
    case 0:
        if(height < 1.70)
            printf("Result: Short\n");
        else if(height < 1.85)
            printf("Result: Normal\n");
        else
            printf("Result: Tall\n");
    break;

    case 1:
        if(height < 1.60)
            printf("Result: Short\n");
        else if(height < 1.75)
            printf("Result: Normal\n");
        else
            printf("Result: Tall\n");
    break;

    default:
        printf("Error: Wrong input\n");
    break;
}
return 0;
}

```

**C.5.24** Write a program that calculates the cost of transporting a passenger's luggage, according to Table 5.4. The program should read the type of the passenger's class and the weight of the luggage and display the cost.

```

#include <stdio.h>
int main(void)
{
    int clas;
    double cost, weight;

```

**TABLE 5.4**

Transport Cost

Class	Weight (lb)	Cost (\$)
Economy	$\leq 25$	0
	$>25 \text{ and } \leq 40$	1.50 for each pound over 25
	$> 40$	2.00 for each pound over 40
Business	$\leq 35$	0
	$>35 \text{ and } \leq 50$	1.25 for each pound over 35
	$> 50$	1.50 for each pound over 50
VIP	$< 60$	0
	$> 60$	30 (fixed cost)

```

printf("Enter class (1-Eco, 2-Business, 3-VIP): ");
scanf("%d", &clas);

printf("Enter weight: ");
scanf("%lf", &weight);

cost = 0; /* Covers all cases where the passenger pays nothing. */
switch(clas)
{
    case 1:
        if(weight > 40)
            cost = 22.5 + 2*(weight-40); /* 22.5 = 1.5 *
15. */
        else if(weight > 25)
            cost = 1.5*(weight-25);
        break;

    case 2:
        if(weight > 50)
            cost = 18.75 + 1.5*(weight-50); /* 18.75 =
1.25 * 15. */
        else if(weight > 35)
            cost = 1.25*(weight-35);
        break;

    case 3:
        if(weight > 60)
            cost = 30;
        break;

    default:
        printf("Error: Wrong traffic class\n");
        return 0;
}
printf("Total cost = %.2f\n", cost);
return 0;
}

```

## Unsolved Exercises

- U.5.1** Write a program that reads the prices of three products and uses the ?: operator to display a message whether one of those costs more \$100 or not. Don't use an **if** statement.
- U.5.2** Write a program that reads two **double** numbers and displays the absolute value of their sum.
- U.5.3** Write a program that reads three integers and checks if they are in successive ascending order (e.g., 5, 6, 7 or -3, -2, -1).

**U.5.4** Write a program that reads three integers, and if the sum of any two of them is equal to the third one, it should display the integers within [0, 10]. Otherwise, the program should read another three integers and display how many of them are multiples of 6 or other than 20.

**U.5.5** A factory produces small and big bottles. A small one costs \$0.008 and a big one \$0.02. For orders of more than \$200 or more than 3000 bottles in total, a discount of 8% is offered. For orders of more than \$600, the discount is 20%. Write a program that reads the number of small and big bottles ordered and displays the total cost.

**U.5.6** Write a program that reads a student's grade and displays its corresponding description, as follows:

- a. (18–20]: Excellent
- b. (16–18]: Very Good
- c. (13–16]: Good
- d. [10–13]: Dangerous Zone
- e. [0–10]: Need Help

**U.5.7** Each Ethernet network card is characterized by a unique 48-bit identifier, which is called Medium Access Control (MAC) address. Typically, the MAC address is represented in hex form. To find the type of the address, we check the value of the first octet at the left. If it is even, the address type is *unicast*. If it is odd, the type is *multicast*. If all octets are 0xFF, the type is *broadcast*. For example:

- a. The FF:FF:FF:FF:FF:FF address is *broadcast*.
- b. The 18:20:3F:20:AB:11 address is *unicast* because 18 is even.
- c. The A3:3F:40:A2:C3:42 address is *multicast* because A3 is odd.

Write a program that reads a MAC address and displays its type. Assume that the user enters the MAC address in x:x:x:x:x:x form, where each x is a hex number.

**U.5.8** Write a program that reads the grades of three students in the lab part and their grades in the theory part. The final grade is calculated as: `lab_grd*0.3 + theory_grd*0.7`. The program should display how many students got a grade between 8 and 10. Don't use more than three variables.

**U.5.9** A water supply company charges the water consumption, as follows:

- a. Fixed amount of \$10.
- b. For the first 30 cubic meters, \$0.6/m<sup>3</sup>.
- c. For the next 20 cubic meters, \$0.8/m<sup>3</sup>.
- d. For the next 10 cubic meters, \$1/m<sup>3</sup>.
- e. For every additional meter, \$1.2/m<sup>3</sup>.

Write a program that reads the water consumption in cubic meters and displays the total cost.

**U.5.10** Write a program that reads a four-digit positive integer and uses the ?: operator to display a message whether it can be read the same in reverse order or not (e.g., the number 7667 can be). Don't use an **if** statement.

**U.5.11** Write a program that reads a floating-point value (e.g.,  $x$ ) and displays the value of  $f(x)$ , as follows:

$$f(x) = \begin{cases} 8 & , x < -5 \\ \frac{1}{x} & , -5 \leq x < 3 \\ x^2 - 4 & , 3 \leq x < 12 \\ \frac{6}{(x - 14)^2} & , x \geq 12 \end{cases}$$

**U.5.12** Write a program that reads four integers and displays the pair with the largest sum. For example, if the user enters 10, -8, 17, and 5, the program should display  $10+17=27$ .

**U.5.13** Write a program that reads the time in the normal form hh:mm:ss and displays how much time is left until midnight (i.e., 24:00:00).

**U.5.14** Use one `printf()` and the ?: operator to write the program of U.5.6.

**U.5.15** Write a program that reads a month number (1 for January, 12 for December) and the day number (for example, if the input month is January, the valid values are from 1 to 31) and displays the date after 50 days. In case of February, the program should prompt the user to enter its days, that is, 28 or 29. For example, if the user enters 3 for month, and 5 for day, the program should display: 4/24.

**U.5.16** Write a program that reads three integers and the user's choice and uses the `switch` statement to support three cases. If the user's choice is 1, the program should check if the integers are different and display a message. If it is 2, it should check if any two of them are equal, and if it is 3, it should display how many of them are within [-5, 5].

**U.5.17** Write a program that reads the numerators and the denominators of two fractions and an integer which corresponds to a math operation (e.g., 1: addition, 2: subtraction,

**TABLE 5.5**

Rental Rates

Vehicle	Cybism (c.c)	Cost per Day (\$)		
		Days 1-2	Days 3-5	Days >5
Moto	≤ 100	30	25	20
	> 100	40	35	30
Auto	≤ 1000	60	55	50
	> 1001	80	70	60

3: multiplication, 4: division) and uses the **switch** statement to display the result of the math operation.

**U.5.18** Write a program that reads the current year and the year of birth and uses the **switch** statement to display the age in words. Assume that the age has not more than two digits. For example, if the user enters 2020 and 1988, the program should display thirty-two.

**U.5.19** Write a program that calculates the cost of renting a vehicle, according to Table 5.5.

The program should read the type of the vehicle and the days to rent it and use the **switch** statement to display the cost. Also, the program should ask the user if an insurance coverage is desired, and if the answer is positive, an extra 5% in the total cost should be charged.

# Loops

Programs often contain blocks of code that must be executed more than once. A statement whose job is to repeatedly execute the same code as long as the value of a controlling expression is true creates an *iteration loop*. In this chapter, we'll discuss about C's iteration statements: `for`, `while`, and `do-while`, which allow us to set up loops, as well as the `break`, `continue`, and `goto` statements, which can be used to transfer control from one point of a program to another.

## The `for` Statement

The `for` statement is one of C's three loop statements. The general form of the `for` statement is:

```
for(exp1; exp2; exp3)
{
    /* a block of statements (loop body), that is repeatedly executed
as long as the value of exp2 is true. */
}
```

The three expressions `exp1`, `exp2`, and `exp3` can be any valid C expressions. The execution of the `for` statement involves the following steps:

1. `exp1` is executed only once. Typically, `exp1` initializes a variable used in the other two expressions.
2. The value of `exp2` is evaluated. Typically, it is a relational condition. If it is false, the loop terminates and the execution of the program continues with the statement after the closing brace. If it is true, the loop body is executed.
3. `exp3` is executed. Typically, `exp3` changes the value of a variable used in `exp2`.
4. Steps 2 and 3 are repeated until `exp2` becomes false.

As with the `if` statement, if the loop body consists of a single statement, the braces can be omitted. The following program uses the `for` statement to display the numbers from 0 to 4.

```
#include <stdio.h>
int main(void)
{
    int a;

    for(a = 0; a < 5; a++) /* The braces can be omitted. */
```

```
{  
    printf("%d ", a);  
}  
return 0;  
}
```

Let's see how it works:

- The statement `a = 0;` is executed.
- The condition `a < 5` is checked. Since it is true, `printf()` is executed and the program displays 0.
- The `a++` statement is executed and `a` becomes 1. The condition `a < 5` is checked again. Because it is still true, `printf()` is executed again and the program displays 1. This process is repeated until `a` becomes 5, at which point the condition `a < 5` becomes false and the loop terminates. As a result, the program displays:  
0 1 2 3 4

Let's ask you, if we write `a != 5` instead of `a < 5` the output would be the same or not?

Yes, it'd be the same. Now, pay attention to that one. A very common error is to use `=` to check if the final value is reached. For example, write `a = 5` instead of `a < 5`. Since this assignment makes the condition true, the loop becomes infinite and displays ..., find it!

Now, what does the following program output?

```
for(a = 0; a > 1; a++)  
{  
    printf("%d\n", a);  
    printf("End\n");  
}
```

Since `a` is 0, the value of `a > 1` is false, the loop body is not executed and the code displays nothing.

The expressions in the `for` statement may contain more than one expression separated with the comma operator `(,)`. It is typically used in the first and third expressions. For example:

```
int a, b;  
for(a = 1, b = 2; b < 10; a++, b++)
```

The expression `a = 1, b = 2` first assigns 1 to `a`, then 2 to `b`, while the expression `a++, b++` first increments `a` and then `b`.

---

As with an `if` statement, a common error is to add accidentally a semicolon `;` at the end of the `for` statement.

For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int i;

    for(i = 2; i < 7; i+=3);
        printf("%d\n", i);
    return 0;
}
```

The semicolon at the end of the `for` statement indicates a *null* statement, a statement that does nothing. Therefore, the compiler assumes that the loop body is empty and `printf()` will be executed only once, after the condition `i < 7` becomes false. Let's trace the iterations:

*First iteration.* Since `i` is 2, the condition `i < 7` (`2 < 7`) is true. Since the loop body is empty, the next statement to be executed is `i+=3` (`i = i+3 = 2+3 = 5`).

*Second iteration.* Since `i` is 5, the condition `i < 7` (`5 < 7`) is still true and the statement `i+=3` makes `i` equal to 8.

*Third iteration.* Since `i` is 8, the condition `i < 7` (`8 < 7`) becomes false and the loop terminates.

Next, `printf()` is executed and the program outputs the value of `i`, that is, 8.

If it is indeed your intention to create an empty loop, it is advisable to put the semicolon on a line by itself, to make it clear to the reader. For example:

```
for(i = 2; i < 7; i+=3)
;
```

Now, the reader won't wonder if the semicolon was accidentally put.

Because in several programs we'll need to generate random numbers, let's make a parenthesis and show you such an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int i, num;

    srand(time(NULL));
    for(i = 0; i < 5; i++)
    {
        num = rand();
        printf("%d\n", num);
    }
    return 0;
}
```

The `rand()` function, together with the `srand()` and `time()` functions, is used to generate random integers each time the program runs. `rand()` returns an integer between 0 and `RAND_MAX`. The value of `RAND_MAX` is defined in `stdlib.h`. The `srand()` function sets the seed for `rand()`. For example, if we want to generate random floating-point numbers in  $[0, 1]$ , we write `(double)rand() / RAND_MAX`. As you may see in Appendix C, in order to use these functions in our program, we must include the `stdlib.h` and `time.h` files. In practice, `rand()` and `srand()` are often used in applications that simulate experiments and gaming applications.

## Omitting Expressions

The three expressions of the `for` statement are optional. In fact, C allows us to omit any or all of them. For example, in the following code, because `a` is initialized before the `for` statement, the first expression can be omitted.

```
int a = 0;
for(; a < 5; a++)
```

Notice that the semicolon before the second expression must be present. In fact, the two semicolons must always be present even if the first and second expressions are both omitted.

In a similar way, we can put the third expression in the loop body and remove it from the `for` statement. For example, in the following code, the third expression `a++` is put inside the body.

```
for(a = 0; a < 5;)
{
    printf("%d ", a);
    a++;
}
```

If the controlling expression is missing, the compiler treats it as always true, and the loop becomes infinite; that is, it never ends. For example, this `for` statement creates an infinite loop:

```
for(a = 0; ; a++)
```

Most programmers omit all three expressions in order to create an infinite loop. For example:

```
for(;;)
```

Next, we'll see that if the first and third expressions are both missing, the `for` statement is equivalent to a `while` statement. For example:

```
for(; a < 5;) is equivalent to: while(a < 5)
```

## Exercises

C.6.1 What is the output of the following program when:

```
#include <stdio.h>
int main(void)
{
    int i, j;

    for(i = j; i < 10; j++)
        printf("One\n");
    return 0;
}
```

- a. j is 10?

**Answer:** Since the condition `i < 10` ( $10 < 10$ ) is false, the loop is not executed and the program displays nothing.

- b. j is 3;

**Answer:** Since `i` is 3, the condition `i < 10` ( $3 < 10$ ) is always true. Therefore, the loop never ends and the program displays One forever, reducing significantly the computer's performance.

C.6.2 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i;
    unsigned int j;

    for(i = 12; i > 2; i-=5)
        printf("%d ", i);
        printf("End = %d\n", i);

    for(j = 5; j >= 0; j--)
        printf("Test\n");
    return 0;
}
```

**Answer:** Since the `for` statement does not contain braces, the compiler assumes that the loop body consists of one statement, which is the first `printf()`. Notice how the indenting can trick you into thinking that the second `printf()` is also included in

the loop body. In fact, this is executed only once, after the loop ends. Let's trace the iterations:

*First iteration.* Since *i* is 12, the condition *i* > 2 (12 > 2) is true and the program displays 12.

*Second iteration.* With the statement *i-=5* (*i* = *i*-5), *i* becomes 7. Therefore, the condition *i* > 2 (7 > 2) is true and the program displays 7.

*Third iteration.* Now, *i* becomes 2. Therefore, the condition *i* > 2 (2 > 2) is false and the loop terminates.

Next, the second `printf()` displays the current value of *i*. Therefore, the program displays: 12 7 End = 2

Here is the second trap. Since *j* is declared as unsigned, it will never become negative (e.g., -1). Therefore, the second loop is infinite and the program outputs Test forever.

**C.6.3** Write a program that reads an integer, and if it is within [10, 20], the program should display the word One a number of times equal to the input integer. Otherwise, the program should read 10 integers and display the negatives.

```
#include <stdio.h>
int main(void)
{
    int i, num;

    printf("Enter number: ");
    scanf("%d", &num);

    if(num >= 10 && num <= 20)
    {
        for(i = 0; i < num; i++)
            printf("One\n");
    }
    else
    {
        for(i = 0; i < 10; i++)
        {
            printf("Enter number: ");
            scanf("%d", &num);

            if(num < 0)
                printf("%d\n", num);
        }
    }
    return 0;
}
```

**Comments:** The braces after the `if` and `else` statements are not needed because only the `for` statement is contained. The reason we are using them is for better readability.

## The break Statement

We have already discussed how a **break** statement terminates the execution of a **switch** statement. It can also be used to terminate a **for**, **while**, or **do-while** loop and transfer control out of the loop. For example:

```
#include <stdio.h>
int main(void)
{
    int i;
    for(i = 1; i < 10; i++)
    {
        if(i == 5)
            break;

        printf("%d ", i);
    }
    printf("End = %d\n", i);
    return 0;
}
```

As long as *i* is not 5, the **if** condition is false and the program displays the values of *i* from 1 to 4. When *i* becomes 5, the **break** statement terminates the loop and the program continues with the execution of the outer **printf()**. Therefore, the program displays: 1 2 3 4 End = 5

Each **break** terminates the execution of the loop or the **switch** statement in which it is contained. For example, what is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i;

    for(i = 1; i < 10; i++)
    {
        switch(i)
        {
            case 4:
                break;

            default:
                if(i == 3)
                    break;
                else
                    printf("%d", i);
                printf("* ");
                break;
        }
        if(i == 6)
            break;
    }
    return 0;
}
```

In the first two iterations, the **default** case outputs 1\* and 2\* and the **break** statement terminates the **switch** statement, not the **for** statement. Be careful now, in the third iteration, it is the **break** statement inside **if** that terminates **switch** and the program displays nothing. In the fourth iteration, it is the **break** statement in the first **case** that terminates **switch** and the program displays nothing. The fifth iteration outputs 5\*, the next one 6\* and the outer **break** terminates the loop. Therefore, the program outputs: 1\* 2\* 5\* 6\*.

And now that we remembered **switch**, here is a new challenge for you; can you tell us what does the following program output?

```
#include <stdio.h>
int main(void)
{
    int i;

    for(i = 1; i < 12; i++)
    {
        switch(i)
        {
            case 1: i += 2;
            case 2: i += 3;
            default: i += 4;
        }
    }
    printf("%d", i);
    return 0;
}
```

As you guessed, we used that indenting in the `switch` statement on purpose to confuse you. In the first iteration, since `i` is 1, the program executes the statement of the first `case` and continues with the next ones, since `break` is missing. Therefore, `i` becomes 10. In the second iteration, because `i` is now 11, only the `default` case is executed and `i` becomes 15. Next, `i` becomes 16, the loop ends, and that is the output value.

---

## The `continue` Statement

The `continue` statement can be used only inside a `for`, `while`, or `do-while` loop. While the `break` statement ends the loop, the `continue` statement terminates the current iteration of the enclosing loop and causes the next iteration to begin. The rest of the loop body is skipped for the current iteration. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int i;

    for(i = 1; i < 10; i++)
    {
        if(i < 5)
            continue;
        printf("%d ", i);
    }
    return 0;
}
```

As long as `i` is less than 5, the `if` condition is true and the `continue` statement terminates the current iteration. The rest of the loop body, that is, `printf()`, is skipped and the program continues with the `i++` statement. When `i` becomes greater or equal to 5, the `continue` statement is not executed and the program displays the value of `i`. As a result, the program outputs: 5 6 7 8 9.

Notice that we can avoid the use of `continue` by testing the reverse condition. For example:

```
if(i >= 5)
    printf("%d ", i);
```

As a matter of style, our preference is to use the `continue` statement rather than reverse logic, in order to make clear to the reader which is the condition that triggers the next iteration. For example, it is clearer to write the following code rather than use reverse logic:

```
if(a >= 5 || b != 10 || c < -8)
    continue;
```

Unlike `break`, the `continue` statement does not apply to `switch`. A `continue` statement inside a `switch`, which is inside a loop, triggers the next loop iteration. Therefore, in

the second example of the previous section, what does the program output if we change the **default** case to:

```
default:  
    if(i == 3)  
        break;  
    else  
        continue;  
printf("* ");  
break;
```

Since the **continue** statement causes the next iteration to begin, the program displays nothing.

---

## Exercises

C.6.4 What is the output of the following program?

```
#include <stdio.h>  
int main(void)  
{  
    int i = 0, j = 5;  
  
    for(i > j; i+j == 5; j < 2)  
    {  
        printf("One\n");  
        i = 4;  
        j = 2;  
    }  
    printf("%d %d\n", i, j);  
    return 0;  
}
```

**Answer:** The value of  $i > j$  is 0; however, this does not affect the execution of the loop. Let's trace the iterations:

*First iteration.* Since  $i$  is 0, the condition  $i+j == 5$  ( $0+5 == 5 == 5$ ) is true and the program outputs One. Next,  $i$  and  $j$  become 4 and 2, respectively.

*Second iteration.* The value of  $j < 2$  is 0; however, this does not affect the execution. Since the condition  $i+j == 5$  ( $4+2 == 6 == 5$ ) becomes false, the loop ends.

The program displays the values of  $i$  and  $j$ , that is, 4 and 2, respectively.

C.6.5 A test consists of 10 multiple choice questions, each of which has three possible answers. The first answer gets three points, the second one point, and the third two points. Write a program that uses the **switch** statement to read the test taker's 10 answers and display the final score.

```

#include <stdio.h>
int main(void)
{
    int i, ans, points;

    points = 0;
    for(i = 0; i < 10; i++)
    {
        printf("Enter answer [1-3]: ");
        scanf("%d", &ans);

        switch(ans)
        {
            case 1:
                points += 3;
                break;

            case 2:
                points += 1;
                break;

            case 3:
                points += 2;
                break;

            default:
                printf("Error: Wrong answer ...\\n");
                break;
        }
    }
    printf("Candidate gets %d points in total\\n", points);
    return 0;
}

```

**C.6.6** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int i, j;

    for(i = 10, j = 2; i != j; i-=2, j+=2)
        printf("%d %d\\n", i, j);
    return 0;
}

```

**Answer:** The loop terminates when i equals j. That happens, after the second iteration, when both become 6. Therefore, the loop is executed twice and the program displays:

10 2  
8 4

**C.6.7** Write a program that reads an integer in  $[0, 170]$  and displays its factorial. The factorial of a positive integer  $n$ , where  $n \geq 1$ , is defined as  $1 \times 2 \times 3 \times \dots \times n$ , while the factorial of 0 equals 1 ( $0! = 1$ ).

```
#include <stdio.h>
int main(void)
{
    int i, num;
    double fact;

    printf("Enter number within [0, 170]: ");
    scanf("%d", &num);

    if(num >= 0 && num <= 170)
    {
        fact = 1; /* This variable holds the factorial of the input
number. It is initialized to 1, to make the multiplications. */
        for(i = 1; i <= num; i++)
            fact = fact*i;
        /* If the user enters 0, the loop won't be executed because
the condition (i <= num) is false (i=1 and num=0). Therefore, the program
would display the initial value of fact, that is, 1. And this is correct,
since  $0! = 1$ . */
        printf("Factorial of %d is %e\n", num, fact);
    }
    else
        printf("Error: Number should be within [0, 170]\n");
    return 0;
}
```

**Comments:** The reason we declared `fact` as `double` is to calculate the factorial of more numbers. Because the maximum value that can be stored in a `double` variable is  $\sim 10^{308}$ , the input is constrained up to 170 since the factorial of 171 exceeds  $10^{308}$ . Had we used the `int` type instead of `double`, the program would have displayed correctly only the factorials of numbers from 0 to 12. Suppose that the user enters 20. Let's trace the first three iterations:

*First iteration.* Since  $i$  is 1, the condition  $i \leq 20$  is true and `fact` becomes  $fact = fact*i = 1*1 = 1$ .

*Second iteration.* Since  $i$  is 2, the condition is still true and `fact` becomes  $fact = fact*i = 1*2 = 2$ .

*Third iteration.*  $i$  becomes 3 and  $fact = fact*i = 2*3 = 6$ .

Therefore, the first three iterations calculate the product  $1 \times 2 \times 3$ . In a similar way, the next iterations calculate the factorial of the input number.

**C.6.8** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i;
```

```

    for(i = 0; i < 0 : i+1; i++)
        printf("%d\n", i);
    return 0;
}

```

**Answer:** In the first iteration, since *i* is 0, the value of the controlling expression is *i+1* = 1. Therefore, the loop is executed and the program displays 0. In the second iteration, *i* becomes 1, so the value of the expression becomes 0 and the loop terminates.

**C.6.9** Write a program that reads an integer and displays its multiplication table. For example, if the user enters 5, the program should output:  $1 \times 5 = 5, 2 \times 5 = 10, \dots, 10 \times 5 = 50$ . The program should force the user to enter an integer within [1, 10].

```

#include <stdio.h>
int main(void)
{
    int i, num;

    for(;;) /* Create an infinite loop. */
    {
        printf("Enter number [1-10]: ");
        scanf("%d", &num);
        if((num >= 1) && (num <= 10))
            break; /* If the user enters a number within [1, 10],
the loop terminates. */
    }
    for(i = 1; i <= 10; i++)
        printf("%d * %d = %d\n", i, num, i*num);
    return 0;
}

```

**C.6.10** Write a program that reads a positive integer *n* and verifies the math formula:  $1 + 3 + 5 + \dots + (2n-1) = n^2$ .

```

#include <stdio.h>
int main(void)
{
    int i, num, sum;

    printf("Enter number > 0: ");
    scanf("%d", &num);
    sum = 0; /* This variable holds the sum of the terms. It is
initialized to 0, to make the additions. */
    for(i = 1; i <= 2*num-1; i+=2)
        sum += i;
    if(sum == num*num)
        printf("Verified\n");
    else
        printf("Not Verified\n");
    return 0;
}

```

**Comments:** Essentially, the left part of the formula calculates the sum of the odd numbers from 1 up to  $2*\text{num}-1$ . Suppose that the user enters 3. Let's trace the iterations:

*First iteration.* Since  $i$  is 1, the condition  $i \leq 5$  is true and  $\text{sum}$  becomes  $\text{sum} = \text{sum} + i = 0 + 1 = 1$ .

*Second iteration.* Since  $i$  becomes  $i+2 = i+2 = 1+2 = 3$ , the condition is still true and  $\text{sum}$  becomes  $\text{sum} = \text{sum} + i = 1 + 3 = 4$ .

*Third iteration.*  $i$  becomes 5 and  $\text{sum} = \text{sum} + i = 4 + 5 = 9$ .

Since  $i$  becomes 7, the loop terminates and the formula is verified.

**C.6.11** Write a program that reads the initial population of a country and its annual population growth (as a percentage). Then, the program should read the number of years and display the new population for each year.

```
#include <stdio.h>
int main(void)
{
    int i, years, pop, pop_incr;
    double rate;

    printf("Enter population: ");
    scanf("%d", &pop);

    printf("Enter increase rate (%%): ");
    scanf("%lf", &rate);

    printf("Enter years: ");
    scanf("%d", &years);

    printf("\nYear\tIncrease\tPopulation\n");
    printf("-----\n");
    for(i = 1; i <= years; i++)
    {
        pop_incr = pop*rate/100; /* Calculate the population's
increase. */
        pop += pop_incr; /* Calculate the new population. */
        printf("%d\t%d\t%d\n", i, pop_incr, pop);
    }
    return 0;
}
```

**C.6.12** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    unsigned char i;

    for(i = 3; i && i-1; i--)
        printf("%d ", i);
```

```

    for(; ++i;)
        printf("\n%d ", i);
    return 0;
}

```

**Answer:** The first loop ends when `i && i-1` becomes false. This happens once `i` becomes 1. Therefore, the loop executes twice and the program outputs 3 and 2. Let's see the second loop. The expression `++i` is equivalent to `++i != 0`. Does this expression ever become 0? Yes, since the type of `i` is `unsigned char`, once it becomes 255 the next increment makes it 0. Therefore, the second loop outputs the values of `i` from 2 up to 255.

**C.6.13** Write a program that reads two integers and displays the sum of the integers between them. For example, if the user enters 3 and 8, the program should display 22 because  $4+5+6+7 = 22$ . The program should check which one of the two input numbers is the greater and act accordingly.

```

#include <stdio.h>
int main(void)
{
    int i, j, sum;

    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);

    sum = 0;
    if(i < j)
    {
        for(i = i+1; i < j; i++)
            sum += i;
    }
    else if(j < i)
    {
        for(j = j+1; j < i; j++)
            sum += j;
    }
    printf("Sum = %d\n", sum);
    return 0;
}

```

**C.6.14** Write a program that reads the grades of 100 students and displays the average of the passed students and the average of the failed students, before it ends. A student passes the exams with a grade equal to or greater than 5. The program should force the user to enter grades within [0, 10]. If the user enters -1, the insertion of grades should end.

```

#include <stdio.h>
int main(void)
{
    int i, suc, fail;
    float grd, sum_suc, sum_fail;

    suc = fail = 0;
    sum_suc = sum_fail = 0;

```

```

for(i = 0; i < 100; i++)
{
    printf("Enter grade: ");
    scanf("%f", &grd);

    if(grd == -1)
        break;
    if(grd > 10 || grd < 0)
    {
        printf("Wrong grade, try again ... \n");
        i--; /* If the input grade is out of [0, 10], the
grade is ignored and i is decremented to repeat the insertion. */
        continue;
    }
    if(grd >= 5)
    {
        suc++;
        sum_suc += grd;
    }
    else
    {
        fail++;
        sum_fail += grd;
    }
}
if(i != 0)
{
    if(suc)
        printf("Avg(+) = %.2f\n", sum_suc/suc);
    else
        printf("Everybody failed\n");

    if(fail)
        printf("Avg(-) = %.2f\n", sum_fail/fail);
    else
        printf("None failed\n");
}
return 0;
}

```

**C.6.15** Write a program that reads 8 bits (each bit is 0 or 1) and displays the corresponding unsigned integer, assuming that the bits are entered from left to right. For example, if the user enters 10000000 the program should display 128.

```

#include <stdio.h>
int main(void)
{
    int i, num, dig;

    num = 0;
    for(i = 7; i >= 0; i--)
    {
        printf("Enter digit_%d (0 or 1): ", i+1);
        scanf("%d", &dig);
    }
}

```

```

        num += (dig << i);
    }
printf("The decimal value = %d\n", num);
return 0;
}

```

C.6.16 Write a program that displays the sum of:  $\frac{1 \times 2}{3 \times 4} + \frac{5 \times 6}{7 \times 8} + \frac{9 \times 10}{11 \times 12} + \dots$  until a term with value greater than 0.995 is going to be added. Also, the program should display how many terms were added and the value of the last valid term.

```

#include <stdio.h>
int main(void)
{
    int i, cnt;
    double sum, term;

    sum = cnt = 0;
    i = 1;
    for(;;)
    {
        term = (double) (i*(i+1))/((i+2)*(i+3));
        if(term > 0.995)
            break;
        cnt++;
        sum += term;
        i += 4;
    }
    i -= 4;
    term = (double) (i*(i+1))/((i+2)*(i+3));
    printf("Sum:%f Terms:%d Last:%f\n", sum, cnt, term);
    return 0;
}

```

C.6.17 What is the output of the following program? Remember that `printf()` returns the number of displayed characters.

```

#include <stdio.h>
int main(void)
{
    int i;

    for(i = 0; printf("%d", i++) < 2;
         ; /* Empty loop. */
    printf("\nEnd = %d\n", i);
    return 0;
}

```

**Answer:** As said, the expressions in the `for` statement can be any valid expression. In the first iteration, `printf()` outputs the value of `i`, that is, 0, and then `i` is incremented by 1. Since `printf()` returns the number of displayed characters, the returned value is 1. Therefore, the loop continues because the condition is true (`1 < 2`).

The same happens with all numbers up to 10. When *i* becomes 10, `printf()` outputs 10 and then *i* becomes 11. Now, since two characters are displayed, `printf()` returns 2, which makes the condition false ( $2 < 2$ ) and the loop terminates. Therefore, the program outputs:

```
012345678910  
End = 11
```

**C.6.18** Write a program that reads the number of students in a class and their grades on a test. The program should display the average test grade of the class, the minimum and maximum test grade, and how many students got the same maximum grade. Assume that the grades are integers within [0, 10].

```
#include <stdio.h>  
int main(void)  
{  
    int i, grd, studs_num, times, min_grd, max_grd, sum_grd;  
  
    printf("Enter number of students: ");  
    scanf("%d", &studs_num);  
  
    if(studs_num <= 0)  
    {  
        printf("Wrong number of students\n");  
        return 0; /* Program termination. */  
    }  
    min_grd = 11;  
    max_grd = -1;  
    sum_grd = 0;  
    for(i = 0; i < studs_num; i++)  
    {  
        printf("Enter grade [0-10]: ");  
        scanf("%d", &grd);  
  
        if(grd < min_grd)  
            min_grd = grd;  
        if(grd > max_grd)  
        {  
            max_grd = grd;  
            times = 1; /* First appearance of the new maximum  
grade. */  
        }  
        else if(max_grd == grd)  
            times++;  
  
        sum_grd += grd;  
    }  
    printf("Min:%d, Max:%d (appeared %d times) Avg:%.2f\n", min_grd,  
max_grd, times, (float)sum_grd/studs_num);  
    return 0;  
}
```

**C.6.19** Consider the following recursive formula:

$$a_n = 2 \times a_{n-1} - a_{n-2} + a_{n-3}, \text{ where } a_0 = 1, a_1 = 2, \text{ and } a_2 = 3.$$

Write a program that reads the value of integer n and displays the value of the n-th term. The program should force the user to enter a number greater than 2.

```
#include <stdio.h>
int main(void)
{
    int i, num, an, an1, an2, an3;

    for(;;)
    {
        printf("Enter number [> 2]: ");
        scanf("%d", &num);
        if(num > 2)
            break;
    }

    an1 = 3;
    an2 = 2;
    an3 = 1;
    for(i = 3; i <= num; i++)
    {
        an = 2*an1 - an2 + an3;
        an3 = an2;
        an2 = an1;
        an1 = an;
    }
    printf("a[%d] = %d\n", num, an);
    return 0;
}
```

**C.6.20** Write a program that reads an integer and displays a message to indicate whether it is a prime number or not. It is reminded that a prime number is any integer greater than 1 with no divisor other than 1 and itself.

```
#include <stdio.h>
int main(void)
{
    int i, num;

    printf("Enter number (>1): ");
    scanf("%d", &num);

    if(num > 1)
    {
        if(num % 2 == 0)
        {
            if(num == 2)
                printf("%d is prime\n", num);
            else
                printf("%d is not prime\n", num);
        }
    }
}
```

```

        return 0;
    }
    for(i = 3; i <= num/2; i+=2)
    {
        if(num % i == 0)
        {
            printf("%d is not prime\n", num);
            return 0; /* Since a divisor is found, it is
not needed to search for other divisors, so the program terminates. */
        }
    }
    printf("The number %d is prime\n", num);
}
else
    printf("Error: Not valid number\n");
return 0;
}

```

**Comments:** If the number is even other than 2, it is not prime and the program terminates. If the number is odd, only the odd divisors are checked. Because any number  $N$  has no divisor greater than  $N/2$ , the program checks if any integer from 2 up to the half of the input number divides that number. If a divisor is found, the program terminates.

Notice that this solution is not the most efficient. It may be further improved. For example, we could check for divisors from 2 up to the square root of the input number, given the math fact, that, if a number  $N$  is not prime, it has at least two divisors greater than 1, and one of them is equal or less than  $\sqrt{N}$ . To find the square root, we may use the `sqrt()` function as described in Appendix C or not use `sqrt()` and compare `i*i` with `num`, instead.

## Nested Loops

When an iteration loop is included in the body of another loop, each iteration of the outer loop triggers the execution of the nested loop. We'll use `for` statements to explain the nested loops. Nested `while` and `do-while` loops are executed in a similar way. For example, let's trace the iterations in the following program:

```

#include <stdio.h>
int main(void)
{
    int i, j;

    for(i = 0; i < 2; i++)
    {
        printf("One ");
        for(j = 0; j < i; j++)
            printf("Two ");
    }
    return 0;
}

```

*First iteration of the outer loop (i=0):* Since the condition ( $i < 2$ ) is true, the program displays One.

*First iteration of the inner loop (j=0):* Since the condition ( $j < i$ ) is false, the loop is not executed.

*Second iteration of the outer loop (i=1):* Another One is displayed.

*First iteration of the inner loop (j=0):* Since the condition ( $j < i$ ) is true, the program displays Two.

*Second iteration of the inner loop (j=1):* Now, the condition is false and the loop terminates.

*Third iteration of the outer loop (i=2):* Since the condition ( $i < 2$ ) is false, the loop terminates.

Therefore, the program displays: One One Two

Let's see another example:

```
#include <stdio.h>
int main(void)
{
    int i, j;

    for(i = 0; i < 2; i++)
    {
        for(j = 0; j < 2; j++)
        {
            if(i+j == 1)
                break;
            printf("Two ");
        }
        printf("One ");
    }
    printf("\nVal1 = %d  Val2 = %d\n", i, j);
    return 0;
}
```

*First iteration of the outer loop (i=0):* Since the condition ( $i < 2$ ) is true, the inner loop is executed.

*First iteration of the inner loop (j=0):* Since the **if** condition is false, the program displays Two.

*Second iteration of the inner loop (j=1):* Now, the **if** condition is true; the **break** statement terminates the inner loop and the program displays One.

*Second iteration of the outer loop (i=1):* The inner loop is executed again.

*First iteration of the inner loop (j=0):* Since the **if** condition is true, the **break** statement terminates the inner loop and the program displays One.

*Third iteration of the outer loop (i=2):* Since the condition ( $i < 2$ ) is false, the loop terminates.

The program also displays the values of  $i$  and  $j$ , that is, 2 and 0, respectively. Therefore, the program displays:

Two One One  
Val1 = 2 Val2 = 0

## Exercises

C.6.21 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i;

    for(i = 0; i < 2; i++)
    {
        printf("One ");
        for(i = 0; i < 2; i++)
            printf("Two ");
    }
    printf("Val = %d\n", i);
    return 0;
}
```

**Answer:** Let's trace the iterations:

*First iteration of the outer loop (i=0):* Since the condition ( $i < 2$ ) is true, the program displays One and the inner loop is executed.

*First iteration of the inner loop (i=0):* Since the condition ( $i < 2$ ) is true, the program displays Two.

*Second iteration of the inner loop (i=1):* Another Two is displayed.

*Third iteration of the inner loop (i=2):* Now, the condition is false and the loop terminates.

*Second iteration of the outer loop (i=3):* Since  $i$  became 2 in the inner loop, the  $i++$  statement makes it 3. Since the condition ( $i < 2$ ) is false, the loop terminates.

As a result, the program outputs: One Two Two Val = 3

C.6.22 Write a program that displays the multiplication table from 1 to 10.

```
#include <stdio.h>
int main(void)
{
    int i, j;

    for(i = 1; i <= 10; i++)
    {
        for(j = 1; j <= 10; j++)
            printf("%d * %d = %d\n", i, j, i*j);

        printf("\n");
    }
    return 0;
}
```

**C.6.23** Write a program that reads an integer that corresponds to a number of lines. The program should display in each line a number of '\*' equal to the line number. For example, if the user enters 5 the program should display:

```
*  
* *  
* * *  
* * * *  
* * * * *  
  
#include <stdio.h>  
int main(void)  
{  
    int i, j, lines;  
  
    printf("Enter lines: ");  
    scanf("%d", &lines);  
  
    for(i = 0; i < lines; i++)  
    {  
        for(j = 0; j <= i; j++)  
            printf("* "); /* Add a space character after '*' */  
        printf("\n"); /* Change line to display the next group of  
        *'s */  
    }  
    return 0;  
}
```

**Comments:** Suppose that the user enters 5. Let's explain the code by tracing the first two iterations of the outer loop.

*First iteration of the outer loop (i=0):* Since the condition ( $i < 5$ ) is true, the inner loop is executed.

*First iteration of the inner loop (j=0):* Since the condition ( $j \leq i$ ) is true, the program displays \*.

*Second iteration of the inner loop (j=1):* Now, the condition is false; the loop terminates and `printf()` changes line.

*Second iteration of the outer loop (i=1):* Since the condition ( $i < 5$ ) is true, the inner loop is executed.

*First iteration of the inner loop (j=0):* Since the condition ( $j \leq i$ ) is true, the program displays \* and leaves a space.

*Second iteration of the inner loop (j=1):* Another \* is displayed.

*Third iteration of the inner loop (j=2):* Now, the condition is false; the loop terminates and `printf()` changes line.

So far, the program has displayed:

```
*  
* *
```

The following lines are produced in a similar way.

**C.6.24** Write a program that reads the grades of five students in three different courses and displays the average grade of each in the three courses, as well as the average grade of all students in all courses.

```
#include <stdio.h>

#define LESSONS      3
#define STUDENTS     5

int main(void)
{
    int i, j;
    float grd, stud_grd, sum_grd;

    sum_grd = 0;
    for(i = 0; i < STUDENTS; i++)
    {
        printf("***** Student_%d\n", i+1);
        stud_grd = 0; /* This variable holds the sum of a student's
grades in all courses. It is initialized to 0 for each one. */
        for(j = 0; j < LESSONS; j++)
        {
            printf("Enter grade for lesson %d: ", j+1);
            scanf("%f", &grd);
            stud_grd += grd;
            sum_grd += grd; /* This variable holds the sum of all
grades. */
        }
        printf("Average grade for student_%d is %.2f\n",
               i+1, stud_grd/LESSONS);
    }
    printf("\nTotal average grade is %.2f\n",
           sum_grd/(STUDENTS*LESSONS));
    return 0;
}
```

**C.6.25** Write a program that displays the integer roots, if any, of the equation:  $3x + 7y - 5z = 10$ , where  $x, y, z$  are integers within  $[-30, 30]$ .

```
#include <stdio.h>
int main(void)
{
    int x, y, z, flag;

    flag = 1;
    for(x = -30; x <= 30; x++)
        for(y = -30; y <= 30; y++)
            for(z = -30; z <= 30; z++)
```

```

        if(3*x + 7*y - 5*z == 10)
    {
        printf("Solution: %d %d %d\n", x, y, z);
        flag = 0;
    }
    if(flag)
        printf("No solution\n");
    return 0;
}

```

---

## The `while` Statement

The `while` statement is the simplest way to create iteration loops in C. Its form is:

```

while(exp)
{
    /* block of statements (loop body) that is repeatedly executed as
long as the value of exp is true. */
}

```

As with the `if` and `for` statements, if the loop body consists of a single statement, the braces can be omitted.

When a `while` statement is executed, the value of the controlling expression `exp` is evaluated first. If it is false, the `while` loop is not executed. If it is true, the loop body is executed and `exp` is evaluated again. If it is false, the `while` loop terminates. If not, the loop body is executed again. This process is repeated until `exp` becomes false. For example, the following program uses the `while` statement to display the integers from 10 to 1:

```

#include <stdio.h>
int main(void)
{
    int i = 10;

    while(i != 0)
    {
        printf("%d\n", i);
        i--;
    }
    return 0;
}

```

*Notice that we could write `while(i)` instead of `while(i != 0)`. Similarly, `while(!i)` is equivalent to `while(i == 0)`.*

The **for** and **while** statements are closely related. In fact, the **for** statement:

```
for (exp1; exp2; exp3)
{
    statements;
}
```

is equivalent to:

```
exp1;
while (exp2)
{
    statements;
    exp3;
}
```

unless the **for** loop contains a **continue** statement.

To answer in the question whether to use the **while** or the **for** statement, given that both produce equivalent code, we'd say that it is largely a matter of personal preference. Typically, the **while** statement is used when the number of the iterations is unknown or when a condition is tested and no initialization takes place or a step is used. For example:

```
while (fscanf(fp, "%d", &i) != 1) /* This statement will be explained in
Chapter 15. */
```

When the number of the iterations is predetermined and a step is used, it is preferable to use the **for** statement because its compact form makes the loop easier to read. In several cases, however, although the **for** statement may seem the better choice, an "awkward" code might be produced. The use of **for** in C.6.14 is such an example. If the user enters a wrong value, the step changes inside its body, which is an "awkward" programming choice for our taste. If that **for** statement is replaced with a **while** statement, the code becomes more elegant. Do it and see the difference.

If **exp** is always true, the loop is executed forever, unless its body contains an exit statement (e.g., **break**). For example, the following loop is infinite:

```
while (1)
    printf("One\n");
```

By convention, most programmers use the value 1 to create an infinite loop. However, any nonzero constant may be used.

As with the **for** statement, a semicolon at the end of the **while** statement declares an empty loop body. For example, putting a semicolon after the parentheses:

```
int a = 10;
while (a != 0);
    a--;
```

makes the loop infinite, and the statement `a--;` will never be executed.

Here is one last example before continuing with the exercises. What does the following program output?

```
#include <stdio.h>
int main(void)
{
    int i, sum;

    i = sum = 0;
    while(i < 5)
    {
        sum += 10;
        i = i+2;
    }
    printf("Sum = %d\n", sum);
    return 0;
}
```

Trace the iterations, make the calculations carefully and give an answer; should be easy for you.

Caught you sleeping? Is 30 your answer?

Who told you that the `=+` is a compound operator? You were warned in Chapter 4 that the `=` in a compound operator comes second, weren't you? Therefore, the statement `sum =+ 10;` is equivalent to `sum = +10;`, which merely assigns 10 to sum, and that is the output value.

Basically, the main purpose of this example is to show you that when reading code, even if it is few lines, imagine the case of thousands, it is very easy to skip an easy bug. Sometimes, the easier to insert a bug, the harder to find it. Just remember, when writing code, you should be particularly concentrated, cautious, and in a good mood. Continue with the exercises, watch out for the next traps though.

---

## Exercises

C.6.26 Write a program that reads integers continuously and displays them until the user enters 0. The number 0 must not be displayed.

```
#include <stdio.h>
int main(void)
{
    int i = 1; /* Initialize with a nonzero value, just to be sure
that the loop will be executed. */
    while(i != 0)
    {
        printf("Enter number: ");
        scanf("%d", &i);
        if(i != 0)
            printf("Num = %d\n", i);
    }
    return 0;
}
```

C.6.27 Find the number of iterations in the following program.

```
#include <stdio.h>
int main(void)
{
    int a = 256, b = 2;

    while(a != b)
    {
        b = b*b;
        a >>= 2;
    }
    return 0;
}
```

## Answer:

*First iteration.* Since  $a = 256$  and  $b = 2$ , the condition is true, so  $b$  becomes 4 and  $a$ , after being shifted two places to the right, becomes 64.

*Second iteration.* The condition is still true, so  $b$  and  $a$  become both 16.

*Third iteration.* Since  $a$  and  $b$  are equal, the condition becomes false and the loop ends.

**C.6.28** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i = -2;

    while(i-6)
    {
        printf("One ");
        i++;
        while(!(i+1))
        {
            printf("Two ");
            i--;
        }
        i += 2;
    }
    return 0;
}
```

**Answer:** As with the nested **for** loops, each iteration of the outer **while** loop triggers the execution of the nested loop. Notice that **while**( $i-6$ ) is equivalent to **while**( $i-6 \neq 0$ ) and **while**( $!(i+1)$ ) is equivalent to **while**( $i+1 == 0$ ). Let's trace the first iteration of the outer loop:

*First iteration of the outer loop:* Since  $i$  is -2, the condition ( $i-6 \neq 0$ ) is true, so the program displays One and  $i$  becomes -1.

*First iteration of the inner loop:* Since  $i$  is -1, the condition ( $i+1 == 0$ ) is true, so, the program displays Two and  $i$  becomes -2.

*Second iteration of the inner loop:* Since  $i$  is -2, the condition ( $i+1 == 0$ ) is false and the loop terminates.

Next, the statement `i+=2;` makes  $i$  equal to 0. Eventually, the program outputs: One  
Two One One

**C.6.29** Write a program that reads an integer continuously and displays the word Hello a number of times equal to the input integer. If the user enters a negative number,

the insertion of integers should end and the program should display the total number of the displayed Hello. Use only **while** loops.

```
#include <stdio.h>
int main(void)
{
    int i, num, times;

    times = 0;
    while(1)
    {
        printf("Enter number: ");
        scanf("%d", &num);
        if(num < 0)
            break;
        i = 0;
        while(i < num)
        {
            printf("Hello\n");
            i++;
        }
        times += num;
    }
    printf("Total number is = %d\n", times);
    return 0;
}
```

C.6.30 Write a program that reads the prices of a shop's products continuously until the user enters -1. The program should display the lowest price, the highest, and the average of those prices within [5, 30], before it terminates. Assume that the highest price is \$100.

```
#include <stdio.h>
int main(void)
{
    int set_prc;
    float min, max, prc, sum_prc;

    set_prc = 0; /* This variable counts the products whose price is
within the specified set. */
    sum_prc = 0; /* This variable holds the sum of the prices within
the specified set. */
    min = 100;
    max = 0;
    while(1)
    {
        printf("Enter price: ");
        scanf("%f", &prc);
        if(prc == -1)
            break;

        if(prc >= 5 && prc <= 30)
        {
            sum_prc += prc;
            set_prc++;
        }
    }
}
```

```

        if(max < prc)
            max = prc;
        if(min > prc)
            min = prc;
    }
    printf("\nMin = %f, Max = %f\n", min, max);
    if(set_prc != 0)
        printf("Avg = %.2f\n", sum_prc/set_prc);
    else
        printf("No product is included\n");
    return 0;
}

```

**C.6.31** Write a program that reads an integer and displays the number of its digits and their sum. For example, if the user enters 1234, the program should display 4 and 10 ( $1+2+3+4 = 10$ ).

```

#include <stdio.h>
int main(void)
{
    int num, sum, dig;

    sum = dig = 0;

    printf("Enter number: ");
    scanf("%d", &num);

    if(num == 0)
        dig = 1;
    else
    {
        while(num != 0)
        {
            sum += num % 10; /* Add the last digit. */
            num /= 10; /* Remove the last digit. */
            dig++;
        }
    }
    if(sum < 0)
        sum = -sum;
    printf("%d digits and their sum is %d\n", dig, sum);
    return 0;
}

```

**C.6.32** Write a program that reads an integer in [0, 255] continuously and displays it in binary. For example, if the user enters 32, the program should display 00100000. For any value out of [0, 255] the program should terminate.

```

#include <stdio.h>
int main(void)
{
    int i, num;

```

```

while(1)
{
    printf("\nEnter number: ");
    scanf("%d", &num);
    if(num < 0 || num > 255)
        break;

    for(i = 7; i >= 0; i--) /* Check each bit. */
    {
        if((num >> i) & 1)
            printf("1");
        else
            printf("0");
    }
}
return 0;
}

```

**C.6.33** Write a program that reads a positive integer and displays the largest positive integer n for which the sum  $1^2 + 2^2 + 3^2 + \dots + n^2$  is less than the given number.

```

#include <stdio.h>
int main(void)
{
    int i, num, sum;

    printf("Enter number: ");
    scanf("%"d", &num);
    if(num <= 0)
    {
        printf("Error: The number should be positive\n");
        return 0;
    }
    sum = 0;
    i = 1;
    while(1)
    {
        sum += i*i;
        if(sum >= num)
            break;
        i++;
    }
    printf("The last number is = %d\n", i-1); /* The number i-1 is the
last number where the value of sum is still less than the given number. */
    return 0;
}

```

**C.6.34** The final grade in a course is calculated as 30% of the lab grade and as 70% of the exam's grade, only if both grades are greater than or equal to 5; otherwise, the final grade will be the lesser. Write a program that reads continuously pairs of grades (lab and exam grades) and displays the final grade for each student, until the user enters a pair of grades containing the value -1. The program should display the average grade of all students in the course, before it ends. The program should force the user to enter grades within [0, 10].

```

#include <stdio.h>
int main(void)
{
    int studs;
    float sum_grd, stud_grd, grd_exc, grd_exam;

    studs = 0;
    sum_grd = 0;
    while(1)
    {
        printf("-----\n");

        printf("Enter exercise grade [0-10]: ");
        scanf("%f", &grd_exc);

        printf("Enter exam grade [0-10]: ");
        scanf("%f", &grd_exam);

        if(grd_exc == -1 || grd_exam == -1)
            break;

        if((grd_exc < 0) || (grd_exc > 10) ||
           (grd_exam < 0) || (grd_exam > 10))
        {
            printf("Error: Grades should be in [0-10]\n");
            continue;
        }
        studs++;
        if(grd_exc >= 5 && grd_exam >= 5)
            stud_grd = 0.3*grd_exc + 0.7*grd_exam;
        else
        {
            if(grd_exc <= grd_exam)
                stud_grd = grd_exc;
            else
                stud_grd = grd_exam;
        }
        printf("Student grade = %.2f\n", stud_grd);
        sum_grd += stud_grd;
    }
    if(studs)
        printf("\nAverage grade = %.2f\n", sum_grd/studs);
    return 0;
}

```

**C.6.35** Write a program that reads continuously a month number (1=Jan, 12=Dec) and the first day of the month (1=Mon, 7=Sun) and displays the calendar for that month. If the input month is February, the program should prompt the user to enter its days, that is, 28 or 29. If the given month is out of [1, 12], the program should terminate.

```

#include <stdio.h>
int main(void)
{
    int i, mon, mon_days, day, rows;

```

```

while(1)
{
    printf("\n\nEnter month: ");
    scanf("%d", &mon);

    if(mon < 1 || mon > 12)
        break;

    if(mon == 2)
    {
        printf("Enter Feb days: ");
        scanf("%d", &mon_days);
    }
    else if(mon==4 || mon == 6 || mon == 9 || mon == 11)
        mon_days = 30;
    else
        mon_days = 31;

    printf("Enter start day (1=Mon,7=Sun): ");
    scanf("%d", &day);

    printf("Mon\tTue\tWed\tThu\tFri\tSat\tSun\n");
    for(i = 1; i < day; i++) /* Add some spaces up to the first
day of the month to format the output. */
        printf("\t");

    rows = 0;
    for(i = 1; i <= mon_days; i++)
    {
        printf("%d\t";
        if(i == 8-day+(rows*7))
        {
            printf("\n");
            rows++;
        }
    }
}
return 0;
}

```

**Comments:** The **if** statement in the last **for** statement checks whether the last day of the week is reached or not. If it is, a new line character is added. The **rows** variable counts how many day rows have been displayed so far. For example, if the user enters 3 as the first month day (**day** = 3), the first new line character will be added when **i** becomes  $8 - \text{day} + (\text{rows} * 7) = 8 - 3 + (0 * 7) = 5$ . Next, **rows** becomes 1. The next new line characters will be added when **i** becomes 12, 19, and 26, respectively.

**C.6.36** Write a program that simulates a theater's ticket office. Assume that the seats are 300. The program should display a menu to perform the following operations:

1. Buy a ticket. The cost is \$15 for adults and \$10 for minors. There is a 10% discount for the purchase of more than three tickets.

2. Display the total income and the number of the free seats.
3. Program termination.

```
#include <stdio.h>

#define SEATS 300

int main(void)
{
    int sel, adults, tkts, rsvd_seats;
    float cost, tot_cost;

    rsvd_seats = 0;
    tot_cost = 0;
    while(1)
    {
        printf("\nMenu selections\n");
        printf("-----\n");

        printf("1. Buy Ticket\n");
        printf("2. Show Information\n");
        printf("3. Exit\n");

        printf("\nEnter choice: ");
        scanf("%d", &sel);
        switch(sel)
        {
            case 1:
                printf("\nHow many tickets would you like to
buy? ");
                scanf("%d", &tkts);
                if(tkts + rsvd_seats > SEATS)
                {
                    printf("\nSorry, the available seats are
%d\n", SEATS - rsvd_seats);
                    break;
                }
                while(1)
                {
                    printf("\nHow many adults? ");
                    scanf("%d", &adults);

                    if(adults <= tkts)
                        break;
                    else
                        printf("Error: Wrong number of
persons\n");
                }
                cost = adults*15 + (tkts-adults)*10;
                if(tkts > 3)
                    cost *= 0.9;
                tot_cost += cost;
                rsvd_seats += tkts;
        }
    }
}
```

```

        if(rsvd_seats == SEATS)
    {
        printf("\nNot available seats. Income:
%.2f\n", tot_cost);
        return 0;
    }
    break;

    case 2:
        printf("\nFree seats: %d Income: %.2f\n\n",
SEATS - rsvd_seats, tot_cost);
        break;

    case 3:
        return 0;

    default:
        printf("\nWrong choice\n");
        break;
    }
}
return 0;
}

```

## The do-while Statement

Unlike the **for** and **while** statements, where the controlling expression is tested *before* the execution of the loop body, the **do-while** statement tests the controlling expression *after* each execution of the loop body. Therefore, a **do-while** loop is executed at least once. It has the form:

```

do
{
    /* block of statements (loop body) that is executed at least once
and then repeatedly executed as long as the value of exp is true. */
} while(exp);

```

The loop body is executed first, then `exp` is evaluated. If it is false, the loop terminates. Otherwise, the loop body is executed again and `exp` is tested once more. If it is false, the loop terminates. If not, the loop body is executed again. This process is repeated until `exp` becomes false.

A **do-while** statement must end with a ;.

For example, the following program uses the **do-while** statement to display the integers from 1 to 10:

```
#include <stdio.h>
int main(void)
```

```

{
    int i = 1;

    do
    {
        printf("%d\n", i);
        i++;
    } while(i <= 10);

    return 0;
}

```

Although the braces around a single statement can be omitted, we always use them; otherwise, the reader might be mistaken and consider the `while` part to be a separate `while` statement. For example:

```

do
    printf("%d\n", i);
while(++i <= 10);

```

In practice, the `do-while` loop is used less often than the `for` and `while` loops, since it can be replaced by them. Typically, it is used to check the validity of the input values. In the following exercises, we could use `for` or `while` statements; we use the `do-while` statement, in order to show you how to use it.

---

## Exercises

C.6.37 Write a program that reads an integer and displays the word This a number of times equal to the input integer.

```

#include <stdio.h>
int main(void)
{
    int i, num;

    printf("Enter number: ");
    scanf("%d", &num);
    i = 1;
    do
    {
        printf("This\n");
        i++;
    } while(i <= num);

    return 0;
}

```

C.6.38 Write a program that reads integers continuously and displays the square of the even numbers until the user enters a number in [10, 20]. The program should display how

many positive and negative numbers were entered and how many are in [300, 500], before it ends. Zero is counted as neither a positive nor a negative number.

```
#include <stdio.h>
int main(void)
{
    int i, pos, neg, cnt;

    pos = neg = cnt = 0;
    do
    {
        printf("Enter number: ");
        scanf("%d", &i);

        if((i & 1) == 0)
            printf("Num = %d\n", i*i);
        if(i > 0)
        {
            pos++;
            if(i >= 300 && i <= 500)
                cnt++;
        }
        else if(i < 0)
            neg++;
    } while(i < 10 || i > 20);

    printf("Pos = %d Neg = %d Cnt = %d\n", pos, neg, cnt);
    return 0;
}
```

C.6.39 Write a program that reads an integer  $N > 3$  and calculates the result of the following math expression. The program should force the user to enter an integer greater than 3.

$$\Pi = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots \frac{1}{N}$$

```
#include <stdio.h>
int main(void)
{
    int i, num;
    double a, val;

    do
    {
        printf("Enter number > 3: ");
        scanf("%d", &num);
    } while(num <= 3);

    val = 0;
    a = 1;
    for(i = 1; i <= num; i++)
    {
        val += a/i;
```

```

        a = -a;
    }
printf("Val = %e\n", val);
return 0;
}

```

**C.6.40** Write a program that a teacher may use to test if the students know the multiplication table. The program should generate two random values within [1, 10] (e.g., a and b) and display  $a \times b$  = (the smaller number should appear first). The program should prompt the student to enter the result and display an informative message to indicate whether the answer is correct or not. If the student enters -1, the program should display the total number of correct and wrong answers and then terminate.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int i, j, num, fails, wins;

    fails = wins = 0;
    srand(time(NULL));
    do
    {
        i = rand()%10+1; /* rand() returns a random positive integer
and the % operator constrains it in [0, 9]. We add one to constrain it in
[1, 10]. */
        j = rand()%10+1;
        if(i < j)
            printf("\n%d\t%d=", i, j);
        else
            printf("\n%d\t%d=", j, i);

        scanf("%d", &num);
        if(num != -1)
        {
            if(num == i*j)
            {
                printf("Correct\n");
                wins++;
            }
            else
            {
                printf("Wrong (answer=%d)\n", i*j);
                fails++;
            }
        }
    } while (num != -1);

    printf("Fails = %d, Wins = %d\n", fails, wins);
    return 0;
}

```

## The `goto` Statement

The `goto` statement is used to transfer control to another statement within the same function, provided that this statement has a label. Its syntax is:

```
goto label;
```

When the `goto` statement is executed, the program transfers control to the statement that follows the label. A label is named as a variable and it must be unique in the function where the `goto` statement is used. Its name must be followed by a colon :. The scope of the label is the entire function. For example, consider the following program. If the user enters -1, the `goto` statement transfers the execution to the START label and the loop is repeated.

```
#include <stdio.h>
int main(void)
{
    int i, num;
START:
    for(i = 0; i < 5; i++)
    {
        printf("Enter number: ");
        scanf("%d", &num);
        if(num == -1)
            goto START; /* We could write i=-1; and get the same
result; it is just an example of how to use goto. */
    }
    return 0;
}
```

Usually, it is better to avoid the use of `goto` because the transition of the program's execution from one point to another and then to another and so on leads to obscure, hard-to-read and complex code that is hard to maintain and modify. In fact, most programmers oppose its use, arguing that it has no place in a well-structured program.

However, `goto` can be helpful in several cases such as to group a block of statements when something goes wrong. For example:

```
if(error_1)
    goto ERROR;
...
if(error_2)
    goto ERROR;
...
ERROR:
... /* Error handling statements. */
```

As another example, since `break` terminates only the `switch` or loop statement in which it is contained, `goto` can be helpful to exit from a nested `switch`, or from a `switch` inside a loop, or from a deeply nested loop. For example:

```
for(i = 0; i < 10; i++)
    for(j = 0; j < 20; j++)
        for(k = 0; k < 30; k++)
```

```
{  
    if (condition)  
        goto NEXT;  
}  
  
NEXT:  
...  
...
```

Although it should be used rarely, don't forget that `goto` can simplify coding in some situations. Ignore statements like "never use `goto`," "`goto` is only for rookies," "no place for `goto` in structural programming," and disparaging comments like these.

Before ending this chapter, let's discuss again the issue of writing readable code. Needless to say, that it is very important to write clear code, particularly in case of large applications. When a company wants to hire a programmer, it might be more useful to ask the candidate to write a short essay and test his or her writing skills in how he or she expresses himself or herself, if the text is clear, concise, well structured, balanced, without repetitions, ..., than to ask him or her what the type of `p` in a declaration such as `int *(*p[5])(int*)` is (don't worry, we'll answer it in Chapter 11) or what the value of `sizeof('a')` is. Even if he or she cannot answer those questions, it is very easy to get that knowledge. On the other hand, learning to organize his or her thoughts in a straightforward, comprehensive, concise, nontiring way is a very tough task. In high school, we underestimated the value of the course "How to Write an Essay"; as programmers, we realized its great significance.

---

## Unsolved Exercises

**U.6.1** Write a program that displays all numbers from 111 to 999, except those beginning with 4 or ending with 6.

**U.6.2** Write a program that reads integers continuously and calculates their sum until it exceeds 100. Then, the program should display the sum and how many numbers were entered.

**U.6.3** What is the output of the following program?

```
#include <stdio.h>  
int main(void)  
{  
    unsigned char i;  
  
    for(i = 1; i <= 256; i*=4)  
        printf("One\n");  
    return 0;  
}
```

**U.6.4** Write a program that reads two numbers that represent miles (e.g., `a` and `b`) and a third number (e.g., `step`). The program should display one column with the miles and a second one with the corresponding kilometers starting from `a` up to `b` (assume that `a < b`) with a step of `step`. 1 mile = 1.6 km.

**U.6.5** What is the output of the following program? You are reminded that `printf()` returns the number of the displayed characters.

```
#include <stdio.h>
int main(void)
{
    int i, sum;
    float j = 1.2345;

    sum = 0;
    for(i = 0; i < 3; i++)
        sum = sum + printf("%.2f\n", j);

    printf("Val = %d\n", sum);
    return 0;
}
```

**U.6.6** Write a program that reads continuously daily temperatures and displays the percentage of those with a temperature equal, less, or more than 40°F. If the user enters 1000, the insertion of temperatures should terminate.

**U.6.7** Write a program that reads up to 100 integers. If an input number is greater than the last entered, the insertion of numbers should terminate and the program should display how many numbers were entered.

**U.6.8** Write a program that reads three integers (e.g., a, b, and c) one after the other, not all together. The program should force the user to enter the numbers in descending order (i.e., a > b > c).

**U.6.9** Write a program that reads the initial height from which a ball is thrown and displays after how many bounces the ball reaches a height less than  $\frac{1}{4}$  of the initial height. The program should prompt the user to enter the decrease height ratio (%). For example, the input value 0.1 means that each time the ball hits the ground, it bounces up to a height 10% less than the previous one.

**U.6.10** Write a program that reads integers continuously until the user enters 0. Then, the program should display the largest positive and the smallest negative input numbers. If the user enters only positive or negative numbers, the program should display an informative message. Zero is counted as neither a positive nor a negative number.

**U.6.11** Write a program that it is mandatory to read 10 positive numbers. If an input number is negative, the program should prompt the user to enter another one. The program should display how many negative numbers were entered, before it terminates. Use a `for` loop. Zero is counted as neither a positive or negative number.

**U.6.12** Write a program that reads 10 integers and displays how many times the user entered successive values. For example, if the user enters: -5, 10, 17, -31, -30, -29, 75, 76, 9, -4, the program should display 3 due to the pairs: {-31, -30}, {-30, -29}, and {75, 76}.

**U.6.13** How many terms should be added by the series:

$$S_1 = 4 + 10 + 16 + 22 + \dots$$

$$S_2 = 1 + 8 + 15 + 22 + \dots$$

so that  $S_2$  becomes at least 100 greater than  $S_1$ . Use a **do-while** loop.

**U.6.14** Write a program that reads an integer and displays the number of its bits set. For example, if the user enters 30 (in binary: 000000000000000000000000001110), the program should display 4.

**U.6.15** Write a program that uses `rand()` to generate the integers 0 and 1 in a random manner. The program should read an integer that corresponds to `rand()` calls and displays the maximum number of successive 0 and 1 pairs in the generated sequence. For example, if the user enters 15, `rand()` should be called 15 times, and if, let's say, the generated sequence is 010011010100100, the program should display 2 because that is the maximum number of pairs with successive 01.

**U.6.16** What is the output of the following program? Explain why.

```
#include <stdio.h>
int main(void)
{
    int i, j, k = 100;

    for(i = 0; i < 2; i++)
    {
        printf("One ");
        for(j = 0; k; j++)
        {
            printf("Two ");
            k -= 50;
        }
    }
    return 0;
}
```

**U.6.17** Write a program that reads an integer that corresponds to a number of lines. The program should display in the first line a number of '\*' equal to the input number, and one less in each next line. For example, if the user enters 5 the program should display:

```
*****
 ****
 ***
 **
 *
```

**U.6.18** Write a program that reads an integer that corresponds to a number of lines. Suppose that the user enters 4. The program should add spaces and '\*' according to the line number, as shown in the following (three spaces at the left of one '\*' in the first line, two spaces at the left of two '\*' in the second line, and so on). When the last line is

reached, the program should display in each line one '\*' less, and the spaces at the right of '\*', until one '\*\*' is displayed.

```
*  
**  
***  
****  
***  
**  
*
```

**U.6.19** Write a program that reads two integers (e.g., M, N) and produces a M×N grid. Each grid cell should be 3×2 characters wide. As an example, a 3×5 grid follows:

```
+---+---+---+---+  
|   |   |   |   |  
+---+---+---+---+  
|   |   |   |   |  
+---+---+---+---+  
|   |   |   |   |  
+---+---+---+---+
```

The three horizontal characters of each cell are: --- and the two verticals are: |

**U.6.20** Write a program that reads a positive integer (e.g., N) and calculates the sum of:

$$\frac{5}{3} + \frac{25}{9} + \frac{125}{27} + \dots$$

until it becomes greater than N. The program should display the last valid sum and the number of its terms, before it terminates.

**U.6.21** Write a program that reads an integer (e.g., N) and displays the sum of:

$$\frac{1}{1 \times 3} + \frac{1}{3 \times 5} + \frac{1}{5 \times 7} + \dots + \frac{1}{(N-2) \times N}.$$

The program should force the user to enter an odd number greater than or equal to 3.

**U.6.22** Write a program that reads an integer (e.g., N) and displays the sum of:  $2^2 + 4^2 + 6^2 + \dots + (2 \times N)^2$ . The program should force the user to enter a positive integer less than 20. Use one loop.

**U.6.23** Write a program that reads 100 integers and displays the integer that was given the most successive times. If more than one number was given the same most successive times, the program should display the last one.

**U.6.24** Write a program that reads 100 integers and displays the two different higher values.

**U.6.25** Write a program that reads an integer and displays the prime numbers that are less than or equal to it. You are reminded that a prime number is any integer greater than 1 with no divisor other than 1 and itself.

**U.6.26** Write a program that reads an integer and displays its digits in words. If the number is negative, it should begin with the word `minus`. The program should accept integers up to five digits. For example, if the user enters `-12`, the program should display `minus one two`.

**U.6.27** Write a program that reads a positive integer and checks if it is an Armstrong number. An Armstrong number is a number that it is equal to the sum of its digits raised to the power of the number of digits. For example, the three-digit number `153` is an Armstrong number, because  $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$ .

# Arrays

The variables we've used so far can store a single value. In this chapter, we'll talk about a new type of variable capable of storing a number of values. This type is called array. An array may be multidimensional. We'll focus on the simplest and most usual kinds: the one-dimensional and two-dimensional arrays. To introduce you to arrays, we'll show you how to use arrays of integers and floating-point numbers. We'll discuss other types of arrays, as well as their close relationship to pointers in later chapters. In Chapter 12, we'll describe some popular algorithms for searching a value in an array and sorting its elements.

## One-Dimensional Arrays

An array is a data structure that contains a number of values, or else elements, of the same type. Each element can be accessed by its position within the array.

### Declaring Arrays

To declare a one-dimensional array, we must specify its name, the number of its elements, and its data type, like this:

```
data_type array_name[number_of_elements];
```

The `number_of_elements`, or else the length of the array, is specified by an integer constant expression greater than 0 enclosed in brackets. A popular error is to use a variable whose value is set during the program execution. No, it is not allowed. All the elements are of the same type; it may be of any type (e.g., `int`, `float`, `char`, ...). For example, the statement:

`int arr[1000];` declares the array `arr` with 1000 elements of type `int`. To realize the importance of arrays, imagine that if C didn't support this type of aggregate variable, we'd have to declare 1000 different integer variables.

The length of an array cannot change during the program execution. It remains fixed.

If the length of the array is used several times in the program, a good practice is to use a macro instead. If you ever need to change it, you just change the macro. For example:

```
#define SIZE 150
float arr[SIZE]; /* The compiler replaces SIZE with 150 and creates an
array of 150 floats. */
```

Unlike macros, it is an error to use a `const` constant to specify the length. For example, the following declaration is illegal:

```
const int size = 150;
float arr[size]; /* Illegal. */
```

When declaring an array, the compiler allocates a memory block to store the values of its elements. These values are stored one after another in consecutive memory locations. Typically, this memory block is allocated in a region called *stack*, and it is automatically released when the function that declares the array terminates. For example, with the statement `int arr[10]`; the compiler allocates 40 bytes to store the values of the 10 integer elements. To find the size of the allocated memory, we use the `sizeof` operator (e.g., `sizeof(arr)`).

---

The maximum memory size that can be allocated for an array depends on the available memory in the stack.

For example, the following program:

```
#include <stdio.h>
int main(void)
{
    double arr[300000];
    return 0;
}
```

may not run in your computer, unless the available stack size is large enough to hold the values. In Chapter 14, we'll see how we can allocate the required memory from a larger area, called *heap*.

---

When memory is scarce, don't declare an array with more length than needed, in order to avoid waste of memory.

## Accessing Array Elements

To access an element, we write the array's name followed by the element's index inside brackets. The index specifies the position of the element within the array, and it must be an integer constant, variable, or expression. In an array of  $n$  elements, the first one is stored in position `[0]`, the second one in position `[1]`, and the last one in `[n-1]`. For example, the statement:

```
float grd[1000];
```

declares the array `grd` with 1000 elements of type `float`, that is, `grd[0]`, `grd[1]`, ..., `grd[999]`. An element can be used just like an ordinary variable. Here are some examples:

```
int i, j, a[10], b[10];
```

```
a[0] = 2; /* The value of the first element becomes 2. */
a[9] = a[0]; /* The value of the last element becomes equal to the value
of the first element, that is, 2. */
```

```
i = j = a[0]+2; /* The values of i and j become equal to 4. */
a[i+j] = 300; /* Since i+j = 4+4 = 8, the value of the ninth element
becomes 300. */
b[i+1] = a[j]; /* The value of the sixth element of array b becomes equal
to the value of the fifth element of array a.*/
```

Avoid side effects in indexing. For example, don't write something like this: `b[i] = a[i++]`; It depends on the compiler when `i` is incremented.

---

A popular error made by novice programmers is to forget that index numbering starts from 0. Just remember, in an array of  $n$  elements, the first element is stored in position 0, and the last one in position  $n-1$ , not  $n$ .

Another common error occurs when we want to copy one array into another. For example, it looks pretty natural to write `b = a;` However, this plausible assignment is illegal. In Chapter 8 and when investigating the close relationship between arrays and pointers, you'll see why. The simplest way to make the copy is to use a loop and copy the elements one by one or use a copy function, such as `memcpy()`.

The following program declares an array of five integers, assigns the values 10, 20, 30, 40, and 50 to its elements, and displays those greater than 20.

```
#include <stdio.h>
int main(void)
{
    int i, arr[5];

    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;
    for(i = 0; i < 5; i++)
        { /* The braces are not necessary; we use them to make the code
clearer. */
            if(arr[i] > 20)
                printf("%d\n", arr[i]);
        }
    return 0;
}
```

---

C does not check if the index is out of the array bounds. It is the programmer's responsibility to ensure that this won't happen. If it does, the behavior of the program is unpredictable.

For example, in the previous program, the valid indexes are from 0 to 4. However, if you write `arr[5] = 500;` the compiler won't raise an error message; it allows the execution of the program. In fact, this statement changes the content of the memory address just after the end of the array. If this memory is used by another variable, overwriting its

value may cause the program to behave unpredictably. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    int i, j = 20, arr[3];

    for(i = 0; i < 4; i++)
        arr[i] = 100;

    printf("%d\n", j);
    return 0;
}
```

Since arr contains three elements, the valid indexes are from 0 to 2. Therefore, in the last iteration (i=3) the statement arr[3] = 100; assigns a value to an element out of the array's bounds. In particular, the value 100 overwrites the content of the memory just after arr[2]. If this memory is allocated for j, j changes, and the program will output 100 instead of 20!!!

Never ever forget, exceeding an array's bounds poisons the program. Play loud *Strychnine*, kind of poison, from *Sonics* (nice sound from *Fuzztones* as well) to stick it in your mind. Later, you'll encounter a similar exercise, watch out.

## Array Initialization

Like ordinary variables, an array can be initialized when it is declared. Uninitialized elements get the arbitrary values of their memory locations, just like an uninitialized variable. In the most common form, the array initializer is a list of values enclosed in braces and separated by commas. The list is allowed to end with a comma. The values must be constant expressions, variables are not allowed. For example, with the declaration:

```
int arr[4] = {10, 20, 30, 40};
```

the values of arr[0], arr[1], arr[2], and arr[3] become 10, 20, 30, and 40, respectively.

If the initialization list is shorter than the number of the elements, the remaining elements are set to 0. It is illegal to be either empty or longer. For example, with the declaration:

```
int a[10] = {10, 20}; the values of a[0] and a[1] become 10 and 20, and the remaining elements a[2] to a[9] are set to 0.
```

If the array's length is omitted, the compiler will create an array with length equal to the number of the values in the list. For example, with the statement:

```
int arr[] = {10, 20, 30, 40};
```

the compiler creates an array of four integers and assigns the values 10, 20, 30, and 40 to its elements.

If we want the elements of an array, whether it is one-dimensional or multidimensional, to remain the same during program execution, we use the **const** qualifier. In that case, the array must be initialized when it is declared. For example, with the declaration:

```
const int b[] = {10, 20, 30, 40};
```

the b array is declared as **const**, and the compiler will produce an error message in an attempt to change the value of any element. For example, it is illegal to write `b[0] = 80;`.

## Exercises

C.7.1 What would be the values of the array arr in the following program?

```
#include <stdio.h>
int main(void)
{
    int i, arr[10] = {0};

    for(i = 0; i < 10; i++)
        arr[++i] = 20;
    return 0;
}
```

**Answer:** When arr is declared, its elements are initialized to 0. The statement `arr[++i] = 20;` first increments i by one and then makes `arr[i]` equal to 20. The **for** statement increments i once more. Therefore, the elements with even index, that is, `arr[0], arr[2], ...` remain 0, whereas the odd ones, that is, `arr[1], arr[3], ...` become 20.

C.7.2 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    unsigned char arr[5];
    int i;

    for(i = 0; i < 5; i++)
    {
        arr[i] = 256+i;
        printf("%d ", arr[i]);
    }
    return 0;
}
```

**Answer:** Since the number 256 is encoded in 9 bits ( $100000000_2$ ) and the type of arr is **unsigned char**, only the 8 lower bits of the expression `256+i` will be stored. For example, when i is 1, only the 8 lower bits of the number 257 ( $100000001_2$ ) will be stored into `arr[1]`. Therefore, `arr[1]` becomes 1. As a result, the program displays: 0 1 2 3 4

**C.7.3** Write a program that declares an array of five elements and uses a **for** loop to assign the values 1.1, 1.2, 1.3, 1.4, and 1.5 to them. Then, the program should display the array's elements in reverse order.

```
#include <stdio.h>
int main(void)
{
    int i;
    double arr[5];

    for(i = 0; i < 5; i++)
        arr[i] = 1.1 + (i*0.1);
    for(i = 4; i >= 0; i--)
        printf("%f\n", arr[i]);
    return 0;
}
```

**Comments:** Another solution could be to replace the first loop with:

```
arr[0] = 1.1;
for(i = 1; i < 5; i++)
    arr[i] = arr[i-1] + 0.1;
```

**C.7.4** What would be the values of the array **a** in the following program?

```
#include <stdio.h>
int main(void)
{
    int i, a[3] = {4, 2, 0}, b[3] = {2, 3, 4};

    for(i = 0; i < 3; i++)
        a[b[i]-a[2-i]]++;
    return 0;
}
```

**Answer:** When **a** is declared, we have **a[0]** = 4, **a[1]** = 2, and **a[2]** = 0. Similarly, when **b** is declared we have **b[0]** = 2, **b[1]** = 3, and **b[2]** = 4. Let's trace the iterations:

*First iteration (i=0). a[b[0]-a[2]]++ = a[2-0]++ = a[2]++ = 1.*

*Second iteration (i=1). a[b[1]-a[1]]++ = a[3-2]++ = a[1]++ = 3.*

*Third iteration (i=2). a[b[2]-a[0]]++ = a[4-4]++ = a[0]++ = 5.*

Therefore, **a[0]**, **a[1]**, and **a[2]** become 5, 3, and 1, respectively.

**C.7.5** Write a program that reads 10 integers and stores them in an array. Then, the program should check if the array is symmetric, that is, if the value of the first element is equal to the last one, the value of the second one is equal to the value of the last but one, and so on.

```
#include <stdio.h>
#define SIZE 10

int main(void)
```

```

{
    int i, a[SIZE] ;

    for(i = 0; i < SIZE ; i++)
    {
        printf("Enter element a[%d]: ", i);
        scanf("%d", &a[i]);
    }
    for(i = 0; i < SIZE/2 ; i++)
        if(a[i] != a[SIZE-1-i])
    {
        printf("Non symmetric array\n");
        return 0; /* Since we found out that the array is not
symmetric, the program terminates. */
    }
    printf("Symmetric array\n");
    return 0;
}

```

**Comments:** If the number of the elements is odd (e.g., SIZE is 11), would you change something in the code?

Since the middle element is not compared with another element, it does not affect the array's symmetry. Therefore, this code works for both odd and even number of elements.

**C.7.6** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int i, j = 30, arr[] = {j, j-10, j-20, j-30, j-40, j-50, j-60};

    for(i = 0; arr[i]; i++)
        printf("%d\n", arr[i]);
    return 0;
}

```

**Answer:** The `arr[i]` condition in the `for` statement is equivalent to `arr[i] != 0`. Since the value of the fourth element is 0, the loop displays the values of the first three elements, that is, 30, 20, and 10, and terminates.

**C.7.7** Write a program that declares an array of 10 floats, assigns random values within [0, 1] to its elements, and displays their average. To remember how to use `srand()` and `rand()`, read again the example in the `for` statement in Chapter 6.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 10

int main(void)

```

```

{
    int i;
    double sum, arr[SIZE];

    sum = 0; /* Initialize with 0 the variable that calculates the sum
of the array elements. */
    srand(time(NULL));
    for(i = 0; i < SIZE; i++)
    {
        arr[i] = (double)rand()/RAND_MAX;
        sum += arr[i];
    }
    printf("Avg = %f\n", sum/SIZE);
    return 0;
}

```

**C.7.8** What would be the values of the array `a` in the following program?

```

#include <stdio.h>
int main(void)
{
    int i, a[3] = {0, 1, 2};

    for(i = 1; i < 4; i++)
        a[a[a[3-i]]] = i-1;
    return 0;
}

```

**Answer:** When `a` is declared, we have  $a[0] = 0$ ,  $a[1] = 1$ , and  $a[2] = 2$ . Let's trace the iterations:

*First iteration (i=1):  $a[a[a[2]]] = a[a[2]] = a[2] = i-1 = 1-1 = 0$ .*

*Second iteration (i=2):  $a[a[a[1]]] = a[a[1]] = a[1] = i-1 = 2-1 = 1$ .*

*Third iteration (i=3):  $a[a[a[0]]] = a[a[0]] = a[0] = i-1 = 3-1 = 2$ .*

Therefore,  $a[0]$ ,  $a[1]$ , and  $a[2]$  become 2, 1, and 0, respectively.

**C.7.9** Write a program that reads the grades of 100 students and stores them in an array. Then, the program should display the average, the best, and the worst grade, as well as the positions in the array of their first occurrences. The program should force the user to enter grades within [0, 10].

```

#include <stdio.h>

#define SIZE 100

int main(void)
{
    int i, min_pos, max_pos;
    float sum, min_grd, max_grd, grd[SIZE];

    sum = 0;
    max_grd = -1;
    min_grd = 11;

```

```

for(i = 0; i < SIZE; i++)
{
    printf("Enter grade in [0-10]: ");
    scanf("%f", &grd[i]);
    /* There are several ways to check whether the input grade
is valid. In C.7.14, we'll use do-while. */
    while(grd[i] < 0 || grd[i] > 10)
    {
        printf("Error - Enter new grade in [0-10]: ");
        scanf("%f", &grd[i]);
    }
    sum += grd[i];
    if(grd[i] > max_grd)
    {
        max_grd = grd[i];
        max_pos = i; /* Store the position with the best
grade. */
    }
    if(grd[i] < min_grd)
    {
        min_grd = grd[i];
        min_pos = i; /* Store the position with the worst
grade. */
    }
}
printf("Avg: %.2f H(%d): %.2f L(%d): %.2f\n", sum/SIZE, max_pos,
max_grd, min_pos, min_grd);
return 0;
}

```

**C.7.10** The purpose of the following program is to generate 10 random integers, store them in an array, and then store in the freq frequency array, the number of occurrences of each number from 0 to 9. Can you find any errors in the program? If yes, correct them.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 10

int main(void)
{
    int i, num, arr[SIZE], freq[SIZE];

    srand(time(NULL));
    for(i = 0; i < SIZE; i++)
        arr[i] = rand();

    for(i = 0; i < SIZE; i++)
    {
        num = arr[i];
        freq[num]++;
    }
}

```

```

printf("\nNumber occurrences\n");
for(i = 0; i < SIZE; i++)
    printf("%d appears %d times\n", i, freq[i]);
return 0;
}

```

**Answer:** The first error is that the elements of the array freq have not been initialized to 0, so the statement freq[num]++; increments the meaningless value of freq[num]. Therefore, we should change the declaration to freq[SIZE] = {0}.

However, the most serious error happens in case arr contains a number out of [0, SIZE-1]. For example, if the number 30 is stored, num becomes 30 and the statement freq[30]++; changes the content of a memory location out of arr. This error can be corrected with the use of an **if** statement. For example:

```

if(num >= 0 && num <= SIZE-1)
    freq[num]++;

```

**C.7.11** Write a program that reads 20 integers and stores them in two arrays of 10 elements each. Then, the program should check if the two arrays have common elements, and if yes, the program should display the value of each common element and its position in both arrays. Otherwise, it should display a message that their elements are different.

```

#include <stdio.h>

#define SIZE 10

int main(void)
{
    int i, j, cnt, arr1[SIZE], arr2[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number for the 1st array: ");
        scanf("%d", &arr1[i]);

        printf("Enter number for the 2nd array: ");
        scanf("%d", &arr2[i]);
    }

    cnt = 0; /* This variable counts the common elements. */
    for(i = 0; i < SIZE; i++)
    {
        for(j = 0; j < SIZE; j++) /* This loop checks if an element
of the first array is contained in the second one. */
        {
            if(arr1[i] == arr2[j])
            {
                cnt++;
                printf("Common = %d (Pos_1 = %d Pos_2 =
%d)\n", arr1[i], i, j);
            }
        }
    }
}

```

```

    if(cnt == 0)
        printf("No common elements were found\n");
    return 0;
}

```

**C.7.12** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int i, a[] = {10, 20, 30, 40, 50};
    double b[] = {2.2, 1.94, 0.5, -1, -2};

    for(i = 0; a[i] = b[i]; i++)
        printf("%d ", a[i]);
    return 0;
}

```

**Answer:** The condition `a[i] = b[i]`; is equivalent to `(a[i] = b[i]) != 0`, which means that the elements of b are copied to the respective elements of a as long as `a[i]` does not become 0. If it does, the loop terminates. Since the type of a is `int`, only the integer parts of the b elements will be stored into the respective a elements. Therefore, when the value 0.5 is copied, `a[2]` becomes 0 and the loop terminates. As a result, the program outputs 2 1

**C.7.13** Write a program that reads the grades of 100 students and stores in successive positions of an array the grades within [5, 10] and in a second array the grades within [0, 5]. If the user enters -1, the insertion of grades should end and the program should display the average of the grades stored in each array.

```

#include <stdio.h>
int main(void)
{
    int i, k, m;
    float grd, sum_suc, sum_fail, arr1[100], arr2[100];

    sum_suc = sum_fail = 0;
    k = m = 0;
    for(i = 0; i < 100; i++)
    {
        printf("Enter grade: ");
        scanf("%f", &grd);
        if(grd == -1)
            break;

        if(grd >= 5 && grd <= 10)
        {
            sum_suc += grd;
            arr1[k] = grd;
            k++; /* The variable k indicates how many grades are
stored into arr1. We could write arr1[k++] = grd; We use two statements
to make it clearer. */
        }
        else if(grd >= 0 && grd < 5)

```

```

    {
        sum_fail += grd;
        arr2[m] = grd;
        m++;
    }
}
if(i != 0)
{
    if(k != 0)
        printf("\nSuccess_Avg: %.2f\n", sum_suc/k);
    else
        printf("\nAll students failed\n");

    if(m != 0)
        printf("\nFail_Avg: %.2f\n", sum_fail/m);
    else
        printf("\nAll students passed\n");
}
return 0;
}

```

**C.7.14** Write a program that reads 10 integers and stores them in an array only if either:

- the current position is even, i.e., 0, 2, 4, ..., and the input number is even or
- the current position is odd, i.e., 1, 3, 5, ..., and the input number is odd.

The program should not accept the values 0 and -1 and prompt the user to enter a new value. Once the insertion of the numbers is completed, the unassigned elements should be set to -1. The program should display the array elements before it ends.

```

#include <stdio.h>

#define SIZE 10

int main(void)
{
    int i, num, arr[SIZE] = {0}; /* Since the value 0 is not an
acceptable value, we use it as a special value to indicate that an
element is unassigned. */
    for(i = 0; i < SIZE; i++)
    {
        do
        {
            printf("Enter number: ");
            scanf("%d", &num);
            if(num == 0 || num == -1)
                printf("Not valid input !!!\n");
        } while(num == 0 || num == -1);

        if(i & 1) /* Check if the position is odd. */
        {
            if(num & 1) /* Store the number only if both the
position and the number are odd. */

```

```

        arr[i] = num;
    }
    else
    {
        if((num & 1) == 0) /* Store the number only if both
the position and the number are even. */
            arr[i] = num;
    }
}
printf("\n*** Array elements ***\n");
for(i = 0; i < SIZE; i++)
{
    if(arr[i] == 0)
        arr[i] = -1;
    printf("%d\n", arr[i]);
}
return 0;
}

```

**C.7.15** Write a program that reads an integer and displays the digits that appear more than once and the number of their appearances. For example, if the user enters 1868, the program should display that digit 8 appears twice. If no digit appears more than once, the program should display a related message.

```

#include <stdio.h>
int main(void)
{
    int i, rem, flag, dig_times[10] = {0}; /* This array holds the
appearances of each digit. For example, dig_times[0] indicates how many
times digit 0 appears. */
    printf("Enter number: ");
    scanf("%d", &i);
    if(i == 0) /* Check if 0 is entered. */
        dig_times[0] = 1;
    while(i != 0)
    {
        rem = i%10;
        if(rem < 0)
            rem = -rem;
        dig_times[rem]++;
        i /= 10;
    }
    flag = 0;
    for(i = 0; i < 10; i++)
    {
        if(dig_times[i] > 1)
        {
            printf("Digit %d appears %d times\n", i,
dig_times[i]);
            flag = 1;
        }
    }
    if(flag == 0)

```

```
    printf("No digit appears multiple times\n");
    return 0;
}
```

C.7.16 Is the following program correct? If yes, what does it output? If there is an error, could you think of a side-effect? Think before looking at the answer.

```
#include <stdio.h>
int main(void)
{
    int i, a[] = {i, i+1, i+2};

    for(i = 0; i < 4; i++)
    {
        a[i] = 0;
        printf("%d ", a[i]);
    }
    return 0;
}
```

**Answer:** Did you answer that the initialization of a array is wrong? Nothing is wrong there; *i* has an arbitrary value, so what? The error is the number of repetitions. Since *a* contains three elements, it should be 3, not 4. In particular, when *i* becomes 3, the statement *a[i] = 0;* assigns a value in a memory location out of *a*. If *i* is stored in that location, *i* becomes 0 and the loop becomes infinite.

*Be careful, and be careful again with the array bounds, if you want to have a peaceful sleep.*

C.7.17 Write a program that reads an integer and displays it in binary. Use an array to store the bits of the number. To display the bits of a negative number, use the two's complement technique. For example, if the user enters -5:

- Convert it to positive (i.e., 5).
- Reverse its bits (i.e.,  $101_2$  becomes  $11111111111111111111111111010_2$ ).
- Add 1 (i.e., the number becomes  $111111111111111111111111111011_2$ ).

```
#include <stdio.h>
int main(void)
{
    int i, j, tmp, bits[32]; /* This array holds the bits of the
                                number, that is 0 or 1. The size is 32, because we assume that the size
                                of an integer is 32 bits. */
    unsigned int num;

    printf("Enter number: ");
    scanf("%d", &tmp);
    if(tmp < 0)
    {
        tmp = -tmp;
        tmp = ~tmp;
        tmp++;
    }
}
```

```

num = tmp;      /* It is stored as positive. */
i = 0;
/* Successive divisions by 2 and store each bit in the respective
array position. */
while(num > 0)
{
    bits[i] = num & 1;
    num >>= 1; /* Equivalent to num /= 2. */
    i++;
}
printf("Binary form: ");
/* Display the bits from left to right. */
for(j = i-1; j >= 0; j--)
    printf("%d", bits[j]);
return 0;
}

```

**Comments:** The reason we used the shift operator is to improve performance. However, it is unnecessary most likely, because the compiler would optimize the operation by itself.

**C.7.18** Write a program that reads products' codes and stores them in an integer array of 50 places. The program should store a code in the array only if it is not already stored. If the array becomes full or the user enters -1, the program should display the stored codes and terminate.

```

#include <stdio.h>

#define SIZE 50

int main(void)
{
    int i, j, pos, num, found, code[SIZE];

    pos = 0;
    while(pos < SIZE)
    {
        printf("Enter code: ");
        scanf("%d", &num);
        if(num == -1)
            break;

        found = 0;
        /* The pos variable indicates how many codes have been
        stored in the array. The loop checks if the input code is already stored.
        If it is, found becomes 1 and the loop terminates. */
        for(j = 0; j < pos; j++)
        {
            if(code[j] == num)
            {
                printf("Error: Code %d exists. ", num);
                found = 1;
                break;
            }
        }
    }
}

```

```

        /* If the code is not stored, we store it and the index
position is incremented. */
        if(found == 0)
        {
            code[pos] = num;
            pos++;
        }
    }
    printf("\nCodes: ");
    for(i = 0; i < pos; i++)
        printf("%d ", code[i]);
    return 0;
}

```

**C.7.19** Write a program that reads 100 integers and stores them in an array. Then, the program should find and display the two different higher values.

```

#include <stdio.h>

#define SIZE 100

int main(void)
{
    int i, max_1, max_2, arr[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%d", &arr[i]);
    }
    max_1 = max_2 = arr[0];
    for(i = 1; i < SIZE; i++)
    {
        if(max_1 != arr[i])
        {
            max_2 = arr[i]; /* We assign that value to max_2, to
compare it with the remaining elements. */
            break;
        }
    }
    if(i == SIZE)
    {
        printf("Array contains the same value: %d\n", max_1);
        return 0;
    }
    /* Continue with the loop execution from the point the second loop
ended. */
    for(; i < SIZE; i++)
    {
        if(arr[i] > max_1)
        {
            max_2 = max_1; /* The second higher value becomes
equal to the first one. */
        }
    }
}

```

```

        max_1 = arr[i];
    }
    else if(arr[i] > max_2 && arr[i] != max_1) /* Search for the
second higher value that is not equal to the first one. */
        max_2 = arr[i];
}
printf("First_Max = %d and Sec_Max = %d\n", max_1, max_2);
return 0;
}

```

**Comments:** The typical approach to find the  $n$ th minimum or maximum element of an array is to use a sorting algorithm, such as one of the algorithms presented in Chapter 12. In particular, the array is first sorted and the element we are searching for is the  $a[n-1]$ .

**C.7.20** Write a program that declares an array of 10 integers and assigns random values within [5, 20] to its elements. Then, the program should check if the array contains duplicated values, and if yes, it should delete them. When an element is deleted, the next elements should be shifted one position to the left. The empty positions should be given the value -1. For example, if the array is {9, 12, 8, 12, 17, 8, 8, 19, 1, 19} it should become {9, 12, 8, 17, 19, 1, -1, -1, -1, -1}.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 10

int main(void)
{
    int i, j, k, size, arr[SIZE];

    srand(time(NULL));
    for(i = 0; i < SIZE; i++)
        arr[i] = rand()%16+5;

    size = SIZE;
    for(i = 0; i < size; i++)
    {
        j = i+1;
        while(j < size)
        {
            if(arr[i] == arr[j])
            {
                for(k = j; k < size-1; k++)
                    arr[k] = arr[k+1]; /* Shift the elements
one position to the left. */
                size--; /* Since the element is deleted, their
number is decremented. */
            }
            else
                j++;
        }
    }
}

```

```

    for(i = size; i < SIZE; i++)
        arr[i] = -1;
    for(i = 0; i < SIZE; i++)
        printf("%d ", arr[i]);
    return 0;
}

```

**C.7.21** A parking station for 20 cars charges \$6 for the first three hours. Each extra hour is charged \$1.50 (even for one extra minute the whole hour is charged). The maximum charge is \$12 and the maximum parking time is 24 hours. Write a program that reads the number of the parked cars and the parking time of each car in hours and minutes and displays the charges in the following form:

Car	Time	Charge
1	2.30'	6.00
2	4.30'	9.00
3	3.12'	7.50
4	4.0'	7.50
5	8.0'	12.00
Total		42.00

```

#include <stdio.h>

#define MAX_CARS 20

int main(void)
{
    int i, tmp, cars, car_mins[MAX_CARS], car_hours[MAX_CARS];
    double bill, total_bill;

    do
    {
        printf("Enter number of cars [1-%d]: ", MAX_CARS);
        scanf("%d", &cars);
    } while(cars <= 0 || cars >= MAX_CARS);

    for(i = 0; i < cars; i++)
    {
        do
        {
            printf("Enter hours for car_%d [max=24h]: ", i+1);
            scanf("%d", &car_hours[i]);
        } while(car_hours[i] > 24 || car_hours[i] < 0);

        if(car_hours[i] == 24) /* Maximum duration. */
        {
            car_mins[i] = 0;
            continue;
        }
        do
        {
            printf("Enter minutes for car_%d [max=59min]: ",
i+1);
            scanf("%d", &car_mins[i]);
        } while(car_mins[i] > 59 || car_mins[i] < 0);
        bill += (car_hours[i] * 6) + ((car_mins[i] / 60) * 1.5);
    }
    total_bill = bill;
    bill = bill / cars;
    printf("Average charge per car: %.2f\n", bill);
    printf("Total bill: %.2f", total_bill);
}

```

```

        } while(car_mins[i] > 59 || car_mins[i] < 0);
    }
total_bill = 0;

printf("\nCar\tTime\tCharge\n");
for(i = 0; i < cars; i++)
{
    bill = 6; /* For the first 3 hours. */
    tmp = car_hours[i]-3;
    if(tmp >= 0)
    {
        bill += tmp*1.5; /* For extra hours. */
        if(car_mins[i] > 0)
            bill += 1.5;
        if(bill > 12)
            bill = 12;
    }
    printf("%d\t%d.%d\t%.2f\n", i+1, car_hours[i], car_mins[i],
bill);
    total_bill += bill;
}
printf("SUM\t\t%.2f\n", total_bill);
return 0;
}

```

**C.7.22** Write a program that simulates an online lottery game. Suppose that the winning numbers are 10 and their values are within [0, 100]. However, the program should be written in a way to control the winning numbers. In particular, the “cheat” is that 3 of the winning numbers in each lottery should have also been drawn in the previous one. The program should ask the user to play, and if the answer is no, the program terminates.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 10

int main(void)
{
    int i, pos1, pos2, pos3, ans, arr[SIZE];

    srand(time(NULL));
    /* First lottery. */
    for(i = 0; i < SIZE; i++)
    {
        arr[i] = rand()%101;
        printf("%d ", arr[i]);
    }
    while(1)
    {
        printf("\nContinue to play? (0:No): ");
        scanf("%d", &ans);
        if(ans == 0)
            return 0;
    }
}

```

```

/* We choose three random places. The numbers stored in
these places will be drawn in the next lottery. */
pos1 = rand()%SIZE;
do
{
    pos2 = rand()%SIZE;
} while(pos1 == pos2);

do
{
    pos3 = rand()%SIZE;
} while((pos1 == pos3) || (pos2 == pos3));

/* Next lottery. */
for(i = 0; i < SIZE; i++)
{
    if((i == pos1) || (i == pos2) || (i == pos3))
    {
        printf("%d ", arr[i]);
        continue;
    }
    arr[i] = rand()%101;
    printf("%d ", arr[i]);
}
return 0;
}

```

**Comments:** The primary goal of this program is to show you that betting software can be written in a way to manipulate the winning results. Therefore, *stay away* from betting sites that advertise big profits in online lotteries. The big profits they promise are not for you, but for their owners.

---

## Two-Dimensional Arrays

The form of a two-dimensional array resembles that of a matrix in math; it is an array of elements of the same type arranged in rows and columns. As with one-dimensional arrays, we'll use arithmetic types in our examples. In the following chapters, you'll see more application examples. Besides numbers, a two-dimensional array is often used to store strings; you'll see this application in Chapter 10.

### Declaring Arrays

To declare a two-dimensional array, we must specify its name, its data type, and the number of its rows and columns, like this:

```
data_type array_name [number_of_rows] [number_of_columns]
```

The number of its elements is equal to the product of rows multiplied by columns. For example, the statement `double arr[10][5];` declares the two-dimensional array `arr` with 50 elements of type `double`.

## Accessing Array Elements

To access an element, we write the name of the array followed by the element's row index and column index enclosed in double brackets. As with one-dimensional arrays, the indexing of rows and columns starts from 0. For example, the statement:

```
int a [3] [4] ;
```

declares a two-dimensional array, whose elements are the  $a[0][0]$ ,  $a[0][1]$ , ...  $a[2][3]$ , as shown in Figure 7.1.

Although a two-dimensional array is visualized as an array with rows and columns, its elements are stored in consecutive memory locations in row-major order, with the elements of row 0 first, followed by the elements of row 1, and so on.

Although we refer to multidimensional arrays, C supports only one-dimensional arrays. Because an element of an array may be another array, we can simulate multidimensional arrays.

For example, the elements of the  $a$  array are the  $a[0]$ ,  $a[1]$ , and  $a[2]$ , where each one of them is an array of four integers. We'll analyze it more in Chapter 8, when discussing about pointers and two-dimensional arrays. Also, we'll see in C.8.39 how we can treat a two-dimensional array as if it were one-dimensional. In another example, if we write `int a[3][4][5];`  $a$  is an array of three elements, where each one is an array of four elements. Each of these four elements is an array of five integers.

As with one-dimensional arrays, when a two-dimensional array is declared, the compiler allocates a memory block in the stack to store the values of its elements. For example, with the declaration `int a[10][5];` the compiler allocates a block of 200 bytes to store the values of its 50 elements. To access an element, we must specify its row index and its column index. Here are some examples:

```
int i = 2, j = 2, a[3][4];
a[0][0] = 100; /* The value of the first element becomes 100. */
a[1][1] = 200; /* The value of the sixth element becomes 200. */
a[2][3] = a[0][0]; /* The value of the last element becomes equal to the
first element. */
a[i-2][j-2] = 300; /* The value of the first element becomes 300. */
```

	Column 0	Column 1	Column 2	Column 3
Row 0	a [ 0 ] [ 0 ]	a [ 0 ] [ 1 ]	a [ 0 ] [ 2 ]	a [ 0 ] [ 3 ]
Row 1	a [ 1 ] [ 0 ]	a [ 1 ] [ 1 ]	a [ 1 ] [ 2 ]	a [ 1 ] [ 3 ]
Row 2	a [ 2 ] [ 0 ]	a [ 2 ] [ 1 ]	a [ 2 ] [ 2 ]	a [ 2 ] [ 3 ]

↑      ↑      ↑      →

Array name      Row index      Column index

FIGURE 7.1

Two-dimensional array ( $3 \times 4$ ).

As with one-dimensional arrays, special care is required not exceed the bounds of any dimension.

Consider the two-dimensional integer array `a[ROWS][COLS]`. Since its elements are stored sequentially in memory, the compiler in order to find the memory address of `a[i][j]` finds the row size, that is, `row_size = COLS × sizeof(int)` and multiplies it with `i`. Then, it multiplies `j` by the size of an element, and adds the products to the array's address. Therefore, we have:

```
memory_address = a + (i × row_size) + (j × sizeof(int))
```

To find the memory address of an element, only the number of columns is needed.

## Array Initialization

A two-dimensional array can be initialized when declared, just like a one-dimensional array. The initialization values are assigned in row order, starting from row 0, then row 1, and so on. For example, with the declaration:

```
int arr[3][3] = {{10, 20, 30},  
                  {40, 50, 60},  
                  {70, 80, 90}};
```

`arr[0][0]` becomes 10, `arr[0][1]` becomes 20, `arr[0][2]` becomes 30, and continue to the next rows.

Alternatively, we can omit the inner braces and write:

```
int arr[3][3] = {10, 20, 30, 40, 50, 60, 70, 80, 90};
```

Once the compiler fills one row, it continues with the next one. However, our preference is always to use the inner braces so that the initialization of each row is clearly shown. When using braces, if the initialization list is shorter than the row's elements, the compiler assigns the value 0 to the remaining elements in that row. If it is larger it is illegal. For example, with the declaration:

```
int arr[3][3] = {{10, 20},  
                  {40, 50},  
                  {70}};
```

the values of `arr[0][2]`, `arr[1][2]`, `arr[2][1]`, and `arr[2][2]` are initialized to 0.

If we omit the initialization of a row, the compiler initializes its elements to 0. For example, with the declaration:

```
int arr[3][3] = {{10, 20, 30}};
```

the elements of the second and third row are set to 0.

If the inner braces are omitted and the initialization list is shorter than the number of the array elements, the remaining elements are set to 0. For example, with the declaration:

```
int arr[3][3] = {10, 20};
```

arr[0][0] becomes 10, arr[0][1] becomes 20, and the remaining elements are set to 0.

When a two-dimensional array is declared, the number of columns must be specified. However, the number of rows is optional. If it is not specified, the compiler will create a two-dimensional array based on the initialization list. For example, with the declaration:

```
int arr[] [3] = {10, 20, 30, 40, 50, 60};
```

since the array has three columns and the initialization values are six, the compiler creates a two-dimensional array of two rows and three columns.

Enough with the initialization lists. Consider the following program and tell us what does it output?

```
#include <stdio.h>
int main(void)
{
    int arr[] [3] = {10, 20, 30, 40, 50, 60, 70};

    printf("%d\n", sizeof(arr)+arr[2][1]+arr[2][2]);
    return 0;
}
```

Since the array has three columns and the values are seven, the compiler creates a two-dimensional array of three rows and three columns, that is, an array of nine elements. Since the initialization list is shorter, the last two, that is, arr[2][1] and arr[2][2] are set to 0. Therefore, the program outputs 36.

If the initialization value is the same or it can be easily produced, a pair of nested loops is the typical choice. For example, the following program declares a two-dimensional array and assigns the value 1 to its elements.

```
#include <stdio.h>
int main(void)
{
    int i, j, arr[50][100];

    for(i = 0; i < 50; i++)
        for(j = 0; j < 100; j++)
            arr[i][j] = 1;
    return 0;
}
```

## Exercises

C.7.23 Write a program that reads 8 integers, stores them in a  $2 \times 4$  array, and displays the array elements in reverse order, from the lower-right element to the upper-left one.

```
#include <stdio.h>

#define ROWS 2
#define COLS 4

int main(void)
{
    int i, j, arr[ROWS] [COLS];

    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
        {
            printf("Enter arr[%d] [%d]: ", i, j);
            scanf("%d", &arr[i] [j]);
        }
    }
    printf("\nArray elements\n");
    printf("-----\n");
    for(i = ROWS-1; i >= 0; i--)
        for(j = COLS-1; j >= 0; j--)
            printf("arr[%d] [%d] = %d\n", i, j, arr[i] [j]);
    return 0;
}
```

C.7.24 Write a program that creates a  $5 \times 5$  upper triangular array; it is in array that the elements below the main diagonal should be set to 0. The program should assign random values within  $[-3, 3]$  to the remaining elements. Then, the program should display the determinant of the array, that is, the product of the main diagonal's elements. Finally, the program should display the array in algebra form.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS 5

int main(void)
{
    int i, j, determ, arr[ROWS] [ROWS];

    determ = 1; /* Initialize the determinant. */
    srand(time(NULL));
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < ROWS; j++)
            if(i == j)
                arr[i] [j] = determ;
            else
                arr[i] [j] = -3 + (rand() % 7);
    }
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < ROWS; j++)
            printf("%d ", arr[i] [j]);
        printf("\n");
    }
    printf("Determinant = %d\n", determ);
}
```

```

    {
        if(i > j)
            arr[i][j] = 0; /* The elements below the main
diagonal should be set to 0. */
        else
            arr[i][j] = rand()%7-3; /* The result of
rand()%7 is an integer in [0, 6]. By subtracting 3, the integer is
constrained in [-3, 3]. */
        printf("%3d", arr[i][j]);
        if(i == j) /* Check if the element belongs in the
main diagonal. */
            determ *= arr[i][j];
    }
    printf("\n"); /* Used to separate the rows of the arrays. */
}
printf("\nThe determinant is: %d\n", determ);
return 0;
}

```

**C.7.25** Write a program that reads 6 integers and stores them in a  $2 \times 3$  array (e.g., a). Then, it should read another 6 integers and store them in a second  $3 \times 2$  array (e.g., b). The program should display the elements of a third  $2 \times 2$  array (e.g., c), which is the result of  $c = ab$ .

It is reminded from the linear algebra that the product of two matrices is produced by adding the products of the elements of each row of the first matrix with the corresponding elements of each column of the second matrix. Therefore, the outcome of a  $N \times M$  matrix multiplied with a  $M \times N$  matrix is a  $N \times N$  matrix. For example, consider the following a ( $2 \times 3$ ) and b ( $3 \times 2$ ) matrices:

$$a = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 2 & 1 \end{bmatrix} \text{ and } b = \begin{bmatrix} 1 & 0 \\ 2 & -2 \\ 2 & 3 \end{bmatrix}$$

The dimension of  $c = ab$  is  $2 \times 2$ , and its elements are

$$c = \begin{bmatrix} 1 \times 1 + (-1) \times 2 + 1 \times 2 & 1 \times 0 + (-1) \times (-2) + 1 \times 3 \\ 0 \times 1 + 2 \times 2 + 1 \times 2 & 0 \times 0 + 2 \times (-2) + 1 \times 3 \end{bmatrix} = \begin{bmatrix} 1 & 5 \\ 6 & -1 \end{bmatrix}$$

In math, the value of its element is the outcome of  $c_{ij} = \sum_{k=1}^M a_{ik} \times b_{kj}$ . Of course, in programming level, k ranges from 0 to  $M-1$ .

```

#include <stdio.h>

#define N 2
#define M 3

int main(void)
{
    int i, j, k, a[N][M], b[M][N], c[N][N] = {0};

```

```

for(i = 0; i < N; i++)
{
    for(j = 0; j < M; j++)
    {
        printf("Enter element a[%d] [%d]: ", i, j);
        scanf("%d", &a[i][j]);
    }
}
for(i = 0; i < M; i++)
{
    for(j = 0; j < N; j++)
    {
        printf("Enter element b[%d] [%d]: ", i, j);
        scanf("%d", &b[i][j]);
    }
}
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        for(k = 0; k < M; k++)
            c[i][j] += a[i][k] * b[k][j];

printf("\nArray c = a x b (%dx%d)\n", N, N);
printf("-----\n");
for(i = 0; i < N; i++)
{
    for(j = 0; j < N; j++)
        printf("%5d", c[i][j]);
    printf("\n");
}
return 0;
}

```

**C.7.26** Write a program that assigns random values within [0, 20] to the elements of a  $6 \times 8$  integer array. Then, the program should read the user's choice, and if it is 1, it should read two more integers, which correspond to two rows, and swap their elements. If it is not 1, the two input integers correspond to columns. The program should check the validity of the two input integers and display the original and final array.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS 6
#define COLS 8

int main(void)
{
    int i, j, a, b, max, tmp, type, arr[ROWS][COLS];

    srand(time(NULL));
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)

```

{}

```
        {
            arr[i][j] = rand()%21;
            printf("%5d", arr[i][j]);
        }
        printf("\n");
    }

printf("Enter swap type (1:rows): ");
scanf("%d", &type);
if(type == 1)
    max = ROWS;
else
    max = COLS;
do
{
    printf("Enter dim_1[1-%d]: ", max);
    scanf("%d", &a);
} while(a < 1 || a > max);

do
{
    printf("Enter dim_2[1-%d]: ", max);
    scanf("%d", &b);
} while(b < 1 || b > max);

a--; /* Subtract 1, since indexing starts from 0.*/
b--;
if(type == 1)
{
    for(i = 0; i < COLS; i++)
    {
        tmp = arr[a][i];
        arr[a][i] = arr[b][i];
        arr[b][i] = tmp;
    }
}
else
{
    for(i = 0; i < ROWS; i++)
    {
        tmp = arr[i][a];
        arr[i][a] = arr[i][b];
        arr[i][b] = tmp;
    }
}
for(i = 0; i < ROWS; i++)
{
    for(j = 0; j < COLS; j++)
        printf("%5d", arr[i][j]);
    printf("\n");
}
return 0;
```

C.7.27 In linear algebra, a square matrix is called *Toeplitz* when the elements of each diagonal parallel to the main diagonal are equal. For example, the following  $5 \times 5$  matrix demonstrates the generic form of a  $5 \times 5$  *Toeplitz* matrix:

$$t = \begin{bmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{bmatrix}$$

Write a program that reads integers and stores them in the first row and first column of a  $5 \times 5$  array. Then, the program should create the *Toeplitz* matrix and display its elements in algebra form.

```
#include <stdio.h>

#define ROWS 5

int main(void)
{
    int i, j, t[ROWS][ROWS];

    printf("Enter [0][0] element: ");
    scanf("%d", &t[0][0]);
    /* Read the remaining values for the first row. */
    for(i = 1; i < ROWS; i++)
    {
        printf("Enter [0][%d] element: ", i);
        scanf("%d", &t[0][i]);
    }
    /* Read the remaining values for the first column. */
    for(i = 1; i < ROWS; i++)
    {
        printf("Enter [%d][0] element: ", i);
        scanf("%d", &t[i][0]);
    }
    /* Create the Toeplitz matrix. */
    for(i = 0; i < ROWS-1; i++)
        for(j = 0; j < ROWS-1; j++)
            t[i+1][j+1] = t[i][j]; /* We traverse the array t and
make each element equal to the upper left, except the elements of the
first row and column. */
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < ROWS; j++)
            printf("%3d", t[i][j]);
        printf("\n");
    }
    return 0;
}
```

**C.7.28** Write a program that reads and stores the grades of 100 students in 10 courses in a  $100 \times 10$  array and displays the average, the best, and the worst grade of each student. The program should also display the positions in the array that hold the best and worst average grades. If more than one student has the same best or worst average grade, the program should display the first position found. The program should force the user to enter grades within  $[0, 10]$ .

```
#include <stdio.h>

#define STUDS    100
#define COURSES 10

int main(void)
{
    int i, j, min_pos, max_pos;
    float sum, min_grd, max_grd, avg_grd, min_avg_grd, max_avg_grd,
grd[STUDS][COURSES];

    max_avg_grd = -1;
    min_avg_grd = 11;
    for(i = 0; i < STUDS; i++)
    {
        sum = 0;
        max_grd = -1;
        min_grd = 11;
        for(j = 0; j < COURSES; j++)
        {
            do
            {
                printf("Enter grade of student %d for
lesson %d: ", i+1, j+1);
                scanf("%f", &grd[i][j]);
            } while(grd[i][j] < 0 || grd[i][j] > 10);

            sum += grd[i][j];
            if(grd[i][j] >= max_grd)
                max_grd = grd[i][j];
            if(grd[i][j] <= min_grd)
                min_grd = grd[i][j];
        }
        avg_grd = sum/COURSES;
        if(avg_grd > max_avg_grd)
        {
            max_avg_grd = avg_grd;
            max_pos = i;
        }
        if(avg_grd < min_avg_grd)
        {
            min_avg_grd = avg_grd;
            min_pos = i;
        }
        printf("Student %d: Avg = %.2f Max = %.2f Min = %.2f\n",
i+1, avg_grd, max_grd, min_grd);
    }
}
```

```

    printf("\nStudent_%d has the higher average %.2f and student_%d
has the lower average %.2f\n", max_pos, max_avg_grd, min_pos,
min_avg_grd);
    return 0;
}

```

**C.7.29** Write a program that reads integers and stores them in a square matrix (e.g., 3x3). Then, the program should check whether the array is a magic square; that is, the sum of each row, column, and diagonal is the same.

```

#include <stdio.h>

#define ROWS 3
#define COLS 3

int main(void)
{
    int i, j, sum, tmp, arr[ROWS][COLS];

    sum = tmp = 0;
    for(i = 0; i < ROWS; i++)
    {
        /* We find the sums of the diagonals. */
        for(j = 0; j < COLS; j++)
        {
            printf("Enter element arr[%d] [%d]: ", i, j);
            scanf("%d", &arr[i][j]);
            if(i == j)
                sum += arr[i][j];
            if(i+j == ROWS-1) /* Check if the element belongs in
the secondary diagonal. */
                tmp += arr[i][j];
        }
        if(sum != tmp)
        {
            printf("Not magic square -> Sum_main_diag: %d Sum_sec_diag:
%d\n", sum, tmp);
            return 0;
        }
        for(j = 0; j < COLS; j++)
        {
            /* Initialize the variable which calculates the sum of the
elements of each row. */
            tmp = 0;
            for(i = 0; i < ROWS; i++)
                tmp += arr[i][j];

            if(sum != tmp)
            {
                printf("Not magic square -> Sum_row_%d: %d Sum_diag:
%d\n", i+1, tmp, sum);
                return 0;
            }
        }
    }
}

```

```

    }
    for(i = 0; i < COLS; i++)
    {
        tmp = 0; /* Initialize the variable which calculates the sum
of the elements of each column. */
        for(j = 0; j < ROWS; j++)
            tmp += arr[j][i];

        if(sum != tmp)
        {
            printf("Not magic square -> Sum_col_%d: %d Sum_diag:
%d\n", i+1, tmp, sum);
            return 0;
        }
    }
    printf("Magic square !!!\n");
    return 0;
}

```

**C.7.30** Write a program that simulates a cinema's ticket office. Assume that the cinema has 30 rows of 20 seats each. The program should display a menu to perform the following operations:

1. *Buy a ticket.* The program should let the spectator to select the row and the seat. If no specific seat is asked, the program should select a random seat. The ticket's price is \$6.
2. *Ticket cancellation.* The program should read the row and the seat and cancel the reservation. The refund is \$5.
3. Display the total income and a diagram to show the reserved and free seats.
4. Program termination.

```

#include <stdio.h>
#include <stdlib.h>

#define ROWS 30
#define COLS 20

int main(void)
{
    int i, j, sel, row, col, rsrd_seats, cost, seats[ROWS][COLS] =
{0}; /* We use the seats array to manage the cinema's seats. If an
element is 0, it implies that the seat is free. */
    rsrd_seats = cost = 0;
    while(1)
    {
        printf("\nMenu Selections\n");
        printf("-----\n");

        printf("1. Buy Ticket\n");
        printf("2. Ticket Refund\n");
        printf("3. Show Information\n");
        printf("4. Exit\n");

```

```

printf("\nEnter choice: ");
scanf("%d", &sel);

switch(sel)
{
    case 1:
        if(rsvd_seats == ROWS*COLS)
        {
            printf("\nSorry, no free seats\n");
            break;
        }
        printf("\nWould you like a specific seat (No:
0)? ");
        scanf("%d", &sel);
        if(sel == 0)
        {
            do
            {
                row = rand() % ROWS; /* Use
rand() to select a random seat. */
                col = rand() % COLS;
            } while(seats[row][col] == 1);
        }
        else
        {
            do
            {
                printf("\nEnter row [1-%d]: ", ROWS);
                scanf("%d", &row);
            } while(row < 1 || row > ROWS);

            do
            {
                printf("Enter seat [1-%d]: ", COLS);
                scanf("%d", &col);
            } while(col < 1 || col > COLS);

            row--; /* Subtract 1, since indexing
starts from 0. */
            col--;
        }
        if(seats[row][col] == 1)
            printf("\nSorry, seat in row_%d and
column_%d is reserved\n", row+1, col+1);
        else
        {
            seats[row][col] = 1;
            cost += 6;
            rsvd_seats++;
        }
    break;
}

```

```

case 2:
    if(rsvd_seats == 0)
    {
        printf("\nAll seats are free\n");
        break;
    }
do
{
    printf("\nEnter row [1-%d] : ", ROWS);
    scanf("%d", &row);
} while(row < 1 || row > ROWS);

do
{
    printf("Enter seat [1-%d] : ", COLS);
    scanf("%d", &col);
} while(col < 1 || col > COLS);

row--;
col--;
if(seats[row][col] != 1)
    printf("\nSeat in row_%d and column_%d
is not reserved\n", row+1, col+1);
else
{
    seats[row][col] = 0;
    cost -= 5;
    rsvd_seats--;
}
break;

case 3:
    printf("\nFree seats: %d, Income: %d\n\n",
ROWS*COLS - rsvd_seats, cost);
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
        {
            if(seats[i][j] == 1)
                printf("%2s", "X");
            else
                printf("%2s", "#");
        }
        printf("\n");
    }
break;

case 4:
return 0;

default:
    printf("\nWrong choice\n");
break;

```

```
    }  
}  
return 0;
```

---

## Unsolved Exercises

**U.7.1** Write a program that reads the grades of 100 students and stores them in an array. Then, the program should read two floats (e.g., a and b) and display how many students got a grade within [a, b]. The program should force the user to enter the second number to be greater than the first one.

**U.7.2** Write a program that reads **double** numbers continuously and stores in an array of 100 places those with a value more than 5. If the array becomes full or the user enters -1 the insertion of numbers should terminate. The program should display the minimum of the values stored in the array.

**U.7.3** Write a program that reads 100 integers and stores them in an array. The program should display how many elements have a value greater than the value of the last element and how many elements have a value greater than the average. The first value to enter should be that of the last element.

**U.7.4** Write a program that reads integers and stores them in an array of 100 places with the restriction that an input number is stored in the array only if it is less than the last stored. The program should terminate once the array becomes full.

**U.7.5** Write a program that reads 100 integers and stores them in an array. Then, the program should rotate the elements one place to the right. For example, if the array was of four elements: 1, -9, 5, 3, the rotated array would be: 3, 1, -9, 5.

**U.7.6** Write a program that reads 100 integers and stores them in an array. The program should display the number of the duplicated values. For example, if the array were of five elements {5, 5, 5, 5} the program should display 4 (since number 5 is repeated four times), and if it were {1, -3, 1, 50, -3} the program should display 2 (since numbers 1 and -3 are repeated once), and if it were {3, -1, 22, 13, -7} the program should display 0 (since no number is repeated).

**U.7.7** Write a program that reads 100 **double** numbers and stores them in an array. The program should calculate the distance between successive elements and display the minimum one. To calculate the distance of two elements, subtract their values and use the absolute value. For example, if the first four elements are 5.2, -3.2, 7.5, 12.22, the distances are  $|-3.2 - 5.2| = 8.4$ ,  $|7.5 - (-3.2)| = 10.7$  and  $|12.22 - 7.5| = 4.72$ .

**U.7.8** Write a program that reads the grades of 100 students and displays the frequency of each grade. The program should also display the grade which is given the most times

and the number of its occurrences. Examine the case that more than one grade might be given the same most times. The program should force the user to enter grades within [0, 10]. Assume that the grades are integers.

**U.7.9** Write a program that reads integers and stores them in a square matrix (e.g.,  $3 \times 3$ ). Then, the program should check whether the array is upper triangular, lower triangular or both. A square matrix is called upper (lower) triangular if all its elements below (above) the main diagonal are zero.

**U.7.10** Write a program that reads integers and stores them in a  $3 \times 4$  array. Then, the program should find the maximum value of each row and assign that value to all preceding elements in the same row.

**U.7.11** Write a program that reads integers and stores them in a square matrix (e.g.,  $3 \times 3$ ). Then, the program should create and display a new matrix having as rows the columns of the original one and columns the rows of the original one. Use only one array.

**U.7.12** Write a program that assigns random values within [1000, 2000] to the elements of the main diagonal of a  $5 \times 5$  integer array. The program should assign random values to the remaining elements with the restriction that the values of the elements below the main diagonal should be less than those of the main diagonal, and the values of the elements above the main diagonal should be greater than those of the main diagonal.

**U.7.13** Write a program that assigns random values within [0, 20] to the elements of a  $3 \times 5$  integer array, except the last row. The elements of the last row should be assigned with proper values so that the sum of the elements in each column is 50. The program should also store into a one-dimensional array the minimum value of each column of the two-dimensional array and display that array, before it terminates.

**U.7.14** Write a program that reads integers and stores them in a  $5 \times 5$  square matrix. Then, the program should read an integer within [1, 5] (e.g.,  $x$ ) and copy in a second array the elements of the first array, except the elements of the  $x$ -row and  $x$ -column.

**U.7.15** Write a program that reads integers and stores them in a  $3 \times 5$  array. The program should display the columns whose elements have different values. For example, if the array were:

$$\begin{bmatrix} 1 & -2 & 2 & 5 & 9 \\ 3 & 0 & 2 & 5 & 1 \\ 1 & 7 & 2 & -3 & 0 \end{bmatrix}$$

the program should display the elements of the second and fifth column.

**U.7.16** Write a program that simulates hotel booking software. Assume that the hotel has 10 wings of 50 rooms each. The program should display a menu to perform the following operations:

1. Display the number of available rooms in each wing.
2. Make a reservation. The program should prompt the user to enter a wing number. Then, it reserves the first available room in that wing. If there is no available room, the program should prompt the user to enter a new wing until the reservation is made.
3. Cancel a reservation. The program should prompt the user to enter the wing number and the room number and cancel the reservation.
4. Program termination.

# Pointers

Pointers are one of the most important features of C; however, it is a complex concept to understand and apply correctly. This chapter uses pointers to numeric variables to introduce you to the pointer concepts. The close relationship between pointers and arrays is investigated. You'll learn how to use arrays of pointers and pointers to functions. Other applications of pointers, such as passing arguments in functions, to handle strings, to form dynamic data structures, as well as pointers to other type of data, will be gradually presented over the following chapters.

## Pointers and Memory

In most modern computers, the main memory consists of millions of consecutively numbered memory cells where each cell stores eight bits of information and is identified by a unique number, called memory address. For example, in a computer with N bytes, the memory address of each cell is a unique number from 0 to N-1, as shown in Figure 8.1.

When a variable is declared, the compiler reserves the required consecutive bytes to store its value. If a variable occupies more than one byte, the variable's address is the address of the first byte. For example, with the declaration: `int a = 10;` the compiler reserves four consecutive unused bytes and stores the value 10 into the addresses 5000-5003, assuming that the less significant byte of the value is stored in the lower address byte (*little-endian* architecture).

The compiler associates the name of a variable with its memory address. When the variable is used in the program, the compiler accesses its address. For example, with the statement `a = 80;` the compiler knows that the memory address of a is 5000 and sets its content to 80.

## Pointer Declaration

A pointer variable is a variable that can hold a memory address. To declare a pointer, we write:

```
data_type *pointer_name;
```

Notice that the name of the pointer variable must be preceded by the \* character, for example, with the declaration: `int *ptr;`

Memory address	Memory content
0	
1	
2	
.	
.	
5000	10
5001	0
5002	0
5003	0
.	
.	
n-1	

**FIGURE 8.1**

Memory layout.

`ptr` is declared as a pointer variable to type `int`. Therefore, `ptr` can hold the memory address of an `int` variable. In general, if the type of the pointer is  $T$ , the type of  $*T$  is  $T$ . For example, the type of the expression `*ptr` is `int`.

Pointer variables can be declared together with other variables of the same type. For example:

```
int *ptr, i, j, k;
```

Notice that it is allowed to put the `*` next to the type (e.g., `int* ptr;`). In fact, many programmers prefer this style. However, this is not our preference, because it might be confusing when multiple variables are declared. For example, with the declaration:

```
int* p1, p2;
```

`p1` is declared as pointer, `p2` is not.

Also, we prefer to declare the pointer variables first, to distinguish them from the ordinary variables.

As with ordinary variables, when a pointer variable is declared, the compiler allocates memory to store its value. For example, the following program uses the `sizeof` operator to find out how many bytes `ptr` allocates:

```
#include <stdio.h>
int main(void)
{
    int *ptr;

    printf("Bytes: %u\n", sizeof(ptr));
    return 0;
}
```

A pointer variable allocates the same size, no matter what data type it points to. This value is platform dependent; its typical value is four bytes. Therefore, if we write `char *ptr;` or `float *ptr;` or `double *ptr;` in our example, the output would be the same, that is, 4.

## Pointer Initialization

After declaring a pointer variable, we can use it to store the memory address of another variable. To find the address of a variable, we use the `& address operator` before its name. For example:

```
#include <stdio.h>
int main(void)
{
    int *ptr, a;

    ptr = &a; /* ptr "points to" the memory address of a. */
    printf("Address = %p\n", ptr); /* Display the memory address of a. */
    return 0;
}
```

With the statement `ptr = &a;` `ptr` becomes equal or else “points to” the memory address of `a`. The `%p` specifier is usually used, in order to display the memory address in hex.

A pointer variable can be initialized when declared, provided that the variable that it points to has already been declared. For example:

```
int a, b, *ptr = &a;
```

As a matter of style, we prefer to initialize the pointer variables in separate statements, not together with their declarations, in order to make it clearer. Due to lack of space, we'll do it in one line in some of our examples.

The address operator `&` cannot be applied to constants (e.g., `&20`), expressions (e.g., `&(a+b)`), variables of type `register` (we'll talk about that type in Chapter 11), and to bit fields (we'll talk about them in Chapter 13). It is applied only to objects in memory, such as variables, array elements, and functions. In the general case, if the type is `T` the type of `&T` is “pointer to `T`.”

## Null Pointers

As with an ordinary variable, the initial value of a pointer variable is the content of its address, that is, “garbage,” an arbitrary value that corresponds to a memory address. When we want to declare explicitly that the pointer points to nowhere, a special value must be assigned. This value is `0` or `0` cast to `void*`, i.e., `(void*)0`, and it is represented by a macro named `NULL`. The `NULL` macro is defined in several header files (e.g., `stdio.h`). A pointer assigned the `NULL` value points to nothing and becomes a null pointer. For example, the following program first outputs the initial value of the pointer and then `0`.

```
#include <stdio.h>
int main(void)
{
    int *ptr;
```

```
    printf("Addr = %p\n", ptr);
    ptr = NULL;
    printf("Addr = %p\n", ptr);
    return 0;
}
```

Although the value 0 can be used in a pointer context, it'd be better to use the `NULL` macro to avoid confusion. For example, the assignment `ptr = 0;` might confuse the reader. Is `ptr` a numeric variable or a pointer? In contrast, the assignment `ptr = NULL;` makes it clear that `ptr` is a pointer. The value of a pointer variable can be compared with `NULL`, as follows:

```
if(ptr != NULL) /* Equivalent to if(ptr) */
if(ptr == NULL) /* Equivalent to if(!ptr) */
```

As a matter of style, the syntax we don't prefer is `if(!ptr)`. As we'll see in the following chapters, many functions return `NULL` to indicate that the execution failed.

---

## Using a Pointer

To access the content of a memory address referenced by a pointer variable we use the `*` (*indirection* or *dereference*) operator before its name. Although the same symbol is used, it is not the same as the multiplication operator. The compiler uses the context to determine whether the dereferencing or the multiplication operator is used. In case you wonder why the standard committee decided to use the same symbol and confuse the learners, we don't know, it could be due to the shortage of available characters. Here is an example:

```
#include <stdio.h>
int main(void)
{
    int *ptr, a;

    a = 10;
    ptr = &a;
    printf("Val = %d\n", *ptr);
    return 0;
}
```

The expression `*ptr` is equivalent to the content of the memory address pointed to by `ptr`. Since `ptr` points to the address of `a`, `*ptr` is an alias for `a`, that is, equal to `a`. Therefore, the program outputs 10.

---

A pointer variable must point to a valid memory address, such as an address of a variable, before being dereferenced.

For example, the following program may crash because `ptr` has not been initialized before used in the statement `a = *ptr;`

```

#include <stdio.h>
int main(void)
{
    int *ptr, a;

    a = *ptr; /* ptr has not been initialized. */
    printf("Val = %d\n", a);
    return 0;
}

```

Typically, in *Unix/Linux* environment, this type of error is indicated with the pop-up runtime message “*Segmentation fault.*” Always remember, initialize the pointer before using it, don’t let it blowing in the wind. Listen to *Blowing in the Wind* from *Bob Dylan* to bear it in mind.

The following program is normally executed, since `ptr` points to a valid address before used in the statement `i = *ptr;` What does it output?

```

#include <stdio.h>
int main(void)
{
    int *ptr, i, j;

    j = 20;
    ptr = &j;

    i = *ptr;
    printf("Val = %d\n", i);
    return 0;
}

```

Since `ptr` points to the address of `j`, `*ptr` is equal to `j`, that is, 20. Therefore, the statement `i = *ptr;` makes `i` equal to 20, and the program outputs `Val = 20`.

The `*` and `&` cancel each other when used together. For example, the following program displays three times the address of `i`.

```

#include <stdio.h>
int main(void)
{
    int *ptr, i;

    ptr = &i;
    printf("%p %p %p\n", &i, *ptr, &*ptr);
    return 0;
}

```

Because the symbol `/*` marks the beginning of a comment, in a statement such as `a = b/*ptr;` whatever follows `/*` until the `*/` is met, is considered part of the comment and the compilation fails. In that case, leave a space or use parentheses. For example, `a = b/*ptr;` or `a = b/(*ptr);`

## Exercises

C.8.1 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, i = 10;

    ptr = &i;
    i += 20;
    printf("%d\n", *ptr);
    return 0;
}
```

**Answer:** Since `ptr` points to the address of `i`, `*ptr` is equal to `i`. The statement `i += 20;` makes `i` equal to 30, and the program outputs 30.

C.8.2 Write a program that reads two integers, stores them in two variables, declares two pointers to them, and displays the memory addresses of both variables, the content of both pointers, as well as their memory addresses.

```
#include <stdio.h>
int main(void)
{
    int *ptr1, *ptr2, i, j;

    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);

    ptr1 = &i;
    ptr2 = &j;

    printf("Num1 address = %p\n", ptr1);
    printf("Num2 address = %p\n", ptr2);

    printf("Ptr1 content = %d\n", *ptr1);
    printf("Ptr2 content = %d\n", *ptr2);

    printf("Ptr1 address = %p\n", &ptr1);
    printf("Ptr2 address = %p\n", &ptr2);
    return 0;
}
```

**Comments:** As with ordinary variables, we use the `&` operator to find the memory addresses of the `ptr1` and `ptr2` pointer variables.

C.8.3 The following program uses a pointer to read and display a float number. Is there any programming bug?

```
#include <stdio.h>
int main(void)
```

```

{
    double *ptr, i;

    scanf("%lf", ptr);
    printf("Val = %f\n", *ptr);
    return 0;
}

```

**Answer:** The code is wrong because `ptr` does not point to the address of `i`, before being used in `scanf()`. Had we added the statement `ptr = &i;` before `scanf()`, the program would be executed normally.

**C.8.4** Write a program that uses a pointer to read a float number and display its absolute value.

```

#include <stdio.h>
int main(void)
{
    double *p, val;

    p = &val;
    printf("Enter number: ");
    scanf("%lf", p);

    if(*p >= 0)
        printf("%f\n", *p);
    else
        printf("%f\n", -*p);
    return 0;
}

```

**C.8.5** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int i = 0, *ptr = &i;

    *ptr = *ptr ? 10 : 20;
    printf("%d\n", i);
    return 0;
}

```

**Answer:** Since `ptr` points to the address of `i`, the expression:

`*ptr = *ptr ? 10 : 20;` is equivalent to `i = i ? 10 : 20;`

Since `i` is 0 (false), the value of the expression is 20. Therefore, `i` becomes 20 and the program outputs 20.

**C.8.6** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int *ptr1, *ptr2, *ptr3, i = 10, j = 20, k = 30;

    ptr1 = &i;
    i = 100;

    ptr2 = &j;
    j = *ptr2 + *ptr1;

    ptr3 = &k;
    k = *ptr3 + *ptr2;
    printf("%d %d %d\n", *ptr1, *ptr2, *ptr3);
    return 0;
}

```

**Answer:** Since ptr1 points to the address of i, \*ptr1 is equal to i. Similarly, \*ptr2 is equal to j. Therefore,  $j = *ptr2 + *ptr1 = 20+100 = 120$ . Since ptr3 points to the address of k, \*ptr3 is equal to k. Therefore,  $k = *ptr3 + *ptr2 = 30+120 = 150$ . Since the values of \*ptr1, \*ptr2, and \*ptr3 are equal to i, j, and k, respectively, the program displays: 100 120 150

**C.8.7** Write a program that uses two pointers to read two float numbers first and then to swap the values they point to. Then, use the same pointers to output the greater value.

```

#include <stdio.h>
int main(void)
{
    float *ptr1, *ptr2, i, j, tmp;

    /* The pointers should be initialized before used in scanf(). */
    ptr1 = &i;
    ptr2 = &j;

    printf("Enter values: ");
    scanf("%f%f", ptr1, ptr2); /* Store the input values in the
addresses pointed to by the pointers. */
    tmp = *ptr2;
    *ptr2 = *ptr1;
    *ptr1 = tmp;

    if(*ptr1 > *ptr2)
        printf("%f\n", *ptr1);
    else
        printf("%f\n", *ptr2);
    return 0;
}

```

**C.8.8** What is the output of the following program?

```

#include <stdio.h>
int main(void)

```

```

{
    int *ptr1, i = 10;
    double *ptr2, j = 1.234;

    ptr1 = &i;
    ptr2 = &j;

    *ptr1 = *ptr2;
    printf("%d %u %u\n", i, sizeof(ptr1), sizeof(ptr2),
sizeof(*ptr2));
    return 0;
}

```

**Answer:** Since ptr1 points to the address of i and ptr2 points to the address of j, the statement \*ptr1 = \*ptr2 is equivalent to i = j. Since i is declared as **int**, it becomes 1. As said, the pointer variables allocate the same size (typically, it is 4 bytes), no matter where they point to. Since ptr2 is a pointer to **double**, the program displays: 1 4 4 8

**C.8.9** Use the p pointer and a **while** loop, and complete the following program to display the integers from 1 to 10.

```

#include <stdio.h>
int main(void)
{
    int *p, i;
    ...
}

```

**Answer:**

```

#include <stdio.h>
int main(void)
{
    int *p, i;

    p = &i;
    *p = 1;
    while (*p <= 10)
    {
        printf("%d\n", *p);
        (*p)++; /* As we'll see later, parentheses are needed for
priority reasons. */
    }
    return 0;
}

```

**C.8.10** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int *ptr1, *ptr2, i = 10, j = 20;

    ptr1 = &i;
    ptr2 = &j;

```

```

ptr2 = ptr1;
*ptr1 = *ptr1 + *ptr2;
*ptr2 *= 2;
printf("%d\n", *ptr1 + *ptr2);
return 0;
}

```

**Answer:** The statement `ptr2 = ptr1;` makes `ptr2` to point to the same address that `ptr1` points to, that is, the address of `i`. Therefore, `*ptr2` is equal to `i`. Since both pointers point to the address of `i`, the statement `*ptr1 = *ptr1 + *ptr2;` is equivalent to `i = i+i = 10+10 = 20`. Similarly, the statement `*ptr2 *= 2;` is equivalent to `*ptr2 = 2*(*ptr2);` that is, `i = 2*i = 2*20 = 40`. As a result, the program displays the result of `*ptr1 + *ptr2 = i+i = 40+40 = 80`.

**C.8.11** Use the `p2` pointer and complete the following program to read students' grades continuously until the user enters `-1`. Use the `p1` pointer to display how many students got a grade within `[5, 10]` and the `p3` pointer to display the best grade.

```

#include <stdio.h>
int main(void)
{
    int *p1, sum;
    float *p2, *p3, grade, max;
    ...
}

```

**Answer:**

```

#include <stdio.h>
int main(void)
{
    int *p1, sum;
    float *p2, *p3, grade, max;

    p1 = &sum;
    *p1 = 0;

    p3 = &max;
    *p3 = 0;

    p2 = &grade;
    while(1)
    {
        printf("Enter grade: ");
        scanf("%f", p2);

        if(*p2 == -1)
            break;
        if(*p2 >= 5 && *p2 <= 10)
        {
            (*p1)++;
            if(*p2 > *p3)

```

```

        *p3 = *p2;
    }
    printf("%d students passed (max = %.2f)\n", *p1, *p3);
    return 0;
}

```

---

## The **void\*** Pointer

A pointer of type **void** is a generic pointer in the sense that it can point to a variable of any type. Any pointer can be cast to **void\*** and back again to the original type without any loss of information. Notice that, except for **void\***, in order to assign a pointer of one type to a pointer of another type, cast is necessary. For example:

```

char *p1;
int *p2;
void *p3;
...
p2 = p3;
p3 = p2;
p2 = p1; /* Wrong. */
p2 = (int*)p1; /* That's ok now. */

```

To access a variable using a **void\*** pointer, cast is necessary in order to inform the compiler about the type of the variable. For example:

```

#include <stdio.h>
int main(void)
{
    void *ptr;
    char s[] = "abcd";
    int i = 10;

    ptr = &i;
    *(int*)ptr += 20;
    printf("%d\n", i);

    ptr = s+2;
    (*(char*)ptr)++;
    printf("%s\n", s);
    return 0;
}

```

To access *i*, we cast the type of *ptr* to the type of *i* using the **int\*** cast. Now, we can change the value of the variable that this generic pointer points to. Therefore, the program displays 30.

Next, *ptr* points to the *s[2]* element with value 'c'. *ptr* is cast to **char\*** and the value of *s[2]* character is incremented. Since it becomes 'd', the program displays abdd. Yes, you are right; since we've not discussed yet about strings and characters, it is hard enough to understand this code. We just added this cast example of **void\*** to come back and read it once you learn strings.

Typically, **void** pointers are used as function parameters in order to pass values of different types. We'll see such an example in C.11.23 and in Chapter 14 with the `memcpy()` function.

## Use of **const** Qualifier

To prevent a pointer from changing the value of the variable it points to, we add the **const** keyword before its type. For example, the following code won't compile because `ptr` is not allowed to change the value of `i`. However, it is allowed to point to some other variable of the same type.

```
int j, i = 10;
const int *ptr;
ptr = &i;
*ptr = 30; /* Illegal action. */
ptr = &j; /* Legal action. */
```

To prevent a pointer from pointing to another variable, we add the **const** keyword before its name. In this case, the pointer must be initialized when declared. For example, the following code won't compile because `ptr` is not allowed to point to the address of `j`. However, it is allowed to change the value of `i`.

```
int i, j;
int* const ptr = &i; /* Initialize pointer. */
ptr = &j; /* Illegal action. */
*ptr = 30; /* Legal action. i becomes 30. */
```

Notice that it is allowed to prevent both actions by using the **const** keyword twice. For example:

```
int i, j;
const int* const ptr = &i; /* Initialize pointer. */
ptr = &j; /* Illegal action. */
*ptr = 30; /* Illegal action. */
```

As you see, there are not many things left to do with `ptr`. Essentially, what is left is to test the pointer value. And yes, that's a very rare declaration to see in practice.

Here is an example to have in mind for something similar:

```
const char* const days[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat",
"Sun"};
```

The first **const** prevents the modification of the strings, and the second prevents the pointers to point to somewhere else. The array of pointers will be discussed later in this chapter.

## Pointer Arithmetic

Pointer arithmetic refers to the application of some arithmetic operations on pointers. The standard says that pointer arithmetic is guaranteed to produce reliable results when applied to elements of the same array; otherwise, the behavior is undefined. An exception to that rule is the address of the first element past the end of the array; the standard guarantees that it is valid to use that place in pointer arithmetic. The allowed operations are adding or subtracting a pointer and an integer and subtracting and comparing pointers provided that they point to the same array. As discussed, assignment or comparison with 0 is also allowed.

### Pointers and Integers

Suppose that a pointer variable `ptr` points to an element of an array. The addition of a positive integer `n` in an assignment such as:

```
ptr = ptr + n;
```

increases the pointer's value by  $n * \text{size}$  of the pointer's data type and makes the pointer point to the address of the  $n$ th element after the one it points to. For example, if `ptr` points to the array element `arr[i]` and `n` is 3, `ptr` will point to `arr[i+3]`. If the resulting element is out of the array, with the exception of the first element past the end, the result is undefined. For example, given the assumptions we've made for the data types in Chapter 2, if `ptr` is declared as a pointer to an array of:

- `char`, its value is increased by  $n * 1 = n$ , because the size of the `char` type is one byte.
- `int` or `float`, its value is increased by  $n * 4$ , because the size of both types is four bytes.
- `double`, its value is increased by  $n * 8$ , because the size of `double` is eight bytes.

Consider the following program:

```
#include <stdio.h>
int main(void)
{
    int *ptr, arr[] = {10, 20, 30};

    ptr = &arr[0];
    printf("Addr:%d\n", ptr);
    ptr = ptr+2;
    printf("Addr:%d Val:%d\n", ptr, *ptr);
    return 0;
}
```

`ptr` is increased by 8, not by 2, because `ptr` is declared as a pointer to `int`. The program displays two addresses and the second one is 8 places higher than the first one. Since `ptr` points to `arr[2]`, the program outputs 30, as well.

Similar to addition, subtracting a positive integer from a pointer in an assignment such as:

```
ptr = ptr - n;
```

decreases the pointer's value by  $n * \text{size}$  of its type and makes the pointer point to the address of the nth element before the one it points to. For example, if we replace the code in the previous example with this one:

```
ptr = &arr[2];
printf("Addr:%d\n", ptr);
ptr = ptr-2;
printf("Addr:%d\n", ptr);
```

the second address would be 8 places lower than the first one. Just like the addition, the result is considered valid if the pointer points to an element inside the array.

Notice that if the pointer does not point to an array element, the use of `++` and `--` operators produces a valid result. The pointer is increased or decreased by the size of the type it points to. For example:

```
#include <stdio.h>
int main(void)
{
    double *ptr, i;

    ptr = &i;
    ptr++;
    printf("Addr:%d\n", ptr);
    ptr--;
    printf("Addr:%d\n", ptr);
    return 0;
}
```

The program displays two addresses with the first one 8 places greater than the second one.

As we'll see in next examples, the `++` and `--` operators are often combined with the `*` indirection operator to process array elements. The value of the expression depends on the operator precedence. Because these combinations are often used, we put them all together for quick reference. For example, let's see the four combinations of `*` and `++`; the same are for `--`.

`i = (*ptr)++;` first the value of `*ptr` is assigned to `i` and then `*ptr` is increased by one.

`i = *ptr++;` first the value of `*ptr` is assigned to `i` and then `ptr` is increased according to its type.

`i = ++*ptr;` first the value of `*ptr` is increased by one and then this value is assigned to `i`.

`i = *++ptr;` first the value of `ptr` is increased according to its type and then the value of `*ptr` is assigned to `i`.

## Subtracting and Comparing Pointers

The result of subtracting two pointers is the number of elements between them. The pointers should point to the elements of the same array or in the next place right after the end of the array. If the value of the subtracted pointer is greater, the result would be the same with a negative sign. The type of the result is implementation dependent, but is defined as `ptrdiff_t` (typically `int`) in the C library. For example, if `p1` points to the second element and `p2` points to the fifth element of the same array, the result of `p2-p1` is 3, while the result of `p1-p2` is -3.

The result of comparing two pointers with the `==` and `!=` operators is reliable. Using the relational operators `<`, `<=`, `>`, and `>=` the result is valid if both point to parts of the same object (e.g., array, structure); otherwise, it is undefined. For example, the result of `p2 > p1` is 1.

---

## Exercises

C.8.12 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, i = 0;

    for(ptr = &i; *ptr < 5; i++)
    {
        (*ptr)++;
        ++*ptr;
        printf("%d ", i);
    }
    return 0;
}
```

**Answer:** Since `ptr` points to the address of `i`, `*ptr` is equal to `i`. Therefore, the statement `(*ptr)++;` is equivalent to `i++`; Since the statement `++*ptr;` is equivalent to `++i;`, the program displays: 2 5.

C.8.13 Write a program that uses three pointers to read the grades of a student in three tests. If all grades are greater than or equal to 5, the program should display them in ascending order. Otherwise, the program should display their average.

```
#include <stdio.h>
int main(void)
{
    float *ptr1, *ptr2, *ptr3, i, j, k;

    ptr1 = &i;
    ptr2 = &j;
    ptr3 = &k;

    printf("Enter grades: ");
```

```

scanf("%f%f%f", ptr1, ptr2, ptr3);

if((*ptr1 >= 5) && (*ptr2 >= 5) && (*ptr3 >= 5))
{
    if(*ptr1 <= *ptr2 && *ptr1 <= *ptr3)
    {
        printf("%f ", *ptr1);
        if(*ptr2 < *ptr3)
            printf("%f %f\n", *ptr2, *ptr3);
        else
            printf("%f %f\n", *ptr3, *ptr2);
    }
    else if(*ptr2 <= *ptr1 && *ptr2 <= *ptr3)
    {
        printf("%f ", *ptr2);
        if(*ptr1 < *ptr3)
            printf("%f %f\n", *ptr1, *ptr3);
        else
            printf("%f %f\n", *ptr3, *ptr1);
    }
    else
    {
        printf("%f ", *ptr3);
        if(*ptr2 < *ptr1)
            printf("%f %f\n", *ptr2, *ptr1);
        else
            printf("%f %f\n", *ptr1, *ptr2);
    }
}
else
{
    printf("Avg = %.2f\n", (*ptr1 + *ptr2 + *ptr3)/3);
    return 0;
}

```

**C.8.14** Use the p1 and p2 pointers and complete the following program to display the product of even numbers from 10 up to 20.

```

#include <stdio.h>
int main(void)
{
    int *p1, *p2, i, mul;
    ...
}

```

**Answer:**

```

#include <stdio.h>
int main(void)
{
    int *p1, *p2, i, mul;

    p1 = &i;
    p2 = &mul;
    for(*p1 = 10, *p2 = 1; *p1 <= 20; (*p1)+=2)
        *p2 = *p2 * *p1;
}

```

```
    printf("Mul = %d\n", *p2);
    return 0;
}
```

C.8.15 Use the p1, p2, and p3 pointers and complete the following program to read two integers and display the sum of the integers between them. For example, if the user enters 6 and 10, the program should display 24 (7+8+9). The program should force the user to enter numbers less than 100 and the first integer should be less than the second.

```
#include <stdio.h>
int main(void)
{
    int *p1, *p2, *p3, i, j, sum;
    ...
}
```

**Answer:**

```
#include <stdio.h>
int main(void)
{
    int *p1, *p2, *p3, i, j, sum;

    p1 = &i;
    p2 = &j;
    p3 = &sum;
    *p3 = 0;
    do
    {
        printf("Enter two numbers (a < b < 100): ");
        scanf("%d%d", p1, p2);
    } while(*p1 >= *p2 || *p2 > 100);

    (*p1)++;
    while(*p1 < *p2)
    {
        *p3 += *p1;
        (*p1)++;
    }
    printf("Sum = %d\n", *p3);
    return 0;
}
```

Break time. We reach one of the most important sections of C. Listen to *Passenger* from *Iggy Pop (Stooges)* and get on board. Pay extra attention.

---

## Pointers and Arrays

The elements of an array are stored in successive memory locations, with the first one stored at the lowest memory address. The type of the array defines the distance of its

elements in memory. For example, in a `char` array, the distance is one byte, while in an `int` array, it is four bytes.

The close relationship between pointers and arrays is based on the fact that the name of an array can be used as a pointer to its first element.

In a similar way, `arr+1` can be used as a pointer to the second element, `arr+2` as a pointer to the third one, and so on. In general, the following expressions are equivalent:

```
arr == &arr[0]
arr + 1 == &arr[1]
arr + 2 == &arr[2]
...
arr + n == &arr[n]
```

As a result, the following program outputs four times the same value:

```
#include <stdio.h>
int main(void)
{
    int *ptr, arr[5];

    ptr = arr;
    printf("%p %p %p %p\n", ptr, &arr[0], arr, &arr);
    return 0;
}
```

Notice that although the expression `&arr` outputs the same value, it is different from the others. This expression evaluates to a pointer to the whole array, whereas the others evaluate to a pointer to its first element. In particular, the type of `&arr` is “pointer to an array of five integers,” while the type of the others is “pointer to integer.” Tough?

And to make it even harder, if we write `&arr+1` instead of `&arr`, what would be the difference with the other values (would be one, four, or something else, ...)? To answer it, use the type of the expression and pointer arithmetic. We won’t tell you the answer; give it some thought, run the program, check the output, and try to figure out why is that. If you need an explanation, don’t get in contact with us, we are out ...

Since the name of an array can be used as a pointer to its first element, its content is equal to the value of its first element. Therefore, `*arr` is equal to `arr[0]`. In a similar way, since `arr+1` is a pointer to the second element, `*(arr+1)` is equal to `arr[1]`, and so on. In general, the following expressions are equivalent:

```
*arr == arr[0]
*(arr+1) == arr[1]
*(arr+2) == arr[2]
...
*(arr+n) == arr[n]
```

The parentheses are necessary because the `*` operator has higher precedence than the `+` operator. Therefore, the expressions `*(arr+n)` and `*arr + n` are evaluated in a different way. For example, consider the following program:

```
#include <stdio.h>
int main(void)
```

```

{
    int *ptr, arr[] = {10, 20, 30, 40, 50};

    ptr = arr;
    printf("Val1 = %d Val2 = %d\n", *ptr+2, *(ptr+2));
    return 0;
}

```

With the statement `ptr = arr;` `ptr` becomes equal to the address of `arr[0]`, so `*ptr` is equal to `arr[0]`, that is, 10. Since the `*` operator has higher precedence than `+`, `*ptr+2` is equal to  $10+2 = 12$ .

The expression `*(ptr+2)` evaluates to the content of the address that `ptr` points to, incremented by two integers' positions. Therefore, `*(ptr+2)` is equal to `arr[2]`. As a result, the program displays: `Val1 = 12 Val2 = 30`

Notice that we don't write `ptr = &arr;` because `&arr` corresponds to a pointer to an array, while `ptr` is a pointer to an integer.

The following program uses array subscripting and pointer arithmetic to display the addresses and the values of `arr` elements.

```

#include <stdio.h>
int main(void)
{
    int i, arr[5] = {10, 20, 30, 40, 50};

    printf("***** Using array notation *****\n");
    for(i = 0; i < 5; i++)
        printf("Addr = %p    Val = %d\n", &arr[i], arr[i]);

    printf("\n***** Using pointer notation *****\n");
    for(i = 0; i < 5; i++)
        printf("Addr = %p    Val = %d\n", arr+i, *(arr+i));
    return 0;
}

```

To process an array, our preference is to use array subscripting rather than pointer arithmetic. The reason is to get a clearer and less error-prone code. In other words, we consider that `arr[i]` is easier to read and safer than `*(arr+i)`. For example, the lack of parentheses introduces a bug, which is not so easy to trace. As far as performance is concerned, the code transformations that to date optimizing compilers make will most probably produce the same efficient code. However, because both ways are quite popular, you must learn and use both of them.

What really matters is to remember that the compiler converts `arr[i]` to `*(arr+i)`. We could say that `arr[i]` is an elegant shortcut for `*(arr+i)`. You'll encounter a really weird application of that rule in an exercise later on. Look forward for your answer; the trap is set.

The following program uses another pointer variable to display the values and the addresses of `arr` elements.

```

#include <stdio.h>
int main(void)
{
    int *ptr, i, arr[5] = {10, 20, 30, 40, 50};

```

```
ptr = arr;
for(i = 0; i < 5; i++)
{
    printf("Addr = %p    Val = %d\n", ptr, *ptr);
    ptr++; /* ptr becomes equal to the memory address of the
next array element. Equivalently, we can write ptr = &arr[i]; */
}
return 0;
```

The statement `ptr = arr;` makes `ptr` point to the first element, while each execution of `ptr++;` makes `ptr` point to the next element.

---

When the name of an array is used as a pointer, C treats it as `const` pointer. Therefore, you are not allowed to change its value and make it point elsewhere. An array is not a modifiable *lvalue*; its value will always be equal to the memory address of its first element.

Therefore, if we write `arr++;` or `arr = &i;` in our last example, the compiler will display an error message for illegal action. Now, you can understand what you were told in Chapter 7, that it is illegal to write `b = a;` in order to copy the elements of the array `a` into `b`. However, you are allowed to copy its value in a pointer variable, as we did with `ptr = arr;` and use that pointer to access the array elements.

Although arrays and pointers are closely related, it must be clear to you that an array is not a pointer and vice versa. For example, the declarations `int a[50];` and `int *a;` are quite different. With the first declaration, the compiler allocates memory for 50 integers. The array can be filled with different values at different times, but it always refers to the same memory address. As said, it is not possible to assign it a new value, that is, it is illegal to write `a = arr.` With the second declaration, the compiler allocates memory to store the value of the pointer (e.g., typically 4 bytes). In contrast to an array, you are allowed to assign a new value to a pointer and make it point somewhere else, for example, it is legal to write `a = arr.`

Just remember that whenever the name of an array is used in an expression, the compiler converts it to a pointer to the address of its first element. Always? Not always, there are a few exceptions:

- a. When it is the operand of the `sizeof` operator. `sizeof` calculates the size of the entire array.
- b. As discussed, when the `&` operator is applied to it. It is the array's address that is retrieved. When discussing complex declarations in Chapter 11, we'll see an example of how to declare and initialize a pointer of that type.
- c. When the array is a literal string initializer.

---

Although a pointer variable is not an array, it can be indexed like an array.

For example, the following program subscripts the pointer variable `ptr` as though it were an array to display the values and the addresses of `arr` elements.

```

#include <stdio.h>
int main(void)
{
    int *ptr, i, arr[5] = {10, 20, 30, 40, 50};

    ptr = arr;
    for(i = 0; i < 5; i++)
        printf("Addr = %p Val = %d\n", &ptr[i], ptr[i]);
    return 0;
}

```

Notice that although `arr[i]` and `ptr[i]` access the same element, the compiler gets there in a different way. For example, to access `arr[2]`, it gets the address of `arr`, adds 8 (as said, we assume that the size of `int` is 4 bytes), and goes to the resulting address to get the value. On the other hand, to access `ptr[2]`, it gets the address of `ptr`, retrieves the value that is stored there, adds 8, and goes to that address to get the value. In other words, `arr[2]` resides 8 places after the address of `arr`, while `ptr[2]` resides 8 places after the address that `ptr` points to.

---

## Exercises

C.8.16 What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int *ptr, arr[] = {10, 20, 30, 40, 50};

    ptr = arr;
    *ptr = 3;

    ptr += 2;
    *ptr = 5;
    printf("%d\n", arr[0]+arr[2]);
    return 0;
}

```

**Answer:** With the statement `ptr = arr`; `ptr` points to the address of `arr[0]`, so `*ptr` is equal to `arr[0]`. Therefore, the statement `*ptr = 3;` is equivalent to `arr[0] = 3;`. The statement `ptr += 2;` makes `ptr` equal to the address of `arr[2]`, so `*ptr` is equal to `arr[2]`. Therefore, the statement `*ptr = 5;` is equivalent to `arr[2] = 5;` As a result, the program displays 8.

C.8.17 What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int i = 10, *p = &i;

```

```

p[0] = 50;
printf("%d\n", i+p[0]);
return 0;
}

```

**Answer:** Since p points to the address of i, we can index it as an array of a single element. Therefore, the statement p[0] = 50; is equivalent to i = 50; and the program displays 100. What would happen if we write p[1] = 50; instead of p[0] = 50;?

The statement p[1] = 50; attempts to change the value of an out-of-bound address, which may cause a program crash.

**C.8.18** Write a program that reads the daily temperatures of January and stores them in an array. Then, the program should read a temperature and display the first day number with a temperature less than this. Use pointer arithmetic to process the array.

```

#include <stdio.h>

#define SIZE 31

int main(void)
{
    int i;
    double temp, arr[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter temperatures: ");
        scanf("%lf", arr+i); /* arr+i is equivalent to &arr[i]. */
    }
    printf("Enter temperature to check: ");
    scanf("%lf", &temp);
    for(i = 0; i < SIZE; i++)
    { /* The braces are not necessary. We put them to make the code
easier to read. */
        if(*(arr+i) < temp)
            break;
    }
    if(i == SIZE)
        printf("No temperature less than %.1f\n", temp);
    else
        printf("The first temperature less than %.1f was %.1f in day
%d\n", temp, *(arr+i), i+1);
    return 0;
}

```

**C.8.19** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int i, *ptr1, *ptr2, arr[] = {10, 20, 30, 40, 50, 60, 70};

    ptr1 = &arr[2];

```

```

ptr2 = &arr[4];
for(i = ptr2 - ptr1; i < 5; i+=2)
    printf("%d ", ptr1[i]);
return 0;
}

```

**Answer:** As said, when two pointers that point to the same array are subtracted, the result is the number of elements between them, not their distance in memory. Therefore, although the address of arr[4] is eight places higher than the address of arr[2], the result of ptr2 - ptr1 is equal to the difference of their subscripts, that is,  $4 - 2 = 2$ . Since ptr1 points to arr[2], ptr1[2] corresponds to arr[4] and ptr1[4] to arr[6]. Therefore, the program outputs 50 and 70.

**C.8.20** Write a program that reads the grades of 10 students, stores them in an array, and displays the best and the worst grade and the positions of their first occurrences in the array. The program should force the user to enter grades within [0, 10]. Use pointer arithmetic to process the array.

```

#include <stdio.h>

#define SIZE 10

int main(void)
{
    int i, max_pos, min_pos;
    float max, min, arr[SIZE];

    max = -1;
    min = 11;
    for(i = 0; i < SIZE; i++)
    {
        do
        {
            printf("Enter grade: ");
            scanf("%f", arr+i);
        } while(*arr+i) > 10 || *arr+i) < 0; /* Check if the
grade is within [0, 10]. */
        if(*arr+i) > max)
        {
            max = *arr+i;
            max_pos = i;
        }
        if(*arr+i) < min)
        {
            min = *arr+i;
            min_pos = i;
        }
    }
    printf("Max grade is %.2f in pos #%d\nMin grade is %.2f in pos
#%d\n", max, max_pos, min, min_pos);
    return 0;
}

```

**C.8.21** Use the `ptr` pointer and an iteration loop and complete the following program to decrement the `arr` elements by one. The program should display their sum, before it terminates. Use pointer arithmetic to process the array.

```
#include <stdio.h>
int main(void)
{
    int *ptr, sum, arr[5] = {11, 21, 31, 41, 51};

    ...
}
```

**Answer:**

```
#include <stdio.h>
int main(void)
{
    int *ptr, sum, arr[5] = {11, 21, 31, 41, 51};

    sum = 0;
    for(ptr = arr; ptr < arr+5; ptr++)
    {
        --*ptr;
        sum += *ptr;
    }
    printf("Sum = %d\n", sum);
    return 0;
}
```

**Comments:** In the first iteration, the statement `ptr = arr;` makes `ptr` point to `arr[0]`. Therefore, `*ptr` is equal to `arr[0]`, that is, 11. The statement `--*ptr;` is equivalent to `--arr[0];`, so `arr[0]` becomes 10. This value is added to `sum`.

In a similar way, the next iterations decrement the remaining elements by one and their values become 20, 30, 40, and 50, respectively. When the loop ends, `sum` is equal to the sum of the new values and the program outputs: `Sum = 150`

**C.8.22** Use the `ptr` pointer and complete the following program to read and store the grades of 50 students into `arr` and then display the array's values in reverse order. Use pointer arithmetic to process the array.

```
#include <stdio.h>

#define SIZE 50

int main(void)
{
    float *ptr, arr[SIZE];
    ...
}
```

**Answer:**

```
#include <stdio.h>
```

```

#define SIZE 50

int main(void)
{
    float *ptr, arr[SIZE];

    ptr = arr;
    while(ptr < arr+SIZE)
    {
        printf("Enter grade: ");
        scanf("%f", ptr);
        ptr++;
    }
    ptr--;
    while(ptr >= arr)
    {
        printf("%f\n", *ptr);
        ptr--;
    }
    return 0;
}

```

C.8.23 What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int *ptr1, *ptr2, i = 10, j = 20;

    ptr1 = &i;
    *ptr1 = 150;

    ptr2 = &j;
    *ptr2 = 50;

    ptr2 = ptr1;
    *ptr2 = 250;

    ptr1 = ptr2;
    *ptr1 += *ptr2;
    printf("%d\n", i+j);
    return 0;
}

```

**Answer:** Since ptr1 points to the address of i, the statement `*ptr1 = 150;` is equivalent to `i = 150;` Similarly, the statement `*ptr2 = 50;` is equivalent to `j = 50;` The statement `ptr2 = ptr1;` makes ptr2 point to the same address that ptr1 points to, that is, the address of i. Therefore, the statement `*ptr2 = 250;` is equivalent to `i = 250.`

The next statement does not change the value of ptr1; that is, it still points to the address of i. Therefore, the statement `*ptr1 += *ptr2;` is equivalent to `i += i;` that is, `i = i+i = 250+250 = 500.` As a result, the program outputs 550.

**C.8.24** Use the `ptr` pointer and complete the following program to read 50 integers and store into `arr` those with a value within [30, 40]. The program should display how many values are stored in the array. The program should set the value -1 to the remaining elements. Use pointer arithmetic to process the array.

```
#include <stdio.h>

#define SIZE 50

int main(void)
{
    int *ptr, i, arr[SIZE];
    ...
}
```

**Answer:**

```
#include <stdio.h>

#define SIZE 50

int main(void)
{
    int *ptr, i, arr[SIZE];

    ptr = arr;
    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%d", ptr);

        if(*ptr >= 30 && *ptr <= 40)
            ptr++;
    }
    printf("%d elements are stored\n", ptr-arr);
    for(; ptr < arr+SIZE; ptr++)
        *ptr = -1;
    return 0;
}
```

**C.8.25** How many compilation errors can you detect?

```
#include <stdio.h>
int main(void)
{
    int i, arr[5] = {10, 20, 30, 40, 50};

    for(i = 0; i < 5; i++)
        printf("%d\n", i[arr]);

    printf("%d\n", 2[arr]-3[arr]);
    return 0;
}
```

**Answer:** Did you escape the trap or not? The normal answer to give is that the expression `i[arr]` is wrong, because `arr` and not `i` is declared as an array. For the same reason, the expressions `2[arr]` and `3[arr]` seem wrong as well. However, the compiler does not complain. Yes, you've read correctly, it is not a typing error; the compiler has no reason to complain about. And now, what shall we do after that concept overturning?

Relax, listen to *Getting Away With It (All Messed Up)* from James and wait for the smoke to clear. Nothing is *messed up*; let's see why. As said, the compiler converts `i[arr]` to `*(i+arr)`, which is equivalent to `*(arr+i)`, equivalent to `arr[i]`. For example, `2[arr]` is equivalent to `arr[2]`. Therefore, the program is compiled successfully and displays the values of the array's elements, as well as the difference of `arr[2]` and `arr[3]`, that is, `-10`.

In the same "abnormal" sense, could we write the following?

```
int *ptr = arr+4;
printf("%d\n", ptr[-2]);
```

Yes, because `ptr[-2]` is converted to `*(ptr-2)`, which is equivalent to `*(arr+4-2)`, that is, `arr[2]`, and the program outputs 30.

Needless to say that we don't recommend this reverse syntax; we just used it to demonstrate the close relationship between arrays and pointers in a really weird way. Forget that you saw it, get it out of your mind, burn the page.

**C.8.26** Use the `p1` and `p2` pointers and complete the following program to read and store 50 integers into `arr1` and `arr2`, respectively. Then, the program should display the values of the common elements and their positions in the two arrays, if found. Use pointer arithmetic to process the arrays.

```
#include <stdio.h>

#define SIZE 50

int main(void)
{
    int *p1, *p2, arr1[SIZE], arr2[SIZE];
    ...
}
```

**Answer:**

```
#include <stdio.h>

#define SIZE 50

int main(void)
{
    int *p1, *p2, arr1[SIZE], arr2[SIZE];
    for(p1 = arr1; p1 < arr1+SIZE; p1++)
    {
        printf("Enter number: ");
        scanf("%d", p1);
    }
}
```

```

printf("\nSecond array\n");
for(p2 = arr2; p2 < arr2+SIZE; p2++)
{
    printf("Enter number: ");
    scanf("%d", p2);
}
for(p1 = arr1; p1 < arr1+SIZE; p1++)
{
    for(p2 = arr2; p2 < arr2+SIZE; p2++)
    {
        if(*p1 == *p2)
            printf("Common element:%d found in positions
%d and %d\n", *p1, p1-arr1+1, p2-arr2+1);
    }
}
return 0;
}

```

**C.8.27** Use the p1 and p2 pointers and complete the following program to read the codes of 100 products and store them into arr. The program should store a code in the array only if it is not already stored. If the user enters -1, the insertion of codes should terminate. The program should display the codes of the stored products, before it terminates. Use pointer arithmetic to process the array.

```

#include <stdio.h>

#define SIZE 100

int main(void)
{
    int *p1, *p2, arr[SIZE];
    ...
}

```

**Answer:**

```

#include <stdio.h>

#define SIZE 100

int main(void)
{
    int *p1, *p2, arr[SIZE];

    p1 = arr;
    while(p1 < arr+SIZE)
    {
        printf("Enter code_%d: ", p1-arr+1);
        scanf("%d", p1);
        if(*p1 == -1)
            break;
        for(p2 = arr; p2 < p1; p2++) /* Traverse the array to check
if the input code is already stored. */

```

```

    {
        if(*p1 == *p2)
        {
            printf("Error: Code %d exists\n", *p1);
            break;
        }
    }
    /* If the code is not stored, increase the pointer. */
    if(p2 == p1)
        p1++;
}
/* Display the codes. */
for(p2 = arr; p2 < p1; p2++)
    printf("C: %d\n", *p2);
return 0;
}

```

**C.8.28** What are the values of arr elements in the following program?

```

#include <stdio.h>
int main(void)
{
    int *ptr, arr[5] = {20};

    for(ptr = arr+1; ptr < arr+4; ptr++)
        *ptr = *(ptr-1) + *(ptr+1) + 1;
    return 0;
}

```

**Answer:** When arr is declared, arr[0] becomes 20 and the remaining elements are set to 20. The statement ptr = arr+1; makes ptr point to arr[1]. In each loop iteration, the statement ptr++ makes it point to the next element. The value of the current element becomes equal to the value of the previous element, plus the value of the next element, plus one. For example, in the first iteration, the statement is equivalent to: arr[1] = arr[0]+arr[2]+1 = 21; As a result, the values of arr[0] up to arr[3] will be 20 to 23, and arr[4] equal to 0.

**C.8.29** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    int a[] = {0, 0, 1, 2, 3}, b[] = {0, 0, 4, 5, 6};
    int *ptr1 = a, *ptr2 = b;

    while(!*ptr1++ && !*ptr2++)
        ;
    printf("%d %d\n", *(b+(ptr1-a)), *(a+(ptr2-b)));
    return 0;
}

```

**Answer:** Yes, you need pen and paper to decipher it. It is a tough one, indeed; sorry to be pushing you. Let's trace the iterations:

*First iteration.* Notice that in `!*ptr1++` the `!` operator is applied first and then `ptr1` is increased. Since `ptr1` points to `a`, `*ptr1` is equal to `a[0]`, that is, 0. The `!` operator makes it 1. Next, `ptr1` is increased and points to `a[1]`. Similarly, the value of `!*ptr2` is 1 and `ptr2` points to `b[1]`. Since both terms are true, the loop continues.

*Second iteration.* Like before, the values of `!*ptr1` and `!*ptr2` are 1. `ptr1` points to `a[2]` and `ptr2` points to `b[2]`.

*Third iteration.* Since `ptr1` points to `a[2]`, `!*ptr1` is 0 and `ptr1` is increased and points to `a[3]`. Here is the key to the answer. Recall from Chapter 4 and the description of the `&&` operator that if an operand is false the rest operands are not evaluated and the value of the expression becomes 0. Therefore, the loop terminates. Since the `!*ptr2++` term is not evaluated, `ptr2` is not increased.

Since `ptr1` points to `a[3]` the result of `ptr1-a` is 3. Therefore, the expression `*(b+(ptr1-a))` is equivalent to `*(b+3)`, that is, `b[3]`. Similarly, since `ptr2` points to `b[2]` the expression `*(a+(ptr2-b))` is equivalent to `*(a+2)`, that is, `a[2]`. Therefore, the program outputs: 5 1.

And to make it even worse, what would be the output if we replace the `&&` operator with the `||` operator and write:

```
while (!*ptr1++ || !*ptr2++) ;
```

If you find it, please let us know...

---

## Array of Pointers

An array of pointers is an array whose elements are pointers to the same data type. To declare such an array, an `*` must prefix its name. For example, the statement:

```
int *p[3] ;
```

declares an array of three pointers to integers.

---

When you declare an array of pointers, don't enclose its name in parentheses.

For example, the statement:

```
int (*p)[3] ;
```

declares `p` as a pointer to an array of three integers and not as an array of three pointers. Don't worry, we'll talk about complex declarations like this one in Chapter 11. The elements of an array of pointers are treated as the ordinary pointers. For example:

```
#include <stdio.h>
int main(void)
{
    int *p[3], i = 100, j = 200, k = 300;
```

```
p[0] = &i;
p[1] = &j;
p[2] = &k;
printf("%d %d %d\n", *p[0], *p[1], *p[2]);
return 0;
}
```

The statement `p[0] = &i;` makes `p[0]` point to the address of `i`, so the value of `*p[0]` is equal to `i`. Similarly, `*p[1]` is equal to `j` and `*p[2]` equal to `k`. Therefore, the program outputs: 100 200 300.

Although we'll talk about strings in Chapter 10, an array of pointers is often used in place of a two-dimensional array to handle strings. One reason is to save memory. For example, the declaration:

```
char names[] [12] = {"First name", "Second name", "Third"};
```

allocates 12 bytes for each row, no matter what the length of the strings is. On the other hand, if we use an array of pointers:

```
char *names[] = {"First name", "Second name", "Third"};
```

the allocated memory matches the length of each string. The elements of the array are pointers to the respective strings. Because of the diverse right-hand ends, this array is often called a *ragged* string array. Let's make a simple question, what is the size of this names array? Easy, count the characters and answer it. Please, read the answer at the next page only if you are absolutely sure; count carefully.

And because we told you to count the characters you did that? Lucky you are, we didn't choose lengthy strings... Here is the answer. Assuming that each pointer allocates 4 bytes, and the array contains three elements, its size is 12 bytes, no matter what the length of the strings is.

Besides saving memory, the use of an array of pointers to handle strings may improve the performance of the program. C.10.37 is such an example.

---

## Exercises

C.8.30 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *p[3], i, num;

    for(i = 0; i < 3; i++)
    {
        printf("Enter number: ");
        scanf("%d", &num);
        p[i] = &num;
    }
    for(i = 0; i < 3; i++)
        printf("Num: %d\n", *p[i]);
    return 0;
}
```

**Answer:** The statement `p[i] = &num;` makes each element of `p` to point to the address of `num`. Since all three pointers point to the same address, their content would be equal to the last value of `num`. Therefore, the second loop outputs three times the last input value.

C.8.31 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *p[3], i, arr[4] = {10, 20, 30, 40};

    for(i = 0; i < 3; i++)
    {
        p[i] = &arr[i]+1;
        printf("%d ", *p[i]);
    }
    return 0;
}
```

**Answer:** The statement `p[i] = &arr[i]+1;` makes each element of `p` point to the address of the next element after `arr[i]`. Therefore, when `i` is 0, `p[0]` points to `arr[1]`, when it is 1, `p[1]` points to `arr[2]`, and when it is 2, `p[2]` points to `arr[3]`. Therefore, the program displays: 20 30 40.

C.8.32 The converse of the famous Pythagoras Theorem says that if a triangle has sides of length  $a$ ,  $b$  and  $c$ , and if:

- $a^2 + b^2 = c^2$ , the triangle is right
- $a^2 + b^2 > c^2$ , the triangle is obtuse
- $a^2 + b^2 < c^2$ , the triangle is acute

Write a program that uses an array of pointers to read the lengths of  $a$ ,  $b$ , and  $c$  sides and determine the type of the triangle. The program should force the user to enter a value for  $c$  greater than the other two.

```
#include <stdio.h>
int main(void)
{
    int *p[3], i, j, k;

    p[0] = &i;
    p[1] = &j;
    p[2] = &k;

    printf("Enter length_a and length_b: ");
    scanf("%d%d", p[0], p[1]);
    do
    {
        printf("Enter length_c: ");
        scanf("%d", p[2]);
    } while((*p[2] <= *p[0]) || (*p[2] <= *p[1]));

    if((*p[0])*(*p[0]) + (*p[1])*(*p[1]) == (*p[2])*(*p[2]))
        printf("Right triangle\n");
    else if((*p[0])*(*p[0]) + (*p[1])*(*p[1]) > (*p[2])*(*p[2]))
        printf("Acute triangle\n");
    else
        printf("Obtuse triangle\n");
    return 0;
}
```

## Pointer to Pointer

Just like an ordinary variable, when a pointer variable is declared, the compiler allocates memory to store its value. Therefore, we can declare another pointer variable to point to this memory address. To declare a pointer to a pointer variable, we use the \* twice. For example, the statement:

```
int **pp;
```

declares  $pp$  as a pointer to another pointer that points to **int**.

To use a pointer to a pointer variable, the single \* provides access to the address of the second pointer, while the double \*\* provides access to the content of the address that the second pointer points to. For example:

```
#include <stdio.h>
int main(void)
{
    int *p, **pp, i = 20;

    p = &i;
    pp = &p;
    printf("%d\n", **pp);
    return 0;
}
```

Since pp points to the address of p, \*pp is equal to p. Since p points to the address of i, \*\*pp is equal to i and the program displays 20.

The most usual application of a pointer to a pointer variable is when we want a function to be able to modify the value of a pointer argument. In Chapter 14, pay attention to the example in the comments of C.14.11. In general, you are allowed to declare a pointer to a pointer to another pointer variable and so on, such as \*\*\*pp, but in practice, it is rarely needed to exceed a depth of two. Let's see an example that combines all together an array, array of pointers, and pointer to a pointer.

```
#include <stdio.h>
int main(void)
{
    int a[] = {10, 20, 30}, *p[] = {a, a+1, a+2}, **pp;

    for(pp = p; pp < p+3; pp++)
        printf("%d ", **pp);
    return 0;
}
```

Initially, pp points to p[0] which points to the address of a[0]. Therefore, \*\*pp is equal to a[0]. When increased, pp points to p[1] which points to a[1]. As a result, the program displays: 10 20 30.

---

## Exercises

C.8.33 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *p, **pp, i = 10, j = 20;

    p = &i;
    pp = &p;
```

```

    **pp += 100;

    p = &j;
    **pp += 100;
    printf("%d\n", i+j);
    return 0;
}

```

**Answer:** Since pp points to p and p points to i, \*\*pp is equal to i. Therefore, the statement `**pp += 100;` is equivalent to  $i = i+100 = 10+100 = 110$ . The statement `p = &j;` makes p point to j, so \*\*pp is equal to j. Therefore, the statement `**pp += 100;` is equivalent to  $j = j+100 = 20+100 = 120$ . As a result, the program displays 230.

**C.8.34** What are the values of elements in the following program?

```

#include <stdio.h>
int main(void)
{
    int k = 0, b = 1, c = 2, d = 3, m, a[3];
    int *p[] = {&k, &b, &c, &d};

    for (m = 0; m < 3; m++)
        a[*p[m]] = **(p+m+1);
    return 0;
}

```

**Answer:** Since  $p[0]$  is equal to `&k`,  $p[1]$  equal to `&b`,  $p[2]$  equal to `&c`, and  $p[3]$  equal to `&d`, the values of  $*p[0]$ ,  $*p[1]$ ,  $*p[2]$ , and  $*p[3]$  are 0, 1, 2, and 3, respectively. In the general case,  $p+m+1$  is a pointer to the  $p[m+1]$  element. Let's trace the iterations:

*First iteration (m=0):*  $a[*p[0]] = **(p+1) = *p[1] = b$ , that is,  $a[0] = 1$ .

*Second iteration (m=1):*  $a[*p[1]] = **(p+2) = *p[2] = c$ , that is,  $a[1] = 2$ .

*Third iteration (m=2):*  $a[*p[2]] = **(p+3) = *p[3] = d$ , that is,  $a[2] = 3$ .

**C.8.35** Use p1 and complete the following program to read continuously an integer and output the word test a number of times equal to the absolute value of the input number. The program should use p2 to display the total number of the output words. If the user enters 0, the insertion of numbers should terminate. Use only `for` loops.

```

#include <stdio.h>
int main(void)
{
    int **p1, **p2, *p3, *p4, i, num, times;
    ...
}

```

**Answer:**

```

#include <stdio.h>
int main(void)

```

```

{
    int **p1, **p2, *p3, *p4, i, num, times;

    p3 = &num;
    p1 = &p3;

    p4 = &times;
    p2 = &p4;

    **p2 = 0;
    for(;;)
    {
        printf("Enter number: ");
        scanf("%d", *p1);
        if(**p1 == 0)
            break;
        if(**p1 < 0)
            **p1 = -**p1;
        for(i = 0; i < **p1; i++)
            printf("test\n");
        **p2 += **p1;
    }
    printf("Total number is = %d\n", **p2);
    return 0;
}

```

---

## Pointers and Two-Dimensional Arrays

Just as pointers are closely related to one-dimensional arrays, they are also related to multidimensional arrays. In this section, we'll focus on the most common case, that of two-dimensional arrays.

Recall from Chapter 7 that when a two-dimensional array is declared, its elements are stored in successive memory locations, starting with the elements of the first row, followed by the elements of the second row, and so on. For example, the following program declares a two-dimensional array and displays the memory addresses of its elements. Their distance is the size of the `int` type.

```

#include <stdio.h>
int main(void)
{
    int i, j, a[2][3];

    for(i = 0; i < 2; i++)
        for(j = 0; j < 3; j++)
            printf("Addr[%d] [%d] :%p\n", i, j, &a[i][j]);
    return 0;
}

```

Also remember that C treats a two-dimensional array as a one-dimensional array, where each element is an array. For example, the elements of `a` are the `a[0]`, `a[1]`, and `a[2]`, where each one is an array of three integers. In particular, C treats `a[0]` as an array and by subscripting its name we've got its elements, that is, `a[0][0]`, `a[0][1]`, and `a[0][2]`. Indeed, if we write `sizeof(a[0])`, the program outputs 12.

To process a two-dimensional array `a[N][M]` through pointer arithmetic, we use the name of each array `a[0]`, `a[1]`, ..., `a[N-1]` as a pointer to the first element of the respective row. For example, since `a[0]` can be used as a pointer to `a[0][0]`, `*a[0]` is equal to `a[0][0]`.

Since `a[0]` points to the first element, `a[0]+1` can be used as a pointer to the second element, that is, `a[0][1]`, and `a[0]+2` as a pointer to the third element, that is, `a[0][2]`. In the general case, `a[0]+j` can be used as a pointer to the `a[0][j]` element of the first row. So, we have:

- `a[0]+j` is equivalent to `&a[0][j]`.
- `*(a[0]+j)` is equivalent to `a[0][j]`.

In a similar way, `a[1]+j` can be used as a pointer to the `a[1][j]` element of the second row. So, we have:

- `a[1]+j` is equivalent to `&a[1][j]`.
- `*(a[1]+j)` is equivalent to `a[1][j]`.

Therefore, in the general case, we have:

- `a[i]+j` is equivalent to `&a[i][j]`.
- `*(a[i]+j)` is equivalent to `a[i][j]`.

and because, as we know by now, `a[i]` is converted to `*(a+i)` we have:

- `*(a+i)+j` is equivalent to `&a[i][j]`.
- `*(*(a+i)+j)` is equivalent to `a[i][j]`.

An example of these equations is that `a[0][0]` is equivalent to `*a[0]` and `**a`. Essentially, when we write `a[i][j]` the compiler converts it to `*(*(a+i)+j)`. To sum up, the following program uses three ways to display the elements of a two-dimensional array.

```
#include <stdio.h>
int main(void)
{
    int i, j, a[2][3] = {10, 20, 30, 40, 50, 60};

    for(i = 0; i < 2; i++)
        for(j = 0; j < 3; j++)
    {
        printf("a[%d] [%d]:%d ", i, j, a[i][j]);
        printf("a[%d] [%d]:%d ", i, j, *(a[i]+j));
        printf("a[%d] [%d]:%d\n", i, j, *(*(a+i)+j));
    }
    return 0;
}
```

Needless to say which one is the simplest ...

Let's add this. Consider the following declarations:

```
int a[2][3], b[10];
```

Our question is: As we say that `b` can be used as a pointer to `b[0]`, is it correct to say that `a` can be used as a pointer to `a[0][0]`?

Because C treats `a` as a one-dimensional array, the answer is no, `a` can be used as a pointer to the first element, that is, `a[0]`. Therefore, if we want to assign `a` to a pointer variable, what should be the type of the pointer? It should be pointer to an array. For example:

```
int (*p)[3]; /* The parentheses are necessary, otherwise the compiler would interpret p as an array of three pointers to integers. */
p = a;
```

and `p` points to the first element of the first row of `a`, that is, `a[0][0]`.

---

Remember, in a declaration such as `int x[5]`, `x` "decays" to a pointer, while in a declaration such as `int y[5][3]`, `y` does not "decay" to a pointer to a pointer, but to a pointer to array.

Make a note of that: the type of the expression `&x` is a pointer to an array of five integers, while the type of the expression `&y` is a pointer to an array of five arrays where each array contains three integers.

And if we write `p++`; where will `p` point to? Yes, we know, it will point to the "*Emergency Exit*," please don't give up, we are aware of the difficulty of these concepts. It makes sense to feel like this, you won't need this stuff for a plain introduction to C, but it might be useful when going deeper. Let's continue, where will `p` point to? Applying pointer arithmetic and because its type is pointer to an array, it will be incremented by the size of the row and point to the first element of the next row. Consider the following program:

```
#include <stdio.h>
int main(void)
{
    int i, a[2][3] = {10, 20, 30, 40, 50, 60}, (*p)[3];

    for(p = a; p < a+2; p++) /* The condition could be written as p <
&a[2] */
    {
        for(i = 0; i < 3; i++)
            printf("%d ", (*p)[i]);
        printf("\n");
    }
    return 0;
}
```

Notice that the type of `p` must be pointer to an array. If it were declared as `int*` or `int**` the program wouldn't compile. Each time `p` is incremented, it points to the first element of the next row and the inner loop displays the elements of that row. What change should we make to display only the elements of a specific row (e.g., the second)?

We'd remove the outer loop and write:  $p = a+1$ ; or equivalently  $p = \&a[1]$ . The inner loop remains as is.

What about if we want to declare pointers to each row of the array? We could use an array of pointers. For example:

```
#include <stdio.h>
int main(void)
{
    int i, a[4][2] = {10, 20, 30, 40, 50, 60, 70, 80}, *p[4] = {a[0],
a[1], a[2], a[3]};
    for(i = 0; i < 4; i++)
        printf("%d ", *(p[i]+1));
    return 0;
}
```

When initialized, each  $p$  element points to the first element of the respective row of  $a$ . For example,  $p[0]$  points to  $a[0][0]$  and  $p[0]+1$  points to  $a[0][1]$ . Therefore, the program displays the values of the second column, that is, 20 40 60 80.

Before finishing a really tough section, the equivalent pointer expressions to access an element of up to a four-dimensional array are:

```
arr[i] = *(arr+i)
arr[i][j] = *(*(arr+i)+j)
arr[i][j][k] = *(*(*arr+i)+j)+k)
arr[i][j][k][l] = *(*(*(*arr+i)+j)+k)+l)
```

---

## Exercises

C.8.36 What does the following program do?

```
#include <stdio.h>
int main(void)
{
    int i, arr[2][5] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    for(i = 0; i < 2; i++)
        *(arr[i]+3) = 0;
    return 0;
}
```

**Answer:** In each iteration,  $arr[i]$  points to the first element of row  $i$ . The expression  $arr[i]+3$  is a pointer to the fourth element of row  $i$ . Therefore,  $*(arr[i]+3)$  is equal to  $arr[i][3]$ . As a result, the program makes zero the elements of the fourth column, so,  $arr[0][3]$  and  $arr[1][3]$  become 0.

C.8.37 What does the following program do?

```
#include <stdio.h>
int main(void)
```

```

{
    int *ptr, arr[2][5] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    for(ptr = arr[1]+2; ptr < arr[1]+5; ptr++)
        *ptr = 0;
    return 0;
}

```

**Answer:** The statement `ptr = arr[1]+2;` makes `ptr` point to the address of `arr[1][2]`. Since `*ptr` is equal to `arr[1][2]`, the statement `*ptr = 0;` is equivalent to `arr[1][2] = 0;`. The statement `ptr++;` makes `ptr` point to the next element of the current row. For example, when `ptr` is first increased, it points to the address of `arr[1][3]` and the next increment makes it point to the address of `arr[1][4]`. Therefore, the program makes zero the last three elements of the second row.

**C.8.38** And because we are “bad characters” and want to press you a bit more, what happens if we change the `for` statement to:

```
for(ptr = arr[0]+4; ptr < arr[0]+7; ptr++)
```

**Answer:** The statement `ptr = arr[0]+4;` makes `ptr` point to the address of `arr[0][4]`. Therefore, the statement `*ptr = 0;` is equivalent to `arr[0][4] = 0;`. Since each row contains five elements, the addresses beyond `arr[0]+4` don’t correspond to array elements. Right?

Don’t think so, see why. When `ptr` is incremented and because the array elements are stored in successive memory locations, `ptr` will point to the first element of the next row. In other words, the address of `arr[0]+5` is the same as of `arr[1][0]`. Similarly, the address of `arr[0]+6` is the same as of `arr[1][1]`. Therefore, the program makes zero the value of the last element of the first row and the first two of the next one.

**C.8.39** Use the `p` pointer and complete the following program to create an identity  $5 \times 5$  matrix. In math, an identity matrix has 1’s on the main diagonal’s elements and 0s everywhere else.

```
#include <stdio.h>

#define SIZE 5

int main(void)
{
    int *p, arr[SIZE][SIZE] = {0};
    ...
}
```

**Answer:**

```
#include <stdio.h>

#define SIZE 5

int main(void)
```

```

{
    int *p, arr[SIZE][SIZE] = {0};

    for(p = &arr[0][0]; p <= &arr[SIZE-1][SIZE-1]; p++)
    {
        if((p-&arr[0][0])%(SIZE+1) == 0)
            *p = 1;
    }
    return 0;
}

```

**Comments:** Since the elements of the array are initialized with 0, it is sufficient to make the elements of the main diagonal equal to 1. The `if` condition checks if the element belongs to the diagonal. The form of that condition is based on the observation that the number of elements between two successive diagonal's elements is SIZE. For example, in the 5x5 array, the elements of the diagonal are in positions 0, 6, 12, 18, and 24.

Also, with this example, we want to show you that it is perfectly legal to treat one two-dimensional as if it were one-dimensional. We exploit the fact that the elements are stored in successive positions in memory.

**C.8.40** Write a program that assigns random integers to a 3x5 array and displays the minimum and the maximum value of each row and column. Use pointer arithmetic to process the array.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS 3
#define COLS 5

int main(void)
{
    int i, j, min, max, arr[ROWS][COLS];

    srand(time(NULL));
    for(i = 0; i < ROWS; i++)
    {
        min = max = *arr[i] = rand();
        for(j = 1; j < COLS; j++)
        {
            *(arr[i]+j) = rand();
            if(*(arr[i]+j) < min)
                min = *(arr[i]+j);
            if(*(arr[i]+j) > max)
                max = *(arr[i]+j);
        }
        printf("Row_%d: Min=%d Max=%d\n", i+1, min, max);
    }
    for(i = 0; i < COLS; i++)
    {
        min = max = *(arr[0]+i);
        for(j = 1; j < ROWS; j++)
        {
            if(*(arr[j]+i) < min)
                min = *(arr[j]+i);
            if(*(arr[j]+i) > max)
                max = *(arr[j]+i);
        }
        printf("Col_%d: Min=%d Max=%d\n", i+1, min, max);
    }
}

```

```

    {
        if(*(arr[j]+i) < min)
            min = *(arr[j]+i);
        if(*(arr[j]+i) > max)
            max = *(arr[j]+i);
    }
    printf("Col_%d: Min=%d Max=%d\n", i+1, min, max);
}
/* Display the array to verify the results. */
for(i = 0; i < ROWS; i++)
{
    for(j = 0; j < COLS; j++)
        printf("%10d", *(arr[i]+j));
    printf("\n");
}
return 0;
}

```

---

## Pointer to Function

Most probably, it'd be better to read this section after reading Chapter 11. However, you may continue and get an idea.

When a function is defined, the compiler allocates memory to store its code. Therefore, every function has an address, as does an ordinary variable. A function pointer points to the memory address, where the function's code is stored. C treats function pointers just like ordinary pointers. For example, we can store them in variables or use them as elements of an array. The general syntax to declare a function pointer is:

```
return_type (*pointer_name) (type_param_1 name_1, type_param_2 name_2,  
..., type_param_n name_n);
```

The **return\_type** specifies the function's return type, while the variables **name\_1**, **name\_2**, ..., **name\_n** indicate the function's parameters, if any. Here are some examples:

```
int (*ptr)(int arr[], int size); /* ptr is declared as a pointer to a  
function, which takes as parameters an array of integers and an integer  
and returns an integer. */  
void (*ptr)(double *arr[]); /* ptr is declared as a pointer to a  
function, which takes as parameters an array of pointers to doubles and  
returns nothing. */  
int test(void (*ptr)(int a)); /* test() returns an integer value and  
takes as parameter a pointer to another function, which takes an integer  
parameter and returns nothing. */
```

The name of the function pointer must be enclosed in parentheses because the function call operator () has higher precedence than the \* operator. For example, the statement:

**int** \*ptr(**int** a); instead of **int** (\*ptr)(**int** a); declares a function named ptr, which takes an integer parameter and returns a pointer to an integer.

An easy way to declare a pointer to a function is first to write the function's prototype and then replace the name of the function with an expression of the form (`*ptr`). Thus, `ptr` is declared as pointer to a function of that prototype. To make a pointer point to a function, the pointer's declaration must match the function's prototype. For example, consider the following program:

```
#include <stdio.h>

void test(int a);

int main(void)
{
    void (*ptr)(int a); /* ptr is declared as a pointer to a function,
which takes an integer parameter and returns nothing. */

    ptr = test; /* ptr points to the address of test(). */
    (*ptr)(10); /* Call the function that ptr points to. */
    return 0;
}

void test(int a)
{
    printf("%d\n", 2*a);
}
```

---

The name of a function can be used as a pointer to its address.

Therefore, the statement `ptr = test;` makes `ptr` point to the address of `test()`. This statement is allowed because the declaration of `ptr` matches the declaration of `test()`. Notice that if we write `ptr = &test;` the effect is the same because the expression `&test` is interpreted as a pointer to `test()`, not as a pointer to a pointer to `test()`.

To call a function through a pointer, we can either use the pointer just like an ordinary call or use the operator `*` to refer to the function. For example, either: `ptr(10);` or `(*ptr)(10);` calls `test()` and the program displays 20. Although the first form is similar to an ordinary call, our preference is the second one, in order to make clear to the reader that `ptr` is a pointer to a function, not a function. Also, we prefer to dereference pointers as we do with ordinary variables, that is, since `ptr` points to the function `*ptr` is the function. However, when using an array of pointers to functions, we usually prefer the first form. Because successive calls to a function through a pointer can use the same syntax, we can extend the second form and write equivalent expressions such as: `(*(*ptr))(10);` or `(*(*(*ptr)))(10);`

In the next example, we write a function that takes as parameters the grades of two students and returns the greater. The program reads two grades and uses a pointer to call the function and display the greater.

```
#include <stdio.h>

float test(float a, float b);

int main(void)
```

```

{
    float (*ptr)(float a, float b); /* ptr is declared as a pointer to
a function, which takes two float parameters and returns a float. */
    float i, j, max;

    printf("Enter grades: ");
    scanf("%f%f", &i, &j);

    ptr = test;
    max = (*ptr)(i, j); /* Call the function that ptr points to. */
    printf("Max = %f\n", max);
    return 0;
}

float test(float a, float b)
{
    if(a > b)
        return a;
    else
        return b;
}

```

We could omit the declaration of `max` and write: `printf("Max = %f\n", (*ptr)(i, j));`

Let's see an example of using a function that takes as parameter a pointer to a function. You may probably wonder, why does this case interest me, am I ever going to need such a function? You are right, we'd most probably wonder the same if we were in your position and pay no attention to that special case. However, there were several situations in which we had to write or call such a function (e.g., we'll see `qsort()` in Chapter 12). Let's see the most usual case that has happened to us.

We use a pointer to a function when we want a part of the program (e.g., B) to notify another part (e.g., A) for the emergence of some event. To achieve this, a function of the part A calls a function of the part B and passes to it a pointer to a function in part A. The function in part B copies that pointer into a variable and that variable is used whenever part B wants to notify part A for some event. This communication method is known as *callback* mechanism. For example, in one of our applications, part B was the controller of a serial port and used that callback mechanism to notify the main program (part A) to receive data. Let's see an example:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void make_rand(void (*ptr)(int evt, int num)); /* Although we could write
make_rand(void ptr(int evt, int num));, we prefer the pointer
declaration. */
void handle_event(int evt, int num);

int main(void)
{
    srand(time(NULL));
    make_rand(handle_event);
    return 0;
}

```

```

void make_rand(void (*ptr)(int evt, int num))
{
    int i, cnt;

    cnt = 0;
    while(1)
    {
        i = rand();
        cnt++;
        if(i >= 1000 && i <= 2000)
        {
            (*ptr)(cnt, i);
            return;
        }
    }
}

void handle_event(int evt, int num)
{
    printf("Times:%d Num:%d\n", evt, num);
}

```

At first, the main program (e.g., part A) calls `make_rand()` (e.g., part B), which takes as parameter a pointer to a function, which returns nothing and accepts two integer parameters. Notice, that we just write the name of `handle_event()`. When the name of a function is not followed by parentheses, the compiler produces a pointer to this function; be careful, it does not call the function. To make sure that you distinguish the two cases, here is an example with two functions a and b:

```

a(b); /* The address of b is passed to a. */
a(b()); /* b is first called and the return value of b is passed to a. */

```

`make_rand()` generates random integers, and once an integer within [1000, 2000] is generated, it uses the pointer to callback `handle_event()` and passes as arguments how many numbers were generated and the number in the specified interval. This example has no programming value, we just wanted to show you an implementation example of the callback mechanism. It might help you in the future.

## Array of Pointers to Functions

An array of pointers to functions is nothing more than an array whose elements are function pointers. Notice that all functions must have the same prototype. For example, the statement:

```
void (*ptr[20])(int a);
```

declares an array of 20 elements, where each element is a pointer to a function that takes an integer parameter and returns nothing. In the following program, each element of the array `ptr` is a pointer to a function, which takes two integer parameters and returns an integer.

```

#include <stdio.h>

int test_1(int a, int b);
int test_2(int a, int b);
int test_3(int a, int b);

int main(void)
{
    int (*ptr[3])(int a, int b);
    int i, j, k;

    ptr[0] = test_1; /* ptr[0] points to the memory address of
test_1(). */
    ptr[1] = test_2;
    ptr[2] = test_3;

    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);

    if(i > 0 && i < 10)
        k = ptr[0](i, j); /* Call the function that ptr[0] points
to. We could also write k = (*ptr[0])(i, j). */
    else if(i >= 10 && i < 20)
        k = ptr[1](i, j); /* Call the function that ptr[1] points
to. */
    else
        k = ptr[2](i, j); /* Call the function that ptr[2] points
to. */
    printf("Val = %d\n", k);
    return 0;
}

int test_1(int a, int b)
{
    return a+b;
}

int test_2(int a, int b)
{
    return a-b;
}

int test_3(int a, int b)
{
    return a*b;
}

```

The program reads two integers, checks the first one, and uses a function pointer to call the respective function. The program displays the return value.

An array of function pointers can be initialized when declared, just like an ordinary array. For example, we could write:

```
int (*ptr[3])(int a, int b) = {test_1, test_2, test_3};
```

Suppose that the user enters the values 20 and 10. What does the following statement output?

```
printf("%d\n", ptr[0](ptr[1](i, j), ptr[2](i, j)));
```

The expression `ptr[1](i, j)` calls `test_2()`, which returns `a-b`, that is, 10. Similarly, the expression `ptr[2](i, j)` calls `test_3()`, which returns `a*b`, that is, 200. Therefore, the expression is translated to `ptr[0](10, 200)` and the program displays the return value of `test_1()`, that is, 210.

When using an array of pointers, we usually prefer to use the pointer's name to call the function because it is easier to read and write and because the reader can figure out that the call is made through a pointer. For example, we prefer to write `ptr[0](ptr[1](i, j), ptr[2](i, j))` than `(*ptr[0])((*ptr[1])(i, j), (*ptr[2])(i, j))`.

And because we know that you are curious, we'll show you an implementation example where an array of pointers to functions can improve the program's performance. This example comes from my experience in the implementation of finite state machines (FSMs) for network protocols. In short, an FSM is a mathematical concept using states and events to describe the operation of the protocol. For example, when an event occurs (e.g., data reception from the network card), the program checks the current state of the protocol, the event code, and calls the proper function to process that event. There are several ways to implement an FSM; using the `switch` statement is one of them. For example:

```

        printf("Error: unexpected event_%d
received at state_(%d)\n", evt_code, state);
                break;
            }

        case STATE_2:
            switch(evt_code)
            {
                case EVT_1:
                    Handle_ST2_Evt1();
                    break;

                case EVT_2:
                    Handle_ST2_Evt2();
                    break;

                ...
                case EVT_N:
                    Handle_ST2_EvtN();
                    break;

                default:
                    printf("Error: unexpected event_%d
received at state_(%d)\n", evt_code, state);
                    break;
            }
            break;

        ...
        default:
            printf("Time to panic: invalid state\n");
            break;
    }
}

```

Alternatively, we can declare a two-dimensional array of pointers and use those pointers to call the event handling functions. For example, suppose that each function does not take parameters and returns an integer value. Then, we could declare an array of pointers like this:

```

typedef int (*Func_Ptr)(void); /* As we'll see in Chapter 14, we use
typedef to create a synonym. */
Func_Ptr func_ptr[MAX_STATES][MAX_EVENTS] = {Handle_ST1_Evt1, Handle_ST1_
Evt2, Handle_ST1_EvtN, Handle_ST2_Evt1, ...};

```

See how much simpler the code becomes:

```

void evt_received(int evt_code)
{
    func_ptr[state][evt_code]();
}

```

Not only is the code much simpler and easier to read, it is more flexible as well. Using an array of pointers, it is easy to remove (e.g., set it to `NULL`) or add a new handling function in some state, if needed. And most importantly, we gain in speed, since the proper function is directly called avoiding the overhead of the `switch` statements.

We've good news and bad news. The good news is that a really tough chapter finished, at last. Close the book and get some rest because the bad news is that the next chapter is bigger and even harder. Don't be scared, just kidding!!!

---

## Unsolved Exercises

**U.8.1** Write a program that uses two pointer variables to read two `double` numbers and display the absolute value of their sum.

**U.8.2** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, i = 10, j = 20, k = 30;

    ptr = &i;
    *ptr = 40;

    ptr = &j;
    *ptr += i;

    ptr = &k;
    *ptr += i + j ;
    printf("%d %d %d\n", i, j, k);
    return 0;
}
```

**U.8.3** Write a program that uses a pointer variable to read a `double` number and display its fractional part. For example, if the user enters -7.21, the program should display 0.21.

**U.8.4** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr1, *ptr2, *ptr3, i = 10, j = 20, k = 30;

    ptr1 = &i;
    ptr2 = &j;
    ptr3 = &k;

    *ptr1 = *ptr2 = *ptr3;
    k = i+j;
    printf("%d\n", *ptr3);
    return 0;
}
```

**U.8.5** Write a program that uses three pointer variables to read three integers and check if they are in successive ascending order (e.g., -5, -4, -3). The program should force the user to enter negative numbers.

**U.8.6** What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, i, j = 1, a[] = {j, j+1, j+2, j+3};

    for(i = 0; i < 3; i++)
    {
        ptr = a+i;
        printf("%d ", a[*ptr]);
    }
    return 0;
}
```

**U.8.7** Write a program that uses three pointer variables to read three integers one after the other. The program should force the user to enter the three numbers in descending order.

**U.8.8** Complete the following program by using the p1 pointer to read 100 integers, the p2 pointer to display the minimum of those with values less than -5, and the p3 pointer to display the maximum of those greater than 10. If no value less than -5 or greater than 10 is entered, the program should display an informative message.

```
int main(void)
{
    int *p1, *p2, *p3, i, num, min, max;
    ...
}
```

**U.8.9** What would be the output of C.8.29 if we replace the `&&` operator with the `||` operator?

**U.8.10** What are the values of a elements in the following program?

```
#include <stdio.h>
int main(void)
{
    int i = 0, a[] = {10, 20, 30, 40, 50}, *p = a;

    while(&p[i] < a+5)
    {
        (*p+i)++;
        p++;
        i++;
    }
    return 0;
}
```

**U.8.11** Use p1, p2, and temp and complete the following program to reverse the elements of arr. Then, use p1 to display the array elements. For example, the new content of arr should be: 2.5 9.4 -3.8 -4.1 1.3

```
#include <stdio.h>
int main(void)
{
    double arr[] = {1.3, -4.1, -3.8, 9.4, 2.5}, temp, *p1 = arr, *p2 =
arr+4;

    ...
}
```

**U.8.12** Write a program that reads 100 integers and stores them in an array. Then, the program should replace the duplicated values with -99. Use pointer arithmetic to process the array. For example, if the array were {5, 5, 5, 5} the program should make it {5, -99, -99, -99, -99} and if it were {-2, 3, -2, 50, 3} it should make it {-2, 3, -99, 50, -99}.

**U.8.13** Use the ptr pointer and complete the following program to read and store integers into arr, with the restriction to store an input number only if it is less than the last stored number. The value -1 should not be stored in the array. If the user enters 0 or if there is no other free place, the insertion of numbers should terminate and the program should display how many numbers were stored in the array. The program should prompt the user to input the numbers, as follows:

```
Enter number_1:
Enter number_2:
...
int main(void)
{
    int *ptr, arr[100];
    ...
}
```

**U.8.14** Use an iterative loop to complete the following program and display the elements of the array in reverse order. It is not allowed to use digits (e.g., to write arr[0], p[1] or p+2).

```
int main(void)
{
    int *p, arr[] = {10, 20, 30, 40};
    ...
}
```

**U.8.15** Use arr and complete the following program to read three integers and display the sum of the even numbers.

```
int main(void)
{
    int *arr[3], i, j, k, m, sum;
    ...
}
```

**U.8.16** Write a program that uses two pointers to two pointer variables to read two integers and swap their values.

**U.8.17** What does the following program do?

```
#include <stdio.h>

#define COLS 4

int main(void)
{
    int i, arr[] [COLS] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
110, 120, 130, 140, 150, 160};

    for(i = 0; i < COLS; i++)
        *(arr[i]+i) = *(arr[i]+COLS-1-i) = 0;
    return 0;
}
```

**U.8.18** Write a program that assigns random integers to a  $5 \times 5$  array and checks if the sum of the elements in the main diagonal is equal to that of its secondary diagonal. Use pointer arithmetic to process the array.

**U.8.19** Write a program that reads integers and stores them in a  $5 \times 5$  array and displays the maximum sum stored in a column and the maximum sum in a row as well. Use pointer arithmetic to process the array.

# Characters

Besides the `int`, `float`, and `double` data types we've mostly used in our programs so far, it is time to discuss more about the `char` type. To show you how to work with characters and strings, we assume that the underlying character set relies on the most popular set, the 8-bit American Standard Code for Information Interchange (ASCII) code. As you can see in Appendix B, ordinary characters, such as letters and digits, are represented by integers from 0 to 255.

C provides a rich set of functions to perform conversions and tests with characters. These functions are declared in the standard file `ctype.h` and don't depend on a particular character set. Therefore, these functions are very useful for writing portable programs. Although we could use them in the next programs, we won't do it, in order to force you program in more depth.

## The `char` Type

Since a character in the ASCII set is represented by an integer between 0 and 255, we can use the `char` type to store its value. Once a character is stored into a variable, it is the character's ASCII value that is actually stored. In the following example:

```
char ch;  
ch = 'c';
```

the value of `ch` becomes equal to the ASCII value of the character '`c`'. Therefore, the statements `ch = 'c'`; and `ch = 99`; are equivalent. Of course, '`c`' is preferable than `99`; not only it is easier to read, but also your program won't depend on the character set as well.

Don't forget to enclose a character constant in single quotes.

For example, if you omit them and write `ch = c`; the compiler will treat `c` as an ordinary variable and its value will be assigned into `ch`. If `c` is not declared, the compiler will produce an error message.

To display a character, we write `%c`, while the `%d` can be used to display its ASCII value. For example:

```
#include <stdio.h>  
int main(void)  
{  
    char ch;  
  
    ch = 'a';
```

```
    printf("Char = %c and its ASCII code is %d\n", ch, ch);
    return 0;
}
```

Let's see the following program. What does it output?

```
#include <stdio.h>
int main(void)
{
    char a = 70, b = 40;

    if('a' > 'b')
        printf("One\n");
    return 0;
}
```

Are the variables a and b related to 'a' and 'b'? Of course not, and since the ASCII value of a is less than b, the program displays nothing.

---

Essentially, when a character appears in an expression, either as constant or variable, C treats it as an integer and uses its ASCII value.

Since C treats the characters as integers, we can use them in numerical expressions. For example:

```
char ch = 'c';
int i;
ch++; /* ch becomes 'd'. */
ch = 68; /* ch becomes 'D'. */
i = ch-3; /* i becomes 'A', that is, 65. */
```

Handling of characters that represent integers usually confuses novice programmers. For example, one way to display 2 is to write `printf("%c", '2');` Be careful, not 2. As shown in Appendix B, the constant '2' corresponds to the character with ASCII value 50, and that is the character 2. Similarly, to test if a character is a digit 0-9 we write

```
if(ch >= '0' && ch <= '9') and not if(ch >= 0 && ch <= 9)
```

Alternatively, we can use the `isdigit()` function declared in `ctype.h` to make this test. For example: `if(isdigit(ch))`

Sometimes, when mixing numeric input with character input the program may behave unexpectedly and make novice programmers crazy enough to smash the computer. Here is a typical example. Do you notice any malfunction in the following program when the user enters an integer and presses *Enter*?

```
#include <stdio.h>
int main(void)
{
    char ch;
    int i;

    printf("Enter number: ");
```

```

scanf("%d", &i);

printf("Enter character: ");
scanf("%c", &ch);

printf("Int = %d and Char = %c\n", i, ch);
return 0;
}

```

When the user enters an integer and presses *Enter*, the generated new line character is stored in `stdin`. The second `scanf()` gets it from `stdin` and stores it automatically into the `ch` variable without letting the user type any other character. The program displays just the input integer. One solution is to make `scanf()` ignore the white spaces such as '`\n`', by adding a space before `%c`: `scanf(" %c", &ch);`

Also, if we reverse the read order, that is, first read the character then the integer, the program will behave normally because `scanf()` skips white spaces before a numeric value.

Let's talk about a special sequence of three characters, the *trigraph sequence*. Because in the past mainly, some systems did not support the following ASCII characters, C defined corresponding trigraph sequences to represent them. All trigraphs begin with `??`.

<code>??=</code>	<code>#</code>	<code>???</code>	<code>]</code>	<code>??!</code>	<code> </code>
<code>??(</code>	<code>[</code>	<code>??'</code>	<code>^</code>	<code>??&gt;</code>	<code>}</code>
<code>??/</code>	<code>\</code>	<code>??&lt;</code>	<code>{</code>	<code>??-</code>	<code>~</code>

Therefore, equivalent C programs can be written even if these characters are missing from the keyboard. For example, what is the output of the following program?

```

??=include <stdio.h>
int main(void)
??<
    int a??(???) = ??<1, ??>;
    printf("%d??/n", a??(0??) ??! a??(1??));
    return 0;
??>

```

The preprocessor replaces the trigraphs with the corresponding ASCII characters and the program is translated to:

```

#include <stdio.h>
int main(void)
{
    int a[2] = {1, 2};
    printf("%d\n", a[0] | a[1]);
    return 0;
}

```

Therefore, the program outputs 3.

Notice that if the sequence `??` is contained in a string, the compiler may treat it as the beginning of a trigraph and replace it. For example, if we write: `printf("ok??=");` the program outputs `ok#`. To avoid this situation, add a `\` before the second `?`, so that it is interpreted as an escape sequence, and `ok??=` is printed.

And yes, if you ever decide to participate in “The International Obfuscated C Code Contest—<http://www.ioccc.org>,” these trigraphs would be useful to write incomprehensible programs.

---

## Exercises

C.9.1 Write a program that displays the upper and lowercase letters and their ASCII values.

```
#include <stdio.h>
int main(void)
{
    int i;

    for(i = 0; i < 26; i++)
        printf("%c(%d)-%c(%d)\n", 'a'+i, 'a'+i, 'A'+i, 'A'+i);
    return 0;
}
```

**Comments:** In ASCII set, the difference between any uppercase letter and the respective lowercase is 32. Therefore, we could write:

```
printf("%c(%d)-%c(%d)\n", 'a'+i, 'a'+i, 'a'+i-32, 'a'+i-32);
```

C.9.2 Write a program that reads three characters and checks if they are consecutive in the ASCII character set.

```
#include <stdio.h>
int main(void)
{
    char ch1, ch2, ch3;

    printf("Enter characters: ");
    scanf("%c%c%c", &ch1, &ch2, &ch3);

    if((ch1+1 == ch2) && (ch2+1 == ch3))
        printf("Consecutive\n");
    else
        printf("Not Consecutive\n");
    return 0;
}
```

**Comments:** If you run the program, don't type a space between the input characters because the space is a character as well (unless the space character is among the characters you want to check).

C.9.3 Write a program that reads two characters and displays the characters between them. For example, if the user enters af or fa, the program should display bcde.

```
#include <stdio.h>
int main(void)
{
    char ch1, ch2;

    printf("Enter characters: ");
    scanf("%c%c", &ch1, &ch2);

    if(ch1 < ch2)
    {
        ch1++;
        while(ch1 != ch2)
        {
            printf("%c", ch1);
            ch1++;
        }
    }
    else
    {
        ch2++;
        while(ch2 != ch1)
        {
            printf("%c", ch2);
            ch2++;
        }
    }
    return 0;
}
```

C.9.4 Write a program that reads a character continuously, and if it is a lowercase letter, it should display the respective uppercase, otherwise the input character. If the last two input characters are ':' and 'q', the program should display how many 'w' and 'x' were entered and then terminate.

```
#include <stdio.h>
int main(void)
{
    char ch, last_ch = 0;
    int sum1 = 0, sum2 = 0;

    while(1)
    {
        printf("Enter character: ");
        scanf("%c", &ch);

        if(last_ch == ':' && ch == 'q') /* If the last input
character is ':' and the current one is 'q', the insertion of characters
should terminate. */

```

```

        break;
    else if(ch >= 'a' && ch <= 'z')
        printf("%c\n", ch-32); /* Display the respective
uppercase letter. */
    else
        printf("%c\n", ch);

    last_ch = ch; /* The input character is stored in last_ch. */
    if(ch == 'w')
        sum1++;
    else if(ch == 'x')
        sum2++;
    getchar();
}
printf("%c:%d times, %c:%d times\n", 'w', sum1, 'x', sum2);
return 0;
}

```

**Comments:** As we'll see later, we use `getchar()` to read the new line character left in `stdin`, when the user presses *Enter*. Alternatively, we could put a space in `scanf()`, as discussed.

**C.9.5** Write a program that displays all lowercase letters in one line, all uppercase letters in a second line, and all characters that represent the digits 0-9 in a third line. Use a single `for` loop.

```

#include <stdio.h>
int main(void)
{
    char ch, end_ch;

    end_ch = 'z';
    for(ch = 'a'; ch <= end_ch; ch++)
    {
        printf("%c ", ch);
        if(ch == 'z')
        {
            ch = 'A'-1; /* Subtract 1, so that the ch++ statement
in the next iteration makes it 'A'. */
            end_ch = 'Z'; /* Change the end character, so that the
loop displays the uppercase letters. */
            printf("\n");
        }
        else if(ch == 'Z')
        {
            ch = '0'-1;
            end_ch = '9';
            printf("\n");
        }
    }
    return 0;
}

```

**Comments:** How about writing the loop without using the `end_ch` variable and any of the `if-else-if` statements? Here you are.

```
for(ch = 'a'; ch != '9'+1; ch++)
{
    printf("%c ", ch);
    ch = (ch == 'z') ? 'A'-1 : (ch == 'Z') ? '0'-1 : ch;
    (ch == 'A'-1 || ch == '0'-1) ? printf("\n") : 1;
}
```

Quite simple, isn't it? We don't think you need any explanation ...

Just kidding, we wanted to show you an incomprehensible version of the same code. Don't forget our advice: write clear code for your own benefit and for the benefit of those who are going to read your code.

---

## The `getchar()` and `putchar()` Functions

The `getchar()` function is used to read a character from `stdin`. It is declared in `stdio.h`:

```
int getchar();
```

`getchar()` starts reading characters when the user presses the *Enter* key. If it is executed successfully, it returns the character read as `unsigned char` cast to `int`. To indicate a read error or when there is no more input, `getchar()` returns a distinctive integer value defined in `stdio.h`, named `EOF` (*End of File*). This value is different from any valid character value, so it cannot be confused with a regular character. Although `EOF` is typically `-1`, it is better to use `EOF`, so that your program does not depend on the specific numeric value.

For example, if the user types abc and presses *Enter*, the first call of `getchar()` returns 'a', the second one returns 'b', the third one 'c', and the fourth one '\n'. If it is called again, the program "stacks" until the user enters new character(s) and presses *Enter* again.

Regarding performance, `getchar()` is executed faster than `scanf()` because `scanf()` is a complex function designed to read various data types, not only characters. Moreover, `getchar()` is usually implemented as a macro to avoid the time overhead of the function call. For example, the next program uses `getchar()` to display the input characters until the user presses *Enter*.

```
#include <stdio.h>
int main(void)
{
    int ch, sum;

    printf("Enter characters: ");
    sum = 0;
    while((ch = getchar()) != '\n' && ch != EOF) /* The inner
parentheses are necessary, because the != operator has greater precedence
than =. */
```

```

    sum++;
    printf("%c", ch);
}
printf("\nTotal number is = %d\n", sum);
return 0;
}

```

`getchar()` returns one by one the input characters until the '`\n`' character is met. Each character is stored into `ch` and that value is tested against EOF to determine if it is valid.

Notice that the return type of `getchar()` should be stored into an `int` variable, not `char`. See why. Suppose that `ch` is declared as `char`. If the `char` type is signed and a character with value 255 is read, it will be stored as -1 into `ch`. As a result, the condition that tests that value against EOF becomes false and the loop stops reading more characters. If the `char` type is unsigned and `getchar()` returns EOF, EOF won't be stored as -1; therefore, the loop won't terminate.

The `putchar()` function writes a character in `stdout`, for example, `putchar('a')`. It returns the character written, or EOF to indicate an error. Since `putchar()` is designed only to print characters, if you want to display a character and performance matters, use `putchar()` instead of `printf()`. Like `getchar()`, `putchar()` is usually implemented as a macro for additional speed.

## Exercises

C.9.6 According to the ITU-T E.161 international standard, the digits of a phone pad correspond to letters as follows: ABC=2, DEF=3, GHI=4, JKL=5, MNO=6, PQRS=7, TUV=8, WXYZ=9. There is no correspondence for the digits 0 and 1. Write a program that reads a phone number up to 10 digits, which may include letters and display that number. For example, if the user enters CALL123456, the program should display 2255123456. The program should accept only uppercase letters and digits; otherwise, it should terminate. Use `putchar()` to write a character.

```

#include <stdio.h>
int main(void)
{
    int ch, dig, arr[26] = {'2', '2', '2', '3', '3', '3', '4', '4',
'4', '5', '5', '5', '6', '6', '6', '7', '7', '7', '7', '8', '8', '8',
'9', '9', '9', '9'}; /* Initialize an array to make right away the
mapping. */
    dig = 0;

    printf("Enter digit or letter: ");
    while((ch = getchar()) != '\n' && ch != EOF)
    {
        if(ch >= 'A' && ch <= 'Z')
        {
            dig++;
        }
    }
}

```

```

        putchar(arr[ch-'A']); /* The output digit depends on
the value of ch. For example, if the user enters 'I' the output would be
arr[73-65] = 4. */
    }
    else if(ch >= '0' && ch <= '9')
    {
        dig++;
        putchar(ch); /* A digit is displayed as is. */
    }
    else
    {
        printf("Error: Not valid character\n");
        return 0;
    }
    if(dig == 10)
        break;
}
if(dig == 10)
    printf("\nNumber completed\n");
return 0;
}

```

**C.9.7** Write a program that reads the digits of a number as characters and displays its value. If the user enters a sign, it should be taken into account. Space characters are allowed only before the number, not between digits. Also, the program should check that the number of digits is up to 10.

```

#include <stdio.h>
int main(void)
{
    int ch, sign, val, dig;

    sign = '+';
    dig = 0;
    val = 0; /* val remains 0 until the user enters the first digit.
Then, if the user enters a non-digit character the program terminates. */
    printf("Enter number: ");
    while((ch = getchar()) != '\n' && ch != EOF)
    {
        if(ch == ' ' || ch == '\t')
        {
            if(val != 0)
            {
                printf("Error: No spaces between digits\n");
                return 0;
            }
        }
        else if(ch == '+' || ch == '-')
        {
            if(val != 0)
            {
                printf("Error: No signs between digits\n");
                return 0;
            }
        }
    }
}

```

```

        sign = ch;
    }
    else if(ch >= '0' && ch <= '9')
    {
        dig++;
        if(dig > 10)
        {
            printf("Error: Maximum number of digits is
passed\n");
            return 0;
        }
        val = 10*val + (ch-'0'); /* To find the numeric value
of the digit the ASCII value of 0 is subtracted. */
    }
    else
    {
        printf("Error: Input is not a digit\n");
        return 0;
    }
}
if(sign == '-')
    val = -val;
printf("%d\n", val);
return 0;
}

```

**Comments:** To find the numeric value of a digit character, we subtract the ASCII value of '0'. For example, if the user enters 4, ch becomes equal to its ASCII value, that is, 52. To get the input digit, we write  $ch - '0' = 52 - 48 = 4$ .

**C.9.8** Write a program that reads an Internet Protocol (IP) version 4 address (*IPv4*) and checks if it is valid. The form of a valid IPv4 address is  $x.x.x.x$ , where each x must be an integer within [0, 255].

```

#include <stdio.h>
int main(void)
{
    int ch, dots, bytes, temp;

    dots = bytes = temp = 0;
    printf("Enter IP address (x.x.x.x): ");

    while((ch = getchar()) != '\n' && ch != EOF)
    {
        if(ch < '0' || ch > '9')
        {
            if(ch == '.')
            {
                dots++;
                if(temp != -1)

```

```

    {
        if(temp > 255)
        {
            printf("Error: The value of each
byte should be in [0, 255]\n");
            return 0;
        }
        bytes++;
        temp = -1; /* The value -1 means that
the current address byte is checked. */
    }
}
else
{
    printf("Error: Acceptable chars are only
digits and dots\n");
    return 0;
}
}
else
{
    if(temp == -1)
        temp = 0; /* Make it 0, to start checking the
next address byte. */
    temp = 10*temp + (ch-'0');
}
}
if(temp != -1) /* Check the value of the last address byte. */
{
    if(temp > 255)
    {
        printf("Error: The value of each byte should be in
[0, 255]\n");
        return 0;
    }
    bytes++;
}
if(dots != 3 || bytes != 4)
    printf("Error: The IP format should be x.x.x.x\n");
else
    printf("The input address is a valid IPv4 address\n");
return 0;
}

```

**C.99** Every mobile phone operating in GSM (2G) and WCDMA (3G) wireless networks is characterized by a unique identifier of 15 digits, called *International Mobile Equipment Identifier* (IMEI). A method to check if the device is really made by the official manufacturer is to compare the IMEI's last digit, called *Luhn digit*, with a check digit. If it is equal, the device is most probably authentic. Otherwise, it is not authentic for sure. The check

digit is calculated as follows: first, we calculate the sum of the first IMEI's 14 digits by adding

- a. The digits in the odd positions.
- b. The double of the digits in the even positions. But if a digit's doubling is a two-digit number, we add each digit separately. For example, suppose that the value of the checked digit is 8. Its double is 16; therefore, we add to the sum the result of  $1+6 = 7$ , not 16.

If the last digit of the calculated sum is 0, the check digit is 0. If not, we subtract the last digit from 10 and that is the check digit. For example, let's check the IMEI 357683036257378. The algorithm applied in the first 14 digits produces

```
3 + (2×5) + 7 + (2×6) + 8 + (2×3) + 0 + (2×3) + 6 + (2×2) + 5 + (2×7) + 3 +  
(2×7) =  
3 + (10) + 7 + (12) + 8 + (6) + 0 + (6) + 6 + (4) + 5 + (14) + 3 + (14) =  
3 + (1+0) + 7 + (1+2) + 8 + (6) + 0 + (6) + 6 + (4) + 5 + (1+4) + 3 +  
(1+4) = 62
```

The check digit is  $10-2 = 8$ , which is equal to the Luhn digit. Therefore, this IMEI is valid.

Write a program that reads the IMEI of a mobile phone (15 digits) and checks if it is authentic or not.

```
#include <stdio.h>  
int main(void)  
{  
    char chk_dig;  
    int i, ch, sum, temp;  
  
    sum = 0;  
    printf("Enter IMEI (15 digits): ");  
    for(i = 1; i < 15; i++) /* Read the first 14 IMEI's digits.*/  
    {  
        ch = getchar();  
        if((i & 1) == 1) /* Check if the digit's position is odd. */  
            sum += ch-'0'; /* To find the numeric value of that  
digit, the ASCII value of 0 is subtracted. */  
        else  
        {  
            temp = 2*(ch-'0');  
            if(temp >= 10)  
                temp = (temp/10) + (temp%10); /* If the digit's  
doubling produces a two-digit number we calculate the sum of these digits.  
*/  
            sum += temp;  
        }  
    }  
    ch = getchar(); /* Read the IMEI's last digit, that is, the Luhn  
digit. */  
    ch = ch-'0';
```

```
chk_dig = sum%10;
if(chk_dig != 0)
    chk_dig = 10-chk_dig;

if(ch == chk_dig)
    printf("*** Valid IMEI ***\n");
else
    printf("*** Invalid IMEI ***\n");
return 0;
}
```

**Comments:** Since the maximum integer has up to 10 digits, we cannot use `scanf()` to read a 15-digit number. Therefore, we use `getchar()` to read each digit separately.

As a new test, check the IMEI of your mobile. How you'll find it? Dial \*#06#. We hope you won't prove to own a fake device as we wouldn't like to spoil your mood now, because a tough chapter follows ...

## Strings

Now that you've seen how to use single characters, it is time to learn how to use strings as well. A string in C is a series of characters that must end with a special character, the *null* character. It is the first character in the American Standard Code for Information Interchange (ASCII) set and it is represented by '\0'. This chapter will teach you how to read and write strings and covers some of the most important string handling functions in the C library.

### String Literals

A string literal is a sequence of characters enclosed in double quotes. C treats it as a nameless character array. The quotes are not considered part of the string but serve only to delimit it. In particular, when the C compiler encounters a string literal, it allocates memory to store its characters plus the null character to mark its end. For example, if the compiler encounters the string literal "message", it allocates eight bytes to store the seven characters of the string plus one for the null character. A string literal may be empty. For example, the string literal "" is stored as a single null character.

### Storing Strings

To store a string in a variable, we use an array of characters. Because of the C convention that a string ends with the null character, to store a string of N characters, the size of the array should be N+1 at least. For example, to declare an array capable of storing a string of up to 7 characters, we write:

```
char str[8];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```
char str[8] = "message";
```

the compiler copies the characters of the "message" into the str array and adds the null character. In particular, str[0] becomes 'm', str[1] becomes 'e', and the value of the last element str[7] becomes '\0'. In fact, this declaration is equivalent to:

```
char str[8] = {'m', 'e', 's', 's', 'a', 'g', 'e', '\0'};
```

Notice that the null character '\0' is one character, not two, since as discussed in Chapter 2, the \ character denotes an escape sequence.

A common error is to confuse the '\0' and '0' characters. The ASCII code of the null character is 0, while the ASCII code of the zero character is 48.

---

As with an ordinary array, if the number of the characters is less than the size of the array, the remaining elements are initialized to 0. If it is greater, it is an error. For example, with the declaration:

```
char str[8] = "me";
```

because the ASCII value of '\0' is 0, str[0] becomes 'm', str[1] becomes 'e' and the rest elements are initialized to 0, or equivalently to '\0'. Similarly, with the declaration:

```
char str[8] = {0};
```

all str elements are initialized to '\0'.

A flexible way to initialize the array is to omit its length and let the compiler compute it. For example, with the declaration:

```
char str[] = "message";
```

the compiler calculates the length of "message" and then allocates eight bytes for str to store the seven characters plus the null character. If we use `sizeof(str)` to output its length, we'll see that it is 8, indeed. Leaving the compiler to compute the length, it is easier and safer, since a careless programmer might make a calculation error or forget to add an extra place for the null character.

---

When declaring an array of characters to store a string, don't forget to reserve an extra place for the null character.

---

If the null character is missing and the program uses a library function to handle the array, unpredictable results may arise because C functions assume that strings are null terminated. Some programmers prefer to emphasize the reservation by adding 1 in the declaration. For example:

```
char s[SIZE+1];
```

Therefore, be cautious, the declarations `char s[] = "abc";` and `char s[3] = "abc";` are different. In the first case the size of s is 4, the null character is stored and we've got a string, while in the second case the size of s is 3 and we've no string.

---

## Exercise

C.10.1 Is the following program error free?

```
#include <stdio.h>
int main(void)
{
```

```
char str1[] = "abc";
char str2[] = "efg";

str2[4] = 'w';
printf("%c\n", str1[0]);
return 0;
}
```

**Answer:** When str1 is declared, the compiler creates an array of four places, to store the 'a', 'b', 'c', and '\0' characters. Similarly, it creates the str2 array with four places, to store the 'e', 'f', 'g', and '\0' characters. The attempt to store the 'w' character in a position that exceeds the length of str2 is wrong. In particular, the assignment str2[4] = 'w' overwrites the data out of str2, which causes unpredictable behavior. For example, the program may display a, but it may also display w if str1 is stored right after str2 in memory.

---

## Writing Strings

The most common functions used to write strings are printf() and puts(). In a simple form, printf() uses the %s format and a pointer to the string. For example, the following program uses the name of the array as a pointer to the first character of the string.

```
#include <stdio.h>
int main(void)
{
    char str[] = "This is text";

    printf("%s\n", str);
    return 0;
}
```

printf() displays the characters of the string beginning from the character that the pointer points to until it encounters the null character. Therefore, the program outputs: This is text. If printf() does not encounter the null character, it continues past the end of the string until it finds a null character somewhere in the memory. Recall from Chapter 2 that if we want to output a specific number of characters, we write %.ns, where n specifies that number.

To display a part of the string, we make the pointer point to the proper position. For example, to display the part beginning from the sixth character, that is, is text, we write printf("%s\n", str+5); or equivalently, printf("%s\n", &str[5]);

If we want to split the string inside printf() in more than one lines, we use the \ character. Notice that the string will be displayed in the same line. For example:

```
printf("This text is displayed \
in the same line\n");
```

Alternatively, because C allows the merging of two or more adjacent literal strings into a single string, we can write:

```
printf("This text is displayed "
"in the same line\n");
```

If the string contains the ", we use the escape sequence \" to print it. For example:

```
printf ("\\"Test\\\""); and the output is: "Test"
```

Once printf() encounters the null character, it stops writing any more characters. For example:

```
#include <stdio.h>
int main(void)
{
    char str[] = "SampleText";

    str[4] = '\0';
    printf ("%s\n", str);
    return 0;
}
```

Since the null character is found in the fifth position, printf() displays Samp and ignores all the rest.

Another function we can use to write strings is the puts(). puts() takes as an argument a pointer to the string to be displayed. Like printf(), puts() writes the characters of the string until the null character is met. After writing the string, puts() writes an additional new line character. For example:

```
#include <stdio.h>
int main(void)
{
    char str[] = "This is text";

    puts(str);
    return 0;
}
```

Because puts() is designed only for writing strings, it tends to be faster than printf(), which is a more complex function.

We wouldn't mention that, but since we've seen it written in several places, we'd like to warn you never use the following way to display a string; it is dangerous. Consider the following program:

```
#include <stdio.h>
int main(void)
{
    char str[100];

    fgets(str, sizeof(str), stdin);
    printf(str);
    return 0;
}
```

We use `fgets()`, we'll see it later, to read a string and store it in `str`. Although it is allowed to call `printf()` in that way, `printf()` expects a pointer as its first argument, as shown in Appendix C, don't do that because if the character `%` is contained in the string, the compiler would interpret it as the beginning of a conversion specification and the behavior of the program would be unpredictable.

## Exercises

C.10.2 Write a program that creates a string with all lowercase and uppercase letters of the English alphabet.

```
#include <stdio.h>
int main(void)
{
    char str[53];
    int i;

    for(i = 0; i < 26; i++)
    {
        str[i] = 'a'+i;
        str[26+i] = 'A'+i;
    }
    str[52] = '\0'; /* At the end, we add the null character. */
    printf("%s\n", str);
    return 0;
}
```

**Comments:** In each iteration, the ASCII value of the respective character is stored in `str`. For example, in the first iteration ( $i=0$ ), we have  $\text{str}[0] = 'a'+0 = 'a' = 97$  and  $\text{str}[26] = 'A'+0 = 'A' = 65$ .

C.10.3 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    char str[] = "Right'0'Wrong";

    printf("%s\n", str);
    return 0;
}
```

**Answer:** Did you answer Right? Sorry, it is wrong.

The '0' does not represent the null character '\0', but it consists of the ' character, the zero character, and another '. Therefore, the program outputs Right'0'Wrong.

What would be the output if we had written the following?

```
char str[] = "Right'\0'Wrong";
```

Since after the first ' the escape sequence \0 represents the null character, this time, you are right if you answered Right'.

---

## Pointers and String Literals

Since a string literal is stored as an array of characters, we can use it also as a pointer of type `char*` to the first character. The purpose of the next program is to show you this conversion in a really odd way. Although it looks absolutely weird, the program uses the string literal in two ways, as an array and as a pointer, and displays twice the fifth character of the string "message", that is, 'a'.

```
#include <stdio.h>
int main(void)
{
    printf("%c %c\n", "message"[4], *(("message"+4));
    return 0;
}
```

Don't get stuck on this program; what is really important to understand is that expressions such as "c" and 'c' are different. The expression "c" is a string literal, which is stored in the memory as an array of two characters, the 'c' and '\0'. Essentially, it is represented by a pointer to that memory. On the other hand, the expression 'c' is just the character constant 'c' represented by its ASCII code.

A convenient way to handle a string literal is to declare a pointer variable and make it point to that string. For example:

```
#include <stdio.h>
int main(void)
{
    char *ptr;
    int i;

    ptr = "This is text";
    for(i = 0; ptr[i] != '\0'; i++)
        printf("%c %c\n", *(ptr+i), ptr[i]);
    return 0;
}
```

With the statement `ptr = "This is text";` the compiler allocates memory to store the string literal and the null character. Then, `ptr` points to the first character of the string, as depicted in Figure 10.1.



**FIGURE 10.1**

Pointers and string literals.

To access the characters of the string, we can use either pointer arithmetic or the pointer as an array and use array subscripting. The program uses both ways to display one by one the characters of the string. The loop is executed until the null character is met.

Because the memory that the compiler allocates to store a string literal is typically read-only, it might not be allowed to modify it.

For example, an error message may be displayed when the following program is executed:

```
#include <stdio.h>
int main(void)
{
    char *ptr = "This is text";

    ptr[0] = 'a';
    return 0;
}
```

As we've seen, we can use an array of characters to store a literal string. However, the declarations

```
char ptr[] = "This is text"; and char *ptr = "This is text";
```

although they may look similar, have quite different properties, as you already know from the section “Pointers and Arrays” in Chapter 8. In particular, the first declaration declares `ptr` as an array of 13 characters. We can store another string into `ptr`, provided that its length won't be more than 13 characters; otherwise, the program may behave unpredictably. The second declaration declares `ptr` as a pointer and makes it point to the memory that holds the string literal. As discussed, in the first case, we can write `ptr[0] = 'a'`; without any concern, while in the second case, this statement may cause a runtime error. The `ptr` pointer can point to another string during program execution, no matter what its length is. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    char *ptr = "First text";

    ptr = "This is a new text";
    printf("%c\n", *ptr);
    return 0;
}
```

At first, `ptr` points to the first character of the string literal "First text". Then, the statement `ptr = "This is a new text";` makes `ptr` point to the new memory allocated

to store that string. Since `ptr` points to its first character, `*ptr` is equal to 'T' and the program displays T.

As we've outlined in Chapter 8, before using a pointer, it must point to a valid address. Consider the following program:

```
#include <stdio.h>
int main(void)
{
    char *ptr;

    ptr[0] = 'a';
    ptr[1] = 'b';
    ptr[2] = '\0';
    printf("%s", ptr);
    return 0;
}
```

Are the assignment `ptr[0] = 'a'`; and the next ones correct? Certainly not. Since `ptr` has not been initialized, writing the 'a', 'b', and '\0' characters somewhere in memory would cause the unpredictable behavior of the program. Even if the program displays ab, the code is erroneous.

---

Never ever forget that using an uninitialized pointer variable is a serious error.

Had we declared an array of characters and make `ptr` to point to it, like this:

```
char str[3], *ptr;
ptr = str;
```

the program would have executed successfully.

---

## Exercises

C.10.4 The following program stores two strings in two arrays, swaps them, and displays their new content. Is there any error?

```
#include <stdio.h>
int main(void)
{
    char temp[100], str1[100] = "Let see", str2[100] = "Is everything
OK?";

    temp = str1;
    str1 = str2;
    str2 = temp;
    printf("%s %s", str1, str2);
    return 0;
}
```

**Answer:** Recall from Chapter 8 that the name of an array when used as a pointer is a constant pointer, meaning that it is not allowed to change its value and point to some other address. Therefore, the statement `temp = str1;` is illegal. The same applies for the next two statements, so the program won't compile.

#### C.10.5 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    char str1[] = "test", str2[] = "test";
    (str1 == str2) ? printf("One\n") : printf("Two\n");
    return 0;
}
```

**Answer:** Since the names of the two arrays are used as pointers, the expression `str1 == str2` checks if the pointers point to the same address, not if the arrays are the same. Do `str1` and `str2` point to the same address?

Of course not; `str1` and `str2` have the same content, but they are stored in different memory. Therefore, the program displays Two.

What would be the output if we write:

```
(*str1 == *str2) ? printf("One\n") : printf("Two\n");
```

Since `str1` can be used as a pointer to its first element, `*str1` is equal to 't'. Similarly, `*str2` is equal to 't'. Therefore, the program would display One.

---

## Read Strings

Like `printf()`, `scanf()` uses the `%s` conversion specification to read a string. By default, `scanf()` reads characters until it encounters a white-space character (i.e., space, tab, or new line character). Then, it appends a null character at the end of the string. In the following program, suppose that the user enters the string `this is the text`:

```
#include <stdio.h>
int main(void)
{
    char str[20];
    int i = 10;

    printf("Enter text: ");
    scanf("%s", str);
    printf("%s %d\n", str, i);
    return 0;
}
```

`scanf()` takes as an argument a pointer to the array that will hold the input string. Since we're using the name of the array as a pointer, we don't add the address operator & before its name. Because `scanf()` stops reading once it encounters the space character, only the word `this` is stored into `str`. Therefore, the program outputs `this`. To force `scanf()` to read multiple words, we can use a more complex form such as `scanf("%[^\\n]", str);`

---

`scanf()` does not check if there is available memory to store all input characters. Therefore, if the user enters more characters than the allocated memory, the program will behave unpredictably.

For example, suppose that in the previous program the user enters a string with more than 20 characters. The first 20 characters will be stored into `str`, but the rest will be stored into a memory out of the bounds of the array, which causes the unpredictable behavior of the program. For example, if the value of `i` is stored into this memory, this value will change. To avoid a memory overflow, use `%ns` instead of `%s`, where `n` specifies the maximum number of characters to be stored.

Alternatively, we can use the `fgets()` function. Read its description in Chapter 15 to get an idea. Another function to read strings is `gets()`. It is declared in `stdio.h`:

```
char *gets(char *str);
```

Like `scanf()`, `gets()` reads characters from `stdin` and stores them in the memory pointed to by `str`. Because it is not possible to specify the maximum number of the characters to be read, `gets()` is extremely unsafe. There is always the possibility of memory overflow. Take a break, and read on the Internet how a virus program exploited this vulnerability of `gets()` to infect thousands of machines in 1988.

---

Don't use `gets()` to read strings. It is unsafe.

`fgets()` is safer than `gets()` because the programmer can specify the maximum number of characters to be read.

---

The pointer that is passed to a read function must point to a memory allocated to store the string.

For example, is the following code well written?

```
char *p, s[10];
p = s;
fgets(p, 10, stdin);
```

Yes, because `p` points to allocated memory. If the statement `p = s;` was missing, the code would be wrong. And, if in `fgets()`, we write `sizeof(p)` instead of 10, would be the same? Of course not, `sizeof` returns the size of the pointer, not the size of the memory that the pointer points to.

If your application runs in a system with limited memory and you don't want to allocate more memory than needed, make a loop and use `getchar()` to read characters one by one

until the new line character is met or EOF is returned. As we'll see in Chapter 14, we can store the characters into a dynamically allocated memory (e.g., initial size of 500 bytes). Each time the memory becomes full, use `realloc()` to increase its size by adding its initial length. For example, the first time it becomes full, add 500 to make its size 1000 bytes, the second time, add another 500 bytes to make it 1500, and so on. Once all the characters are read, make a last call to `realloc()` to shrink the size of the allocated memory and make it equal to the length of the input string.

*For simplicity, we are going to use `fgets()` to read strings, assuming that the maximum length of the input string would be a reasonable number, for example, less than 100 characters.*

In particular, we implemented the `read_text()` function. Although we've not discussed about functions yet, it is not so hard to realize how it works. It uses `fgets()` to read characters from `stdin`, which, as we've said, is associated with the keyboard by default. It stores them in the `str` array of length `size` bytes. Because the '`\n`' might be stored, the function provides the ability to remove it by passing a nonzero value of `flag`. The function uses `strlen()`, we'll see it shortly, to return the length of the string. If something goes wrong, the `exit()` function is called.

The `exit()` function is declared in `stdlib.h` and terminates the program. It takes as an argument an integer that indicates the termination status. The value 0 indicates normal termination. Also, C provides the `EXIT_SUCCESS` and `EXIT_FAILURE` macros declared in `stdlib.h`. Their values depend on the implementation; typical values are 0 and 1. The main difference between the `return` statement and the `exit()` function is when they are called from inside a function. In particular, `return` terminates the function, while `exit()` terminates the program.

```
int read_text(char str[], int size, int flag)
{
    int len;

    if(fgets(str, size, stdin) == NULL)
    {
        printf("Error: fgets() failed\n");
        exit(EXIT_FAILURE);
    }
    len = strlen(str);
    if(len > 0)
    {
        if(flag && (str[len-1] == '\n'))
        {
            str[len-1] = '\0';
            len--;
        }
    }
    else
    {
        printf("Error: No input\n");
        exit(EXIT_FAILURE);
    }
    return len;
}
```

To save space, we won't repeat its code in the programs that use it. We'll just add its prototype to remind you how it is used.

An unexpected behavior that makes many novice programmers wonder why their program does not work correctly arises when the program reads a string after having read a numerical value. In C.9, we saw a quite similar example. As another example, suppose that the user enters an integer and presses *Enter*. What does the program output?

```
#include <stdio.h>
int main(void)
{
    char str[100];
    int num;

    printf("Enter number: ");
    scanf("%d", &num);

    printf("Enter text: ");
    fgets(str, sizeof(str), stdin);

    printf("%d %s\n", num, str);
    return 0;
}
```

`scanf()` reads the integer and stores it in `num`. The new line character generated when *Enter* is pressed is stored in `stdin`, and `fgets()` will read it and store it in `str`. Since `fgets()` terminates once the new line character is read, the user cannot enter more characters. Therefore, the program outputs only the input number. A solution is to use `getchar()` before `fgets()`, in order to read the new line character.

---

## Exercises

C.10.6 Write a program that reads a string of less than 100 characters and displays the number of its characters, the number of 'b' occurrences, and the input string after replacing the space character with the new line character and 'a' with 'p'. Don't use the return value of `read_text()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100];
    int i, cnt;

    printf("Enter text: ");
    read_text(str, sizeof(str), 1);

    cnt = 0;
    for(i = 0; str[i] != '\0'; i++)
    {
        if(str[i] == ' ')
            str[i] = '\n';
        else if(str[i] == 'a')
            str[i] = 'p';
        else if(str[i] == 'b')
            cnt++;
    }
    printf("Len = %d Times = %d\nText = %s\n", i, cnt, str);
    return 0;
}

```

**Comments:** The loop is executed until the end of the string is met, that is, once `str[i]` becomes equal to '`\0`'. After the loop ends, the value of `i` specifies the length of the string.

**C.10.7** Use a pointer variable to replace the `for` loop of the previous program with a `while` loop. Use this pointer to make the replacements and calculate the length of the input string, as well.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);
int main(void)
{
    char *ptr, str[100];
    int cnt;

    printf("Enter text: ");
    read_text(str, sizeof(str), 1);

    cnt = 0;
    ptr = str;
    while(*ptr != '\0')
    {
        if(*ptr == ' ')
            *ptr = '\n';
        else if(*ptr == 'a')
            *ptr = 'p';
        else if(*ptr == 'b')
            cnt++;
    }
}
```

```

        cnt++;
        ptr++;
    }
    printf("Len = %d Times = %d\nText = %s\n", ptr-str, cnt, str);
    return 0;
}

```

**C.10.8** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    char *str = "this";
    for(; *str; printf("%s ", str++));
    return 0;
}

```

**Answer:** The expression `*str` is equivalent to `*str != 0`. Therefore, the loop is executed until the null character is met. In the first iteration, `str` points to the first character of the string. Therefore, `printf()` displays `this` and `str` is advanced to point to the next character. The next call of `printf()` displays `his`, and so on. Therefore, the program outputs: `this his is s`

**C.10.9** What is the output of the following program?

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char *ptr[3], str[100];
    int i;

    for(i = 0; i < 3; i++)
    {
        printf("Enter text: ");
        read_text(str, sizeof(str), 1);
        ptr[i] = str;
    }
    for(i = 0; i < 3; i++)
        printf("%c\n", *ptr[i]);
    return 0;
}

```

**Answer:** The `ptr` variable is declared as an array of three pointers to character. In each iteration, the statement `ptr[i] = str;` makes all pointers up to `i` point to the first character

of the string stored in str array. Since all pointers point to the same address, the second loop displays three times the first character of the last input string.

C.10.10 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    char *p, *q, s[] = "play";

    p = s+1;
    q = s;
    p[1] = 'x';
    *s = 'a';
    printf("%d %c\n", *q+2, *(q+2));
    return 0;
}
```

**Answer:** Since p points to the address of the second element of the s array, s[2] changes to 'x'. Also, s[0] becomes 'a'. Because q points to s[0] and the \* operator has greater precedence than the + operator, we have \*q+2 = 'a'+2. Therefore, the program outputs the ASCII value of the character two places after 'a'. This is 'c' and the program outputs 99. Since the value of \*(q+2) is equal to s[2], the program outputs x, as well.

C.10.11 Write a program that uses getchar() to read characters and creates a string with more than 5 characters and less than 100. If the user enters a number of characters out of this interval, the insertion of characters should be repeated.

```
#include <stdio.h>

#define MIN_SIZE 5
#define MAX_SIZE 100

int main(void)
{
    char str[MAX_SIZE];
    int i, ch;

    printf("Enter text (> %d && < %d): ", MIN_SIZE, MAX_SIZE);
    while(1)
    {
        i = 0;
        while((ch = getchar()) != '\n' && ch != EOF)
        {
            if(i < MAX_SIZE-1)
            {
                str[i] = ch;
                i++;
            }
        }
    }
}
```

```

        else
            i++;
    }
    if((i > MAX_SIZE-1) || (i <= MIN_SIZE))
        printf("Enter text (> %d && < %d): ", MIN_SIZE,
MAX_SIZE);
    else
    {
        str[i] = '\0';
        break;
    }
}
printf("%s\n", str);
return 0;
}

```

**C.10.12** Write a program that reads a string of less than 100 characters and displays a message to indicate if it is a palindrome, which means if it can be read the same in either direction. For example, the string level is a palindrome, since it is read the same in both directions.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100];
    int i, diff, len;

    printf("Enter text: ");
    len = read_text(str, sizeof(str), 1);

    diff = 0;
    for(i = 0; i < len/2; i++)
    {
        if(str[i] != str[len-1-i]) /* If two characters are not the
same, the loop terminates. */
        {
            diff = 1;
            break;
        }
    }
    if(diff == 1)
        printf("%s is not a palindrome\n", str);
    else
        printf("%s is a palindrome\n", str);
    return 0;
}

```

**Comments:** The loop compares the characters from the first character up to the middle one with the characters from the last one back to the middle. That's why the loop is executed from 0 up to `len/2`. The last character is stored in `str[len-1]`.

C.10.13 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    char *str = "Example";
    int *ptr = (int*)str;

    ptr++;
    printf("%s\n", (char*)ptr+3);
    return 0;
}
```

**Answer:** Since `ptr` is declared as a pointer to `int`, the statement `ptr++;` makes it point to the fifth character of the string. In `printf()`, since the type of `ptr` is cast to `char*`, the program displays the part of the string from the eighth character and on. Since the eighth character is the null character, the program displays nothing.

C.10.14 Write a program that reads a string of less than 100 characters, and if it is less than three characters, the user should enter a new one. Then, the program should read a character and check if the string contains the input character three times in a row. The program should display the position of the first triad found.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char ch, str[100];
    int i, len;

    do
    {
        printf("Enter text (more than 2 chars): ");
        len = read_text(str, sizeof(str), 1);
    } while(len < 3);

    printf("Enter character: ");
    ch = getchar();

    for(i = 0; i <= len-3; i++)
        if(str[i] == ch && str[i+1] == ch && str[i+2] == ch)
```

```
{  
    printf("There are three successive '%c's in position  
%d\n", ch, i+1);  
    return 0;  
}  
printf("There aren't three successive '%c's\n", ch);  
return 0;  
}
```

---

## String Functions

The C library provides a rich set of functions to perform operations on strings. This section presents some of the most commonly used. We won't cover every aspect of them, but we'll give you enough of what you need in order to use them in your programs. Although we have yet to discuss functions, we believe that you can understand how they work.

### The `strlen()` Function

The `strlen()` function is declared in `string.h`:

```
size_t strlen(const char *str);
```

The `size_t` type is defined in the C library as an unsigned integer type (usually as `unsigned int`). `strlen()` returns the number of characters in the string pointed to by `str`, not counting the null character. For example:

```
int len;  
len = strlen("Test"); /* len is 4. */  
char *p = "";  
len = strlen(p); /* len is 0. */
```

The pointer is declared as `const`, so that `strlen()` cannot modify the content of the string. In Chapter 11, we'll implement the `str_len()` function, to show you an implementation example of `strlen()`.

---

## Exercises

C.10.15 What is the output of the following program?

```
#include <stdio.h>  
#include <string.h>  
int main(void)  
{  
    char str[] = "Text";  
  
    printf("%d %d\n", strlen(str+4), strlen("Text"+1));  
    return 0;  
}
```

**Answer:** Since the name of an array can be used as a pointer to its first element—we do know that we've written it many times, we just want to be sure that you've got it—`str+4` is a pointer to the fifth character of `str`, that is, the null character. Therefore, the first `strlen()` returns 0. Since the string literal can be used as a pointer, the second `strlen()` returns the number of the characters after the second one. As a result, the program outputs: 0 3.

**C.10.16** Write a program that reads a string of less than 100 characters and displays it after replacing all 'a' characters that may appear at the beginning and at the end of the string with the space character. For example, if the user enters "aabcaaa", the program should display " bcd ", while if the input string is "bcdaa", the program should display "bcd ".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100];
    int i, len;

    printf("Enter text: ");
    len = read_text(str, sizeof(str), 1);

    for(i = 0; i < len; i++)
    {
        if(str[i] == 'a')
            str[i] = ' ';
        else
            break;
    }
    for(i = len-1; i >= 0; i--)
    {
        if(str[i] == 'a')
            str[i] = ' ';
        else
            break;
    }
    printf("Text: %s\n", str);
    return 0;
}
```

**Comments:** The first loop compares the characters in the beginning of the string with 'a'. If it is an 'a', it is replaced with the space character; otherwise, the loop is terminated. Similarly, the second loop replaces all 'a' characters at the end of the string with the space character.

**C.10.17** Write a program that reads continuously strings (less than 100 characters each). The program should copy each input string in a second string variable after replacing each single '\*' with a double '\*\*', and the lowercase letters with uppercase letters and

vice versa. The program should display the second string and its number of lowercase and uppercase letters. If the user enters end, the input of strings should terminate.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100], new_str[200]; /* The new string will be stored into
new_str. It is declared with double size, just for the case that the
input string contains only '*'. */
    int i, j, len, small_let, big_let;

    while(1)
    {
        printf("Enter text: ");
        len = read_text(str, sizeof(str), 1);

        if(str[0] == 'e' && str[1] == 'n' && str[2] == 'd')
            break;

        j = small_let = big_let = 0;
        for(i = 0; i < len; i++)
        {
            if(str[i] >= 'a' && str[i] <= 'z')
            {
                str[i] -= 32; /* The difference of an
uppercase letter with the respective lowercase is 32, according to the
ASCII code. */
                big_let++;
            }
            else if(str[i] >= 'A' && str[i] <= 'Z')
            {
                str[i] += 32;
                small_let++;
            }
            new_str[j] = str[i]; /* Copy each character of the
input string in the position indicated by j. */
            if(str[i] == '*')
            {
                j++; /* Increase j to store another '*'. */
                new_str[j] = '*';
            }
            j++; /* Increase j to store the next character. */
        }
        new_str[j] = '\0';
        printf("%s contains %d lowercase and %d uppercase
letters\n", new_str, small_let, big_let);
    }
    return 0;
}
```

### C.10.18 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "csfbl", *p = str;

    while (*p)
    {
        --*p; /* To make it harder, we could merge the two
statements into --*p++; */
        p++;
    }
    printf("%s\n", p-strlen(str));
    return 0;
}
```

**Answer:** The statement `--*p;` decreases the content of the address that `p` points to. For example, in the first iteration, `p` points to the first character of the string, that is, '`c`'. Therefore, `*p` is equal to '`c`' and the statement `--*p;` changes the value of `str[0]` and makes it equal to the previous character in the ASCII set. Therefore, `str[0]` becomes '`b`'. Next, `p` is increased and points to the next element of `str`. In the same way, the next iterations make `str[1]`, `str[2]`, `str[3]`, and `str[4]` equal to '`r`', '`e`', '`a`', and '`k`', respectively.

The loop terminates once `p` becomes equal to `str+5`, that is, once the null character is met. Since `strlen()` returns the length of the string, the expression `p-strlen(str)` is equivalent to `str+5-5 = str`. Therefore, the program displays the new string stored into `str`, that is, `break`. Indeed, have a break before moving on to the next exercise.

### C.10.19 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char str[] = "Text", *p = str;
    int i;

    for(i = 0; i < strlen(str)-1; i++, p++)
        printf("%c", p[i]);
    return 0;
}
```

**Answer:** Since `strlen()` returns 4, the loop is executed three times. Let's trace the iterations:

*First iteration (`i = 0`):* The value of `p[0]` is displayed, that is, `T`.

*Second iteration (`i++ = 1`):* Since `p` is incremented by one, `p` points to the second character of the string, that is, '`e`'. Since we handle `p` as an array, `p[0]` is '`e`' and `p[1]` is '`x`'. Therefore, the program outputs `x`.

*Third iteration (i++ = 2):* Now, p points to the third character, that is, 'x'. Therefore, p[0] is 'x', p[1] is 't' and p[2] is equal to '\0'. As a result, the program displays nothing.

To sum up, the program displays: Tx

C.10.20 Write a program that reads a string of less than 100 characters and displays the number of appearances of its lowercase letters and its digits. Also, the program should display the lowercase letter which appears the most times and the number of its appearances. If more than one letter appears the same most times, the program should display the one found first.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char ch, max_ch, str[100];
    int i, len, max_times, low Let[26] = {0}; /* The size of the array
is equal to the number of the lowercase letters. This array holds the
number of the appearances of each letter. For example, low Let[0] holds
the appearances of 'a', and low Let[25] the appearances of 'z'. */
    int dig[10] = {0}; /* Similarly, dig[0] holds the appearances of
digit 0, and dig[9] the appearances of digit 9. */

    printf("Enter text: ");
    len = read_text(str, sizeof(str), 1);
    max_ch = max_times = 0;
    for(i = 0; i < len; i++)
    {
        if((str[i] >= 'a') && (str[i] <= 'z'))
        {
            ch = str[i] - 'a';
            low Let[ch]++;
            /* For example, if the character is
'a', the value of low Let['a'-'a'] = low Let[0], which holds the
appearances of 'a', will be incremented by one. */
            if(low Let[ch] > max_times)
            {
                max_times = low Let[ch];
                max_ch = str[i];
            }
        }
        else if((str[i] >= '0') && (str[i] <= '9'))
            dig[str[i] - '0']++;
    }
    printf("***** Lowercase letters appearances\n");
    for(i = 0; i < 26; i++)
        if(low Let[i] != 0) /* Check if the letter appears once at
least. */
            printf("Letter %c appeared %d times\n", 'a'+i,
low Let[i]);
```

```

printf("***** Digits appearances\n");
for(i = 0; i < 10; i++)
    if(dig[i] != 0)
        printf("Digit %d appeared %d times\n", i, dig[i]);
if(max_times != 0)
    printf('%c' appears %d times\n", max_ch, max_times);
return 0;
}

```

**Comments:** If the string contains more than one character, which appears the same most times, the program displays the one found first. For example, if the user enters exit1, the output would be: 'e' appears 1 times, because all characters appear once and the character 'e' is the first one.

**C.10.21** Write a program that simulates the popular word-guessing game “hangman.” The program reads a secret word of less than 30 characters, which is the word to guess. This word is entered by the first player. Then, the second player enters letters one by one. The program should check each letter and inform the player. If the letter occurs in the word, the program should display the letters of the correct guesses in its positions and set the underscore character in the positions of the unknown letters. If it does not occur, the program should display a related message. The game is over if either the player finds the word or 7 incorrect guesses are made. Assume that the second player is not allowed to enter the same letter more than once.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ERROR_TRIES 7

int read_text(char str[], int size, int flag);

int main(void)
{
    char ch, secret[30], hide[30] = {0};
    int i, found, len, error, correct;

    printf("Enter secret word: ");
    len = read_text(secret, sizeof(secret), 1);

    for(i = 0; i < len; i++)
        hide[i] = '_';

    error = correct = 0;
    while(error < ERROR_TRIES)
    {
        printf("Enter character: ");
        ch = getchar();
        found = 0;
        for(i = 0; i < len; i++)
        {
            if(secret[i] == ch)

```

```

    {
        hide[i] = ch;
        found = 1;
        correct++;
        if(correct == len)
        {
            printf("Secret word is found !!!\n");
            return 0;
        }
    }
    if(found == 0)
    {
        error++;
        printf("Error, %c does not exist. You've got %d more
attempts\n", ch, ERROR_TRIES - error);
    }
    else
        printf("%s\n", hide);
    getchar(); /* Get '\n' from the previous getchar(). */
}
printf("Sorry, the secret word was %s\n", secret);
return 0;
}

```

**C.10.22** What is the output of the following program?

```

#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "noteeasy";

    printf("%s\n", str+(* (str+3)-1)-str[strlen(str+3)]);
    return 0;
}

```

**Answer:** Since the expression  $*(str+3)$  is equivalent to  $str[3]$ , that is, 'e', the value of  $*(str+3)-1$  is equal to  $str[3]-1$ , that is, 'e'-1 = 'd'.  $strlen()$  returns the length of the string after the third character, that is, 4. Therefore, the expression is equivalent to  $str+'d'-'a' = str+3$  and the program outputs easy.

**C.10.23** A Universal Product Code (UPC) barcode consists of 12 digits. The last digit is a check digit used for error detection. To calculate its value, we use the first 11 digits, as follows:

- Add the digits in the odd positions and multiply the result by 3.
- Add the digits in the even positions to the previous result.
- Divide the result by 10. Subtract the remainder from 10, and that is the check digit. If the subtraction gives 10 (meaning that the remainder is 0), use 0 as check digit.

For example, let's see if the barcode 123456789015 is correct. The check digit is calculated as follows:

- a.  $1+3+5+7+9+1 = 26$ . Multiplied by 3 gives  $26 \times 3 = 78$ .
- b.  $2+4+6+8+0 = 20$ . Added to the previous result gives  $78+20 = 98$ .
- c. The value of the check digit is  $10 - (98 \% 10) = 10 - 8 = 2$ .

Since the last digit is 5 and the calculated check bit is 2 the barcode is not correct.

Write a program that reads a UPC barcode and checks if it is correct. The program should force the user to enter a valid barcode, meaning that the length of the string should be 12 and it must contain digits only.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char upc[13];
    int i, len, flag, chk_dig, sum;

    while(1)
    {
        printf("Enter UPC (12 digits): ");
        len = read_text(upc, sizeof(upc), 1);
        if(len != 12)
        {
            printf("Error: wrong length\n");
            continue;
        }
        flag = 1;
        for(i = 0; i < 12; i++)
        {
            if(upc[i] < '0' || upc[i] > '9')
            {
                printf("Error: only digits allowed\n");
                flag = 0;
                break;
            }
        }
        if(flag == 1)
            break;
    }
    sum = 0;
    for(i = 0; i < 11; i+=2)
        sum += upc[i]-'0'; /* Subtract '0' to get the numeric value
of the digit character. */
    sum *= 3;
```

```

for(i = 1; i < 11; i+=2)
    sum += upc[i] - '0';

chk_dig = 10 - (sum%10);
if(chk_dig == 10)
    chk_dig = 0;

if(chk_dig == (upc[11] - '0'))
    printf("Valid barcode\n");
else
    printf("Wrong check digit. The correct is %d\n", chk_dig);
return 0;
}

```

## The `strcpy()` and `strncpy()` Functions

A very common error is to use the = operator to copy strings. For example, see again C.10.4 or this one:

```

char str[10];
str = "something"; /* Wrong, since str is an array. */

```

One way to copy strings is to use the `strcpy()` function. It is declared in `string.h`.

```
char *strcpy(char *dest, const char *src);
```

`strcpy()` copies the string pointed to by `src` into the memory pointed to by `dest`. Once the null character is copied, `strcpy()` terminates and returns the `dest` pointer. Since `src` is declared as `const`, `strcpy()` cannot modify the string. If the source and destination strings overlap, the behavior of `strcpy()` and `strncpy()` is undefined. In the following example, `strcpy()` copies the string "something" into `str`.

```

char str[10];
strcpy(str, "something");

```

Continue with a second `strcpy()`:

```

strcpy(str, "new");
printf("%c\n", str[6]);

```

Can you tell what it outputs?

Since `strcpy()` terminates once the null character is copied, the first four elements change and the rest remain the same. Therefore, the program outputs `i`.

The following program reads a string of less than 100 characters and uses `strcpy()` to copy it into an array:

```

#include <stdio.h>
#include <string.h>
int main(void)
{
    char str1[100], str2[100];

```

```
    printf("Enter text: ");
    fgets(str2, sizeof(str2), stdin);

    strcpy(str1, str2);
    printf("Copied text: %s\n", str1);
    return 0;
}
```

---

Because `strcpy()` does not check if the string pointed to by `src` fits into the memory pointed to by `dest`, it is the programmer's responsibility to ensure that the destination memory is large enough to hold all characters. Otherwise, the memory past the end of the `dest` will be overwritten, causing unpredictable behavior of the program.

For example, consider the following program:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char c = 'a', str[10];

    strcpy(str, "Longer text. The program may crash");
    printf("%s %c\n", str, c);
    return 0;
}
```

Since the size of `str` is not large enough to hold the characters of the string, the data past the end of `str` will be overwritten.

And yes, it is a serious error to pass an uninitialized pointer to `strcpy()`. For example:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str;

    strcpy(str, "something");
    printf("%s\n", str);
    return 0;
}
```

Since `str` does not point to an allocated memory, the program may crash.

The `strncpy()` function is declared in `string.h`:

```
char *strncpy(char *dest, const char *src, size_t count);
```

`strncpy()` is similar to `strcpy()`, with the difference that only the first `count` characters of the string pointed to by `src` will be copied into `dest`. If the value of `count` is less than or equal to the length of the string that `src` points to, a null character won't be appended to the memory pointed to by `dest`. If it is greater, null characters are appended, up to the value of `count`.

Since `strncpy()` specifies the maximum number of the copied characters, it is safer than `strcpy()`.

Here is an example of `strncpy()`:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str1[] = "Old text", str2[] = "New", str3[] = "Get";
    strncpy(str1, str2, 3);
    printf("%s\n", str1);

    strncpy(str1, str3, 5);
    printf("%s\n", str1);
    return 0;
}
```

Since the number 3 is less than the length of the string stored into `str2`, the first `strncpy()` copies the first three characters into `str1` and does not append a null character. Therefore, the program displays `New text`. Since the number 5 is greater than the length of the string stored into `str3`, the second `strncpy()` copies three characters from `str3` into `str1` and appends two null characters to reach the value 5. Therefore, the program displays `Get`.

## Exercises

C.10.24 Does the following program contain an error?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str1[] = "abcd", str2[] = {'e', 'f', 'g'};

    strcpy(str1, str2);
    printf("%s\n", str1);
    return 0;
}
```

**Answer:** Since `str2` does not contain the null character, the copy operation won't perform successfully and a runtime error may occur.

C.10.25 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str1[5], str2[5];

    printf("%c\n", strcpy(str1, strcpy(str2, "test"))[0]);
    return 0;
}
```

**Answer:** The inner strcpy() copies the string test into str2 and returns a pointer to str2. The outer strcpy() copies the string pointed to by the returned pointer of the inner strcpy() into str1. Therefore, the string test is copied into str1. Since the outer strcpy() returns a pointer to str1, printf() is translated to printf("%c\n", str1[0]); and the program displays t.

C.10.26 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[10] = {0};

    printf("%c\n", *(str+strlen(strncpy(str, "example", 5))/2));
    return 0;
}
```

**Answer:** strncpy() copies the first five characters into str. The return value of strncpy(), that is, the pointer to str is used as the argument of strlen(). Therefore, strlen() returns the length of the string stored into str, that is, 5. As a result, printf() is translated to printf("%c\n", \*(str+5/2)); and the program displays the character stored in position str+2, that is, a.

C.10.27 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "test";

    printf("%d %s\n", *strcpy(str, "n")**strcpy(str+2, "xt"), str);
    return 0;
}
```

**Answer:** The first `strcpy()` copies the characters of the string "n", that is, the characters 'n' and '\0' to `str[0]` and `str[1]`, respectively, and returns a pointer to `str`. Therefore, the expression `*strcpy(str, "n")` can be replaced by `*str`, that is, 'n'. The second `strcpy()` copies the string "xt" in the third position of `str` and returns the `str+2` pointer. Therefore, the expression `*strcpy(str+2, "xt")` can be replaced by `*(str+2)`, that is, 'x'.

As a result, the program displays the product of the ASCII codes of 'n' and 'x', that is, 13200. However, it does not display next as you might expect, but just n, because the first `strcpy()` replaced 'e' with '\0'.

## The `strcat()` Function

The `strcat()` function is declared in `string.h`:

```
char *strcat(char *dest, const char *src);
```

`strcat()` appends the string pointed to by `src` to the end of the string pointed to by `dest`. `strcat()` appends the null character and returns `dest`, which points to the resulting string. For example, the following program reads two strings of less than 100 characters and uses `strcat()` to merge them and store the resulting string into an array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str1[100], str2[100], str3[200] = {0}; /* Store the null
character in str3 for the first strcat(). */

    printf("Enter first text: ");
    read_text(str1, sizeof(str1), 1);

    printf("Enter second text: ");
    read_text(str2, sizeof(str2), 1);

    strcat(str3, str1);
    strcat(str3, str2);
    printf("The merged text is: %s\n", str3);
    return 0;
}
```

It is the programmer's responsibility to ensure that there is enough room in the memory pointed to by `dest` to append the string pointed to by `src`. Otherwise, it is a potential cause of memory overrun with unpredictable results.

For example, consider the following program:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[20] = "example";

    strcat(str, "not available memory");
    printf("The merged text is: %s\n", str);
    return 0;
}
```

Since the size of `str` is not large enough to hold the characters of both strings, the data beyond the end of `str` will be overwritten with unpredictable results on the program's operation.

Let's see the "crazy" one. What is the output?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[30];
    return !printf("%s\n", strcat(strcpy(strcpy(str, "Shaws") + 5, "hank
Red"), "emption") - 5));
}
```

A bit scary, isn't it? The inner `strcpy()` copies the string `Shaws` and returns a pointer to `str`. The outer `strcpy()` copies the string `hank Red` after the fifth place and returns a pointer to that place. `strcat()` appends the string `emption`. What about the `-5` at the end? Because the pointer that `strcat()` returns points to `str+5` we subtract `5` to go back to the beginning.

And, what about this ! in front of `printf()`? Is it a type error? No, it is added to make `printf()` more beautiful ...

Just kidding, since `printf()` returns the number of the printed characters, we add ! so that `main()` returns 0. Oh yes, almost forgot it, the program outputs `Shawshank Redemption`. Break time. If you have not seen that film, first watch it and then continue to the next section. Make yourself comfortable and enjoy the movie.

## The `strcmp()` and `strncmp()` Functions

The `strcmp()` function is declared in `string.h`:

```
int strcmp(const char *str1, const char *str2);
```

`strcmp()` compares the string pointed to by `str1` with the string pointed to by `str2`. If the strings are identical, `strcmp()` returns 0. If the first string is less than the second, `strcmp()` returns a negative value, whereas if it is greater, it returns a positive value. A string is considered less than another if either one of the following conditions is true:

- a. The first  $n$  characters of the strings match, but the value of the  $n+1$  character in the first string is less than the value of the  $n+1$  character in the second string.
- b. All characters match, but the first string is shorter than the second.

For example, the statement `strcmp("onE", "one");` returns a negative value because the ASCII code of the first nonmatching character '`E`' is less than the ASCII code of '`e`'.

The statement `strcmp("w", "many");` returns a positive value because the ASCII code of the first nonmatching character '`w`' is greater than the ASCII code of '`m`'. In another example, the statement `strcmp("some", "something");` returns a negative value because the first four characters match, but the first string is smaller than the second.

A very common error is to use the `==` operator to compare strings. For example, see C.10.5 once more. Therefore, the `if` condition in the next code:

```
char str[20];
fgets(str, sizeof(str), stdin);
if(str == "test")
```

does not compare the strings, but the values of `str` and that of the pointer to the literal string. Because the values are different, the condition is always false, no matter what string the user enters.

In Chapter 11, we'll implement the `str_cmp()` function, to show you an implementation example of `strcmp()`.

`strncpy()` is similar to `strcmp()`, with the difference that it compares a specific number of characters. It is declared in `string.h`.

```
int strcmp(const char *str1, const char *str2, int count);
```

The parameter `count` specifies the number of the compared characters.

---

## Exercises

C.10.28 Write a program that reads two strings of less than 100 characters and uses `strcmp()` to compare them. If the strings are different, the program should use `strncpy()` to compare their first three characters and display a related message.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str1[100], str2[100];
    int k;

    printf("Enter first text: ");
    read_text(str1, sizeof(str1), 1);

    printf("Enter second text: ");
    read_text(str2, sizeof(str2), 1);

    k = strcmp(str1, str2);
    /* We could omit the declaration of k and write: if(strcmp(str1,
str2) == 0) */
    if(k == 0)
        printf("Same texts\n");
    else
    {
        printf("Different texts\n");
        if(strncmp(str1, str2, 3) == 0)
            printf("But the first 3 chars are the same\n");
    }
    return 0;
}
```

C.10.29 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[5];

    str[0] = 't';
    str[1] = 'e';
    str[2] = 's';
    str[3] = 't';

    if(strcmp(str, "test") == 0)
        printf("One\n");
    else
        printf("Two\n");
    return 0;
}
```

**Answer:** Did you answer One but Two is displayed? Bad luck, next time ...

The str array contains the 't', 'e', 's', and 't' characters, but str[4] is unassigned. On the other hand, the string literal "test" ends with the null character. If the random value of str[4] is not 0 the two contents are different, so strcmp() returns a nonzero value and the program displays Two.

To continue... If we write `char str[5] = {'t', 'e', 's', 't'};`; what will be the output? One or Two?

Since the noninitialized elements are set to 0, str[4] becomes '\0' and the program displays One. What about if we change the condition to:

```
if(strcmp(str, "test\0") == 0)
```

No difference, the program displays One.

**C.10.30** Write a program that continuously reads strings (less than 100 characters each) and displays the “shortest” and “longest” strings. If the input string begins with end, the insertion of strings should terminate; end does not participate in the comparison.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100], min_str[100], max_str[100];

    printf("Enter text: ");
    read_text(str, sizeof(str), 1);
    /* Store the first string as shortest and longest as a first
reference to make the comparisons. */
    strcpy(min_str, str);
    strcpy(max_str, str);
    if(strncmp(str, "end", 3) == 0)
    {
        printf("\nMax = %s Min = %s\n", max_str, min_str);
        return 0;
    }
    while(1)
    {
        printf("Enter text: ");
        read_text(str, sizeof(str), 1);

        if(strncmp(str, "end", 3) == 0)
            break;

        if(strcmp(str, min_str) < 0)
            strcpy(min_str, str);
    }
}
```

```

        if(strcmp(str, max_str) > 0)
            strcpy(max_str, str);
    }
    printf("\nMax = %s Min = %s\n", max_str, min_str);
    return 0;
}

```

C.10.31 What is the output of the following program?

```

#include <stdio.h>
#include <string.h>
int main(void)
{
    char str1[10], str2[] = "engine";

    printf("%c\n", ++str1[strlen(strcpy(str1, "ine"), str2+3)+2]);
    return 0;
}

```

**Answer:** strcpy() copies the string "ine" into str1 and returns a pointer to str1. strcmp() compares the string pointed to by the return value of strcpy(), that is, str1, with the part of the string stored in the str2+3 position, that is, "ine". Since both strings are the same, strcmp() returns 0 and the program displays the value of ++str1[0+2]. Since str1[2] is 'e', the program outputs f.

C.10.32 The data compression algorithm run length encoding (RLE) is based on the assumption that a symbol within the data stream may be repeated many times in a row. This repetitive sequence can be replaced by an integer that declares the number of the repetitions and the symbol itself. Write a program that reads a string of less than 100 characters and uses the RLE algorithm to compress it. Don't compress digits and characters that appear once.

For example, the string: fffmmmmm1234jjjjjjjjjjx should be compressed to 3f4m123410jx

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100];
    int i, len, cnt;

    printf("Original text: ");
    len = read_text(str, sizeof(str), 1);

    printf("Compressed text: ");
    i = 0;
    while(i < len)
    {

```

```

        cnt = 1;
        if(str[i] < '0' || str[i] > '9') /* Digits are not compressed.
*/
    {
        while(str[i+cnt] == str[i]) /* Check if the current
character, that is str[i], is repeated in the next places. */
            cnt++;

        if(cnt == 1)
            printf("%c", str[i]);
        else
            printf("%d%c", cnt, str[i]);
    }
    else
        printf("%c", str[i]);

        i += cnt;
}
return 0;
}

```

**C.10.33** Write a program that reads a string of less than 100 characters and displays the words that it consists of and their number. Suppose that a word is a sequence of character(s) that does not contain space character(s). For example, if the user enters how many words ? (notice that more than one space may be included between the words), the program should display.

```

how
many
words
?
The text contains 4 words

```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100];
    int i, words;

    i = words = 0;

    printf("Enter text: ");
    read_text(str, sizeof(str), 1);

    if(str[0] != ' ' && str[0] != '\t' && str[0] != '\0') /* If the
first character is other than the space character, it means that a word
begins, so the value of words is incremented. */
        words++;

    while(str[i] != '\0')

```

```

{
    if(str[i] == ' ' || str[i] == '\t')
    {
        /* Since more than one space character may be
        included between words, we check if the next character, that is str[i+1],
        is also a space character. If it isn't, it means that a new word begins,
        so the value of words is increased. */
        if(str[i+1] != ' ' && str[i+1] != '\t' && str[i+1] != '\0')
        {
            words++;
            printf("\n");
        }
    }
    else
        printf("%c", str[i]);
    i++;
}
printf("\nThe text contains %d words\n", words);
return 0;
}

```

**C.10.34** A simple algorithm to encrypt data is the algorithm of the single transformation. Let's see how it works. In one line, we write the letters of the used alphabet. In a second line, we write the same letters in a different order. This second line constitutes the cryptography key. For example, see Figure 10.2.

Each letter of the original text is substituted with the respective key letter. For example, the word `test` is encrypted as `binb`.

Write a program that reads a string of less than 100 characters, the key string consisting of 26 characters, and encrypts the lowercase letters of the input string. Then, the program should decrypt the encrypted string and display it (should be the same with the original string). The program should check that the key characters are 26 and appear only once.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LETTERS 26

int read_text(char str[], int size, int flag);

int main(void)
{
    char str[100], key[LETTERS+1];
    int i, j, len, error;

```

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
d	y	p	r	i	a	j	u	h	t	q	w	e	s	f	o	v	c	n	b	l	x	m	k	z	g

**FIGURE 10.2**

Single transformation algorithm.

```
printf("Enter text: ");
len = read_text(str, sizeof(str), 1);

do
{
    error = 0;
    printf("Enter key (%d different characters): ", LETTERS);
    read_text(key, sizeof(key), 1);
    if(strlen(key) != LETTERS)
    {
        printf("Error: Key must be of %d characters\n", LETTERS);
        error = 1;
    }
    else
    {
        for(i = 0; i < LETTERS; i++)
        {
            for(j=i+1; j < LETTERS; j++)
            {
                if(key[i] == key[j])
                {
                    error = 1;
                    printf("Key characters must be
different\n");
                    getchar();
                    break;
                }
            }
            if(error == 1)
                break;
        }
    }
}
while(error != 0);

for(i = 0; i < len; i++)
{
    if(str[i] >= 'a' && str[i] <= 'z')
        str[i] = key[str[i]-'a'];
}
printf("Encrypted text: %s\n", str);
for(i = 0; i < len; i++)
{
    for(j = 0; j < LETTERS; j++)
    {
        if(str[i] == key[j])
        {
            str[i] = 'a'+j;
            break;
        }
    }
}
printf("Original text: %s\n", str);
return 0;
}
```

## Two-Dimensional Arrays and Strings

Two-dimensional arrays are often used to store multiple strings. For example, the statement:

```
char str[10][40];
```

declares the `str` array with 10 rows. Each row can store a string of up to 40 characters. In case of shorter strings, there is a waste of memory.

We can store string literals in a two-dimensional array when it is declared. For example:

```
char str[3][40] = {"One", "Two", "Three"};
```

the characters of "One" are stored in the first row of `str`, the characters of "Two" in the second row, and those of "Three" in the third row. Since the strings are not long enough to fill the rows, null characters are padded, as depicted in Figure 10.3.

Recall from Chapter 8 and the section "Pointers and Two-Dimensional Arrays" that we can treat each of the elements `str[0]`, `str[1]`, ..., `str[N-1]` of a two-dimensional array `str[N][M]` as a pointer to an array of M elements. For example, `str[0]` can be used as a pointer to an array of 40 characters, which holds the string One.

Also remember from the section "Array of Pointers" that in order to save memory space, we could use an array of pointers instead of a two-dimensional array. For example:

```
char *str[] = {"One", "Two", "Three"};
```

Although some memory is allocated for the `str` pointers, the important thing is that the memory allocated for the strings is exactly as needed. Thanks to the close relationship between pointers and arrays, we can subscript `str` as a two-dimensional array to access a character. For example:

```
for(i = 0; i < 3; i++)
    if(str[i][0] == 'T')
        printf("%s\n", str[i]);
```

The loop displays the strings beginning from T.

C.10.37 illustrates another example where using an array of pointers to handle strings provides efficiency.

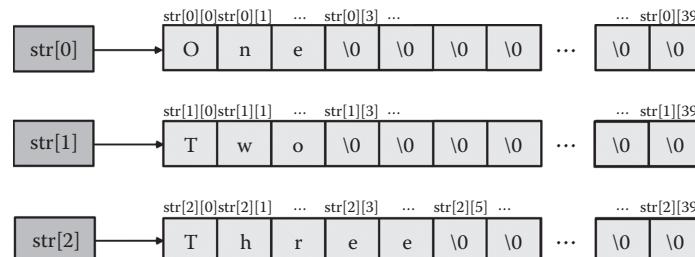


FIGURE 10.3

Array of strings.

## Exercises

C.10.35 What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    char arr[7][10] = {"Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"};
    int i;

    for(i = 0; i < 7; i++)
        if(arr[i][2] == 'n' && *(arr[i]+3) == 'd' && *(*(arr+i)+4)
== 'a')
            printf("%s is No.%d week day\n", arr[i], i+1);
    return 0;
}
```

**Answer:** The characters of "Monday" are stored in the first row of arr, the characters of "Tuesday" in the second row, and so on. Remember that  $*(\text{arr}[i]+3)$  is equivalent to  $\text{arr}[i][3]$  and  $*(\text{arr}+i)+4$  is equivalent to  $*(\text{arr}[i]+4)$ , that is,  $\text{arr}[i][4]$ . Therefore, the loop checks each row of arr and displays the strings whose third, fourth, and fifth characters are 'n', 'd' and 'a', respectively. Therefore, the program displays:

```
Monday is No.1 week day
Sunday is No.7 week day
```

C.10.36 Write a program that reads 10 strings (less than 40 characters each) and stores in an array those that begin with an 'a' and end with 's'. Then, the program should read a character and display in which column it occurs most. If there is more than one column with the same most occurrences, the program should display the one found first.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM 10

int read_text(char str[], int size, int flag);

int main(void)
{
    char tmp[40], str[NUM][40] = {0};
    int i, j, ch, num, len, max_len, cnt, max_cnt, col;

    num = max_len = 0;
    for(i = 0; i < NUM; i++)
    {
        printf("Enter text: ");
        len = read_text(tmp, sizeof(tmp), 1);
        if(tmp[0] == 'a' && tmp[len-1] == 's')
```

```

    {
        strcpy(str[num], tmp);
        num++;
        if(len > max_len)
            max_len = len;
    }
}
if(num == 0)
{
    printf("None string is stored\n");
    return 0;
}
printf("Enter character: ");
ch = getchar();

max_cnt = 0;
/* We use the length of the longest string and the number of the
stored strings, to search for the character only in those rows and
columns where characters are stored. */
for(j = 0; j < max_len; j++)
{
    cnt = 0;
    for(i = 0; i < num; i++)
    {
        if(str[i][j] == ch)
            cnt++;
    }
    if(cnt > max_cnt)
    {
        max_cnt = cnt;
        col = j;
    }
}
if(max_cnt == 0)
    printf("%c is not stored\n", ch);
else
    printf("Max appearances of %c is %d in col_%d\n", ch, max_
cnt, col+1);
return 0;
}

```

**C.10.37** Write a program that reads 20 names (less than 100 characters each), stores them in an array, and uses an array of pointers to display them in alphabetical order.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 20

int read_text(char str[], int size, int flag);

int main(void)

```

```

{
    char *temp, *ptr[SIZE], str[SIZE] [100];
    int i, j;

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter name: ");
        read_text(str[i], sizeof(str[i]), 1);
        ptr[i] = str[i]; /* The elements of the array point to the
input strings. */
    }
    for(i = 0; i < SIZE-1; i++)
    {
        for(j = i+1; j < SIZE; j++)
        {
            /* If the string pointed to by ptr[j] is less than
the string pointed to by ptr[i], swap the respective pointers. */
            if(strcmp(ptr[j], ptr[i]) < 0)
            {
                temp = ptr[j];
                ptr[j] = ptr[i];
                ptr[i] = temp;
            }
        }
    }
    for(i = 0; i < SIZE; i++)
        printf("%s\n", ptr[i]);
    return 0;
}

```

**Comments:** The main reason we added this exercise is to show you how useful an array of pointers might be. When two strings must be swapped, it is the respective pointers that are swapped, not the strings. As a result, we gain in performance. The sorting algorithm we used is called selection sort and it will be discussed in Chapter 12. The “negative” point in this solution is the potential waste of space. Each row of str allocates 100 characters, even if the string contains less. In Chapter 14 and the exercise C.14.8, you’ll see a more efficient way where the exact memory needed is dynamically allocated instead of using a fixed two-dimensional array.

## Unsolved Exercises

**U.10.1** Write a program that reads characters until the sum of their ASCII values exceeds 500 or the user enters 'q'. The program should display how many characters were read.

**U.10.2** Write a program that reads characters and displays how many characters between the first two consecutive '\*' are: (a) letters, (b) digits, and (c) other than letters and digits. If no two '\*' occur, the program should display a related message. For example, if the user enters: 1abc\*D2Efg\_#!\*345Higkl~mn+op\*qr the program should display the following: Between first two stars (letters:4, digits:1, other:3).

**U.10.3** Write a program that reads three strings of less than 100 characters and stores them in three arrays (e.g., `str1`, `str2`, and `str3`). Then, the program should copy their contents one place right, meaning that the content of `str3` should be copied to `str1`, the content of `str1` to `str2`, and that of `str2` to `str3`.

**U.10.4** Write a program that reads a string of less than 100 characters, and if it ends with `aa`, the program should display it in reverse order.

**U.10.5** What is the output of the following program?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "test", *p = str;

    *str = str[strlen(str)];
    while(*p)
        printf("%s", p++);
    return 0;
}
```

**U.10.6** Write a program that reads a string of less than 100 characters and stores it in an array. Then, the program should reverse the stored string and display the new one. For example, if the string `code` is stored, the program should reverse it and store in the array `edoc`. Use just one array.

**U.10.7** Write a program that reads a string of less than 100 characters, how many characters are going to be deleted, and the position of the first character to be deleted. After the deletion, the rest part of the string should be shifted to the left, a number of places equal to the number of the deleted characters. The program should display the string before it ends. For example, if the input string is `test case` and the numbers are 4 and 2, the program should display `t case`. The program should check the validity of the input numbers in order to ensure that the characters are contained in the string.

**U.10.8** Write a program that reads two strings of less than 100 characters and displays how many times the second string is contained in the first one. The length of the second string should be less than or equal to the first one.

**U.10.9** Write a program that reads a string of less than 100 characters and stores it in an array. Then, the program should delete from the array all the characters that are not letters, the multiple repetitions of the same letter, and display the new string. For example, if the string `jA*8%cr1^c` is stored in the array, the program should change its content to `jAcr`. There is one restriction; don't use a second array to store the characters that are not deleted.

**U.10.10** Modify C.10.33 in order to display the words in reverse order. For example, if the user enters `imagine the case`, the program should display `case the imagine`.

**U.10.11** Write a program that reads characters and stores them in an array of 100 places with the restriction that no duplicated character is stored. If the user enters 'q' or the array is full, the insertion of characters should terminate.

**U.10.12** Write a program that reads a string of less than eight characters that represents a hexadecimal number (accepted characters: 0-9, a-f, A-F) and displays the corresponding decimal value. For example, if the user enters 1AF, the program should display 431.

**U.10.13** Suppose that each lowercase letter has a value as follows. The letters 'a' up to 'i' correspond to units; 'a' has the value 1 and 'i' is 9. The letters 'j' up to 'r' correspond to tens; 'j' has the value 10 and 'r' is 90. The rest correspond to hundreds; 's' has the value 100 and 'z' is 800.

Write a program that continuously reads strings (less than 100 characters each) and displays the strings with the largest and smallest value and those values as well, before it ends. For example, the value of cky is 723, while the value of A9m is 40, since the characters 'A' and '9' don't have a value. If more than one string has the same largest or smallest value, the program should display the one entered last. If the user enters \*\*\*, the insertion of strings should terminate.

**U.10.14** Write a program that reads two strings of less than 100 characters and removes every occurrence of the second string inside the first one. After each removal, the remaining part of the first string should be shifted to the left, a number of places equal to the characters of the second string. The program should display the first string, before it ends. For example, if the first string is thisthat though and the second is th, the program should display isat ough.

**U.10.15** Write a program that reads two strings of less than 100 characters and displays the longest part of the first string that does not contain any character of the second string. For example, if the first string is example and the second one is day, the program should display mple because that part does not contain any character of day.

**U.10.16** Write a program that reads an integer and converts it to a string. For example, if the user enters 12345, the program should store the characters '1', '2', '3', '4' and '5' into an array.

**U.10.17** Write a program that reads two strings of less than 100 characters and displays the largest part of the first string that contains exclusively characters from the second string. For example, if the first string is programming and the second string is im, the program should display mmi. Assume that the second string contains different characters.

**U.10.18** Write a program that reads an integer in the form of a string and converts the string into that number. For example, if the user enters the string -12345, the program should convert that string to the number -12345, assign it to a numerical variable and display it. Assume that the user enters up to 10 digits.

**U.10.19** To continue the previous exercise, the program should check if the input string corresponds to a float number and assign that number to a numerical variable. For example,

if the user enters the string 12, the program should assign the number 12 to a variable. If the user enters the string -1.234 the assigned value should be -1.234.

**U.10.20** Write a program that reads 20 strings (less than 100 characters each) and stores them in a two-dimensional array. Then, the program should read a string, and if it is found in the array, it should be removed. To remove the string, move the strings below that point one row up. In the place of the last row moving up, insert the null character. For example, if the array is:

```
one  
two  
three  
four  
...  
...
```

and the user enters `two`, the new content should be:

```
one  
three  
four  
...  
...
```

**U.10.21** Write a program that reads 20 strings (less than 100 characters each) and stores them in a two-dimensional array. Then, the program should:

- Read a column number and display the words that are contained in that column, including those that consist of a single character. In an example with five strings, if the user enters:

```
abcdefghijklm  
abcdefghijklm  
abcd  
abc  
abcdklmop  
abcdklmop
```

and the number 6 as column, the program should display:

```
word_1: gp  
word_2: m
```

- Read a row number, except the first one, and if the string in that row contains less than five different lowercase letters, the program should swap that string with the string in the first row.

**U.10.22** Write a program that reads the names of 20 bookstores and 500 book titles. Use two two-dimensional arrays to store them. Assume that all names are less than 100 characters. The program should also read for each bookstore the number of copies left in stock for each title. Then, the program should read a book title and display the names of the bookstores that offer this title and the number of copies of that title in those bookstores. Finally, the program should read the name of a bookstore and display its total number of copies left in stock.

# Functions

In C, a function is an independent block of code that performs a specific task when called, and it may return a value. Each function is essentially a small program, with its own variables and statements. Functions may exist in separate files that are compiled separately. Using functions, a program is divided into smaller parts, making it easier to understand, control, and modify. Functions are reusable. For example, the `printf()` and `scanf()` functions can be used in every C program. In addition, functions allow us to avoid duplicating the same code. Up to this point, we've written just one function, that is, `main()`. In this chapter, you'll learn how to write and use your own functions within your programs.

## Function Declaration

A function declaration, which is also called function prototype, specifies the name of the function, its return type, and a list of parameters if present. The general form of a function declaration is:

```
return_type function_name(parameter_list);
```

Try to choose descriptive names for your functions. It is much easier to read the code of a function when its name indicates its role. For example, if you write a function that calculates the average of some numbers, give it a name like `average` rather than an arbitrary name like `func`, `test` or `lala`.

Usually, the declarations of functions are put in a separate file. For example, the prototypes of the library functions reside in several header files. For each library function used in the program, the `#include` directive is used to add the file that contains its declaration. For example, to use the `printf()` and `scanf()` functions, the `stdio.h` file is included. Otherwise, the compiler will produce an error message for undeclared identifier.

Although it is not mandatory to declare a function, we always do it. One reason is to improve the readability of the program, since the reader can get an idea of how the program is structured. You'll see more reasons later. For simplicity, we'll declare the functions we write in the same file with `main()`.

## Return Value

A function may return one value at most. The `return_type` specifies the type of the returned value. A function may return any type, for example, `char`, `int`, `float`, ... or a pointer to some type. The only restriction is that it is not allowed to return an array or another function. However, it can return a pointer to an array or a function. If the return

type is missing, the function is presumed to return a value of type **int**. The return type **void** specifies that the function does not return any value.

## Function Parameters

A function may take a list of parameters, separated by commas. Each parameter is preceded by its type. As we'll see later, a function parameter is essentially a variable of the function, which will be assigned with a value when the function is called.

If the function has no parameters, the word **void** should be used inside parentheses to indicate the empty list. However, many programmers omit the word **void** and use an empty pair of parentheses. Most probably, the reason for that originates either from an older version of C or due to their involvement with C++, where it is not needed to use the word **void**. However, as in the case of `main()`, if we want to be precise with the C standard, we should use the word **void**. The reason is that a pair of empty parentheses indicates that the function takes an unknown number of parameters, not any. Let's see some examples of function declarations:

```
void show(char ch); /* Declare a function with name show that takes a char parameter and returns nothing. */
double show(int a, float b); /* Declare a function with name show that takes an integer and a float parameter and returns a value of type double. */
int *show(int *p, double a); /* Declare a function with name show that takes as parameters a pointer to integer and a double parameter and returns a pointer to integer. */
```

Notice that it is not needed to specify the names of the parameters. For example, we could write:

```
double show(int, float);
```

Our preference is always to use names because it helps the reader to get an idea about the information that is passed to the function, the purpose of each parameter, and the order in which arguments should be passed.

The type of each parameter must be specified, even if several parameters have the same type. For example, you can't write:

```
void test(a, b, c); instead of void test(int a, int b, int c);
```

---

## Function Definition

The general form of a function definition is:

```
return_type function_name(parameter_list)
{
    /* Function body */
}
```

The first line of the function's definition must agree with its declaration, with the difference that no semicolon is added at the end. The names of the parameters don't have to match the names given in the function's declaration. The code, or else the body of the function, contains declarations and statements enclosed in braces. The body of a function can be empty. Functions can be defined in several files; however, the definition of a function cannot be split between files. If a library function is used, the function has already been defined and compiled.

The function's body is executed only if the function is called. The execution of a function terminates if either an exit statement (e.g., `return`) is called or its last statement is executed. Let's see an example of using a function:

```
#include <stdio.h>

void test(void); /* Function declaration. */

int main(void)
{
    test(); /* Function call. */
    return 0;
}

void test(void) /* Function definition. */
{
    /* Function body. */
    printf("In\n");
}
```

As will be explained later, the program outputs In.

As an example, to show you how to declare a function in a different file, create a file, e.g., `prototype.h`, copy the declaration inside, remove the declaration from the source file, and use the `#include` directive to include `prototype.h`. In Chapter 17, we'll write a program consisting of multiple source and header files and discuss about what is typically put in a header file.

```
#include <stdio.h>
#include "prototype.h"

int main(void)
...
...
```

Although C does not require function definitions to be put after `main()`, this is our preference, so that the reader first reads the heart of the program, that is, `main()`, which generally provides an idea for the program structure, and then to continue with the next functions.

As said, it is not mandatory to declare the function. However, if the declaration is missing and the function is defined after being called, the compiler may make wrong assumptions about the types of the parameters, their number, and the return type. As a result, the program might not be compiled or behave unpredictably if compiled and run. For example, let's see what might happen in the previous program if we remove the declaration of `test()` and replace its call with the statement:

```
int i = test();
```

Since the prototype is missing, `test()` is implicitly declared when the compiler encounters the call of `test()`. In particular, the function is assumed to return the default type, that is, `int`, and nothing is assumed about its arguments. The compiler may produce the warning message “*implicit declaration of function ‘test’*” to indicate that the declaration of `test()` is implied. However, this assumption is wrong because, later, it is found that the function returns nothing. The message “*conflicting types for ‘test’*” may be displayed, to indicate that an inconsistency is found. If the compiler allows the program to run, `i` will have an arbitrary value, and if that value is used in the program, its behavior becomes unpredictable.

A function is not allowed to be defined inside another function. However, the declaration of a function can be put inside the body of the calling function. For example:

```
int main(void)
{
    void test(void);
    test();
    return 0;
}
```

However, because the declaration of `test()` is visible only inside `main()`, it should be declared in any other function needs to call `test()`. And, if the declaration must change in the future, the programmer should search for and modify all declarations. For these reasons, we never declare functions inside functions.

So far, we assumed that the function is defined after the point at which it is first called. If the definition precedes its call, it is safe to omit the declaration of the function. For example:

```
#include <stdio.h>

void test(void)
{
    printf("In\n");
}

int main(void)
{
    test();
    return 0;
}
```

Although our programs are short and we could define our functions before `main()`, we'll always use prototypes, in order to get used to that practice. Let's see some reasons. Suppose that you define all functions before `main()`. In case a function calls another, you must be very careful in the definition order. How about the case that two functions call each other? No matter which function is defined first, one function will be called without being defined, which, as discussed, is a potential cause of program malfunction. Moreover, when you start writing larger programs, the functions would be most probably split in several files. Then, you'll need to use prototypes, in order to inform the compiler about functions in other files.

To end up, our practice is always to use prototypes and put them in header files when writing larger applications. This is what you are going to learn, end of discussion.

---

Wherever you choose to define a function, the important thing is to remember that the compiler should have seen its definition before the function is called, in order to check that it is properly called.

---

## Complex Declarations

Now that we've learned how to define functions, let's make a parenthesis to describe a method for deciphering complex declarations. Always decipher the declaration from inside out. Find the name of the variable and, if it is enclosed in parentheses (), start deciphering from there. Then, continue with the next operator where the precedence order, from high to low, is:

- a. The postfix operators. The parentheses () indicate a function, and the brackets indicate [] an array.
- b. The prefix operator \* indicates "pointer to."

For example, consider the declaration `int *p[5];`

Let's apply these precedence rules. The variable p is not enclosed in parentheses. Because [] take precedence over \* we go right, so we have that p is "an array of five ..." then, go left to \* and have "an array of five pointers ..." we add the type and end up with "an array of five pointers to integers."

Let's see another example: `int (*p)[5];`

Now, p is enclosed in parentheses. Deciphering starts inside there, so we go left to \* and we have that p is "a pointer to ..." then we go right, so we have "a pointer to an array of five ..." then, go left to add the type and end up with "a pointer to an array of five integers." By the way, here is an example of how to initialize a pointer of that type. Suppose that the array arr is declared as `int arr[5];`. To initialize p we write `p = &arr;` or together with the declaration we write `int (*p)[5] = &arr;`

Another one: `int *(*p)(int);`

Like before, we start that p is "a pointer to ..." the () take precedence over \*, so we go right and have "a pointer to a function with an integer argument ..." then, go left to add the return type and end up with "a pointer to a function with an integer argument that returns a pointer to an integer."

Leave the best for the end: `int *(*p[5])(int*);`

Like before, we start deciphering p inside the parentheses. p is "..." because [] take precedence over \*, we go right and have "an array of five ..." then, go left to \*, "an array of five pointers ..." then, because () take precedence over \* go right and have "an array of five pointers to functions with argument pointer to an integer ..." then, go left to add the return type and end up with the train "an array of five pointers to functions with argument pointer to an integer that returns a pointer to an integer."

With so many left-right zigzagging, we feel a little dizzy; just remember, *never drive drunk*. And always fasten your seatbelt or put your helmet on.

## The `return` Statement

The `return` statement is used to terminate immediately the execution of a function. In some programs so far, we've used the `return` statement to terminate the `main()` function, that is, the program itself. In the following example, the program terminates if the user enters the value 2, otherwise it outputs the input value.

```
#include <stdio.h>
int main(void)
{
    int num;

    while(1)
    {
        printf("Enter number: ");
        scanf("%d", &num);

        if(num == 2)
            return 0; /* Terminate the program. */
        else
            printf("Num = %d\n", num);
    }
    return 0;
}
```

Notice that the last `return` will never be executed, since the first `return` terminates the program. The value returned by `main()` indicates the termination status of the program. The value 0 indicates normal termination, whereas a nonzero value typically indicates abnormal termination.

If the function returns nothing, we just write `return`. The `return` statement at the end of a `void` function is unnecessary because the function will return automatically. In the following example, the `avg()` function compares the values of the two parameters, and if they are different, it displays their average. If they are the same, the function terminates.

```
void avg(int a, int b)
{
    /* Function body. */
    if(a == b)
        return;

    printf("%f\n", (a+b)/2.0);
    /* The return statement is unnecessary. */
}
```

If the function is declared to return a value, each `return` statement should be followed by a value. As we'll see later, the value of the executed `return` is returned to the point at which the function was called. The calling function is free to use or discard the returned value. For example, we modified `avg()` to return an integer value:

```
int avg(int a, int b)
{
    if(a == b)
```

```
    return 0;
    printf("%f\n", (a+b)/2.0);
    return 1;
}
```

Notice that the second `return` statement at the end of the function can be omitted. In that case, the compiler would most probably issue a message such as “*‘avg’ : not all control paths return a value*” or “*control reaches end of non-void function.*” The function would return an undefined value, which is definitely undesirable. Therefore, we strongly recommend that a non-`void` function always return a value, even if we are allowed not to do it.

The type of the returned value should match the function’s return type. If it does not match, the compiler will convert the returned value, if possible, to the return type. For example:

```
int test()
{
    return 4.9;
}
```

Since `test()` is declared to return an `int` value, the returned value is implicitly converted to `int`. Therefore, the function returns 4.

---

## Function Call

A function can be called as many times as needed. When a function is called, the execution of the program continues with the execution of the function’s code. In particular, the compiler allocates memory to store the function’s parameters and the variables that are declared within its body. Typically, this memory is reserved from a specific part of the memory, called *stack*; however, it might not apply in all cases. For example, parameters are typically passed in system registers for faster access. For simplicity, we assume that the memory is allocated in the stack. Moreover, the compiler allocates memory to store the address where the program returns once the function terminates. The allocated memory is automatically released when the function terminates. A more detailed discussion about the actions performed in the memory when a function is called is beyond the scope of this book.

### Function Call without Parameters

A call to a function that does not take any parameters is made by writing the function name followed by a pair of empty parentheses. The calling function does not pass any data to the called function. In the following program, the calling function, that is, `main()`, calls twice the `test()` function:

```
#include <stdio.h>

void test(void);

int main(void)
```

```
{  
    printf("Call_1 ");  
    test(); /* Function call. The parentheses are empty, because the  
function does not take any parameters. */  
    printf("Call_2 ");  
    test(); /* Second call. */  
    return 0;  
}  
  
void test(void)  
{  
    /* Function body. */  
    int i;  
    for(i = 0; i < 2; i++)  
        printf("In ");  
}
```

At the first call of `test()`, the program continues with the execution of the function body. When `test()` terminates, the execution of the program returns to the calling point and continues with the execution of the next statement. Therefore, the main program displays `Call_2` and calls `test()` again. As a result, the program displays `Call_1 In In Call_2 In In`.

---

A common oversight is to omit the parentheses when the function is called.

For example, if you write `test` the program will compile, but the function won't get called. Recall from Chapter 8 that the compiler treats the name of a function as a pointer; therefore, it permits the statement. However, this statement has no effect; it evaluates the address of the function but does not call it. The compiler may issue a warning message such as *"function call missing argument list"* or *"statement with no effect"* to inform the programmer. In the following program, `test()` returns an integer value.

```
#include <stdio.h>  
  
int test(void);  
  
int main(void)  
{  
    int sum;  
  
    sum = test(); /* Function call. The returned value is stored in  
sum. */  
    printf("%d\n", sum);  
    return 0;  
}  
  
int test(void)  
{  
    int i = 10, j = 20;  
    return i+j;  
}
```

`test()` declares two integer variables with values 10 and 20 and returns their sum. This value is stored into `sum` and the program displays 30. Notice that it is not needed to declare `sum`; we could write `printf("%d\n", test());` `test()` is called first and the returned value is used as an argument in `printf()`.

Before ending this section, we'd like to point out that the order of the function calls in a compound expression, such as `d = a() + b() * c();` is undefined. In other words, these three functions can be called in any order. Don't assume that the `b()` and `c()` functions will be called first because of the multiplication's precedence. And if any of them changes a variable on which some other may depend, the value assigned to `d` will depend on the evaluation order.

## Function Call with Parameters

A call to a function that takes parameters is made by writing the function name followed by a list of arguments, enclosed in parentheses. Using arguments is one way to pass information to a function. The difference between *parameter* and *argument* is that the term *parameter* refers to the variables that appear in the definition of the function, while the term *argument* refers to the expressions that appear in the function call. For example, consider the following program:

```
#include <stdio.h>

int test(int x, int y);

int main(void)
{
    int sum, a = 10, b = 20;

    sum = test(a, b); /* The variables a and b become the function's
arguments. */
    printf("%d\n", sum);
    return 0;
}

int test(int x, int y) /* The variables x and y are the function's
parameters. */
{
    return x+y;
}
```

The argument can be any valid expression, such as constant, variable, math, or logical expression, even another function with a return value. Notice that we could omit `sum` and write `printf("%d\n", test(a, b));` `test()` is called first, and then `printf()` outputs the return value.

Because the order in which arguments are evaluated is not specified, don't write something like `printf("%d %d\n", ++a, test(a, b));`

It depends on the compiler whether `a` is incremented, before or after `test()` is called.

The number of the arguments and their types should match the function declaration. If the arguments are less, the compiler will produce an error message. If the types of the arguments don't match the types of the parameters, the compiler will try to convert implicitly the types of the mismatched arguments to the types of the corresponding parameters. If it is successful, the compiler may display a warning message to inform the programmer

for the type conversion. If not, it will produce an error message. For example, consider the call `test(10.9, b);`. Since the type of the first parameter is `int`, the compiler will pass the value 10 to `test()` and the program would compile.

In the absence of a function prototype, `char` and `short` arguments are promoted to `int`, and `float` arguments to `double`. Let's see another example, where the absence of the prototype causes wrong behavior:

```
#include <stdio.h>

int main(void)
{
    test(5);
    return 0;
}

void test(double a)
{
    printf("%f\n", a);
}
```

When `test()` is called, the compiler has not seen a prototype, so it does not know that it accepts an argument of type `double`. Therefore, it does not convert the `int` value to `double`, and since the `int` type is not promoted by default, the function outputs an invalid value. Once more, always declare functions before calling them.

Let's return to the first program. When it is executed, the compiler allocates eight bytes to store the values of `a` and `b`. When `test()` is called, the compiler allocates another eight bytes to store the values of `x` and `y`. Then, the arguments are evaluated and their values are assigned one to one to the corresponding memory locations of the parameters `x` and `y`. Essentially, each parameter is a variable that is initialized with the value of the corresponding argument. Therefore, `x` becomes 10 and `y` becomes 20. When `test()` terminates, the memory allocated for `x` and `y` is automatically released.

Since the memory locations of `x` and `y` are different from those of `a` and `b`, any changes in the values of `x` and `y` have no effect on the values of `a` and `b`. In fact, all function arguments are passed by value in C. In other words, the called function always receives the values of the arguments in copies of the original variables. It cannot modify the value of the variable used as an argument; it can only modify the value of its copy. As we'll see later, an exception to that rule is the arrays. When an array is passed to a function, no copy of the array is made. It is the address of its first element that is actually passed. The function can access and modify any element of the array.

If we want the function to be able to modify the value of the original variable, we should pass its address, not its value. The function must declare the corresponding parameter to be a pointer and use that pointer to access the variable. For example, consider the following program:

```
#include <stdio.h>

void test(int *p);

int main(void)
{
    int i = 10;
```

```
    test(&i);
    printf("%d\n", i);
    return 0;
}

void test(int *p)
{
    *p = 20;
}
```

Since the & operator produces the address of i, &i is a pointer to i and the call matches the declaration. When test() is called, we have p = &i. Since p points to i, test() may change the value of i. Therefore, \*p is an alias for i, and the statement \*p = 20 changes i from 10 to 20. As a result, the program outputs 20.

---

Since a function cannot return more than one value, we can pass the addresses of the arguments and change their values.

It is worth noting that passing the address of an argument is often referred to as another type of call, named call by reference. However, it is no different from the call by value, since the function still cannot change the value of the passing argument. For example, consider the following program:

```
#include <stdio.h>

void test(int *p);

int main(void)
{
    int *ptr, i;

    ptr = &i;
    printf("%p\n", ptr);

    test(ptr);
    printf("%p\n", ptr);
    return 0;
}

void test(int *p)
{
    int j;
    p = &j;
}
```

What do you believe, has the value of ptr changed after calling test()?

The answer is no. Although the value of p changed and points to j, ptr remains as is because it is the value that is passed; p is a copy of ptr. In Chapter 18, we'll see that C++ provides another way to modify arguments without passing pointers. In particular, we'll talk about references and call by reference. One who makes the distinction between call by value and call by reference probably wants to say that a pointer, like a reference, can be used to change the value of an argument. However, the reference is a C++ concept, not of C.

When a function is called, it is allowed to mix pointers and plain values. For example:

```
#include <stdio.h>

void test(int *p, int a);

int main(void)
{
    int i = 100, j = 200;

    test(&i, j);
    printf("%d %d\n", i, j);
    return 0;
}

void test(int *p, int a)
{
    *p = 300;
    a = 400;
}
```

When `test()` is called, we have `p = &i`, and `test()` changes `i` from 100 to 300, whereas the value of `j` does not change. Therefore, the program outputs 300 200.

Here is a more difficult example. How can a function change the value of a pointer? We just pass to the function an argument of type pointer to a pointer. For example:

```
#include <stdio.h>

void test(int **arg);

int var = 100;

int main(void)
{
    int *ptr, i = 30;

    ptr = &i;
    test(&ptr); /* The value of &ptr is the memory address of ptr,
which points to the address of i. So, the type of the argument is a
pointer to a pointer to an integer and matches the function declaration.
*/
    printf("%d\n", *ptr);
    return 0;
}

void test(int** arg)
{
    *arg = &var;
}
```

Since the memory address of `ptr` is passed to `test()`, `test()` may change its value. When `test()` is called, we have `arg = &ptr`, so `*arg = ptr`. Therefore, the statement `*arg = &var;` is equivalent to `ptr = &var;` which means that the value of `ptr` changes

and points to the address of var. Therefore, the program displays 100. We'll talk about the scope of var shortly. We do understand that you might find this example a bit difficult. However, because you'll most probably need to read or write code that changes the value of a pointer, we added it here so that it is easy to find and consult it. Enough with the theory, let's go to the exercises.

---

## Exercises

C.11.1 In the following program, which one of the scanf() and printf() functions may change the value of a;

```
#include <stdio.h>
int main(void)
{
    int a;

    scanf("%d", &a);
    printf("Value: %d\n", a);
    return 0;
}
```

**Answer:** scanf() may change the value of a, because the passing argument is its address. On the other hand, printf() cannot change a, because the passing argument is its value.

C.11.2 Write two functions that take an integer parameter and return the square and the cube of this number, respectively. Write a program that reads an integer and uses the functions to display the sum of the number's square and cube.

```
#include <stdio.h>

int square(int a);
int cube(int a);

int main(void)
{
    int i, j, k;

    printf("Enter number: ");
    scanf("%d", &i);

    j = square(i);
    k = cube(i);
    printf("sum = %d\n", j+k); /* Without declaring the j and k
variables, we could write printf("sum = %d\n", square(i)+cube(i)); */
    return 0;
}

int square(int a)
{
```

```
    return a*a;
}

int cube(int a)
{
    return a*a*a;
}
```

C.11.3 Write a function that takes as parameters three values and returns the minimum of them. Write a program that reads the grades of three students and uses the function to display the minimum.

```
#include <stdio.h>

float min(float a, float b, float c);

int main(void)
{
    float i, j, k;
    printf("Enter grades: ");
    scanf("%f%f%f", &i, &j, &k);

    printf("Min grade = %f\n", min(i, j, k));
    return 0;
}

float min(float a, float b, float c)
{
    if(a <= b && a <= c)
        return a;
    else if(b < a && b < c)
        return b;
    else
        return c;
}
```

C.11.4 Write a function that takes as parameters an integer and a character and displays the character as many times as the value of the integer. Write a program that reads an integer and a character and uses the function to display the character.

```
#include <stdio.h>

void show_char(int num, char ch);

int main(void)
{
    char ch;
    int i;

    printf("Enter character: ");
    scanf("%c", &ch);

    printf("Enter number: ");
```

```

    scanf ("%d", &i);

    show_char(i, ch);
    return 0;
}

void show_char(int num, char ch)
{
    int i;

    for(i = 0; i < num; i++)
        printf ("%c", ch);
}

```

**Comments:** The return type of `show_char()` is declared `void`, because it does not return any value.

**C.11.5** What is the output of the following program?

```

#include <stdio.h>

int f(int a);

int main(void)
{
    int i = 10;

    printf ("%d\n", f(f(f(i))));
    return 0;
}

int f(int a)
{
    return a+1;
}

```

**Answer:** Each time `f()` is called, it returns the value of its argument incremented by one. The calls are executed inside out. Therefore, the first call returns 11, which becomes the argument in the second call. The second call returns 12, which becomes the argument in the third call. Therefore, the program displays 13.

**C.11.6** Write a function that takes as parameter a character and uses the `switch` statement to return the same character if it is 'a', 'b' or 'c', otherwise the character 'z'. Write a program that reads a character, calls the function, and displays the return value.

```

#include <stdio.h>

char check(char ch);

int main(void)
{
    char ch;

```

```

printf("Enter character: ");
scanf("%c", &ch);

printf("%c\n", check(ch));
return 0;
}

char check(char ch)
{
    switch(ch)
    {
        case 'a':
        case 'b':
        case 'c':
            return ch;

        default:
            return 'z';
    }
}

```

**C.11.7** Write a function that takes as parameters three floats and returns the average of those within [1,2]. Write a program that reads three floats, calls the function, and displays the return value.

```

#include <stdio.h>

double avg(double a, double b, double c);

int main(void)
{
    double i, j, k, ret;

    printf("Enter prices: ");
    scanf("%lf%lf%lf", &i, &j, &k);

    ret = avg(i, j, k);
    if(ret == -1)
        printf("No value in [1, 2]\n");
    else
        printf("Avg = %f\n", ret);
    return 0;
}

double avg(double a, double b, double c)
{
    int k = 0;
    double sum = 0;

    if(a >= 1 && a <= 2)
    {
        sum += a;
    }
}

```

```

        k++;
    }
if(b >= 1 && b <= 2)
{
    sum += b;
    k++;
}
if(c >= 1 && c <= 2)
{
    sum += c;
    k++;
}
if(k != 0)
    return sum/k;
else
    return -1;
}

```

**C.11.8** What is the output of the following program?

```

#include <stdio.h>

void test(int *ptr1, int *ptr2);

int main(void)
{
    int i = 10, j = 20;

    test(&i, &j);
    printf("%d %d\n", i, j);
    return 0;
}

void test(int *ptr1, int *ptr2)
{
    int m, *tmp;

    tmp = ptr1;
    ptr1 = &m;
    *ptr1 = 100;

    *ptr2 += m;
    ptr2 = tmp;
    *ptr2 = 100;
}

```

**Answer:** When `test()` is called, we have `ptr1 = &i` and `ptr2 = &j`. The statements `ptr1 = &m;` and `*ptr1 = 100;` make `m` equal to 100. Since `ptr2` points to the address of `j`, `*ptr2` is 20. Therefore, the statement `*ptr2 += m;` makes `j` equal to  $20+100 = 120$ . Since `tmp` points to the address of `i`, the statement `ptr2 = tmp;` is equivalent to `ptr2 = &i`. Therefore, the statement `*ptr2 = 100;` changes the value of `i` to 100. As a result, the program outputs: 100 120

**C.11.9** Write a function that takes an integer parameter (e.g., n) and returns the result of  $1^3 + 2^3 + 3^3 + \dots + n^3$ . Write a program that reads a positive integer up to 1000 and uses the function to display the result of the expression.

```
#include <stdio.h>

double sum_cube(int num);

int main(void)
{
    int i;

    do
    {
        printf("Enter number: ");
        scanf("%d", &i);
    } while(i < 0 || i > 1000);
    printf("Result = %.0f\n", sum_cube(i));
    return 0;
}

double sum_cube(int num)
{
    int i;
    double sum; /* It is declared as double in order to store larger
numbers. */
    sum = 0;
    for(i = 1; i <= num; i++)
        sum += i*i*i;

    return sum;
}
```

**C.11.10** Write a function that takes as parameters two pointers to floats and swaps the values they point to. Write a program that reads two floats and uses the function to swap them.

```
#include <stdio.h>

void swap(float *ptr1, float *ptr2);

int main(void)
{
    float i, j;

    printf("Enter numbers: ");
    scanf("%f%f", &i, &j);
    swap(&i, &j);
    printf("i = %f j = %f\n", i, j);
    return 0;
}
```

```

void swap(float* ptr1, float* ptr2)
{
    float m;

    m = *ptr1; /* Equivalent to m = i; */
    *ptr1 = *ptr2; /* Equivalent to i = j; */
    *ptr2 = m; /* Equivalent to j = m = i; */
}

```

**C.11.11** Write the power(**double** a, **int** b); function that returns the result of  $a^b$ . Write a program that reads a float number (e.g., a) and an integer (e.g., b) and uses the function to display the result of  $a^b$ .

```

#include <stdio.h>

double power(double a, int b);

int main(void)
{
    int exp;
    double base;

    printf("Enter base and exponent: ");
    scanf("%lf%d", &base, &exp);
    printf("%f power %d = %f\n", base, exp, power(base, exp));
    return 0;
}

double power(double a, int b)
{
    int i, exp;
    double val;

    val = 1; /* Necessary initialization. */
    exp = b;
    if(exp < 0) /* If the exponent is negative, make it positive. */
        exp = -exp;
    for(i = 0; i < exp; i++)
        val *= a;
    if(b < 0)
        val = 1/val;
    return val;
}

```

**Comments:** If we want to output the result in a scientific form of greater precision, we can use %e instead of %f. Notice that the program does not check extreme conditions such as an overflow case or the use of more efficient algorithms. For example, instead of executing the loop `exp` times, we could check if `exp` is even and, if it is, execute the loop `exp/2` times and then square the result. If it is odd, multiply the result once more with `a`. For example, if `exp` is 8, the loop is executed 4 times ( $a^4$ )<sup>2</sup>. For the same purpose, the C library provides the `pow()` function as shown in Appendix C.

C.11.12 Write a **void** function that generates a random number from 0 to 1 with two decimal digits and uses a proper parameter to return it. Write a program that calls the function and displays the return value.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void make_rand(double *p);

int main(void)
{
    double d;

    make_rand(&d);
    printf("%.2f\n", d);
    return 0;
}

void make_rand(double *p)
{
    int i;

    srand(time(NULL)); /* It is reminded that srand() together with
time() are used to generate random positive integers, each time the
program runs. */
    i = rand()%101; /* rand() returns a random integer which is
constrained in [0, 100]. */
    *p = i/100.0; /* This division constrains the value within [0, 1]
with two decimal digits. */
}
```

**Comments:** Since the return type is **void** we use a pointer parameter, so that the function can return the value.

C.11.13 Write a **void** function that takes as parameters the coefficients of a quadratic trinomial and returns its real roots, if any. Write a program that reads the coefficients of a trinomial (e.g., a, b, and c) and uses the function to solve the equation. The program should force the user to enter a nonzero value for a. It is reminded from algebra that in order to find the roots of the trinomial  $ax^2 + bx + c$  with  $a \neq 0$ , the discriminant  $D = b^2 - 4ac$  is tested.

- a. If  $D > 0$ , it has two real roots  $r_{1,2} = (-b \pm \sqrt{D})/2a$ .
- b. If  $D = 0$ , it has one real double root  $r = -b/2a$ .
- c. If  $D < 0$ , it has no real root.

To calculate the square root, use the **sqrt()** function, as shown in Appendix C.

```
#include <stdio.h>
#include <math.h>

void find_roots(double a, double b, double c, double *r1, double *r2, int
*code);
```

```

int main(void)
{
    int code;
    double a, b, c, r1, r2;

    do
    {
        printf("Enter coefficients (a<>0): ");
        scanf("%lf%lf%lf", &a, &b, &c);
    } while(a == 0);

    find_roots(a, b, c, &r1, &r2, &code);
    if(code == 2)
        printf("Two roots: %f %f\n", r1, r2);
    else if(code == 1)
        printf("One root: %f\n", r1);
    else
        printf("Not real roots\n");
    return 0;
}

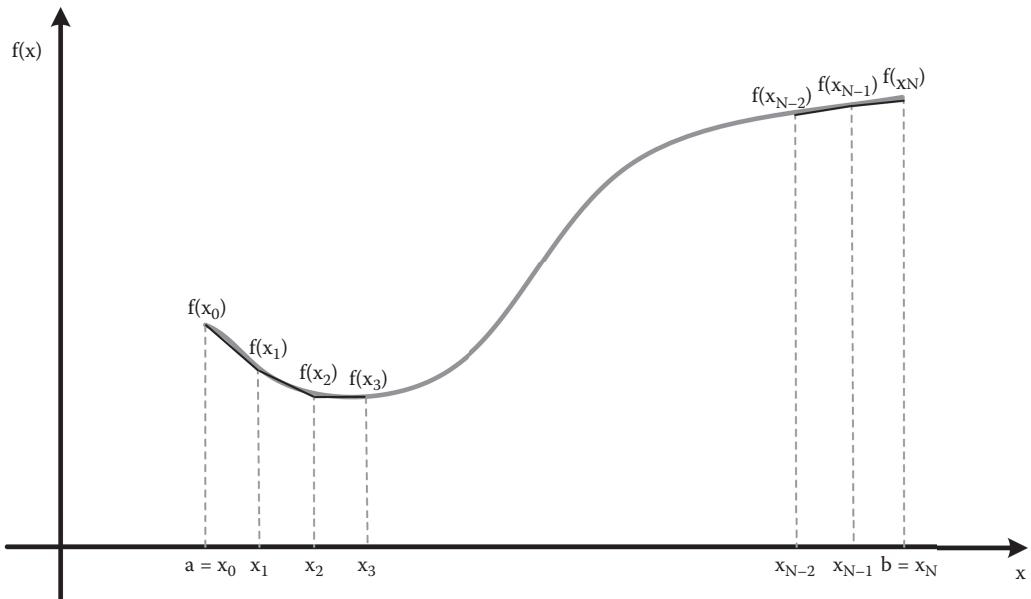
void find_roots(double a, double b, double c, double *r1, double *r2, int
*code)
{
    double d;

    d = b*b-4*a*c;
    if(d > 0)
    {
        *code = 2;
        *r1 = (-b+sqrt(d))/(2*a);
        *r2 = (-b-sqrt(d))/(2*a);
    }
    else if(d == 0)
    {
        *code = 1;
        *r1 = *r2 = -b/(2*a);
    }
    else
        *code = 0;
}

```

**Comments:** If you are using the `gcc` compiler and a message similar to “*undefined reference to sqrt*” is displayed, it means that the compiler didn’t find the code for `sqrt()`. Suppose that the file name is `test.c`; write in the command line: `gcc test.c -lm`, in order to load the library with the math functions.

**C.11.14** In math, the trapezoidal rule is a technique for approximating the definite integral of a function  $f(x)$  within an interval  $[a, b]$ . The rule works by adding the areas of the trapezoids that are formed when approximating the function with a polyline, as depicted in Figure 11.1.



**FIGURE 11.1**

Trapezoidal rule approximation.

The procedure involves the following steps:

- Split the interval  $[a, b]$  into  $N$  subintervals of the same width. Therefore,  $x_i = x_0 + \frac{i \times (b - a)}{N}$ ,  $i = 0, \dots, N$ . For larger values of  $N$ , the approximation is closer the value of the integral.
- Calculate the values  $f(x_i)$  for  $i = 0, \dots, N$ .
- Calculate the areas of the  $N$  trapezoids formed under the polyline and add them. The integral is approximately

$$\int_a^b f(x) dx = E_{tr1} + \dots + E_{trN} = \frac{(x_1 - x_0) \times (f(x_0) + f(x_1))}{2} + \dots + \frac{(x_N - x_{N-1}) \times (f(x_{N-1}) + f(x_N))}{2}$$

Write a program that uses the trapezoidal rule to approximate the value of the integral  $\int_a^b x^2 dx = \left[ \frac{x^3}{3} \right]_a^b$ . The program should read the number  $N$ , the limits  $a$  and  $b$ , and check their validity.

```
#include <stdio.h>

double f(double x);

int main(void)
{
    int i, a, b, n;
    double sum, width, x0, x1;
```

```

double step, x, area;

do
{
    printf("Enter number of intervals: ");
    scanf("%d", &n);
} while(n <= 0);

do
{
    printf("Enter low and up limit (a < b): ");
    scanf("%d%d", &a, &b);
} while(a >= b);

step = (double) (b-a)/n;
x = a;
area = 0;
for(i = 0; i < n; i++)
{
    area += 0.5*step*(f(x)+f(x+step));
    x += step;
}
printf("Calculus = %f\n", area);
return 0;
}

double f(double x) /* The reason we're using a function is that if we've
to calculate another integral it is easy to change the code in the
function. */
{
    return x*x;
}

```

## Storage Classes and Variables Scope

Every variable is characterized by three properties: *storage class*, *scope*, and *linkage*. The default properties of a variable depend on the point of its declaration.

The *storage class* of a variable determines when memory is allocated for the variable and when that memory is released. There are two storage classes: *automatic* and *static*. The storage class is specified by using one of the keywords: **auto**, **register**, **extern** or **static**. The *scope* of a variable is the part of the program in which the variable can be accessed, or else it is visible; it depends on the point of its declaration. The *linkage* of a variable concerns the linker and determines whether it can be shared by different parts of the program. A variable with *external linkage* may be shared by several files in a program, while a variable with *internal linkage* is restricted to the file that it is declared; however, it may be shared by the functions in that file. A variable with *no linkage* belongs to a function and cannot be shared.

## Local Variables

A variable declared within the body of a function is called *local*. A local variable has block scope; its scope is constrained inside the function from the point of its declaration to the end of the enclosing block. Therefore, no other function can access it. Since a local variable is not visible outside the function in which it is declared, we can use the same name to declare variables in other functions. For example, in the following program, the local variable *i* declared in *main()* is different from the local variable *i* declared in *test()*, although they have the same name.

```
#include <stdio.h>

void test(void);

int main(void)
{
    int i = 10;

    test();
    printf("I_main = %d\n", i);
    return 0;
}

void test(void)
{
    int i = 200;
    printf("I_test = %d\n", i);
}
```

Since the variables are different, each has its own value, and the program outputs:

```
I_test = 200
I_main = 10
```

What happens if we remove the declaration `int i;` from `test()`?

Since the two *i* variables are unrelated, the compiler would issue an error message that the variable *i* in *test()* is undeclared.

The parameters of the function are also considered local variables of the function. For example:

```
#include <stdio.h>

void test(int i, int j);

int main(void)
{
    int i = 100, j = 100;

    test(i, j);
    printf("%d %d\n", i, j);
    return 0;
}
```

```
void test(int i, int j)
{
    int a = 2000; /* The local variables of the function are a, i and
j. */
    i = j = a;
}
```

Since the parameters of the function are treated as local variables, the variables `i` and `j` of `test()` are not related with those of `main()`. Therefore, the program displays 100 100.

The default storage class of the variables we've used so far in our programs is `auto`. The `auto` class can be used only for variables declared in a function. For a local variable `a`, the declarations `auto int a;` and `int a;` are equivalent; therefore, the `auto` specifier can be omitted. Each time the function is called, the compiler allocates memory to store the values of the local variables. This memory is automatically released when the function terminates; therefore, a local variable does not retain its value between successive calls. Because a local variable ceases to exist, it is usually referred to as an *automatic* variable having *automatic* storage duration. To sum up, the default properties of a local variable is block scope, no linkage, and automatic storage duration. Because the parameters of a function are in effect local variables, they have the same properties. The only storage class that can be applied to a parameter is the `register` class.

The `register` class can only be applied to automatic variables and to the parameters of a function. Declaring a local variable with the `register` class (e.g., `register int i;`) informs the compiler that the variable will be used a lot and requests to save it in a system register for faster access. However, the compiler is free to ignore the request and store the variable in the main memory as it does with other variables. The initial value of an automatic or `register` variable that is not explicitly initialized is undefined. A `register` variable has the same scope, linkage, and lifetime as an `auto` variable. However, it is not allowed to use the `&` operator to get the address of a `register` variable, even if the compiler didn't store the variable in a register. Today's compilers have been evolved a lot and can determine by themselves which variables should be kept in registers.

C allows us to declare variables after the `{` of any compound statement, not only after the `{` of a function. The general form of a compound statement is `{ declarations statements }`, and that form specifies a block. The scope of a variable declared in a block extends up to the matching `};` therefore, it does not conflict with any other variable outside the block having the same name. By default, the storage duration is automatic; the compiler allocates memory to store it when the block is entered and releases that memory when the block is exited. Consider the following program:

```
#include <stdio.h>
int main(void)
{
    int i = 20, num = 10;

    if(num == 10)
    {
        int i; /* Allocate memory for the new i. */
        i = 50;
    } /* Release memory. */
    printf("%d\n", i);
    return 0;
}
```

Because the `i` variable declared inside the `if` block is unrelated to the first `i`, the program displays 20. A reason to declare a variable inside a compound statement instead of the beginning of a function is that memory will be allocated only if needed. This might be useful when writing applications in systems with limited memory. A compound statement is not needed to join a conditional or loop statement. For example, the following program outputs 50 and 20.

```
#include <stdio.h>
int main(void)
{
    int i = 20;

    {
        int i = 50;
        printf("%d\n", i);
    }
    printf("%d\n", i);
    return 0;
}
```

Since the memory of a local variable is released, a function should not return its address. For example:

```
#include <stdio.h>

int *test(void);

int main(void)
{
    int *ptr, j;

    ptr = test();
    printf("%d\n", *ptr);

    j = *ptr;
    printf("%d\n", j);
    return 0;
}

int *test(void)
{
    int i = 10;
    return &i;
}
```

When `test()` is called, the compiler allocates memory for the local variable `i`. Then, it returns the address of `i`. Since this memory is released, it may be used for other purposes, so the value 10 might be overwritten. Therefore, the program might output 10 and a random value, or two random values, instead of 10 and 10. The behavior of the program is unpredictable. Most probably, the compiler will warn you with a message such as *"returning address of local variable."*

However, as we'll see later, if the variable `i` is declared as static you can safely return its address because the memory allocated for a `static` variable is not released when the function terminates.

---

Don't return the address of a local variable unless it is declared as `static`.

Somewhere later, there is an exercise to test if you digested this rule.

## Global Variables

A variable that is declared outside of any function is called *global*. Its scope extends from the point of its declaration to the end of the file in which it is declared. Therefore, it can be accessed by all functions that follow its declaration. Unlike an automatic variable, a global variable has a permanent lifetime and retains its last assigned value. To sum up, the default properties of a global variable is file scope, external linkage, and static storage duration; that is, it remains in existence permanently.

When the same variable is used in many functions, many programmers choose to declare it as global instead of passing it as an argument in function calls. In fact, a global variable can be used as an alternative for communicating data between functions.

Typically, we avoid the use of global variables because it may be more difficult to identify erroneous conditions. For example, if an undesired value is assigned to a global variable, it'd be harder to identify the guilty function. Excessive relying on global variables may have a bad effect on program structure and make its comprehension more difficult. For example, it'd be more difficult for the reader to correlate each global variable with the functions that use it. We prefer to use arguments instead, so when reading the function declarations, it is clear which functions need them and if these functions can change their values. For example, if a wrong value is assigned, the list of the guilty functions is narrowed to those which can change its value.

When naming a global variable, it is very important to choose a descriptive name; it helps the reader to understand its role. Don't use meaningless names that are often given in local variables, like `i`. By default, the value of an uninitialized global variable or the elements of an aggregate type (e.g., array, structure) are set to 0, however, our preference is to set it explicitly. The following program declares the global variable `glob`. Since it is declared before the definitions of the functions, all functions may access it.

```
#include <stdio.h>

void add(void);
void sub(void);

int glob = 10;

int main(void)
{
    add();
    printf("Val = %d\n", glob);
    sub();
```

```

    printf("Val = %d\n", glob);
    return 0;
}

void add(void)
{
    glob++;
}

void sub(void)
{
    glob--;
}

```

Since the initial value of glob is 10, the program outputs:

```
Val = 11
Val = 10
```

A local variable may have the same name as a global variable. The scope rule says that when a variable inside a block is named the same as another variable that is already visible, the new declaration hides the old one in that block. For example:

```

#include <stdio.h>

void test(void);
void last(int a);

int a = 100;

int main(void)
{
    test();
    a += 50;
    printf("%d ", a);
    last(200);
    return 0;
}

void test(void)
{
    int a = 300;

    if(a == 300)
    {
        int a = 500;
        a++;
        printf("%d ", a);
    }
    a += 20;
    printf("%d ", a);
}

void last(int a)

```

```
{  
    a++;  
    printf("%d\n", a);  
}
```

The four variables named `a` are different. Let's apply the scope rule. The `a` tested in the `if` condition is the one declared in `test()`, while the statement `a++;` refers to the `a` declared in the `if` block. Its scope is up to the `}`, and the next statement `a+=20;` increases the `a` of `test()`. Its scope is up to the `}` at the end of the function. `main()` increases the value of the global `a` which becomes 150. `last()` increases the local parameter `a` which becomes 201. As a result, the program outputs 501 320 150 201.

Needless to say, it'd be better to avoid using the same names, since the code becomes harder to read and error prone.

A global variable can be visible in more than one source file; that is, it may have external linkage. This is very useful, particularly in large programs, because several source files may need to share the same variable. The `extern` storage class enables a variable to be shared among several files. Typically, such a variable is declared as global in the file used most and the other files refer to it using the `extern` keyword.

Up to this point, we've not discussed about the distinction between *declaring* and *defining* a variable. Let's see the difference. The statement `int size;` not only declares the type of `size` to be `int` but defines `size` as well; that is, the compiler allocates memory to store its value. A variable can have only one definition; other files may contain `extern` declarations to access it. Initialization of an external variable goes only with its definition. For example, to use it in another file we write:

```
extern int size;
```

Because of the word `extern`, the variable `size` is only declared, not defined; that is, the compiler does not allocate new memory for it. In other words, the word `extern` informs the compiler that the variable `size` is defined somewhere else. `size` can be accessed and modified if needed, in any of those files that is referred to as `extern`. Since the compiler does not allocate new memory, we can omit the length of an array in an `extern` declaration. For example:

```
extern int arr[];
```

Essentially, `extern` is another way for data exchange between functions that reside in different files. Although sharing variables among files is very popular, care has to be taken because a wrong assignment in one file may induce unpredictable results in the other files that use it. We'll see examples of `extern` declarations later in this chapter and in the program of Chapter 17.

## Static Variables

As discussed, the memory that is allocated to store a local variable is released when the function terminates. Therefore, there is no guarantee that a local variable would still have its old value when the function is called again. If we want a local variable to retain its value, we should use the `static` storage class. Unlike an ordinary local variable, a `static` variable has static storage duration; that is, it has a permanent lifetime. It resides at the same memory location throughout program execution, and that

memory is not released when the function terminates. To sum up, the default properties of a **static** variable declared inside a function are block scope, no linkage, and static storage duration.

A **static** variable is initialized only once, the first time that the function is called. As with a global variable, the initializer must be a constant expression; if it is not explicitly initialized, the default value is 0. In the next calls, it retains its last value and it is not initialized again. For example, consider the following program:

```
#include <stdio.h>

void test(void);

int main(void)
{
    test();
    test();
    test();
    return 0;
}

void test(void)
{
    static int i = 100, arr[1];
    int j = 0;

    i++;
    arr[0] += 2;
    j++;
    printf("%d %d %d\n", i, arr[0], j);
}
```

When `test()` is first called, `i` becomes 101. Since `i` is declared as **static**, it retains its value and the next call of `test()` makes it 102. Similarly, the third call makes it 103. Since the elements of a static array are set to 0 by default, `arr[0]` becomes 2, 4 and 6, respectively. On the other hand, since `j` is an automatic variable, it does not retain its value between `test()` calls. Therefore, the program outputs:

```
101 2 1
102 4 1
103 6 1
```

As another example, a **static** variable declared inside a block of a compound statement is initialized only the first time the block is entered. For example:

```
#include <stdio.h>

void test(int num);

int main(void)
{
    test(50);
    test(10);
    test(10);
```

```
    return 0;
}

void test(int num)
{
    if(num == 10)
    {
        static int i = 100;
        i++;
        printf("%d ", i);
    }
}
```

The program outputs: 101 102

The use of **static** may improve the performance of a program. For example, if a **const** variable is declared inside a function that is called many times, it'd be more efficient to declare it as **static** as well, so that it is initialized only once. Therefore, the performance of the function is increased. Here is an example:

```
void test(void)
{
    static const int c = 200;
    ...
}
```

The linkage of a global variable or function is external by default, meaning that they can be shared across files. However, if it is declared as **static**, the linkage changes to internal, which means that it is visible only within the file which it is defined. For example:

```
#include <stdio.h>

static void test(int i);
static int flag;

int main(void)
{
    ...
}

static void test(int i)
{
    ...
}
```

`flag` and `test()` are visible only within this file, which means that they cannot be accessed by functions in other files. In particular, `flag` has file scope, static storage duration, but internal linkage. The primary reason of declaring them as **static** is to hide them from other files, thus preventing use as **extern** when it is not desired. In fact, in a large program split in several files, typically written by different programmers, it is a good idea to declare your functions as **static**. Besides information hiding, another reason is that if the code of a **static** function should be modified later, you won't be concerned if those

changes affect functions in other files (with the exception that a pointer to that function is passed to a function of another file). Also, you don't have to worry if the names you choose are used in other files as well, because even if it happens, there will be no conflict. Since the linkage is internal, the same names can be reused in other files. Here is a last example that puts these concepts together. Suppose that this program consists of two files. Both files must be compiled and linked together to get the complete program. You'll see how to do it in Chapter 17. Could you tell us what does it output?

```
/* First file. */

#include <stdio.h>

void test(void);
int ext = 10;
static int st = 20;

int main(void)
{
    test();
    printf("%d %d\n", ext, st);
    return 0;
}

/* Second file. */

#include <stdio.h>

extern int ext;
static int st;

void test(void)
{
    st++;
    printf("%d %d\n", ext, st);
    ext = 30;
}
```

Since the `st` variable is declared as `static` in the first file, it does not conflict with the `st` declared in the second file (either if this second `st` is declared as `static` or not). Notice that the definition of `test()` resides in the second file. Since `st` is initialized to 0 by default, `test()` outputs 10 and 1. Since `test()` changed the value of `ext` to 30, the program outputs 30 and 20.

---

## One-Dimensional Array as Argument

When a parameter of a function is a one-dimensional array, we write the name of the array followed by a pair of brackets. The length of the array can be omitted; in fact, this is the common practice. We'll explain later why. For example:

```
void test(int arr[]);
```

When passing an array to a function, we write only its name, without brackets. For example:

```
test(arr);
```

---

When an array name is passed to a function, it is always treated as a pointer. In effect, the passing argument is the memory address of its first element, not a copy of the array itself. Since no copy of the array is made, the time required to pass an array to a function does not depend on the size of the array.

Just remember, when an ordinary variable is passed to a function, the function works with a copy. But if an array is passed, no copy of the array is made; the function works with the original. Why is that decision taken? To improve performance, since it could be expensive in memory and time to copy an entire array. It is just a pointer to its first element that is passed and the function may access any of its elements. For example:

```
#include <stdio.h>

void test(int arr[]);

int main(void)
{
    int i, a[5] = {10, 20, 30, 40, 50};

    test(a); /* Equivalent to test(&a[0]); */
    for(i = 0; i < 5; i++)
        printf("%d ", a[i]);
    return 0;
}

void test(int arr[])
{
    arr[0] = arr[1] = 0;
}
```

Notice that we could use the name `a` instead of `arr`, since you know by now that local variables of different functions are not related even if they are named the same. When `test()` is called, we have `arr = a = &a[0]`. Therefore, the statements `arr[0] = 0;` and `arr[1] = 0;` change the values of the first two elements and the program displays

```
0 0 30  
40 50
```

Let's see an example of passing an array to a library function. What would be the output?

```
#include <stdio.h>
int main(void)
{
    char s[] = "%c\n";
    *(s+1) = 'd';
    printf(s, *s);
    return 0;
}
```

Since `*(s+1)` is equivalent to `s[1]`, the string changes to "%d\n" and the program displays the ASCII value of the '%' character.

An array parameter can be declared as pointer, as well. For example, the declarations:

```
void test(int arr[]); and void test(int *arr);
```

are equivalent. The compiler treats both the same and passes the pointer to the function. Therefore, the second declaration is more accurate since it clearly states that only a pointer is passed, not a copy of the array. However, our preference is the first one, to show explicitly that the intention is to pass an array. The second declaration is ambiguous; the reader cannot figure out whether the function accepts a number of values or a pointer to a single variable.

Alternatively, we could use pointer arithmetic to access the array elements. For example:

```
void test(int arr[])
{
    *arr = 0; /* Equivalent to arr[0] = 0. */
    arr++;
    *arr = 0;
}
```

What is really interesting is the statement `arr++`. Can we change the value of an array? Certainly not, you know by now that when the name of an array is used as pointer, it is a `const` pointer. So, why does the compiler allow this statement? Because, as said, although `arr` is declared as array, it is actually a pointer. Therefore, it can be assigned with a new value. One more example, does the following code compile, and if yes, what is the output?

```
void test(int arr[])
{
    int i = 20, b[10];

    arr = b;
    *arr = 10;
    arr = &i;
    *arr = 30;
    printf("%d %d", b[0], i);
}
```

Of course it compiles; `arr` is treated as an ordinary pointer. The function outputs 10 and 30.

---

To prevent a function from changing the values of the array elements, declare the parameter as `const`.

For example, with the declaration:

`void test(const int arr[]);` `test()` cannot modify the value of any `arr` element. Note that this does not mean that the original array needs to be declared as constant. It just says that `test()` can treat `arr` as read-only data.

When passing an array to a function, we can pass a part of it. For example:

```
#include <stdio.h>

void test(int ptr[]);

int main(void)
{
    int i, arr[6] = {1, 2, 3, 4, 5, 6};

    test(arr+3); /* Alternatively, test(&arr[3]). */
    for(i = 0; i < 6; i++)
        printf("%d ", arr[i]);
    return 0;
}

void test(int ptr[])
{
    int i, tmp[3] = {10, 20, 30};

    for(i = 0; i < 3; i++)
        ptr[i] = tmp[i];

    *ptr = *(ptr-1);
}
```

When `test()` is called, we have `ptr = arr+3`, that is, the part of `arr` from the fourth element is passed. Since we use `ptr` as array, `ptr[0]` corresponds to `arr[3]`, `ptr[1]` corresponds to `arr[4]`, and `ptr[2]` to `arr[5]`. Therefore, the loop makes the values of `arr[3]`, `arr[4]` and `arr[5]` equal to 10, 20, and 30, respectively. Since `ptr` points to `arr[3]`, the statement `*ptr = *(ptr-1);` is equivalent to `arr[3] = arr[2];`. As a result, the program displays 1 2 3 3 20 30

Let's make a tricky question. Could we write the following?

```
void test(int ptr[])
{
    printf("%d ", ptr[-1]);
    ...
}
```

Strange as it may seem, the answer is yes. As you already know from Chapter 8, `ptr[-1]` is equivalent to `*(ptr-1)` and since `ptr` is equal to `arr+3`, it is equivalent to `*(arr+3-1)`, that is, `arr[2]`. If it is certain that elements in the backward direction do exist, yes, it is safe to use negative indexing and access them. Of course, if someone chooses this syntax, we've nothing to say but dedicate *Personality Crisis* from *New York Dolls*.

---

Although we can use the `sizeof` operator to find the size of an array variable, we cannot use it in a function to find the size of an array parameter.

---

For example, consider the following program:

```

#include <stdio.h>

void test(int arr[]);

int main(void)
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    test(a);
    printf("Array = %u bytes\n", sizeof(a));
    return 0;
}

void test(int arr[])
{
    printf("Ptr = %u bytes\n", sizeof(arr));
}

```

Since an array parameter is treated as a pointer, the `sizeof` operator calculates the size of a pointer variable, not the actual size of the array. Therefore, the program outputs 4 for the pointer and 40 for the array.

Because it is the pointer that is actually passed to the function, it has no effect if we put the length of the array inside the brackets. The compiler ignores it; it won't check if the array actually has the indicated length. In fact, the passing pointer is a pointer to the first element of an array of unknown length. That's why we leave the brackets empty; we don't want to create the illusion that a specific number of elements is passed. An easy way to make known to the function the length of the array is to pass it as an additional argument. For example, we could declare `test()` as follows:

```
void test(int arr[], int size);
```

and write `test(a, 10);` to call it.

Alternatively, we could define a constant, e.g., `#define SIZE 10`, and use that constant wherever needed instead of passing an additional argument. Another way is to use a special value to indicate the end of the array, provided, of course, that this value wouldn't occur in the array. For example, if an array contains a string, it is not needed to pass its length, since the null character indicates its end.

An easy question before finishing, to see that you fully understood this very important section. Does the following program output the same values? If yes, explain why.

```

#include <stdio.h>

void test(int arr[]);

int main(void)
{
    int a[10];

    printf("Addr: %p\n", a);
    test(a);
}
```

```
    return 0;
}

void test(int arr[])
{
    printf("Addr: %p\n", &arr);
}
```

Don't hurry to see the answer in the next page, give it some thought.

Not only did you answer yes, but did you justify your answer? We tricked you once again... watch out for the traps coming next. Don't be angry, listen to *Don't Look Back in Anger* from *Oasis* and check the program again.

The first `printf()` outputs the address of the `a` variable. `test()` outputs the address of the `arr` variable. Are they the same? Of course not; they reside in different memory locations. However, the value of `arr` is equal to `a`; that is, if we remove `&`, the output values would be the same.

---

## Exercises

C.11.15 Write a function that takes as parameters an array containing the students' grades in a test and two grades (e.g., `a` and `b`) and returns the average of the grades within  $[a, b]$ . Write a program that reads the grades of 50 students and the two grades `a` and `b` and uses the function to display the average. The program should force the user to enter a value for `a` less than or equal to `b`.

```
#include <stdio.h>

#define SIZE 50

float avg_arr(float arr[], float min, float max);

int main(void)
{
    int i;
    float a, b, k, arr[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter grade: ");
        scanf("%f", &arr[i]);
    }
    do
    {
        printf("Enter min and max grades: ");
        scanf("%f%f", &a, &b);
    } while(a > b);

    k = avg_arr(arr, a, b);
    if(k == -1)
        printf("None grade in [%f, %f]\n", a, b);
    else
        printf("Avg = %.2f\n", k);
    return 0;
}

float avg_arr(float arr[], float min, float max)
{
    int i, cnt = 0;
```

```

float sum = 0;

for(i = 0; i < SIZE; i++)
{
    if(arr[i] >= min && arr[i] <= max)
    {
        cnt++;
        sum += arr[i];
    }
}
if(cnt == 0)
    return -1;
else
    return sum/cnt;
}

```

**C.11.16** Write a function that takes as parameter a string and returns its length. Don't use `strlen()`.

```

unsigned int str_len(const char *str)
{
    unsigned int i = 0;
    while(str[i] != '\0') /* Count the characters up to the null
character. */
    {
        i++;
    }
    return i;
}

```

**Comments:** `str_len()` is an implementation example of the `strlen()` library function.

**C.11.17** What is the purpose of the following function?

```

unsigned int test(const char *str)
{
    const char *ptr = str;

    while(*str++) /* Equivalent to while(*str++ != '\0') */
    ;
    return str - ptr - 1;
}

```

**Answer:** Tough, don't be scared.

In each iteration, the loop compares the value of `*str` with 0, equivalently to '`\0`', and then `str` is increased, to point to the next array element. For example, in the first iteration, `*str` is equal to `str[0]`, then it becomes equal to `str[1]`, and so on. Once `*str` becomes equal to the null character, the loop terminates.

Recall from Chapter 8 and pointer arithmetic that the result of the subtraction of two pointers that point to the same array is the number of elements, so, in this example, the number of characters between them. `ptr` points to the first element, while `str` after its last increase points to the next place following the null character. That's why we put `-1`, to subtract this place.

So, what really does this function do? It does the same thing as the `str_len()` function of the previous exercise. It returns the length of the string stored into `str`. The reason we added this exercise is to show you that a problem may be solved in several ways, some simpler and others more complex.

Even if `test()` is executed a bit faster than `str_len()` the reader of `test()` needs much more time to realize what exactly this function does. Once more, if your program does not set very strict requirements in speed and efficiency, try to write simple and clear code for your own benefit and for those who are going to read your code.

In another example of complex coding, the following code uses a `for` loop instead of a `while` loop to produce the same result.

```
unsigned int test(const char *str)
{
    const char *ptr = str;

    for(; *str; str++)
    ;
    return str - ptr;
}
```

C.11.18 Write a function that takes as parameters two strings and returns a pointer to the longer string. If the strings have the same number of characters, it should return `NULL`. Write a program that reads two strings of less than 100 characters and uses the function to display the longer one.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *max_str(char str1[], char str2[]);
int read_text(char str[], int size, int flag);

int main(void)
{
    char *ptr, str1[100], str2[100];

    printf("Enter first text: ");
    read_text(str1, sizeof(str1), 1);

    printf("Enter second text: ");
    read_text(str2, sizeof(str2), 1);

    ptr = max_str(str1, str2);
    if(ptr == NULL)
        printf("Same number of characters.\n");
    else
        printf("Result: %s\n", ptr);
    return 0;
}

char *max_str(char str1[], char str2[])
{
```

```

int i, j;

i = strlen(str1);
j = strlen(str2);

if(i > j)
    return str1;
else if(i < j)
    return str2;
else
    return NULL;
}

```

**Comments:** Just to remind you, we'll omit the definition of `read_text()` presented in Chapter 10, in order to save space.

**C.11.19** Does the following program contain any error? If not, what does it output?

```

#include <stdio.h>
#include <string.h>

char *test(void);

int main(void)
{
    char ptr[100] = "sample";

    strcpy(ptr, test());
    printf("%s\n", ptr);
    return 0;
}

char *test(void)
{
    char str[] = "example";
    return str;
}

```

**Answer:** You've been warned that you'll encounter such an exercise. Let's see, did you find the error or you answered that it is correct?

When `test()` is called, the compiler allocates memory for the `str` array and stores the string into it. This memory location is returned. Remember that the memory of a local variable is released when the function terminates. Most probably, the program won't display `example`. But even if it is displayed, the code is erroneous.

*Always remember, don't return the address of a local variable unless it is declared as `static`.*

If you want to change the contents of an array, the simplest way is to pass the array as an argument. For example, `test()` is modified like this:

```

void test(char str[])
{
    strcpy(str, "example");
}

```

C.11.20 Write a function that takes as parameters two strings and uses them as pointers to copy the second one into the first one. Don't use `strcpy()`. Write a program that reads two strings of less than 100 characters and uses the function to swap them and display their content.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void copy(char s1[], char s2[]);
int read_text(char str[], int size, int flag);

int main(void)
{
    char str1[100], str2[100], tmp[100];

    printf("Enter first text: ");
    read_text(str1, sizeof(str1), 1);

    printf("Enter second text: ");
    read_text(str2, sizeof(str2), 1);

    copy(tmp, str1); /* Copy the first string into tmp. */
    copy(str1, str2); /* Copy the second string into str1. */
    copy(str2, tmp); /* Copy the first string into str2. */
    printf("\nFirst text: %s\n", str1);
    printf("Second text: %s\n", str2);
    return 0;
}

void copy(char s1[], char s2[])
{
    while(*s2 != '\0')
    {
        *s1 = *s2;
        s1++;
        s2++;
    }
    *s1 = '\0';
}
```

C.11.21 Write a function that takes as parameters two strings and returns 1 if the second string is contained at the end of the first one. Otherwise, it should return 0. Write a program that reads two strings of less than 100 characters and uses the function to check whether the second string is contained at the end of the first one or not.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int str_end(char str1[], char str2[]);
int read_text(char str[], int size, int flag);
```

```

int main(void)
{
    char str1[100], str2[100];
    int i;

    printf("Enter first text: ");
    read_text(str1, sizeof(str1), 1);

    printf("Enter second text: ");
    read_text(str2, sizeof(str2), 1);

    i = str_end(str1, str2);
    if(i == 0)
        printf("%s is not at the end of %s\n", str2, str1);
    else
        printf("%s is at the end of %s\n", str2, str1);
    return 0;
}

int str_end(char str1[], char str2[])
{
    int i, len1, len2;

    len1 = strlen(str1);
    len2 = strlen(str2);

    if(len1 < len2) /* If the length of the second string is greater,
the function returns. */
        return 0;
    for(i = 1; i <= len2; i++)
        if(str1[len1-i] != str2[len2-i]) /* If two characters are
not the same, it is not needed to compare the rest, so the function
returns. */
            return 0;
    /* If this point is reached, it means that all compared characters
were the same. */
    return 1;
}

```

**C.11.22** Write a function that takes as parameters two pointers to floats and returns the pointer to the float with the greater value. Write a program that reads two floats and uses the function to display the greater.

```

#include <stdio.h>

double *max(double *ptr1, double *ptr2);

int main(void)
{
    double *ptr, i, j;

    printf("Enter numbers: ");

```

```

scanf("%lf%lf", &i, &j);

ptr = max(&i, &j);
printf("The max of %f and %f is %f\n", i, j, *ptr);
return 0;
}

double *max(double *ptr1, double *ptr2)
{
    if(*ptr1 > *ptr2)
        return ptr1;
    else
        return ptr2;
}

```

**Comments:** max() compares the two input values and returns the pointer to the greater one. This pointer is stored into ptr and printf() displays its content. Notice that we could remove the declaration of ptr and write:

```
printf("The max value of %f and %f is %f\n", i, j, *max(&i, &j));
```

**C.11.23** Modify the previous function in order to take **void** types of pointers and a third argument to declare the data type. For example, if it is 0, the passed type is **int**, otherwise, it is **double**. Write a program to test the operation of the function.

```

#include <stdio.h>

void *max(void *ptr1, void *ptr2, int type);

int main(void)
{
    int *p1, i1, i2;
    double *p2, d1, d2;

    printf("Enter integers: ");
    scanf("%d%d", &i1, &i2);

    printf("Enter floats: ");
    scanf("%lf%lf", &d1, &d2);

    p1 = (int*) max(&i1, &i2, 0);
    p2 = (double*) max(&d1, &d2, 1);
    printf("Max1:%d Max2:%f\n", *p1, *p2);
    return 0;
}

void *max(void *ptr1, void *ptr2, int type)
{
    if(type == 0)
    {
        if(*(int*)ptr1 > *(int*)ptr2)
            return (int*)ptr1;
        else
    }
}
```

```

        return (int*)ptr2;
    }
else
{
    if(*(double*)ptr1 > *(double*)ptr2)
        return (double*)ptr1;
    else
        return (double*)ptr2;
}
}

```

**Comments:** We could make max() more generic, in order to compare more data types.

C.11.24 What is the output of the following program?

```

#include <stdio.h>
#include <string.h>

void test(char ch, char *ptr);

int main(void)
{
    char str[] = "bacdefghij";

    test(*str-1, &str[5]);
    printf("%s\n", str);
    return 0;
}

void test(char ch, char *ptr)
{
    strcpy(ptr, "12345");
    *ptr = ch;
}

```

**Answer:** When test() is called, we have ch = \*str-1 = str[0]-1 = 'b'-1, so ch becomes equal to 'a'. We also have ptr = &str[5]. Since ptr points to the sixth element of str, the statement strcpy(ptr, "12345"); changes its content to "bacde12345". The statement \*ptr = ch; is equivalent to str[5] = ch = 'a'. Therefore, the program displays: bacdea2345

C.11.25 Write a function that takes as parameters two strings, uses them as pointers, and returns 0 if they are identical or the difference of the first two nonmatching characters. Write a program that reads two strings of less than 100 characters and uses the function to display the result of the comparison.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int str_cmp(const char *str1, const char *str2);

```

```

int read_text(char str[], int size, int flag);

int main(void)
{
    char str1[100], str2[100];
    int i;

    printf("Enter first string: ");
    read_text(str1, sizeof(str1), 1);

    printf("Enter second string: ");
    read_text(str2, sizeof(str2), 1);

    i = str_cmp(str1, str2);
    if(i == 0)
        printf("%s = %s\n", str1, str2);
    else if(i < 0)
        printf("%s < %s\n", str1, str2);
    else
        printf("%s > %s\n", str1, str2);
    return 0;
}

int str_cmp(const char *s1, const char *s2)
{
    while(*s1 == *s2)
    {
        if(*s1 == '\0')
            return 0;
        s1++;
        s2++;
    }
    return *s1-*s2;
}

```

**Comments:** If two different characters are found, the loop terminates and the function returns their difference. The `str_cmp()` function is an implementation example of the `strcmp()` library function.

**C.11.26** What is the output of the following program?

```

#include <stdio.h>

void test(int *ptr1, int *ptr2, int a);

int main(void)
{
    int i = 1, j = 2, k = 3;

    test(&i, &j, k);
    printf("%d %d %d\n", i, j, k);
    return 0;
}

```

```

void test(int *ptr1, int *ptr2, int a)
{
    ptr1 = ptr2;

    *ptr1 = 100;
    *ptr2 = 200;
    a = *ptr1 + *ptr2;
    printf("%d\n", a);
}

```

**Answer:** When `test()` is called, we have `ptr1 = &i`, `ptr2 = &j` and `a = k`. The statement `ptr1 = ptr2;` makes `ptr1` point to the address of `j`. Therefore, the statement `*ptr1 = 100;` makes `j` equal to 100. Since `ptr2` points to the address of `j`, the statement `*ptr2 = 200;` changes the value of `j` to 200. Since the statement `a = *ptr1 + *ptr2;` is equivalent to `a = j+j = 200+200 = 400`, `test()` displays 400. Since any change in the value of `a` does not affect `k`, `k` remains the same. As a result, the program displays:

```

400
1 200 3

```

**C.11.27** Write a function that takes as parameters a character, an integer, and a string and uses it as a pointer to check whether the character exists in the string or not. If not, it should return NULL. Otherwise, if the integer is 0, it should return a pointer to its first occurrence, otherwise to the last one. Write a program that reads a string of less than 100 characters, a character, and an integer, calls the function, and displays the part of the string. For example, if the user enters "bootstrap", 't' and 0, the program should display tstrap. If it is "bootstrap", 't' and 3, the program should display trap.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);
char *find(char str[], char ch, int f);

int main(void)
{
    char *ptr, ch, str[100];
    int flag;

    printf("Enter text: ");
    read_text(str, sizeof(str), 1);

    printf("Enter character to search: ");
    scanf("%c", &ch);

    printf("Enter choice (0-first, other-last): ");
    scanf("%d", &flag);

    ptr = find(str, ch, flag);
    if(ptr == NULL)
        printf("'%c' is not included in the text\n", ch);
    else

```

```

        printf("The rest string is: %s\n", ptr);
    return 0;
}
char *find(char str[], char ch, int f)
{
    char *tmp = NULL; /* Initial value in case the character is not
found. */
    while(*str != '\0')
    {
        if(*str == ch)
        {
            tmp = str;
            if(f == 0) /* If the character is found and the
choice is 0, the loop terminates and the function returns the pointer.
Otherwise, tmp points to the place of the last occurrence. */
                break;
        }
        str++;
    }
    return tmp;
}

```

**C.11.28** What is the output of the following program?

```

#include <stdio.h>

int *test(int p[], int *ptr2);

int main(void)
{
    int *ptr, i = 1, arr[] = {10, 20, 30, 40, 50, 60, 70};

    ptr = test(arr+2, &i);
    printf("%d %d\n", arr[4], *ptr);
    return 0;
}

int *test(int p[], int *ptr2)
{
    p[2] = 200;
    return p+*ptr2;
}

```

**Answer:** When `test()` is called, we have `p = arr+2 = &arr[2]` and `ptr2 = &i`. Therefore, `p[0]` is equal to `arr[2]`, `p[1]` equals `arr[3]`, and `p[2]` equals `arr[4]`. Therefore, with the statement `p[2] = 200`; the value of `arr[4]` becomes 200. Since `ptr2` points to the address of `i`, the expression `p+*ptr2;` is equivalent to `p+i = p+1`. Since `p` points to the third element of the array `arr`, `test()` returns a pointer to its fourth element. Since that pointer is stored into `ptr`, `*ptr` becomes equal to `arr[3]`. As a result, the program displays: 200 40

**C.11.29** Write a function that calculates the maximum common divisor (MCD) of two positive integers, according to the following Euclid's algorithm. Suppose we have the integers `a` and `b`, with `a > b`. If `b` divides `a` precisely, this is the MCD. If the remainder `r` of the division `a/b` is not 0, then we divide `b` with `r`. If the new remainder of the division is 0,

then the MCD is r; otherwise, this procedure is repeated. Write a program that reads two positive integers and uses the function to calculate their MCD.

```
#include <stdio.h>

int mcd(int a, int b);

int main(void)
{
    int num1, num2;
    do
    {
        printf("Enter the first number: ");
        scanf("%d", &num1);
        printf("Enter the second number (equal or less than the
first one): ");
        scanf("%d", &num2);
    } while((num2 > num1) || (num1 <= 0) || (num2 <= 0));
    printf("MCD of %d and %d is %d\n", num1, num2, mcd(num1, num2));
    return 0;
}

int mcd(int a, int b)
{
    int r;

    while(1)

    {
        r = a%b;
        if(r == 0)
            return b;
        else /* According to the algorithm, we divide b by r, so we
change the values of a and b. */
        {
            a = b;
            b = r;
        }
    }
}
```

C.11.30 What is the output of the following program?

```
#include <stdio.h>

void test(int *arg);

int var = 100;

int main(void)
{
    int *ptr, i = 30;

    ptr = &i;
    test(ptr);
    printf("%d\n", *ptr);
    return 0;
}

void test(int *arg)
{
    arg = &var;
}
```

**Answer:** You answered 100, didn't you? *Basket Case* from *Green Day*, play it loud. Let's see the trap.

Since the value of `ptr` and not its address is passed to `test()`, any change in the value of `arg` does not affect `ptr`. Therefore, the program displays 30.

**C.11.31** Write a function that returns the value of the polynomial  $a_0 + a_1x + a_2x^2 + \dots + a_nx^n$  for a given  $x$ . The function prototype is `double poly(double a[], double x, int n);` where the coefficients of the polynomial are stored into `a` and `n` represents its degree. Write a program that reads the degree of a polynomial, its coefficients, and a value (e.g., `x`) and uses the function to calculate the value of the polynomial. Assume that the maximum degree is 100. To calculate the power of the coefficients, use the function `pow()`, as shown in Appendix C.

```
#include <stdio.h>
#include <math.h>

#define MAX_DGR 100

double poly(double a[], double x, int n);

int main(void)
{
    int i, dgr;
    double num, arr[MAX_DGR+1];

    do
    {
        printf("Enter degree (<= %d) : ", MAX_DGR);
        scanf("%d", &dgr);
    } while(dgr < 0 || dgr > MAX_DGR);

    for(i = 0; i <= dgr; i++)
    {
        printf("Enter coefficient arr[%d] : ", i);
        scanf("%lf", &arr[i]);
    }
    printf("Enter number: ");
    scanf("%lf", &num);
    printf("Poly(%f) = %f\n", num, poly(arr, num, dgr));
    return 0;
}

double poly(double a[], double x, int n)
{
    int i;
    double sum;

    sum = 0;
    for(i = 0; i <= n; i++)
        sum += a[i]*pow(x, i);
    return sum;
}
```

C.11.32 What is the output of the following program?

```
#include <stdio.h>

float *test(float *ptr1, float *ptr2);

int main(void)
{
    float a = 1.2, b = 3.4;

    *test(&a, &b) = 5.6;
    printf("val1 = %.1f val2 = %.1f\n", a, b);

    *test(&a, &b) = 7.8;
    printf("val1 = %.1f val2 = %.1f\n", a, b);
    return 0;
}

float *test(float *ptr1, float *ptr2)
{
    if(*ptr1 < *ptr2)
        return ptr1;
    else
        return ptr2;
}
```

**Answer:** test() returns a pointer to the parameter with the smaller value. In the first call, we have ptr1 = &a, so \*ptr1 = a = 1.2. Similarly, we have ptr2 = &b, so \*ptr2 = b = 3.4. Therefore, test() returns the ptr1 pointer. Since test() returns a pointer to a, the statement \*test(&a, &b) = 5.6; makes a equal to 5.6. Therefore, the program displays val1 = 5.6 val2 = 3.4.

The second call of test() returns the ptr2 pointer because \*ptr1 = a = 5.6 and \*ptr2 = b = 3.4. Like before, since test() returns a pointer to b, b becomes 7.8 and the program displays: val1 = 5.6 val2 = 7.8

C.11.33 Write a **void** function that takes as parameters an array that contains the prices of some products in a shop and their number and uses proper variables to return the lowest, the highest, and the average of the prices. Write a program that reads the prices of less than 100 products and stores them in an array. If the user enters -1, the insertion of prices should terminate. The program should use the function to display the lowest, the highest, and the average of the prices.

```
#include <stdio.h>

void stat_arr(float arr[], int size, float *min, float *max, float *avg);

int main(void)
{
    int i;
    float min, max, avg, arr[100];
```

```

for(i = 0; i < 100; i++)
{
    printf("Enter price: ");
    scanf("%f", &arr[i]);
    if(arr[i] == -1)
        break;
}
if(i == 0)
    return 0;
/* The variable i indicates how many prices were stored into the
array. For example, if the user does not enter the value -1, i would be
equal to 100. */
stat_arr(arr, i, &min, &max, &avg);
printf("Max=% .2f Min=% .2f Avg=% .2f\n", max, min, avg);
return 0;
}

void stat_arr(float arr[], int size, float *min, float *max, float *avg)
{
    int i;
    float sum;

    sum = *min = *max = arr[0];
    for(i = 1; i < size; i++)
    {
        if(arr[i] > *max)
            *max = arr[i];
        if(arr[i] < *min)
            *min = arr[i];
        sum += arr[i];
    }
    *avg = sum/size;
}

```

C.11.34 What is the output of the following program?

```

#include <stdio.h>

int *test(int *ptr1, int *ptr2);

int main(void)
{
    int arr[] = {1, 2, 3, 4};

    *test(arr, arr+3) = 30;
    printf("%d %d %d %d\n", arr[0], arr[1], arr[2], arr[3]);
    return 0;
}

int *test(int *ptr1, int *ptr2)
{
    *(ptr1+1) = 10;
    *(ptr2-1) = 20;
}

```

```
    return ptr1+3;
}
```

**Answer:** When test() is called, we have `ptr1 = arr`, so the pointer `ptr1+1` points to `arr[1]`. Therefore, the statement `*(ptr1+1) = 10;` makes `arr[1]` equal to 10. Similarly, we have `ptr2 = arr+3`, so `ptr2` points to `arr[3]`. Therefore, the statement `*(ptr2-1) = 20;` makes `arr[2]` equal to 20. Since `ptr1` points to `arr`, the expression `ptr1+3` returns a pointer to `arr[3]`. Therefore, `arr[3]` becomes 30. As a result, the program displays: 1 10 20 30

**C.11.35** Write a function that takes as parameter an integer (e.g., N) and calculates the  $N^{\text{th}}$  term of the Fibonacci sequence, according to the formula  $F(N) = F(N-1) + F(N-2)$ , where  $F(0) = 0$  and  $F(1) = 1$ . Write a program that reads an integer N between 2 and 40 and uses the function to display the  $N^{\text{th}}$  term.

```
#include <stdio.h>

unsigned int fib(int num);

int main(void)
{
    int num;

    do
    {
        printf("Enter a number between 2 and 40: ");
        scanf("%d", &num);
    } while(num < 2 || num > 40);

    printf("F(%d) = %u\n", num, fib(num));
    return 0;
}

unsigned int fib(int num)
{
    unsigned int term1, term2, sum;

    term1 = 1;
    term2 = 0;
    while(num > 1)
    {
        sum = term1 + term2;

        term2 = term1;
        term1 = sum;

        num--;
    }
    return sum;
}
```

**Comments:** Given that  $F(N) = F(N-1) + F(N-2)$ , the first terms are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, .... For example,  $F(7) = 13$ , which is the sum of  $F(6)$  and  $F(5)$ . To calculate the value of a term, we store into `term1` the last sum, while in `term2` the previous one.

C.11.36 What is the output of the following program?

```
#include <stdio.h>

double *f(double ptr[]);

int main(void)
{
    int i;
    double a[8] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8};

    printf("%.1f\n", *f(f(f(a))));
    for(i = 0; i < 8; i++)
        printf("a[%d] = %.1f\n", i, a[i]);
    return 0;
}

double *f(double ptr[])
{
    (*ptr)++;
    return ptr+2;
}
```

**Answer:** The `f()` calls are executed inside out. When `f()` is first called, we have `ptr = a = &a[0]`. Therefore, the statement `(*ptr)++;` is equivalent to `a[0]++`; and `a[0]` becomes 1.1.

The return value `a+2` is used as an argument in the second call, so `f(f(a))` is translated to `f(a+2)`. Therefore, `a[2]` becomes 1.3. Similarly, the return value `a+4` is used as an argument in the third call and `a[4]` becomes 1.5. Since the last call to `f()` returns the address of `a[6]`, the `*` operator is used to display its value, that is, 0.7. As a result, the content of `a` is 1.1 0.2 1.3 0.4 1.5 0.6 0.7 0.8

C.11.37 Write a function that takes as parameter a string and displays how many digits (0-9) it contains, how many words begin with a lowercase letter (a-z), and how many with an uppercase (A-Z). Write a program that reads a string of less than 100 characters and calls the function.

There are two restrictions. The first one is to use pointer notation inside the function, and the second is to use functions from `ctype.h`. Read the descriptions of `ctype.h` functions in Appendix C and select the proper ones.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

int read_text(char str[], int size, int flag);
void parse_str(char str[]);
```

```

int main(void)
{
    char str[100];

    printf("Enter text: ");
    read_text(str, sizeof(str), 1);
    parse_str(str);
    return 0;
}

void parse_str(char s[])
{
    int dig, uplet_wrd, lowlet_wrd;

    dig = uplet_wrd = lowlet_wrd = 0;
    while(*s != '\0')
    {
        while(isisspace(*s)) /* Skip the whitespace characters until a
word begins. */
            s++;
        if(*s == '\0')
            break;
        /* We check if the word begins with a digit or letter.*/
        if(isdigit(*s))
            dig++;
        else if(isupper(*s))
            uplet_wrd++;
        else if(islower(*s))
            lowlet_wrd++;
        s++;
        if(*s == '\0')
            break;
        while(isisspace(*s) == 0) /* Skip the characters of the word;
check only for digits. */
        {
            if(isdigit(*s))
                dig++;
            s++;
            if(*s == '\0')
                break;
        }
    }
    printf("UL:%d LL:%d DG:%d\n", uplet_wrd, lowlet_wrd, dig);
}

```

**C.11.38** In our experience, we've often used the C language in the field of "Implementing Communication Protocols." For example, we are going to show you a method to create network data frames. Before that, here is some theory:

Two computers, A and B, that reside in the same Internet Protocol (IP) local network may communicate if both know the physical address of each other. The physical address is called Medium Access Control (MAC). For example, when A needs to communicate with B, it must know the MAC address of B. In order to learn it, A should transmit a MAC frame

to the local network, which contains a special message, called ARP\_Request. When B gets this MAC frame, it replies to A with another MAC frame, which contains another special message, called ARP\_Reply. The ARP\_Reply contains the MAC address of B. In this way, A learns the MAC address of B, so they can communicate.

Write a program that reads the MAC address of A and the IP addresses of A and B and creates a MAC frame that contains the ARP\_Request message. The program should display the content of the MAC frame in lines, where each line should contain 16 bytes in hex format. The MAC address consists of six octets (bytes) and it should be entered in the x.x.x.x.x.x form (each x is an integer in [0, 255]), while the IP address consists of four octets and it should be entered in the x.x.x.x form. Figure 11.2 shows the format of the MAC frame:

- a. Set the first seven octets of the Preamble field equal to 85 and the eighth octet equal to 171.
- b. Set the six octets of the MAC destination address equal to 255.
- c. Set the six octets of the MAC source address equal to the MAC address of A.
- d. Set the first octet of the Type field equal to 8 and the second one equal to 6.
- e. Set the four octets of the CRC field equal to 1.

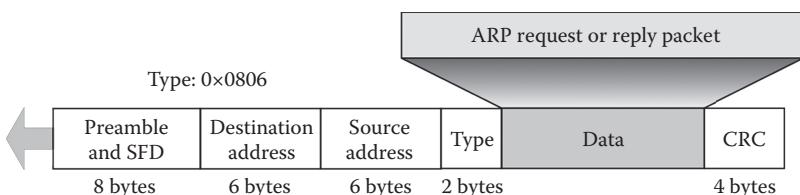
Figure 11.3 depicts the format of the ARP\_Request message.

- a. The length of the Hardware Type field is two octets. The first octet is 0 and the second one is 1.
- b. The length of the Protocol Type field is two octets. The first octet is 8 and the second one is 6.
- c. The length of the Hardware Length field is one octet with value 6.
- d. The length of the Protocol Length field is one octet with value 4.
- e. The length of the Operation field is two octets.
- f. Set the six octets of the Target hardware address equal to 0.
- g. Set the four octets of the Target protocol address equal to the IP address of B.

```
#include <stdio.h>

void Build_Frm(int MAC_src[], int IP_src[], int IP_dst[]);
void Show_Frm(unsigned char pkt[], int len);
```

```
int main(void)
{
    int MAC_src[6], IP_src[4], IP_dst[4];
```



**FIGURE 11.2**

MAC frame structure.

Hardware type		Protocol type
Hardware length	Protocol length	Operation request 1, reply 2
Sender hardware address (for example: 6 bytes for ethernet)		
Sender protocol address (for example: 4 bytes for IP)		
Target hardware address (for example: 6 bytes for ethernet) (it is not filled in a request)		
Target protocol address (for example: 4 bytes for IP)		

**FIGURE 11.3**

ARP packet structure.

```

printf("Enter src MAC (x.x.x.x.x.x): ");
scanf("%d.%d.%d.%d.%d", &MAC_src[0], &MAC_src[1], &MAC_src[2],
&MAC_src[3], &MAC_src[4], &MAC_src[5]);

printf("Enter src IP (x.x.x.x): ");
scanf("%d.%d.%d.%d", &IP_src[0], &IP_src[1], &IP_src[2],
&IP_src[3]);

printf("Enter dst IP (x.x.x.x): ");
scanf("%d.%d.%d.%d", &IP_dst[0], &IP_dst[1], &IP_dst[2],
&IP_dst[3]);
Build_Frm(MAC_src, IP_src, IP_dst);
return 0;
}

void Build_Frm(int MAC_src[], int IP_src[], int IP_dst[])
{
    unsigned char pkt[54] = {85, 85, 85, 85, 85, 85, 85, 171}; /*
Initialize the first eight octets of the frame and make zero the rest. */
    int i, j;

    for(i = 8; i < 14; i++)
        pkt[i] = 255; /* Broadcast MAC address. It is a special
address used in Ethernet technology to broadcast the frame. */
    for(i = 14, j = 0; i < 20; i++, j++)
        pkt[i] = MAC_src[j]; /* MAC source. */

    pkt[20] = 8; /* Type. */
    pkt[21] = 6;

    pkt[22] = 0; /* Hardware Type. */
    pkt[23] = 1;

    pkt[24] = 8; /* Protocol Type. */
    pkt[25] = 6;
}

```

```

pkt[26] = 6; /* Hardware Length. */
pkt[27] = 4; /* Protocol Length. */

pkt[28] = 0; /* Operation (ARP_Request). */
pkt[29] = 1;

for(i = 30, j = 0; i < 36; i++, j++)
    pkt[i] = MAC_src[j]; /* MAC source. */
for(i = 36, j = 0; i < 40; i++, j++)
    pkt[i] = IP_src[j]; /* IP source. */
/* The MAC destination in places [40-45] is initialized to 0. */
for(i = 46, j = 0; i < 50; i++, j++)
    pkt[i] = IP_dst[j]; /* IP destination. */
for(i = 50; i < 54; i++)
    pkt[i] = 1; /* CRC. */

Show_Frm(pkt, i);
}

void Show_Frm(unsigned char pkt[], int len)
{
    int i;
    for(i = 0; i < len; i++)
    {
        if(i%16 == 0)
            printf("\n");
        printf("%02X ", pkt[i]);
    }
}

```

**Comments:** In a real network application, the network card of the system transmits this MAC frame to the IP network.

C.11.39 What is the output of the following program?

```

#include <stdio.h>

void swap(char *s1, char *s2);

int main(void)
{
    char *p[] = {"Shadow", "Play"};

    swap(p[0], p[1]);
    printf("%s %s\n", p[0], p[1]);
    return 0;
}

void swap(char *s1, char *s2)
{
    char *p;

    p = s1;

```

```
s1 = s2;  
s2 = p;  
}
```

**Answer:** What kind of play is that? Did you answer Play Shadow? Read the comments of C.11.30. It is only the song to change: *Shadow Play* from *Rory Gallagher*.

As in `test()` of C.11.30, `swap()` takes as arguments the values of the pointers, not their addresses. Therefore, the swapping has no effect and the program displays *Shadow Play*. What we want you to do is to modify the program in order to output *Play Shadow*. Let's see, can you do that? To give you a hint, change the type of the parameters to... find it, no more to say.

## Two-Dimensional Array as Argument

The most common way to declare a function that takes as parameter a two-dimensional array is to write the name of the array followed by its dimensions. For example, `test()` takes as parameter a two-dimensional integer array with 5 rows and 10 columns.

```
void test(int arr[5][10]);
```

Because, as discussed in Chapter 7, it is not necessary for the compiler to know the number of rows to calculate the memory location of an element, the first dimension may be omitted. For example:

```
void test(int arr[][10]);
```

However, the number of columns must be specified. In the general case, when a multidimensional array is passed to a function, the first dimension may be omitted. All the others must be specified and match the declaration of the array. In this way, the compiler would have the necessary information to make the correct scaling for address arithmetic and access an element.

Since C treats a two-dimensional array as an array of one-dimensional arrays, the compiler actually translates the two-dimensional parameter to a pointer to an array. Therefore, we could equivalently write `void test(int (*arr)[10]);`. Because [] have greater precedence than \*, parentheses are necessary; otherwise, `arr` would be interpreted as an array of 10 pointers to integers instead of pointer to an array of 10 integers. Although that declaration is more accurate, our preference is to use the first one and omit the first dimension; we prefer to show clearly that the parameter is a two-dimensional array.

Play some defense, a burst of questions is coming. Where does `arr` point to? It points to the first row of the array. And if we write `arr++` where does it point to? In the next one, since it is increased by the size of the row. And what is the `(*arr)[1]`? It is the second element of the row that `arr` points to. And how do we know that the last row of the array is reached? Typically, we either use a constant (e.g., `#define ROWS 20`) or pass an argument to declare the total number of rows. End of alert. If you've queries, read again the "Pointers and Two-Dimensional Arrays" section in Chapter 8.

We write it once more, the compiler translates the two-dimensional parameter to a pointer to an array, not to a pointer to pointer as you might have expected. Therefore, you can't write `void test(int **arr);`. When we present the `malloc()` function in Chapter 14, you'll see how to use a pointer to a pointer variable to create dynamically a two-dimensional array and pass it to a function. Now, let's summarize how the compiler translates array arguments:

*One-dimensional:* `arr[10]` is translated to pointer `*arr`.

*Two-Dimensional:* `arr[10][20]` is translated to pointer to array `(*arr)[20]`.

*Array of pointers:* `*arr[10]` is translated to pointer to pointer `**arr`.

As with one-dimensional arrays, to pass a two-dimensional array, as an argument, we write only its name, without brackets. If we want to pass a row (e.g., `i`), we pass the pointer to that row (e.g., `arr[i]`), as we do when passing a one-dimensional array. For example, the following program displays the elements of the first row.

```
#include <stdio.h>

#define ROWS 2
#define COLS 3

void show_row(int a[]);

int main(void)
{
    int arr[ROWS][COLS] = {{1, 2, 3}, {4, 5, 6}};
    show_row(arr[0]);
    return 0;
}

void show_row(int a[])
{
    int i;
    for(i = 0; i < COLS; i++)
        printf("%d ", a[i]);
}
```

---

## Exercises

C.11.40 Write a function that takes as parameter a  $3 \times 4$  two-dimensional array and returns one array where each element is equal to the sum of the elements of the respective row of the two-dimensional array and another array where each element is equal to the sum of the elements of the respective column of the two-dimensional array. Write a program that reads 12 integers, stores them in a  $3 \times 4$  two-dimensional array, and uses the function to display the sum of the elements in each row and the sum of the elements in each column as well.

```

#include <stdio.h>

#define ROWS 3
#define COLS 4

void find_sums(int arr1[] [COLS], int arr2[], int arr3[]);

int main(void)
{
    int i, j, arr1[ROWS] [COLS], arr2[ROWS], arr3 [COLS];

    for(i = 0; i < ROWS; i++)
        for(j = 0; j < COLS; j++)
    {
        printf("arr1[%d] [%d] = ", i, j);
        scanf("%d", &arr1[i][j]);
    }

    find_sums(arr1, arr2, arr3);
    for(i = 0; i < ROWS; i++)
        printf("sum_line_%d = %d\n", i, arr2[i]);
    for(i = 0; i < COLS; i++)
        printf("sum_col_%d = %d\n", i, arr3[i]);
    return 0;
}

void find_sums(int arr1[] [COLS], int arr2[], int arr3[])
{
    int i, j, sum;

    for(i = 0; i < ROWS; i++)
    {
        sum = 0;
        for(j = 0; j < COLS; j++)
            sum += arr1[i][j];
        arr2[i] = sum;
    }
    for(i = 0; i < COLS; i++)
    {
        sum = 0;
        for(j = 0; j < ROWS; j++)
            sum += arr1[j][i];
        arr3[i] = sum;
    }
}

```

**Comments:** Since a function cannot return an array, the arrays are declared in `main()` and passed to the function.

**C.11.41** Write a function that takes as parameters an array of names and another name. The function should check if that name is contained in the array. If it is, the function should return a pointer to the position of that name in the array, otherwise NULL. Write a program that reads the names of 20 students (less than 100 characters each) and stores them in an array. Then, it reads another name and uses the function to check if that name is contained in the array.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM 20
#define SIZE 100

char *find_name(char name[] [SIZE], char str[]);
int read_text(char str[], int size, int flag);

int main(void)
{
    char *ptr, str[SIZE], name[NUM] [SIZE]; /* The name array holds the
names of the students. */
    int i;

    for(i = 0; i < NUM; i++)
    {
        printf("Enter name: ");
        read_text(name[i], sizeof(name[i]), 1); /* We use the
name[i] as a pointer to the respective i row of SIZE characters. */
    }
    printf("Enter name to search: ");
    read_text(str, sizeof(str), 1);

    ptr = find_name(name, str);
    if(ptr == NULL)
        printf("%s is not contained\n", str);
    else
        printf("%s is contained\n", ptr);
    return 0;
}

char *find_name(char name[] [SIZE], char str[])
{
    int i;
    for(i = 0; i < NUM; i++)
        if(strcmp(name[i], str) == 0)
            return name[i]; /* name[i] points to the first
character of the row that contains the name. */
    return NULL; /* If this point is reached, the name is not found in
the array. */
}

```

**C.11.42** A popular card game among children is a memory matching game. The game starts with a deck of identical pairs of cards face down on a table. The player selects two cards and turns them over. If they match, they remain face up. If not, they are flipped face down. The game ends when all cards are face up.

To simulate that game, write a program that uses the elements of a two-dimensional array as the cards. To test your program, use a 4x4 array and assign the values 1-8 to its elements (cards). Each number must appear twice. Set the values in random positions. An example of the array might be the following:

$$\begin{bmatrix} 5 & 3 & 4 & 8 \\ 4 & 2 & 6 & 1 \\ 3 & 8 & 7 & 6 \\ 2 & 5 & 1 & 7 \end{bmatrix}$$

The program should prompt the user to select the positions of two cards and display a message to indicate if they match or not. The program ends when all cards are matched.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS 4
#define COLS 4

void show_board(int c[] [COLS], int s[] [COLS]);
void sel_card(int c[] [COLS], int s[] [COLS], int *row, int *col);

int main(void)
{
    int i, j, m, r, c, r2, c2, cnt, cards[ROWS] [COLS], status[ROWS]
[COLS] = {0}; /* The status array indicates if a card faces up or down (0
is for down). */
    cnt = 0; /* This variable counts the number of the faced up cards.
*/
    for(i = r = 0; i < ROWS; i++) /* Assign the values 1 to 8,
sequentially. */
    {
        for(j = 0; j < COLS; j+=2)
        {
            cards[i] [j] = cards[i] [j+1] = r+1;
            r++;
        }
    }
    /* Now, shuffle the cards. */
    srand(time(NULL));
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
        {
            c = cards[i] [j];
            m = rand()%ROWS;
            r = rand()%COLS;
            cards[i] [j] = cards[m] [r];
            cards[m] [r] = c;
        }
    }
    show_board(cards, status);
    m = 0;
    while(cnt != ROWS*COLS) /* The game ends when all cards are faced
up. */
    {
```

```

    sel_card(cards, status, &r, &c);
    printf("Card_1 = %d\n", cards[r][c]);

    sel_card(cards, status, &r2, &c2);
    printf("Card_2 = %d\n", cards[r2][c2]);

    for(i = 0; i < 18; i++) /* Blank line to delete history and
make it harder for the player to remember the card positions. */
        printf("\n");

    if(cards[r][c] == cards[r2][c2])
    {
        printf("Cards matched !!!\n");
        cnt += 2;
    }
    else
    {
        printf("Sorry. No match !!!\n");
        status[r][c] = status[r2][c2] = 0; /* Reset the cards
to face down condition. */
    }
    m++;
    show_board(cards, status);
}
printf("Congrats: You did it in %d tries\n",
return 0;
}

void show_board(int c[][] [COLS], int s[][] [COLS])
{
    int i, j;

    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
        {
            if(s[i][j] == 1)
                printf("%d ", c[i][j]);
            else
                printf("* ");
        }
        printf("\n");
    }
}

void sel_card(int c[][] [COLS], int s[][] [COLS], int *row, int *col)
{
    while(1)
    {
        printf("Enter row and column: ");
        scanf("%d %d", row, col);
        (*row)--; /* Subtract 1, because the indexing of the array
starts from 0. */
        (*col)--;
    }
}

```

```

    if(*row >= ROWS || *row < 0 || *col >= COLS || *col < 0)
    {
        printf("Out of bound dimensions\n");
        continue;
    }
    if(s[*row][*col] == 1)
    {
        printf("Error: This card is already flipped\n");
        continue;
    }
    s[*row][*col] = 1; /* Change the card position to face up.
*/
    return;
}
}

```

---

## Passing Data in main() Function

When we run a program from the command line, we'll often need to pass data to it. For example, suppose that the executable file hello.exe is stored in C disk. The command line:

```
C:\>hello 100 200
```

executes the program hello and passes the values 100 and 200 to main(). To get the data, main() must be defined like this:

```
int main(int argc, char *argv[])
```

Although you can use any names, argc (*argument count*) and argv (*argument vector*) are, by convention, the typical choices. And because the compiler translates an array parameter to pointer, many prefer to write `**argv` instead of `*argv[]`.

The value of argc is equal to the number of the command line arguments, including the name of the program itself. For example, in the previous command line, argc is 3. The arguments should be separated with space(s), to distinguish the values.

The argv parameter is an array of pointers to the command line arguments, which are stored in string form. argv[0] pointer points to the name of the program, while the other pointers up to argv[argc-1] point to the remaining arguments. The last argv element is the argv[argc], which is always NULL. In our example, the arguments hello, 100, and 200 are passed to main() as strings. In particular, argv[0] points to the string "hello", argv[1] points to "100", argv[2] points to "200", and argv[3] is NULL.

For example, the following program checks if the user entered the correct user name and password. Suppose that these are user and pswd, respectively.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{

```

```

if(argc == 1)
    printf("Error: missing user name and password\n");
else if(argc == 2)
    printf("Error: missing password\n");
else if(argc == 3)
{
    if(strcmp(argv[1], "user") == 0 &&
strcmp(argv[2], "pswd") == 0)
        printf("Valid user. The program \"%s\" will be
executed ...\n", argv[0]);
    else
        printf("Wrong input\n");
}
else
    printf("Error: too many parameters\n");
return 0;
}

```

The **if** statements check the value of argc. If it is 3, the program checks if the entered user name and password are correct. If not, the program displays a related message.

---

## Exercises

C.11.43 What is the output of the following program?

```

#include <stdio.h>
int main(int argc, char *argv[])
{
    while(--argc)
        printf((argc > 1) ? "%s " : "%s\n", ++argv);
    return 0;
}

```

**Answer:** argc is equal to the number of the command line arguments. If the only argument is the name of the program, argc would be 1 and the loop won't be executed because its value becomes 0. If there are more arguments, the program displays their values, separated by a space, and it adds a new line character after printing the last one.

For example, suppose that the command line arguments are one and two. Therefore, argc would be 3 and the loop condition makes it 2. Since (argc > 1) is true, the "%s " will be replaced by ++argv and a space. Since argv points to argv[0], the expression ++argv makes it point to argv[1]. Therefore, ++argv is equivalent to argv[1], and the program outputs one. In the next iteration, argc becomes 1, so the "%s\n" will be replaced by argv[2]. As a result, the program outputs two and a new line character.

Now, answer this one. What would be the output if we write \*argv++ instead of ++argv?

**Answer:** The program would display the name of the program and the command line arguments, but the last one.

C.11.44 The popular ping command is used to test the communication between two systems in an IP network, for example, ping www.ntua.gr. Write a program that reads the command line argument and checks if it is a valid hostname. To be valid, assume that it should have the form of the example; that is, begin with "www.", and the part after the second dot should be two or three characters long (e.g., gr).

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    int i, len;

    if(argc != 2)
    {
        printf("Wrong number of arguments\n");
        return 0;
    }
    if(strncmp(argv[1], "www.", 4) != 0)
    {
        printf("Name must begin with www.\n");
        return 0;
    }
    len = strlen(argv[1]);
    for(i = 4; i < len; i++)
        if(argv[1][i] == '.')
            break;

    if(i == len)
    {
        printf("Second . is missing\n");
        return 0;
    }
    if((len-i-1) != 2 && (len-i-1) != 3)
    {
        printf("The last part should be two or three characters
long\n");
        return 0;
    }
    printf("The hostname %s is valid\n", argv[1]);
    return 0;
}
```

---

## Functions with Variable Number of Parameters

A function may accept a variable number of parameters. To declare such a function, first, we put the fixed parameters, that is, the parameters that must always be present in the function call, and then the ... symbol, known as *ellipsis*. For example, the declaration:

```
void test(int num, char *str, ...);
```

indicates that `test()` takes two fixed parameters, an integer and a pointer to a character, which may be followed by a variable number of additional parameters. In practice, you'll rarely need to write functions with a variable number of parameters. It might happen if you want to write a generic function, for example, a function that accepts any number of values and returns their sum. Another example is `printf()` and `scanf()`; both accept a variable number of parameters.

---

A function with a variable number of parameters must have one named fixed parameter, at least.

---

To call such a function, first we write the values of the fixed arguments and then the values of the optional arguments. For example, we could call `test()` like this:

```
test(3, "example", 5, 8.9, "sample");
```

The data types of the optional arguments are `int`, `float`, and `char*` with values 5, 8.9, and "sample", respectively.

To handle the optional argument list, we use a set of macros defined in the standard file `stdarg.h`. Macros are discussed in Chapter 16. For now, assume that their behavior is similar to functions.

`va_list`: A variable of type `va_list` is used to point to the optional argument list.

`va_start()`: The `va_start()` macro takes two parameters. The first one is of type `va_list`, and the second one should be the name of the last fixed parameter. For example, the name of the last fixed parameter in `test()` is `str`. `va_start()` sets the `va_list` argument to point to the first optional argument. `va_start()` must be called before `va_arg()` is used for first time.

`va_arg()`: The `va_arg()` macro takes two parameters. The first one is of type `va_list` and the second one is the type of an optional argument. For example, the type of the first optional argument in `test()` is `int`. The `va_list` argument must be the one initialized with `va_start()`. `va_arg()` returns the value of the optional argument and advances the `va_list` argument to point to the next optional argument. In order to get the values of all optional arguments, we should call `va_arg()` as many times as the number of the optional arguments.

`va_end()`: The `va_end()` macro takes one parameter of type `va_list`. It must be called to end the processing of the optional argument list.

In the following program, `test()` takes a variable number of parameters of type `char*`. The fixed parameter `num` indicates their number.

```
#include <stdio.h>
#include <stdarg.h>

void test(int num, ...);

int main(void)
```

```

}

    test(3, "text_1", "text_2", "text_3");
    return 0;
}

void test(int num, ...)
{
    char *str;
    int i;
    va_list arg_ptr;

    va_start(arg_ptr, num); /* arg_ptr points to the first optional
argument. Notice that the name of the second argument should be the same
with the name of the last fixed parameter in the definition of test(). */

    for(i = 0; i < num; i++)
    {
        str = va_arg(arg_ptr, char*); /* Each call of va_arg()
returns the value of the respective optional argument of type char* and
arg_ptr advances to point to the next optional argument. The second
argument is char*, because it is said that the type of all optional
arguments is char*. */
        printf("%s ", str);
    }
    va_end(arg_ptr);
}

```

The program outputs the values of the optional arguments: `text_1 text_2 text_3`

The primary difficulty in using a function with a variable argument list is that there is no easy way to determine the number and types of its optional arguments. To find the number, a simple solution is to add a fixed parameter, which declares that number. In the previous example, we used the parameter `num`. If the arguments are all of the same type, we could use a special value to mark the end of the list. Moreover, the function must somehow be informed about their types. For example, `printf()` and `scanf()` use the format string to get the necessary type information to process the argument list.

## Exercise

**C.11.45** Write a function that takes a variable parameter list of type `double` and returns the maximum value. Write a program that reads three floats and uses the function to display the maximum value.

```

#include <stdio.h>
#include <stdarg.h>

double find_max(int num, ...);

int main(void)
{
    va_list arg_ptr;
    double max = -1.0;
    double current;
    int i;
    char str[10];
    char *ptr;
    int len;
    int sign;
    double d;
    double sum = 0.0;
    double avg;
    double min = 1.0;
    double max2;
    double max3;
    double max4;
    double max5;
    double max6;
    double max7;
    double max8;
    double max9;
    double max10;
    double max11;
    double max12;
    double max13;
    double max14;
    double max15;
    double max16;
    double max17;
    double max18;
    double max19;
    double max20;
    double max21;
    double max22;
    double max23;
    double max24;
    double max25;
    double max26;
    double max27;
    double max28;
    double max29;
    double max30;
    double max31;
    double max32;
    double max33;
    double max34;
    double max35;
    double max36;
    double max37;
    double max38;
    double max39;
    double max40;
    double max41;
    double max42;
    double max43;
    double max44;
    double max45;
    double max46;
    double max47;
    double max48;
    double max49;
    double max50;
    double max51;
    double max52;
    double max53;
    double max54;
    double max55;
    double max56;
    double max57;
    double max58;
    double max59;
    double max60;
    double max61;
    double max62;
    double max63;
    double max64;
    double max65;
    double max66;
    double max67;
    double max68;
    double max69;
    double max70;
    double max71;
    double max72;
    double max73;
    double max74;
    double max75;
    double max76;
    double max77;
    double max78;
    double max79;
    double max80;
    double max81;
    double max82;
    double max83;
    double max84;
    double max85;
    double max86;
    double max87;
    double max88;
    double max89;
    double max90;
    double max91;
    double max92;
    double max93;
    double max94;
    double max95;
    double max96;
    double max97;
    double max98;
    double max99;
    double max100;
    double max101;
    double max102;
    double max103;
    double max104;
    double max105;
    double max106;
    double max107;
    double max108;
    double max109;
    double max110;
    double max111;
    double max112;
    double max113;
    double max114;
    double max115;
    double max116;
    double max117;
    double max118;
    double max119;
    double max120;
    double max121;
    double max122;
    double max123;
    double max124;
    double max125;
    double max126;
    double max127;
    double max128;
    double max129;
    double max130;
    double max131;
    double max132;
    double max133;
    double max134;
    double max135;
    double max136;
    double max137;
    double max138;
    double max139;
    double max140;
    double max141;
    double max142;
    double max143;
    double max144;
    double max145;
    double max146;
    double max147;
    double max148;
    double max149;
    double max150;
    double max151;
    double max152;
    double max153;
    double max154;
    double max155;
    double max156;
    double max157;
    double max158;
    double max159;
    double max160;
    double max161;
    double max162;
    double max163;
    double max164;
    double max165;
    double max166;
    double max167;
    double max168;
    double max169;
    double max170;
    double max171;
    double max172;
    double max173;
    double max174;
    double max175;
    double max176;
    double max177;
    double max178;
    double max179;
    double max180;
    double max181;
    double max182;
    double max183;
    double max184;
    double max185;
    double max186;
    double max187;
    double max188;
    double max189;
    double max190;
    double max191;
    double max192;
    double max193;
    double max194;
    double max195;
    double max196;
    double max197;
    double max198;
    double max199;
    double max200;
    double max201;
    double max202;
    double max203;
    double max204;
    double max205;
    double max206;
    double max207;
    double max208;
    double max209;
    double max210;
    double max211;
    double max212;
    double max213;
    double max214;
    double max215;
    double max216;
    double max217;
    double max218;
    double max219;
    double max220;
    double max221;
    double max222;
    double max223;
    double max224;
    double max225;
    double max226;
    double max227;
    double max228;
    double max229;
    double max230;
    double max231;
    double max232;
    double max233;
    double max234;
    double max235;
    double max236;
    double max237;
    double max238;
    double max239;
    double max240;
    double max241;
    double max242;
    double max243;
    double max244;
    double max245;
    double max246;
    double max247;
    double max248;
    double max249;
    double max250;
    double max251;
    double max252;
    double max253;
    double max254;
    double max255;
    double max256;
    double max257;
    double max258;
    double max259;
    double max260;
    double max261;
    double max262;
    double max263;
    double max264;
    double max265;
    double max266;
    double max267;
    double max268;
    double max269;
    double max270;
    double max271;
    double max272;
    double max273;
    double max274;
    double max275;
    double max276;
    double max277;
    double max278;
    double max279;
    double max280;
    double max281;
    double max282;
    double max283;
    double max284;
    double max285;
    double max286;
    double max287;
    double max288;
    double max289;
    double max290;
    double max291;
    double max292;
    double max293;
    double max294;
    double max295;
    double max296;
    double max297;
    double max298;
    double max299;
    double max300;
    double max301;
    double max302;
    double max303;
    double max304;
    double max305;
    double max306;
    double max307;
    double max308;
    double max309;
    double max310;
    double max311;
    double max312;
    double max313;
    double max314;
    double max315;
    double max316;
    double max317;
    double max318;
    double max319;
    double max320;
    double max321;
    double max322;
    double max323;
    double max324;
    double max325;
    double max326;
    double max327;
    double max328;
    double max329;
    double max330;
    double max331;
    double max332;
    double max333;
    double max334;
    double max335;
    double max336;
    double max337;
    double max338;
    double max339;
    double max340;
    double max341;
    double max342;
    double max343;
    double max344;
    double max345;
    double max346;
    double max347;
    double max348;
    double max349;
    double max350;
    double max351;
    double max352;
    double max353;
    double max354;
    double max355;
    double max356;
    double max357;
    double max358;
    double max359;
    double max360;
    double max361;
    double max362;
    double max363;
    double max364;
    double max365;
    double max366;
    double max367;
    double max368;
    double max369;
    double max370;
    double max371;
    double max372;
    double max373;
    double max374;
    double max375;
    double max376;
    double max377;
    double max378;
    double max379;
    double max380;
    double max381;
    double max382;
    double max383;
    double max384;
    double max385;
    double max386;
    double max387;
    double max388;
    double max389;
    double max390;
    double max391;
    double max392;
    double max393;
    double max394;
    double max395;
    double max396;
    double max397;
    double max398;
    double max399;
    double max400;
    double max401;
    double max402;
    double max403;
    double max404;
    double max405;
    double max406;
    double max407;
    double max408;
    double max409;
    double max410;
    double max411;
    double max412;
    double max413;
    double max414;
    double max415;
    double max416;
    double max417;
    double max418;
    double max419;
    double max420;
    double max421;
    double max422;
    double max423;
    double max424;
    double max425;
    double max426;
    double max427;
    double max428;
    double max429;
    double max430;
    double max431;
    double max432;
    double max433;
    double max434;
    double max435;
    double max436;
    double max437;
    double max438;
    double max439;
    double max440;
    double max441;
    double max442;
    double max443;
    double max444;
    double max445;
    double max446;
    double max447;
    double max448;
    double max449;
    double max450;
    double max451;
    double max452;
    double max453;
    double max454;
    double max455;
    double max456;
    double max457;
    double max458;
    double max459;
    double max460;
    double max461;
    double max462;
    double max463;
    double max464;
    double max465;
    double max466;
    double max467;
    double max468;
    double max469;
    double max470;
    double max471;
    double max472;
    double max473;
    double max474;
    double max475;
    double max476;
    double max477;
    double max478;
    double max479;
    double max480;
    double max481;
    double max482;
    double max483;
    double max484;
    double max485;
    double max486;
    double max487;
    double max488;
    double max489;
    double max490;
    double max491;
    double max492;
    double max493;
    double max494;
    double max495;
    double max496;
    double max497;
    double max498;
    double max499;
    double max500;
    double max501;
    double max502;
    double max503;
    double max504;
    double max505;
    double max506;
    double max507;
    double max508;
    double max509;
    double max510;
    double max511;
    double max512;
    double max513;
    double max514;
    double max515;
    double max516;
    double max517;
    double max518;
    double max519;
    double max520;
    double max521;
    double max522;
    double max523;
    double max524;
    double max525;
    double max526;
    double max527;
    double max528;
    double max529;
    double max530;
    double max531;
    double max532;
    double max533;
    double max534;
    double max535;
    double max536;
    double max537;
    double max538;
    double max539;
    double max540;
    double max541;
    double max542;
    double max543;
    double max544;
    double max545;
    double max546;
    double max547;
    double max548;
    double max549;
    double max550;
    double max551;
    double max552;
    double max553;
    double max554;
    double max555;
    double max556;
    double max557;
    double max558;
    double max559;
    double max560;
    double max561;
    double max562;
    double max563;
    double max564;
    double max565;
    double max566;
    double max567;
    double max568;
    double max569;
    double max570;
    double max571;
    double max572;
    double max573;
    double max574;
    double max575;
    double max576;
    double max577;
    double max578;
    double max579;
    double max580;
    double max581;
    double max582;
    double max583;
    double max584;
    double max585;
    double max586;
    double max587;
    double max588;
    double max589;
    double max590;
    double max591;
    double max592;
    double max593;
    double max594;
    double max595;
    double max596;
    double max597;
    double max598;
    double max599;
    double max600;
    double max601;
    double max602;
    double max603;
    double max604;
    double max605;
    double max606;
    double max607;
    double max608;
    double max609;
    double max610;
    double max611;
    double max612;
    double max613;
    double max614;
    double max615;
    double max616;
    double max617;
    double max618;
    double max619;
    double max620;
    double max621;
    double max622;
    double max623;
    double max624;
    double max625;
    double max626;
    double max627;
    double max628;
    double max629;
    double max630;
    double max631;
    double max632;
    double max633;
    double max634;
    double max635;
    double max636;
    double max637;
    double max638;
    double max639;
    double max640;
    double max641;
    double max642;
    double max643;
    double max644;
    double max645;
    double max646;
    double max647;
    double max648;
    double max649;
    double max650;
    double max651;
    double max652;
    double max653;
    double max654;
    double max655;
    double max656;
    double max657;
    double max658;
    double max659;
    double max660;
    double max661;
    double max662;
    double max663;
    double max664;
    double max665;
    double max666;
    double max667;
    double max668;
    double max669;
    double max670;
    double max671;
    double max672;
    double max673;
    double max674;
    double max675;
    double max676;
    double max677;
    double max678;
    double max679;
    double max680;
    double max681;
    double max682;
    double max683;
    double max684;
    double max685;
    double max686;
    double max687;
    double max688;
    double max689;
    double max690;
    double max691;
    double max692;
    double max693;
    double max694;
    double max695;
    double max696;
    double max697;
    double max698;
    double max699;
    double max700;
    double max701;
    double max702;
    double max703;
    double max704;
    double max705;
    double max706;
    double max707;
    double max708;
    double max709;
    double max710;
    double max711;
    double max712;
    double max713;
    double max714;
    double max715;
    double max716;
    double max717;
    double max718;
    double max719;
    double max720;
    double max721;
    double max722;
    double max723;
    double max724;
    double max725;
    double max726;
    double max727;
    double max728;
    double max729;
    double max730;
    double max731;
    double max732;
    double max733;
    double max734;
    double max735;
    double max736;
    double max737;
    double max738;
    double max739;
    double max740;
    double max741;
    double max742;
    double max743;
    double max744;
    double max745;
    double max746;
    double max747;
    double max748;
    double max749;
    double max750;
    double max751;
    double max752;
    double max753;
    double max754;
    double max755;
    double max756;
    double max757;
    double max758;
    double max759;
    double max760;
    double max761;
    double max762;
    double max763;
    double max764;
    double max765;
    double max766;
    double max767;
    double max768;
    double max769;
    double max770;
    double max771;
    double max772;
    double max773;
    double max774;
    double max775;
    double max776;
    double max777;
    double max778;
    double max779;
    double max780;
    double max781;
    double max782;
    double max783;
    double max784;
    double max785;
    double max786;
    double max787;
    double max788;
    double max789;
    double max790;
    double max791;
    double max792;
    double max793;
    double max794;
    double max795;
    double max796;
    double max797;
    double max798;
    double max799;
    double max800;
    double max801;
    double max802;
    double max803;
    double max804;
    double max805;
    double max806;
    double max807;
    double max808;
    double max809;
    double max810;
    double max811;
    double max812;
    double max813;
    double max814;
    double max815;
    double max816;
    double max817;
    double max818;
    double max819;
    double max820;
    double max821;
    double max822;
    double max823;
    double max824;
    double max825;
    double max826;
    double max827;
    double max828;
    double max829;
    double max830;
    double max831;
    double max832;
    double max833;
    double max834;
    double max835;
    double max836;
    double max837;
    double max838;
    double max839;
    double max840;
    double max841;
    double max842;
    double max843;
    double max844;
    double max845;
    double max846;
    double max847;
    double max848;
    double max849;
    double max850;
    double max851;
    double max852;
    double max853;
    double max854;
    double max855;
    double max856;
    double max857;
    double max858;
    double max859;
    double max860;
    double max861;
    double max862;
    double max863;
    double max864;
    double max865;
    double max866;
    double max867;
    double max868;
    double max869;
    double max870;
    double max871;
    double max872;
    double max873;
    double max874;
    double max875;
    double max876;
    double max877;
    double max878;
    double max879;
    double max880;
    double max881;
    double max882;
    double max883;
    double max884;
    double max885;
    double max886;
    double max887;
    double max888;
    double max889;
    double max890;
    double max891;
    double max892;
    double max893;
    double max894;
    double max895;
    double max896;
    double max897;
    double max898;
    
```

```

{
    double i, j, k;

    printf("Enter numbers: ");
    scanf("%lf%lf%lf", &i, &j, &k);

    printf("Max = %f\n", find_max(3, i, j, k));
    return 0;
}

double find_max(int num, ...)
{
    int i;
    double max, tmp;
    va_list arg_ptr;

    va_start(arg_ptr, num);
    max = va_arg(arg_ptr, double);
    for(i = 1; i < num; i++)
    {
        tmp = va_arg(arg_ptr, double);
        if(max < tmp)
            max = tmp;
    }
    va_end(arg_ptr);
    return max;
}

```

**Comments:** Notice that `find_max()` can be used to find the maximum of any number of `double` arguments, not just 3.

---

## Recursive Functions

A function that calls itself is called recursive. For example:

```

#include <stdio.h>

void show(int num);

int main(void)
{
    int i;

    printf("Enter number: ");
    scanf("%d", &i);

    show(i);
    return 0;
}

```

```

void show(int num)
{
    if(num > 1)
        show(num-1);

    printf("val = %d\n", num);
}

```

To see how recursion works, assume that the user enters a number greater than 1, e.g., 3, so that `show()` is called again.

- In the first call of `show()`, since `num = 3 > 1`, `show()` calls itself with argument `num-1 = 3-1 = 2`. `printf()` is not executed. Since `show()` is not terminated, the memory allocated for the variable `num` with value 3 and other information related to the function call is not released.
- In the second call of `show()` new memory is allocated for the `num` variable. Since `num = 2 > 1`, `show()` calls again itself with argument `num-1 = 2-1 = 1`. Like before, `printf()` is not executed and the memory allocated for the new variable `num` with value 2 is not released.
- Like before, in the third call of `show()` new memory is allocated for the `num` variable. `show()` is not called again because `num` is not greater than 1. Therefore, this `printf()` is executed and outputs `val = 1`. The memory allocated for that `num` is released.

Next, all unexecuted `printf()` will be executed sequentially, starting from the last one. In each termination of `show()`, the memory allocated for the respective `num` and for each particular call is released. Therefore, the program displays:

```

val = 1
val = 2
val = 3

```

---

A recursive function must contain a termination condition in order to prevent infinite recursion.

In our example, this condition was the statement `if(num > 1)`.

Each time a recursive function calls itself, new memory is allocated from the stack to store its automatic variables. In our example, three different `n` variables were created, each with a different value. A simple way to describe that is to consider that each `num` is placed on top of the other (e.g., a pile of dirty dishes) and at the end of the recursion we access the `num` created last (e.g., we get the dish at the top of the pile). For static variables, no new memory is allocated; that is, all function calls share the same static variables. The information for which part of the code is left unexecuted is also stored in the stack. That code will be executed backward when the function won't call itself again.

In our example, the value of each `num` is stored in the stack and each `printf()` will be executed in the opposite order from the function calls, that is, from the last one to the first one. This will happen once `show()` does not call itself again, that is, when `num` becomes 1.

However, the size of the stack might not be large enough to store the information associated with each function call. For example, if the user enters a large value, say 100000, the execution of the program might terminate and the message “*Stack overflow*” appears, which indicates that there is no other available memory in the stack. The default size of the stack depends on the implementation, however, a compiler typically provides the ability to change it.

In practice, recursion is often used to implement math algorithms or operations in data structures. For example, we’ll use recursion in Chapter 14 to implement several functions in a binary search tree. However, if you can write equivalent code, it’d be probably a better idea. For example, use an iteration loop instead. Although the recursive code might be easier to read and write, the loop performance would be probably better.

---

## Exercises

C.11.46 What is the output of the following program?

```
#include <stdio.h>

int a = 4;

int main(void)
{
    if(a == 0)
        return 0;
    else
    {
        printf("%d ", a--);
        main();
    }
    return 0;
}
```

**Answer:** Notice that `main()` can be also called recursively. In each call, `a` is decremented by 1. The program stops calling `main()` once `a` becomes 0. Therefore, the program displays: 4 3 2 1

C.11.47 What is the output of the following program?

```
#include <stdio.h>

int unknown(int arr[], int num);

int main(void)
{
    int arr[] = {10, 20, 30, 40};

    printf("%d\n", unknown(arr, 4));
    return 0;
}
```

```

int unknown(int arr[], int num)
{
    if(num == 1)
        return arr[0];
    else
        return arr[num-1] + unknown(arr, num-1);
}

```

**Answer:** When unknown() is called, it returns

```

arr[4-1 = 3] + unknown(arr, 4-1 = 3) =
arr[3] + (arr[3-1] + unknown(arr, 3-1)) =
arr[3] + arr[2] + (arr[2-1] + unknown(arr, 2-1)) =
arr[3] + arr[2] + arr[1] + unknown(arr, 1)

```

The last call of unknown(arr, 1) returns arr[0], because num = 1. Therefore, the return value is arr[3]+arr[2]+arr[1]+arr[0] and the program outputs 100. As a result, the function calculates recursively the sum of the array's elements.

**C.11.48** Write a recursive function that takes as parameter an integer n and returns its factorial, using the formula  $n! = n \times (n-1)!$ . The factorial of an integer  $n$ , where  $n \geq 1$ , is the product of the integers from 1 to  $n$ , that is,  $1 \times 2 \times 3 \times \dots \times n$ . The factorial of 0 is 1 ( $0!=1$ ). Write a program that reads a positive integer less than 170 and uses the function to display its factorial.

```

#include <stdio.h>

double fact(int num);

int main(void)
{
    int num;

    do
    {
        printf("Enter a positive integer less than 170: ");
        scanf("%d", &num);
    } while(num < 0 || num > 170);

    printf("Factorial of %d is %e\n", num, fact(num));
    return 0;
}

double fact(int num)
{
    if((num == 0) || (num == 1))
        return 1;
    else
        return num * fact(num-1);
}

```

**Comments:** Notice that for large values of num the calls to fact() increase; therefore, the time to calculate its factorial also increases. In that case, the alternative solution with the **for** loop in C.6.7 calculates the factorial's number faster.

**C.11.49** Write a recursive function that takes as parameter an integer (e.g., N) and returns the  $N^{\text{th}}$  term of the Fibonacci sequence, using the formula  $F(N) = F(N-1) + F(N-2)$ , where  $F(0) = 0$  and  $F(1) = 1$ . Write a program that reads an integer N between 2 and 40 and uses the function to display the  $N^{\text{th}}$  term.

```
#include <stdio.h>

unsigned int fib(int num);

int main(void)
{
    int num;

    do
    {
        printf("Enter a number between 2 and 40: ");
        scanf("%d", &num);
    } while(num < 2 || num > 40);

    printf("F(%d) = %d\n", num, fib(num));
    return 0;
}

unsigned int fib(int num)
{
    if(num == 0)
        return 0;
    else if(num == 1)
        return 1;
    else
        return fib(num-1) + fib(num-2);
}
```

**Comments:** This recursive solution of printing *Fibonacci* numbers is not so efficient because the recalculation of lower terms is repeated. For example,  $\text{fib}(5) = \text{fib}(4) + \text{fib}(3)$ . To calculate  $\text{fib}(4)$ ,  $\text{fib}(3)$  and  $\text{fib}(2)$  are calculated. When the second term  $\text{fib}(3)$  is calculated, the same calculations are repeated. As a result, the execution time of  $\text{fib}()$  is significantly increased. Comparing the solutions, the code described earlier in C.11.35 is executed faster.

**C.11.50** What does the following program do?

```
#include <stdio.h>

int unknown(int num1, int num2);
```

```

int main(void)
{
    int num1, num2, sign;

    printf("Enter numbers: ");
    scanf("%d%d", &num1, &num2);

    sign = 1;
    if((num1 < 0) && (num2 > 0))
    {
        num1 = -num1;
        sign = -1;
    }
    else if((num1 > 0) && (num2 < 0))
    {
        num2 = -num2;
        sign = -1;
    }
    else if((num1 < 0) && (num2 < 0))
    {
        num1 = -num1;
        num2 = -num2;
    }
    if(num1 > num2)
        printf("%d\n", sign*unknown(num1, num2));
    else
        printf("%d\n", sign*unknown(num2, num1));
    return 0;
}

int unknown(int n1, int n2)
{
    if(n2 == 1)
        return n1;
    else
        return n1 + unknown(n1, n2-1);
}

```

**Answer:** First, let's see what the function is doing. Suppose that it is called with arguments 10 and 4. The function returns:

```

n1 + unknown(n1, n2-1 = 3) =
n1 + n1 + unknown(n1, n2-1 = 2) =
n1 + n1 + n1 + unknown(n1, n2-1 = 1)

```

The last call of `unknown(n1, 1)` returns `n1`, because `n2 = 1`. Therefore, the function returns:

$$n1 + n1 + n1 + n1 = 4 * n1 = n2 * n1$$

The **if-else-if** series finds the sign of the product and the absolute values of the integers. Next, the **if-else** statement is used, in order to make the less recursive calls.

As a result, the purpose of this program is to calculate the product of the input numbers through the use of a recursive function.

C.11.51 Write a recursive function that uses the `strchr()` function (see its description in Appendix C) to display the number of occurrences of a character inside a string. Write a program that reads a string of less than 100 characters and a character and calls the function. Don't rush to see the answer, try it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);
void find(char *p, int ch, int cnt);

int main(void)
{
    char ch, str[100];

    printf("Enter text: ");
    read_text(str, sizeof(str), 1);

    printf("Enter character: ");
    ch = getchar();
    find(str, ch, 0);
    return 0;
}

void find(char *p, int ch, int cnt)
{
    p = strchr(p, ch);
    if(p == NULL)
        printf("%d\n", cnt);
    else
        find(p+1, ch, cnt+1);
}
```

C.11.52 Sometimes, in math, it is very difficult to prove some problems that seem quite simple, like the one of the mathematician Lothar Collatz, who first proposed it. Think of a positive integer  $n$  and execute the following algorithm:

- If it is even, divide it by two ( $n/2$ ).
- If it is odd, triple it and add one ( $3n+1$ ).

Repeat the same process for each produced number and you'll come to a surprising result: for any integer you choose, you'll always end up with ...1!!! For example, if we choose the number 53, the produced numbers are  $53 \rightarrow 160 \rightarrow 80 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

This math problem, well known as “Collatz conjecture,” remains unsolved, although the use of computing machines confirms that for any positive integer up to  $2^{60}$ , we'll eventually reach 1.

What we are asking from you is not to prove the Collatz conjecture and contest for the next Fields prize, but only that, to write a recursive function that takes as parameter a positive integer and displays the produced sequence of numbers.

```
#include <stdio.h>

int collatz(int n);

int main(void)
{
    int a;

    do
    {
        printf("Enter a positive integer: ");
        scanf("%d", &a);
    } while(a <= 0);

    printf("The result is %d indeed!!!\n", collatz(a));
    return 0;
}

int collatz(int n)
{
    printf("%d\n", n);

    if(n == 1)
        return 1;
    else if(n & 1) /* If n is odd. */
        return collatz(3*n+1);
    else /* If n is even. */
        return collatz(n/2);
}
```

**Comments:** Execute the program for several positive integers. The result is amazing, indeed. You'll always reach 1.

---

## Unsolved Exercises

**U.11.1** Write a function that takes as parameters three integers and checks if the sum of the first two numbers is equal to the third one. If it is, the function should return the larger of the first two numbers; otherwise, the smaller of the second and the third one. Write a program that reads three integers, calls the function, and displays the return value.

**U.11.2** Write the functions  $f()$  and  $g()$ , as follows:

$$f(x) = \begin{cases} x+2, & x > 0 \\ -3x+7, & x \leq 0 \end{cases} \quad g(x) = \begin{cases} x^2 + 2, & x > 0 \\ 7x - 5, & x \leq 0 \end{cases}$$

Write a program that reads an integer (e.g.,  $x$ ) and uses  $f()$  and  $g()$  to display the result of  $f(g(x))$ , with the restriction that  $g()$  must be called from inside  $f()$ .

**U.11.3** Write a function that takes as parameters two integers (e.g.,  $a$  and  $b$ ), reads 100 integers, and displays the minimum of those within  $[a, b]$ . Write a program that reads two integers and calls the function. *Note:* the first argument should be less than the second.

**U.11.4** Write a function that takes as parameter a string and a character and returns the number of occurrences of the character in the string. Write a program that reads continuously a character and a string of less than 100 characters, calls the function, and displays the return value. If the user enters end, the insertion of strings should terminate.

**U.11.5** Write a **void** function that takes as parameters two arrays and the number of the elements (e.g.,  $N$ ) to compare. If the  $N$  first elements are the same, the function should return 1, 0 otherwise. Write a program that reads 200 **double** numbers and stores them in two arrays of 100 elements each. Then, the program should read the number of the elements to be compared, use the function to compare them, and display the result. *Hint:* since the return type is **void** add a pointer argument to return the result of the comparison.

**U.11.6** Write a function that takes as parameter an array and checks if it is sorted. If it is sorted in ascending order, the function should return 1, if in descending order, it should return 2, otherwise, another value. Write a program that generates 10 random integers in  $[5, 20]$ , stores them in an array and uses the function to display an informative message if it is sorted or not.

**U.11.7** Write a **void** function that takes as parameters a string and other proper parameters in order to return the number of its lowercase, uppercase letters, and digits. Write a program that reads a string of less than 100 characters, and if it begins with 'a' and ends with 'q', the program should call the function and display the return values.

**U.11.8** Write a function that takes as parameter an array and checks if it contains duplicated values. If so, the function should return a pointer to the element that appears the most times, otherwise **NULL**. Write a program that reads 100 integers, stores them in an array, and uses the function to display the element with the most occurrences. *Note:* if more than one element appears the same most times, the function should return the first found.

**U.11.9** Write a function that takes as parameters two strings and returns a pointer to the longest part in the first string that does not contain any character of the second string. If no part is found, the function should return **NULL**. Write a program that reads two strings of less than 100 characters, calls the function, and displays that part.

**U.11.10** Write a function that takes as parameters three  $2 \times 4$  two-dimensional arrays, calculates the sum of the first two, and stores that sum into the third one. Write a program that reads 16 integers, stores them in two  $2 \times 4$  two-dimensional arrays, and uses the function to display their sum.

**U.11.11** Write a **void** function that takes as parameters a  $3 \times 4$  two-dimensional array, a row number, and a column number and uses proper parameters to return the largest value

in that row and the smallest value in that column. Write a program that reads 12 integers and stores them in a  $3 \times 4$  two-dimensional array. Then, the program should read a row number and a column number and use the function to display the largest value in that row and the smallest value in that column.

**U.11.12** Write a function that takes as parameters a two-dimensional array and an integer and returns a pointer to the row in which that number appears the most times. If the number is not found, the function should return `NULL`. Write a program that assigns random values to a  $5 \times 5$  array of integers, it reads an integer and uses the function to display the elements of the row in which it appears the most times. *Note:* if the number appears in more than one row the same most times, the function should return the pointer to the first found.

**U.11.13** A parking lot has 200 parking spaces for cars and 40 parking spaces for motos. Write a function (e.g., `find_plate()`) that takes as parameters a two-dimensional array that contains plate numbers of less than 10 characters, the plate number of a vehicle, and a third one to specify the vehicle's type (e.g., 1 for cars). If the number is found in the array, the function should return its position in the array, -1 otherwise.

Write a program that reads continuously plate numbers and their vehicle types. For each plate, the program should call `find_plate()`, and if it returns -1, it means that the vehicle is not parked and its plate number should be stored in the first available position of a respective two-dimensional array. Use a two-dimensional plate array for cars and another one for motos. To find an available position, write a second function. If `find_plate()` does not return -1, it means that the plate number is found and the program should set 0 to that position in order to free the space and charge it. The parking fee is \$10 for the cars and \$4 for the motos. If the user enters `end` for plate number, the program should terminate and display the total charge and the number of free spaces.

**U.11.14** Write a program that accepts three command line arguments and displays them in alphabetical ascending order.

**U.11.15** Write a program that displays the characters of its command line arguments in reverse order. For example, if the arguments are `one two`, the program should display `two one`.

**U.11.16** Write a program that accepts as command line arguments the sign of a math operation and two one-digit numbers and displays the result of the operation. For example, if the arguments are +, 5, -3, the program should display 2. The program should check that the input data are valid.

**U.11.17** Write a function that accepts a variable number of pointers to integer arguments and returns the pointer to the largest number. Write a program that reads three integers and uses the function to display the largest value.

**U.11.18** Write a simple version of `printf()` to take as parameter a format string and a variable argument list. The string should contain only the `%c`, `%d`, `%f`, and `%s` specifiers and characters. The function should process the string and the arguments and call `printf()` to display their values. For example:

```

void simple_printf(char fmt[], ...);

int main(void)
{
    int i = 5;
    double j = 1.2, k = i+j;

    simple_printf("%s %c = %d+%f = %f\n", "Result: ", 'k', i, j, k);
    return 0;
}

```

The program should display Result: k = 5+1.2 = 6.2 (plus the 0s).

**U.11.19** Write a function (e.g., f()) that takes as parameters an integer (e.g., a) and a pointer to another function (e.g., g()). g() accepts an integer argument, and if that argument is positive, it returns the corresponding negative; otherwise, it returns the argument as is. If a is even, f() should use the pointer to call g() and display the return value. If it is odd (e.g., 5), it should change it to the next even (e.g., 6) and then call g(). Write a program that reads an integer and uses a pointer variable to call f(). *Note:* revisit the section “Pointer to Function” in Chapter 8.

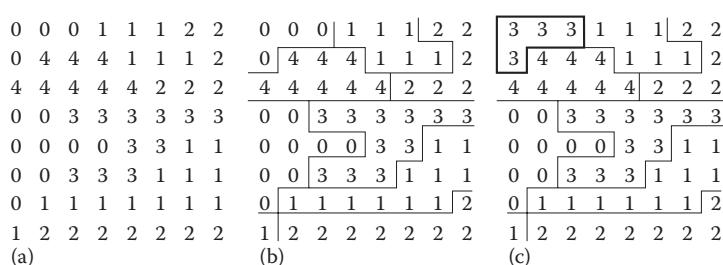
**U.11.20** Modify the power() function in C.11.11 and use the formula  $m^n = m \times m^{n-1}$  to calculate  $m^n$  recursively.

**U.11.21** In math, a triangular number counts the objects that can form an equilateral triangle, as shown next.

$T_0=0$	$T_1=1$	$T_2=3$	$T_3=6$
*	* *	* * *	*

The  $T(n)$  triangular number declares the number of the objects composing the equilateral triangle, and it is equal to the sum of the numbers from 1 to n. Therefore,  $T(n)$  is expressed as  $T(n) = 1 + 2 + 3 + \dots + (n - 1) + n$

and in a recursive form:  $T(n) = \begin{cases} n, & \text{for } n = 0 \text{ or } n = 1 \\ n + T(n-1), & \text{for } n > 1 \end{cases}$



**FIGURE 11.4**

Flood fill algorithm.

Write a program that reads a positive integer (e.g.,  $n$ ) up to 20 and uses a recursive function to display the  $T(n)$  triangular number.

**U.11.22** Write a program that generates 10 random integers and stores them in an array. Then, the program should call a recursive function to check if the array is symmetric, that is, if the value of the first element is equal to the last one, the value of the second one equal to the value of the last but one, and so on. If the array is not symmetric, the function should return the position of the first element that breaks the symmetry.

**U.11.23** Image editing programs often use the *flood fill* algorithm to fill similarly colored connected areas with a new color. Suppose that the two-dimensional  $8 \times 8$  array of Figure 11.4a represents the pixels of an image, where 0 represents the black color, 1: white, 2: red, 3: green, and 4: blue. Consider that a pixel is similarly colored connected with another, if they have the same color and it is adjacent to it. The similarly colored areas are depicted in Figure 11.4b.

To implement the *flood fill* algorithm, write a recursive `floodfill()` function, which changes the color (e.g.,  $c$ ) of a pixel at a random location (e.g.,  $i$ ,  $j$ ) to a new color and then it changes the color of its neighboring pixels (i.e., the pixels to its left, right, up, and down) having the same color  $c$ . Notice that this process should continue recursively on the neighbors of the changed pixels, until there are no other pixels to check. For example, if we choose to change the color of the pixel in the position 0, 0 from black (e.g., 0) to green (e.g., 3), the color of the top-left area of four pixels changes to green, as shown in Figure 11.4c.

Write a program that uses a two-dimensional  $8 \times 8$  array of integers and assigns to its elements random values in  $[0, 4]$ . Then, the program should read the location of a pixel (e.g.,  $i$ ,  $j$ ) and a new color and use `floodfill()` to change the existing color of its similarly colored area with the new one.

## Searching and Sorting Arrays

This chapter describes some of the most common algorithms used for searching a value in an array and for sorting the elements of an array in ascending or descending order. For more algorithms, performance analysis, metrics, and best implementations, you should read a book that focuses on this subject.

### Searching Arrays

To check if a value is stored in an array, we are going to describe the linear and binary search algorithms. In Chapter 14, we'll discuss another technique based on the hashing method.

#### Linear Search Algorithm

The linear search algorithm (also called sequential search) is the simplest algorithm to search for a value in a *nonsorted* array. The searched value is compared with the value of each element until a match is found. In an array of  $n$  elements, the maximum number of searches is  $n$ . This may occur if the searched value is not found or is equal to the last element. In the following program, the `linear_search()` function implements the linear search algorithm.

### Exercises

C.12.1 Write a function that searches for a number in an array of doubles. If the number is stored, the function should return the number of its occurrences and the position of its first occurrence, otherwise -1. Write a program that reads up to 100 doubles and stores them in an array. If the user enters -1, the insertion of numbers should terminate. Then, the program should read a double number and use the function to display its occurrences and the position of its first occurrence in the array.

```
#include <stdio.h>

#define SIZE 100

int linear_search(double arr[], int size, double num, int *t);
```

```

int main(void)
{
    int i, times, pos;
    double num, arr[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%lf", &num);
        if(num == -1)
            break;
        arr[i] = num;
    }
    printf("Enter number to search: ");
    scanf("%lf", &num);

    pos = linear_search(arr, i, num, &times); /* The variable i
indicates the number of the array's elements. */
    if(pos == -1)
        printf("%f isn't found\n", num);
    else
        printf("%f appears %d times (first pos = %d)\n", num, times,
pos);

    return 0;
}

int linear_search(double arr[], int size, double num, int *t)
{
    int i, pos;

    pos = -1;
    *t = 0;
    for(i = 0; i < size; i++)
    {
        if(arr[i] == num)
        {
            (*t)++;
            if(pos == -1) /* Store the position of the first
occurrence. */
                pos = i;
        }
    }
    return pos;
}

```

**C.12.2** Write a function that takes as parameter an integer array and returns the maximum number of the same occurrences. For example, if the array is {1, 10, -3, 5, -3, 8}, the function should return 2 because -3 appears the most times, that is, 2. Write a program that reads 10 integers, stores them in an array, and uses the function to display the maximum number of the same occurrences.

```

#include <stdio.h>

#define SIZE 10

int num_occurs(int arr[]);

int main(void)
{
    int i, arr[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%d", &arr[i]);
    }
    printf("\nMax occurrences is %d\n", num_occurs(arr));
    return 0;
}

int num_occurs(int arr[])
{
    int i, j, k, max_times;

    max_times = 1;
    for(i = 0; i < SIZE; i++)
    {
        k = 1; /* Each number appears once at least. */
        for(j = i+1; j < SIZE; j++) /* Compare arr[i] with the rest
elements. */
        {
            if(arr[i] == arr[j])
                k++;
        }
        if(k > max_times)
            max_times = k;
    }
    return max_times;
}

```

## Binary Search Algorithm

The binary search algorithm is used to search for a value in a *sorted* array (either in ascending or descending order). To understand how the algorithm works, suppose that we are searching for a value in an array sorted in ascending order:

*Step 1:* We use the variables `start` and `end` to indicate the start and the end of the part of the array, in which we are searching for the value. We use the variable `middle` to calculate the middle position of that part: `middle = (start+end)/2`. For example, if we have a sorted array of 100 integers, `start` should be initialized to 0, `end` to 99, so `middle` becomes 49.

*Step 2:* We compare the value we are searching for with the middle element:

- If they are equal, the searched value is found and the algorithm terminates.
- If it is greater, the search continues in the part starting from the middle position and up to the end. The algorithm goes back to Step 1, and start becomes `start = middle+1`.
- If it is less, the search continues in the part starting from the start position and up to the middle. The algorithm goes back to Step 1, and end becomes `end = middle-1`.

In short, the binary search algorithm divides the array into two parts. Then, the searched value is compared with the middle element and the search continues in the proper part. The algorithm terminates if either the searched value is found or start becomes greater than end. In the following program, the `binary_search()` function implements the binary search algorithm.

---

## Exercises

C.12.3 Write a function that searches for a number in an array of integers. If the number is found, the function should return its position, otherwise -1. Write a program that initializes an array of integers with values sorted in ascending order. The program should read an integer and use the function to display its position in the array.

```
#include <stdio.h>

int binary_search(int arr[], int size, int num);

int main(void)
{
    int num, pos, arr[] = {10, 20, 30, 40, 50, 60, 70};

    printf("Enter number to search: ");
    scanf("%d", &num);

    pos = binary_search(arr, 7, num);
    if(pos == -1)
        printf("%d isn't found\n", num);
    else
        printf("%d is found in position %d\n", num, pos);
    return 0;
}

int binary_search(int arr[], int size, int num)
{
    int start, end, middle;

    start = 0;
    end = size-1;
    while(start <= end)
```

```

    {
        middle = (start+end)/2;

        if(num < arr[middle])
            end = middle-1;
        else if(num > arr[middle])
            start = middle+1;
        else
            return middle;
    }
    return -1; /* If the execution reaches this point it means that
the number was not found. */
}

```

**Comments:** To see how the algorithm works, suppose that the user enters the number 45.

*First iteration.* The initial value of start is 0 and end is 6. middle becomes  $(start+end)/2 = 6/2 = 3$ . Since  $arr[middle]$  is 40, less than 45, the next statement to be executed is  $start = middle+1 = 3+1 = 4$ .

*Second iteration.* middle becomes  $(start+end)/2 = (4+6)/2 = 5$ . Since  $arr[middle]$  is 60, greater than 45, the next statement is  $end = middle-1 = 4$ .

*Third iteration.* middle becomes  $(start+end)/2 = (4+4)/2 = 4$ . Since  $arr[middle]$  is 50, greater than 45, the next statement is  $end = middle-1 = 3$ .

Since start is greater than end, the loop terminates and the function returns -1.

**C.12.4** Write a program that reads an integer within  $[0, 1000]$  and uses the binary search algorithm to “guess” that number. The program should make questions to determine if the number we are searching for is less or greater than the middle of the examined interval. The answers must be given in the form of 0 (no) or 1 (yes). The program should display in how many tries the number was found.

```

#include <stdio.h>
int main(void)
{
    int x, ans, low, high, middle, times;

    do
    {
        printf("Enter number in [0, 1000]: ");
        scanf("%d", &x);
    } while(x < 0 || x > 1000);

    times = 1;
    low = 0;
    high = 1000;
    middle = (high+low)/2;
    while(high >= low)
    {
        printf("Is %d the hidden number (0 = No, 1 = Yes) ? ",
middle);
        scanf("%d", &ans);
        if(ans == 1)
            high = middle - 1;
        else
            low = middle + 1;
        middle = (high+low)/2;
    }
}

```

```

if(ans == 1)
{
    printf("Num = %d is found in %d tries\n", x, times);
    return 0;
}
times++;
printf("Is the hidden number < %d (0 = No, 1 = Yes) ? ",
middle);
scanf("%d", &ans);
if(ans == 1)
{
    high = middle-1;
    middle = (high+low)/2;
}
else
{
    low = middle+1;
    middle = (high+low)/2;
}
}
printf("Num = %d isn't found. You probably gave a wrong answer\n", x);
return 0;
}

```

---

## Sorting Arrays

There are several algorithms to sort an array. We are going to describe some of the most popular, the selection sort, the insertion sort, the bubble sort, and the quick sort algorithms.

### Selection Sort Algorithm

To describe the algorithm, we'll show you how to sort an array in ascending order. At first, we find the element with the smallest value and we swap it with the first element of the array. Therefore, the smallest value is stored in the first position.

Then, we find the smallest value among the remaining elements, except the first one. Like before, we swap that element with the second element of the array. Therefore, the second smallest value is stored in the second position. This procedure is repeated with the rest of the elements and the algorithm terminates once the last two elements are compared.

To sort the array in descending order, we find the largest value instead of the smallest. In the following program, the `sel_sort()` function implements the selection sort algorithm to sort an array in ascending order.

## Exercises

C.12.5 Write a function that takes as parameters an array of doubles and uses the selection sort algorithm to sort it in ascending order. Write a program that reads the grades of 10 students, stores them in an array, and uses the function to sort it.

```
#include <stdio.h>

#define SIZE 10

void sel_sort(double arr[]);

int main(void)
{
    int i;
    double grd[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter grade of stud_%d: ", i+1);
        scanf("%lf", &grd[i]);
    }
    sel_sort(grd);

    printf("\n***** Sorted array *****\n");
    for(i = 0; i < SIZE; i++)
        printf("%.2f\n", grd[i]);
    return 0;
}

void sel_sort(double arr[])
{
    int i, j;
    double tmp;

    for(i = 0; i < SIZE-1; i++)
    {
        for(j = i+1; j < SIZE; j++)
        {
            if(arr[i] > arr[j])
            {
                /* Swap values. */
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
            }
        }
    }
}
```

**Comments:** In each iteration of the inner loop, `arr[i]` is compared with the elements from `i+1` up to `SIZE-1`. If an element is less than `arr[i]`, their values are swapped. Therefore,

in each iteration of the outer loop, the smallest value of the elements from  $i$  up to  $\text{SIZE}-1$  is stored at  $\text{arr}[i]$ . To sort the array in descending order, change the **if** statement to:

```
if( $\text{arr}[i] < \text{arr}[j]$ )
```

**C.12.6** To continue the previous exercise, the program should also read the names of the 10 students (less than 100 characters each) and display the final rating in ascending order of the given grades.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 10

void sel_sort(char str[] [100], double arr[]);
int read_text(char str[], int size, int flag);

int main(void)
{
    char name[SIZE] [100]; /* Array of SIZE rows to store the names of
the students. Each name[i] can be used as a pointer to the corresponding
row. */
    int i;
    double grd[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter name: ");
        read_text(name[i], sizeof(name[i]), 1);

        printf("Enter grade: ");
        scanf("%lf", &grd[i]);
        getchar();
    }
    sel_sort(name, grd);

    printf("\n***** Final Rating *****\n");
    for(i = 0; i < SIZE; i++)
        printf("%s\t%.2f\n", name[i], grd[i]);
    return 0;
}

void sel_sort(char str[] [100], double arr[])
{
    char tmp[100];
    int i, j;
    double k;

    for(i = 0; i < SIZE-1; i++)
    {
        for(j = i+1; j < SIZE; j++)
        {
            if(arr[i] > arr[j])

```

```

    {
        /* Parallel swapping of grades and names. */
        k = arr[i];
        arr[i] = arr[j];
        arr[j] = k;

        strcpy(tmp, str[j]);
        strcpy(str[j], str[i]);
        strcpy(str[i], tmp);
    }
}
}
}

```

**C.12.7** Write the `add_sort()` function that inserts a number into a sorted array, so that the array remains sorted. Write a program that reads 9 integers, stores them in an array of 10 integers, and sorts these 9 elements in ascending order. Then, the program should read the last integer, use the `add_sort()` to insert it in the array, and display the sorted array.

```

#include <stdio.h>

#define SIZE 10

void sel_sort(int arr[], int size);
void add_sort(int arr[], int size, int num);

int main(void)
{
    int i, num, a[SIZE];

    for(i = 0; i < SIZE-1; i++) /* Read 9 integers and store them in
the array. */
    {
        printf("Enter number: ");
        scanf("%d", &a[i]);
    }
    sel_sort(a, SIZE-1); /* Sort the 9 elements. */
    printf("Insert number in sorted array: ");
    scanf("%d", &num);

    add_sort(a, SIZE-1, num); /* Insert the last integer in the array. */
    for(i = 0; i < SIZE; i++)
        printf("%d\n", a[i]);
    return 0;
}

void add_sort(int arr[], int size, int num)
{
    int i, pos;

    if(num <= arr[0])
        pos = 0;

```

```

    else if(num >= arr[size-1]) /* If it greater than the last one,
store it in the last position and return. */
{
    arr[size] = num;
    return;
}
else
{
    for(i = 0; i < size-1; i++)
    {
        /* Check all adjacent pairs up to the last one at
positions SIZE-3 and SIZE-2 to find the position to insert the number. */
        if(num >= arr[i] && num <= arr[i+1])
            break;
    }
    pos = i+1;
}
for(i = size; i > pos; i--)
    arr[i] = arr[i-1]; /* The elements are shifted one position
to the right, starting from the last position of the array, that is [SIZE-1],
up to the position in which the new number will be inserted. For example,
in the last iteration, i = pos+1, so, arr[pos+1] = arr[pos]. */
arr[pos] = num; /* Store the number. */
}

void sel_sort(int arr[], int size)
{
    int i, j, temp;

    for(i = 0; i < size-1; i++)
    {
        for(j = i+1; j < size; j++)
        {
            if(arr[i] > arr[j])
            {
                /* Swap values. */
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

```

**C.12.8** Write a function that takes as parameter a two-dimensional array and sorts its rows in ascending order and another function that sorts its columns in descending order. Write a program that generates 20 random integers and stores them in a 4×5 array. Then, the program should read an integer, and if it is 1, the program should call the first function, otherwise the second one.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

```

```
#define ROWS 4
#define COLS 5

void sort_rows(int arr[] [COLS]);
void sort_cols(int arr[] [COLS]);

int main(void)
{
    int i, j, type, arr[ROWS] [COLS];

    srand(time(NULL));
    for(i = 0; i < ROWS; i++)
        for(j = 0; j < COLS; j++)
            arr[i] [j] = rand();

    printf("Enter sort type (1: rows): ");
    scanf("%d", &type);
    if(type == 1)
        sort_rows(arr);
    else
        sort_cols(arr);

    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
            printf("%10d", arr[i] [j]);
        printf("\n");
    }
    return 0;
}

void sort_rows(int arr[] [COLS])
{
    int i, j, k, temp;

    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS-1; j++)
        {
            for(k = j+1; k < COLS; k++)
            {
                if(arr[i] [j] > arr[i] [k])
                {
                    temp = arr[i] [j];
                    arr[i] [j] = arr[i] [k];
                    arr[i] [k] = temp;
                }
            }
        }
    }
}

void sort_cols(int arr[] [COLS])
{
    int i, j, k, temp;
```

```

for(j = 0; j < COLS; j++)
{
    for(i = 0; i < ROWS-1; i++)
    {
        for(k = i+1; k < ROWS; k++)
        {
            if(arr[i][j] < arr[k][j])
            {
                temp = arr[i][j];
                arr[i][j] = arr[k][j];
                arr[k][j] = temp;
            }
        }
    }
}

```

## Insertion Sort Algorithm

To describe the algorithm, we'll sort an array in ascending order. The operation of the algorithm is based on sequential comparisons between each element (starting from the second and up to the last element) and the elements on its left, which form the "sorted subarray." The elements on its right form the "unsorted subarray." In particular, the left-most element of the unsorted subarray (i.e., the examined element) is compared against the elements of the sorted subarray from right to left; according to this:

*Step 1:* At first, the examined element is stored in a temporary variable (Figure 12.1a).

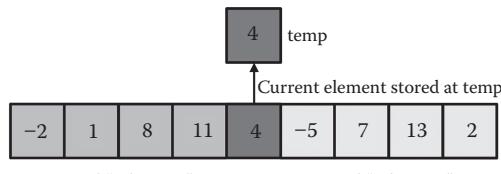
*Step 2a:* If it is not less than the rightmost element of the sorted subarray, its position does not change and the algorithm continues with the testing of the next most left element.

*Step 2b:* If it is less than the rightmost element of the sorted subarray, the latter is shifted one position to the right while the examined element is compared against the rightmost but one element (Figure 12.1b and c). If it is not less, the examined element is stored at the position of the last shifted element; otherwise, the same procedure is repeated (the rightmost but one element is shifted one position to the right and the examined element is compared against the rightmost but two elements) (Figure 12.1c and d). Step 2b terminates either if the examined element is not less than an element of the sorted subarray (Figure 12.1d and e) or the first element of the sorted subarray is reached. Then, Step 1 is repeated for the new leftmost element (Figure 12.1e and f).

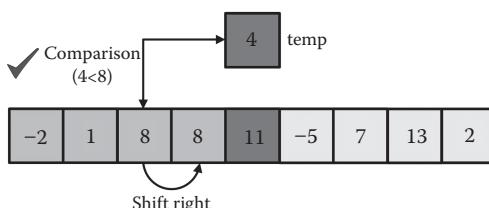
The algorithm terminates when the rightmost element of the unsorted subarray is tested.

The algorithm resembles the way that a card player would sort a card game hand, assuming that he or she starts with an empty left hand and all cards face down on the table. The player picks up a card with his right hand and inserts it in the correct position in the left hand. To find the correct position, that card is compared against the cards in his left hand, from right to left.

To sort the array in descending order, an element of the sorted subarray is shifted one position to the right if it is less than the examined element. In the following program, the `insert_sort()` function implements the insertion sort algorithm to sort an array in ascending order.



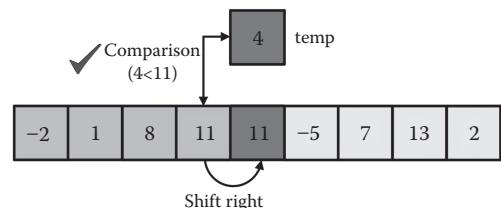
(a)



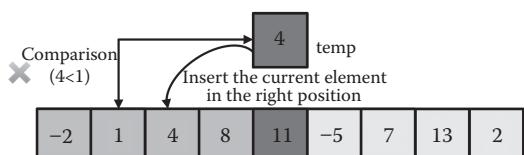
(c)



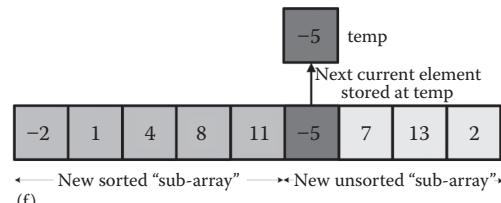
(e)



(b)



(d)



(f)

**FIGURE 12.1**

Insertion sort algorithm.

## Exercise

**C.12.9** Write a function that takes as parameter an array of integers and uses the insertion sort algorithm to sort it in ascending order. Write a program that reads five integers, stores them in an array, and uses the function to sort it.

```
#include <stdio.h>

#define SIZE 5

void insert_sort(int arr[]);

int main(void)
{
    int i, a[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%d", &a[i]);
    }
    insert_sort(a);
    printf("\n***** Sorted array *****\n");
    for(i = 0; i < SIZE; i++)
        printf("%d\n", a[i]);
}
```

```

    return 0;
}

void insert_sort(int arr[])
{
    int i, j, temp;

    for(i = 1; i < SIZE; i++)
    {
        temp = arr[i];
        j = i;
        while((j > 0) && (arr[j-1] > temp))
        {
            arr[j] = arr[j-1]; /* Shift this element one position
to the right. */
            j--;
        }
        arr[j] = temp;
    }
}

```

**Comments:** The **for** loop compares the elements starting from the second one. In each iteration, **temp** holds the examined element. The **while** loop shifts one position to the right the elements being on the left of the examined element and are greater than it. Suppose that the array elements are 7, 3, 1, 9, 4.

First **for** loop iteration (*i* = 1 and **temp** = *arr*[1] = 3)

(*j* = 1) First **while** loop iteration: 7 --> 3 1 9 4

So, the array is transformed to: 3 7 1 9 4

Second **for** loop iteration (*i* = 2 and **temp** = *arr*[2] = 1)

(*j* = 2) First **while** loop iteration: 3 7 --> 1 9 4

(*j* = 1) Second **while** loop iteration: 3 --> 7 1 9 4

So, the array is transformed to: 1 3 7 9 4

Third **for** loop iteration (*i* = 3 and **temp** = *arr*[3] = 9)

(*j* = 3) No shifting takes place because the fourth array element (i.e., *arr*[3] = 9) is greater than the rightmost element (i.e., *arr*[2] = 7).

So, the array remains the same: 1 3 7 9 4

Fourth **for** loop iteration (*i* = 4 and **temp** = *arr*[4] = 4)

(*j* = 4) First **while** loop iteration: 1 3 7 9 --> 4

(*j* = 3) Second **while** loop iteration: 1 3 7 --> 9 4

The sorting is completed and the array is transformed to: 1 3 4 7 9

To sort the array in descending order, change the **while** statement to

```
while((j > 0) && (arr[j-1] < temp))
```

in order to shift one position to the right, the elements of the sorted subarray, which are less than the examined element.

## Bubble Sort Algorithm

The bubble sort algorithm is based on sequential comparisons between adjacent array elements. Each element “bubbles” up and is stored in the proper position. For example, suppose that we want to sort an array in ascending order.

At first, the last element is compared against the last but one. If it is less, the elements are swapped, so the smaller value bubbles up. Then, the last by one element is compared against the last but two. Like before, if it is less, the elements are swapped, so the smaller value keeps bubbling up. The comparisons continue up to the beginning of the array, and eventually the smallest value “bubbles” to the top of the array and it is stored at its first position.

This procedure is repeated from the second element and up to the last one, so the second smallest value of the array bubbles to the top and stored in the second position. The same is repeated for the part of the array from its third element and up to the last one, and so forth. The algorithm terminates when no element bubbles to the top.

To sort the array in descending order, the bubbling value is the greatest and not the smallest one. In the following program, the `bubble_sort()` function implements the bubble sort algorithm to sort an array in ascending order.

---

## Exercises

C.12.10 Write a function that takes as parameter an array of integers and uses the bubble sort algorithm to sort it in ascending order. Write a program that reads five integers, stores them in an array, and uses the function to sort it.

```
#include <stdio.h>

#define SIZE 5

void bubble_sort(int arr[]);

int main(void)
{
    int i, a[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%d", &a[i]);
    }
    bubble_sort(a);

    printf("\n***** Sorted array *****\n");
    for(i = 0; i < SIZE; i++)
        printf("%d\n", a[i]);
    return 0;
}

void bubble_sort(int arr[])
{
```

```

int i, j, temp, reorder;

for(i = 1; i < SIZE; i++)
{
    reorder = 0;
    for(j = SIZE-1; j >= i; j--)
    {
        if(arr[j] < arr[j-1])
        {
            /* Swap values. */
            temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
            reorder = 1;
        }
    }
    if(reorder == 0)
        return;
}
}

```

**Comments:** The `reorder` variable checks if the sorting is completed in order to avoid unnecessary iterations. If two elements are swapped, it is set to 1. Otherwise, the value 0 means that the array is sorted and the function terminates. Suppose that the array elements are 10, 9, 4, 7, 6.

First iteration of the outer `for` loop ( $i = 1$ )

First iteration of the inner `for` loop ( $j = 4$ ): 10 9 4 6  $\leftrightarrow$  7

Second iteration of the inner `for` loop ( $j = 3$ ): 10 9 4 6 7

Third iteration of the inner `for` loop ( $j = 2$ ): 10 4  $\leftrightarrow$  9 6 7

Fourth iteration of the inner `for` loop ( $j = 1$ ): 4  $\leftrightarrow$  10 9 6 7

So, the array is transformed to: 4 10 9 6 7

Second iteration of the outer `for` loop ( $i = 2$ )

First iteration of the inner `for` loop ( $j = 4$ ): 4 10 9 6 7

Second iteration of the inner `for` loop ( $j = 3$ ): 4 10 6  $\leftrightarrow$  9 7

Third iteration of the inner `for` loop ( $j = 2$ ): 4 6  $\leftrightarrow$  10 9 7

So, the array is transformed to: 4 6 10 9 7

Third iteration of the outer `for` loop ( $i = 3$ )

First iteration of the inner `for` loop ( $j = 4$ ): 4 6 10 7  $\leftrightarrow$  9

Second iteration of the inner `for` loop ( $j = 3$ ): 4 6 7  $\leftrightarrow$  10 9

So, the array is transformed to: 4 6 7 10 9

Fourth iteration of the outer `for` loop ( $i = 4$ )

First iteration of the inner `for` loop ( $j = 4$ ): 4 6 7 10  $\leftrightarrow$  9

The sorting is completed and the array is transformed to 4 6 7 9 10

To sort the array in descending order, change the `if` statement to `if(arr[j] > arr[j-1])`

**C.12.11** Write a program that reads the names of 50 countries (less than 100 characters each) and the number of tourists who visited them on monthly basis. The program should use proper arrays to store the data. Then, the program should read the name of a country and display the annual number of tourists who visited that country. The program should display the five most visited countries (check if more than one country ties in the fifth place) before it ends.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define CNTRS      50
#define MONTHS     12

void bubble_sort(char str[] [100], int arr[]);
int read_text(char str[], int size, int flag);

int main(void)
{
    char cntr[CNTRS] [100], str[100];
    int i, j, tmp, flag, tour[CNTRS] = {0};

    for(i = 0; i < CNTRS; i++)
    {
        printf("Enter name of country_%d: ", i+1);
        read_text(cntr[i], sizeof(cntr[i]), 1);

        for(j = 0; j < MONTHS; j++)
        {
            printf("Enter tourists of month_%d: ", j+1);
            scanf("%d", &tmp);
            tour[i] += tmp; /* This array holds the annual number
of tourists for each country. */
        }
        getchar();
    }
    printf("Enter country to search: ");
    read_text(str, sizeof(str), 1);

    flag = 0;
    for(i = 0; i < CNTRS; i++)
    {
        if(strcmp(str, cntr[i]) == 0)
        {
            flag = 1;
            printf("%d tourists visited %s\n", tour[i], str);
            break;
        }
    }
    if(flag == 0)
        printf("%s not registered\n", str);

    bubble_sort(cntr, tour); /* Sort the tourist array and update the
countries array in parallel. */
}
```

```

printf("\n***** Tourists in decrease order *****\n");
for(i = 0; i < 5; i++)
    printf("%d.%s\t%d\n", i+1, cntr[i], tour[i]);
/* Check if more than one country ties in the fifth place. */
while((tour[i] == tour[4]) && i < CNTRS)
{
    printf("%d.%s\t%d\n", i+1, cntr[i], tour[i]);
    i++;
}
return 0;
}

void bubble_sort(char str[] [100], int arr[])
{
    char temp[100];
    int i, j, k, reorder;

    for(i = 1; i < CNTRS; i++)
    {
        reorder = 0;
        for(j = CNTRS-1; j >= i; j--)
        {
            if(arr[j] > arr[j-1]) /* Parallel swapping of the
tourist numbers and the respective countries. */
            {
                k = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = k;

                strcpy(temp, str[j]);
                strcpy(str[j], str[j-1]);
                strcpy(str[j-1], temp);
                reorder = 1;
            }
        }
        if(reorder == 0)
            return;
    }
}

```

## Quick Sort Algorithm

The quick sort algorithm is of *divide and conquer* type. It is a particularly popular algorithm and the best choice for a variety of sorting applications. Its operation is based on the partition of the array into two parts and the separate sorting of each part. At first, the element of the partition is selected (i.e.,  $a[i]$ ). The array is reordered according to the following:

- a.  $a[i]$  is set in its final position.
- b. No element in the left subarray  $a[1], \dots, a[i-1]$  is greater than  $a[i]$ .
- c. No element in the right subarray  $a[i+1], \dots, a[r]$  is less than  $a[i]$ .

This process is recursively repeated for each of the subarrays, until those parts become single elements.

The performance of the algorithm depends on the selection of the partition element. If we are aware of the array content, the best choice is an element that divides the array near the middle. To give an implementation example, we sort an array in ascending order. To improve the performance of the algorithm, several methods have been proposed for the optimum selection of the partition element. For simplicity, we select the most right element.

We scan the array from left to right until we find an element greater or equal than the partition element. Then, we scan the array from right to left until we find an element less than the partition element. Since those two elements are not in right positions, we swap them. Repeating this procedure, the result is that no element in the left subarray is greater than the partition element and no element in the right subarray is less than the partition element. We use index i for the left scan and index j for the right scan. When the two indexes are met, the rightmost element is swapped with the element indexed by i and the partition is completed. To implement the algorithm, we'll use recursion.

```
#include <stdio.h>

#define SIZE 7

int partition(int a[], int l, int r);
void quick_sort(int a[], int l, int r);

int main(void)
{
    int i, a[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%d", &a[i]);
    }
    quick_sort(a, 0, SIZE-1);

    printf("\n***** Sorted array *****\n");
    for(i = 0; i < SIZE; i++)
        printf("%d\n", a[i]);
    return 0;
}

void quick_sort(int a[], int l, int r)
{
    int i;

    if(r <= l)
        return;
    i = partition(a, l, r);
    quick_sort(a, l, i-1);
    quick_sort(a, i+1, r);
}
```

```

int partition(int a[], int l, int r)
{
    int i, j, v, tmp;

    i = l;
    j = r-1;
    v = a[r];
    while(1)
    {
        while(a[i] < v)
            i++;
        while(a[j] >= v)
        {
            j--;
            if(j == l) /* We check the case that the partition
element is the lower in the examined part. */
                break;
        }
        if(i >= j)
            break;
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
    tmp = a[i];
    a[i] = a[r];
    a[r] = tmp;
    return i;
}

```

**Comments:** Suppose that the user enters the values 2, 15, 4, 1, 7, 20, 5. Let's trace the first call of `partition()`. We have  $i=1=0$ ,  $j=r-1=5$ ,  $v=a[6]=5$ . The first loop performs the left scan and terminates once  $i$  becomes 1 (since  $a[1] > 5$ ). The second loop performs the right scan and terminates once  $j$  becomes 3 (since  $a[3] < 5$ ). The two elements are swapped and the array is transformed to 2, 1, 4, 15, 7, 20, 5.

Left scan continues and terminates once  $i$  becomes 3 (since  $a[3] > 5$ ). Right scan continues and terminates once  $j$  becomes 2 (since  $a[2] < 5$ ). Because  $i > j$  the infinite loop terminates.  $a[3]$  and  $a[6]$  are swapped and the array is transformed to 2, 1, 4, 5, 7, 20, 15. Notice that 5 is set in its final position and all elements in its left are less, while all elements in its right are greater. The function returns the value of  $i$ , that is, 3. Then, recursion is used to repeat the same procedure for the left subarray (positions 0 to 2) and for the right subarray (positions 4 to 6).

## The `bsearch()` and `qsort()` Library Functions

The `bsearch()` library function is used to search for a value in a sorted array. The `qsort()` library function is used to sort an array. Appendix C provides a short description for both of them.

## Exercise

C.12.12 Write a program that reads 10 integers and stores them in an array. The program should use `qsort()` to sort the array in ascending order. Then, the program should read an integer and use `bsearch()` to check if it is stored in the array.

```
#include <stdio.h>
#include <stdlib.h>

#define NUM 10

int compare(const void *elem1, const void *elem2);

int main(void)
{
    int *pos, i, arr[NUM];

    for(i = 0; i < NUM; i++)
    {
        printf("Enter number: ");
        scanf("%d", &arr[i]);
    }
    qsort(arr, NUM, sizeof(int), compare);

    printf("\nSorted array: ");
    for(i = 0; i < NUM; i++)
        printf("%d ", arr[i]);

    printf("\nEnter number to search: ");
    scanf("%d", &i);
    pos = (int*)bsearch(&i, arr, NUM, sizeof(int), compare);
    if(pos == NULL)
        printf("\n%d isn't found\n", i);
    else
        printf("\n%d is in position %d\n", i, pos-arr+1);
    return 0;
}

int compare(const void *elem1, const void *elem2)
{
    return *(int*)elem1 - *(int*)elem2;
}
```

**Comments:** Appendix C describes the parameters for both `qsort()` and `bsearch()`. Notice that the name of `compare()` is used as a pointer to that function. According to the declarations of `qsort()` and `bsearch()`, the parameters of `compare()` must be declared as `const void*` pointers. Since we compare integers, we cast their types to `int*`. Then, the values that the two pointers point to are compared and `compare()` returns a value, according to the description of `bsearch()`.

If the searched value is found, `bsearch()` returns a pointer to the respective element, otherwise `NULL`. We cast the return type from `void*` to `int*` to make it clear. The program

uses pointer arithmetic to display its position, if found. To sort the array in descending order, just reverse the comparison:

```
int compare(const void *elem1, const void *elem2)
{
    return *(int*)elem2 - *(int*)elem1; /* For descending order. */
}
```

---

## Unsolved Exercises

**U.12.1** Write a program that reads the populations of 100 cities and stores them in ascending order in an array when entered. The program should display the array before it ends. For example, if the user enters 2000 and then 1000, the first two elements should be 1000, 2000. If the next input value is 1500, the first three elements should be 1000, 1500, and 2000.

**U.12.2** Write a program that reads the names of 50 students (less than 100 characters each) and stores them in an array. Suppose that the user enters the names in alphabetical order. Use a second array and do the same for another group of 70 students. The program should merge the two arrays into a third one, where all names should be stored in alphabetical order.

**U.12.3** Write a program that reads the names of 20 cities (less than 100 characters each) and the average monthly temperatures for each city. Then, the program should read a city name and display the warmest month (e.g., 1:Jan – 12:Dec) for that city. Also, the program should display for the same city the months with higher temperature than the previous and next month.

**U.12.4** Modify C.12.3 so that the program reads a string of less than 100 characters and uses `binary_search()` to display its position in the array, if found. Use the following sorted array and the new prototype to modify the function.

```
int binary_search(char *arr[], int size, char *str);

int main(void)
{
    char str[100], *arr[] = {"alpha", "beta", "gamma", "delta",
"epsilon", "eta", "omega"};
    int pos;
    ...
}
```

**U.12.5** Write a recursive version of the `binary_search()` in C.12.3. Use the following prototype to modify the function. Here is an example how to call it:

```
int binary_search(int arr[], int num, int start, int end);

int main(void)
```

```
{  
    ...  
    pos = binary_search(arr, num, 0, sizeof(arr)/sizeof(int)-1);  
    ...  
}
```

**U.12.6** Modify `sel_sort()` of C.12.5 using the `p1` and `p2` pointers to sort `arr` in ascending order.

```
void sel_sort(double arr[])  
{  
    double *p1, *p2, tmp;  
  
    ...  
}
```

**U.12.7** Write a program that reads the names of 10 students (less than 100 characters each), stores them in a two-dimensional array, and uses the bubble sort algorithm to sort the array in alphabetical order. The program should display the array before it ends.

**U.12.8** Modify the previous exercise to use `qsort()` to sort the names.

**U.12.9** Write a program that reads the names of 10 courses (less than 100 characters each) and the grades of 50 students on those courses. The program should display the names of the courses and the average grade next to them, starting with the course with the best average. In case of same average, the program should display the “tied” courses in alphabetical order. To sort the array, use the bubble sort algorithm.

**U.12.10** In a music contest, 200 judges vote for the top songs out of 50 songs. Write a program that reads the names of the songs (less than 100 characters each) and displays the top five songs. If there is a tie in the fifth place, the program should display the “tied” songs. To vote for a song, the judge enters its title. Each judge can vote up to five songs or less and enter the word `end` as a song title. Assume that the judge does not vote for the same song more than once. To sort the array, use the selection sort algorithm.

**U.12.11** Write a program that reads the prices and the book titles (less than 100 characters each) of two bookstores. Assume that the maximum number of books in each bookstore is 200. If the user enters `end` for book title, the insertion of books should terminate. The program should display the book titles and prices of both bookstores sorted by price descending order. There is one restriction. If a title is found in both bookstores, the program should display that title once with the price in the first bookstore.

**U.12.12** Write a program that reads the names of 1000 candidates (less than 100 characters each), their score, and the name of the school they were accepted to. Then, the program should read a candidate’s name and display the school, as well as the score and the names of all candidates accepted in that same school sorted by score descending order. For example, if the user enters

P.Smith	2300	CIT
A.Leep	1200	LU
R.Bachir	1600	LU
B.Koong	2800	BPL

and then the user enters A.Leep, the program should display LU and

R.Bachir	1600	LU
A.Leep	1200	LU

# Structures and Unions

This chapter introduces two new types: *structures* and *unions*. Like arrays, structures and unions group a set of values into a single entity. Both help to organize data in a more natural way. However, their properties are quite different from an array's. Unlike arrays, the members of a structure or union may have *different* types. Furthermore, to access a member of a structure or union, we specify its *name*, not its position. This chapter discusses how to declare structures and unions types, declare variables, and perform operations on them.

## Structures

When we want to group related data items into a single entity, a structure is a typical choice. For example, a structure may hold information for a company, such as its name, business core, tax number, number of employees, contact information, and other data.

### Declare a Structure

To declare a structure, we use the **struct** keyword. The general syntax for a structure declaration is:

```
struct structure_tag
{
    member_list;
} structure_variable_list;
```

A **struct** declaration defines a type. Although the `structure_tag` is optional, we prefer to name the structures we declare and use that name later to declare variables. We'll see another use of the `structure_tag` in Chapter 14 when declaring self-referential structures to form linked lists. In that case, it is required to use a structure tag. The *members* or *fields* of a structure are used like the ordinary variables. A structure is more versatile than an array, because it may contain elements of different data types. As we'll see next, we can declare an optional `structure_variable_list` of that type. Don't forget the ; at the end. For example, to store information about a company, we could define the type `company`:

```
struct company
{
    char name[50];
    int start_year;
    int field;
    int tax_num;
```

```
int num_empl;
char addr[50];
float balance;
};
```

Although members of the same type can be declared in the same line, our preference is to declare them separately to make it easier for the reader to read the stored data.

---

The declaration of a structure not followed by a list of variables does not invoke any memory allocation. It just describes the template of the structure.

If a structure tag is declared, we can use that tag to declare variables. For example:

```
struct company c1, c2;
```

The c1 and c2 variables are declared as structures of the type company. Each variable contains its own members. Alternatively, we can declare the variables in the `structure_variable_list`. For example:

```
struct book
{
    char title[100];
    int year;
    float price;
} b1, b2;
```

The b1 and b2 variables are declared as structures of the type book. If we declare all the variables we need in the `structure_variable_list`, the tag (i.e., book) may be omitted. However, to be able to declare variables whenever we need, our suggestion is always to name the structure and use that tag to declare variables. Alternatively, as we'll see in the next section, we can omit the tag and use the `typedef` to create a synonym. The typical practice is to put the structure declarations together with other declarations like function prototypes and macros in a header file and use the `#include` directive to include it, if needed. We'll see such an example later, in the program of Chapter 17. For simplicity, in all the following programs, we'll define each structure type with global scope so that all functions may use it. If it is defined inside a function its scope becomes local and other parts of the program would ignore its existence.

In C, we say that each structure (or union) specifies a separate *namespace* for its members. Therefore, it is allowed to declare ordinary variables with the same names as the members or the tag of a structure. For the same reason, the same member names can be reused in other structures. For example:

```
struct person
{
    char name[50];
    int tax_num;
    char addr[50];
};
```

These three members have no relation with the respective members of the company type. As another example, check the following declarations:

```
int a;  
struct S1 {int a[5]; } s1;  
struct S2 {int *a; } s2;
```

Next, you'll learn that we can write `s1.a[0] = a;` or `s2.a = &a;` or `s2.a = s1.a;` without any problem.

Although a structure type is completed with the right brace `},` a structure can contain a pointer to an instance of itself because C allows the declaration of pointers to incomplete types. For example:

```
struct A  
{  
    struct A *p;  
};
```

When a structure variable is declared, the compiler allocates memory to store its members in the order in which they are declared. For example, the following program displays how many bytes the variable `d` allocates.

```
#include <stdio.h>  
  
struct date  
{  
    int day;  
    int month;  
    int year;  
};  
  
int main(void)  
{  
    struct date d;  
  
    printf("%u\n", sizeof(d));  
    return 0;  
}
```

Since `d` contains three integer members, the allocated memory is  $3 \times 4 = 12$  bytes. However, the allocated size may be larger than the sizes of the members added together. For example, if we change the type of the `day` member from `int` to `char`, the program may display again 12, not 9 as might expected.

This can happen if the compiler requires that each member is stored in an address multiple of some number (typically four). This requirement may be set for a faster access of the structure members. If we assume that the `month` member should be stored in an address multiple of four, the compiler would allocate three more bytes, a "hole," right after the `day` member. In that case, the program would display 12. Holes may exist between members and at the end of the structure, not at the beginning.

We won't get into details, but if you ever write an application that uses many structures and the memory is limited, keep in mind that the order of the member declarations might affect the alignment and create holes. For example:

```
struct test1
{
    char c;
    double d;
    short s;
};

struct test2
{
    double d;
    short s;
    char c;
};
```

Although the members are the same, reordering the members, from the larger to smaller, may save memory. For example, the first structure might allocate more bytes than the second one, i.e., 24 bytes instead of 16, in case the compiler leaves holes to align the members to the larger type (i.e., `double`).

---

To calculate the memory size of a structure variable, always use the `sizeof` operator; don't add the sizes of its members.

Although it is not often used, it is good to know about the `offsetof` macro. It is defined in `stddef.h` and takes two arguments. The first one is the type of the structure and the second one the name of a member. The macro calculates the number of bytes from the beginning of the structure up to the member. For example, consider the latter structure. Since C guarantees that the address of the first member is the same as the address of the entire structure, the statement

`printf("%d\n", offsetof(struct test2, d));` outputs 0. If we replace `d` with `s`, `printf()` outputs 8. Because a structure may contain holes, if you write an application to run in different platforms and member alignment matters, that macro might be useful to check the portability of your program.

## The `typedef` Specifier

The `typedef` specifier is used to create a synonym for an existing data type. For example, the statement:

```
typedef unsigned int size_t;
```

makes `size_t` synonym of the `unsigned int` type. Therefore, the declarations:

`unsigned int i;` and `size_t i;` are equivalent.

The syntax resembles a variable declaration preceded by the **typedef** keyword. The name represents the stated type. For example, the statement:

```
typedef float arr[100];
```

makes arr synonym for an array of 100 **floats**. Pay attention, **typedef** does not create a variable, nor a new type, just a new name. Therefore, the statement:

```
arr arr1;
```

declares the variable arr1 as an array of 100 **floats**. If we ever need to make the type of the array **double**, we just change **float** to **double**. The declaration of the variable remains the same.

In the same way, we can use **typedef** to create an alias for a structure type. For example:

```
typedef struct Mybook
{
    char title[100];
    int year;
    float price;
} book;
```

Mybook is a tag and we must add the word **struct** when used. On the other hand, because the **typedef** name, that is, book, is a synonym for the structure type, it is not allowed to add the word **struct**. For example:

```
book b1, b2;
```

Notice that we can use the same identifier for both the structure tag and the **typedef** name; their meaning is still different though. Also, we can omit the tag and use the **typedef** name to declare variables. For example:

```
typedef struct
{
    char title[100];
    int year;
    float price;
} book;
```

**typedef** is used a lot with structures and unions to save writing the word **struct** or **union**, when declaring variables. Although this is very common, we don't use it so much. We prefer to clearly show the type of the variable; why hide it? It is easier and faster to read the code; why waste time searching for hidden types? The most times we are using **typedef** is either if the same type appears a lot in the code or we are using a data structure (e.g., list) with elements structures (or unions).

If the data types of your application depend on the running platform, **typedef** might be useful. For example:

```
typedef short int size_i;
```

the type of all **size\_i** variables will be **short int**. In case these variables must be **int** in another platform, just change **short int** to **int** and they become **int**.

Another common use of **typedef** is to create synonyms for complex types. For example:

```
typedef int (*pcpy)(const char *s1, const char *s2, int len);
```

pcpy is a synonym for a pointer to a function that returns an integer and takes two arguments of type **const char \*** and an integer argument. Now, it is much easier to declare variables of that type. For example:

```
pcpy p_cpy1, p_cpy2, p_cpy3;
```

is not only easier to write but also easier to read.

## Initializing a Structure

The most common method to access a structure member is to write the name of the structure variable followed by the **.** operator and the name of the member. For example, the following program displays the values of the b members:

```
#include <stdio.h>
#include <string.h>

struct book
{
    char title[100];
    int year;
    float price;
};

int main(void)
{
    struct book b;

    strcpy(b.title, "Literature");
    b.year = 2016;
    b.price = 10.85;
    printf("%s %d %.2f\n", b.title, b.year, b.price);
    return 0;
}
```

*Besides the **.** operator, you'll learn next how to use the **->** operator to access the members of a structure.*

A structure variable can be initialized when it is declared. The declaration is followed by a list of initializers enclosed in braces, separated with a comma. The list must appear in the same order as the members of the structure and should match their data types. As with arrays, the values must be constant expressions, variables are not allowed. Also, there must not be more initializers than members. Consider the following declaration:

```
struct book b = {"Literature", 2020, 10.85};
```

The value of b.title becomes "Literature", b.year becomes 2016 and b.price becomes 10.85.

As with arrays, any unassigned member is given the value 0. For example, with the declaration

```
struct book b = {"Literature"};
```

the values of year and price members are set to 0. Similarly, with the declaration:

```
struct book b = {0};
```

all members are set to 0.

A structure with automatic storage duration can be initialized by assignment. For example:

```
struct book b2 = b1;
```

Notice that the initializer (i.e., b1) can be any expression of the proper type. For example, we could replace b1 with a function that returns a structure of type book or write \*p where p is a pointer to a structure of that type. We'll talk about operations on structures in a short.

Finally, a structure variable declared in the structure\_variable\_list can be initialized at the same time. For example:

```
struct book
{
    char title[100];
    int year;
    float price;
} b = {"Literature", 2016, 10.8};
```

Let's see another example. The following program defines the structure type computer with members: manufacturer, model, processor, and price. The program uses the structure to read the characteristics of a computer and display them.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

struct computer
{
    /* Assume that 50 characters are enough to hold the
characteristics. */
    char comp[50];
    char model[50];
    char cpu[50];
    float prc;
};
```

```

int main(void)
{
    struct computer pc;

    printf("Enter company: ");
    read_text(pc.comp, sizeof(pc.comp), 1);

    printf("Enter model: ");
    read_text(pc.model, sizeof(pc.model), 1);

    printf("Enter cpu: ");
    read_text(pc.cpu, sizeof(pc.cpu), 1);

    printf("Enter price: ");
    scanf("%f", &pc.prc);

    printf("\nC:%s M:%s CPU:%s P:%.2f\n", pc.comp, pc.model, pc.cpu,
pc.prc);
    return 0;
}

```

## Pointer to a Structure Member

A pointer to a structure member is used like an ordinary pointer. For example, the following program uses pointers to display the values of the b members:

```

#include <stdio.h>
#include <string.h>

struct book
{
    char title[100];
    int year;
    float price;
};

int main(void)
{
    char *ptr1;
    int *ptr2;
    float *ptr3;
    struct book b;

    strcpy(b.title, "Literature");
    b.year = 2016;
    b.price = 10.8;

    ptr1 = b.title;
    ptr2 = &b.year;
    ptr3 = &b.price;

    printf("%s %d %.2f\n", ptr1, *ptr2, *ptr3);
    return 0;
}

```

To make a pointer variable point to a structure member, its type should be compatible with the type of the member. For example, since the type of the title member is `char`, `ptr1` is declared as `char*`. As you know by now, the reason we didn't add the `&` operator before the name of `b` in the `ptr1 = b.title;` is that the name of the array is used as a pointer.

## Structure Operations

Although we cannot use the `=` operator to copy one array into another, we can use it to copy one structure into another. The structures must be of the same type. Consider the following program:

```
#include <stdio.h>

struct student
{
    int code;
    float grd;
};

int main(void)
{
    struct student s1, s2;

    s1.code = 1234;
    s1.grd = 6.7;
    s2 = s1; /* Copy structure. */
    printf("C:%d G:%.2f\n", s2.code, s2.grd);
    return 0;
}
```

The statement `s2 = s1;` copies `s1` members into `s2`. It is equivalent to:

```
s2.code = s1.code;
s2.grd = s1.grd;
```

Notice that if `s1` and `s2` were not variables of the same structure type, the statement `s2 = s1;` wouldn't compile even if the two types contained the same exact members.

Notice also that if a structure contains an array (e.g., `name`), although it is not allowed to write `s2.name = s1.name;`, once the structure is copied (e.g., `s2 = s1`), the `s1` array is copied to `s2`. Furthermore, if it contains a pointer member, both pointers will point to the same place after the copy. We'll see such examples in the next two sections.

Besides assignment, no other operation can be performed on entire structures. For example, the operators `==` and `!=` cannot be used to test whether two structures are equal or not. Therefore, you are not allowed to write:

```
if(s1 == s2) or if(s1 != s2).
```

To test whether two structures are equal, we must compare their members one by one. For example:

```
if((s1.code == s2.code) && (s1.grd == s2.grd))
```

In the preface, we've told you that you might need some luck when programming. Here is an example. One time, I had to compare two structures. The structure didn't contain members whose comparison could be dangerous, like floating point members, pointers, or arrays of characters. Because the structure contained many members, I decided to use `memcmp()` (we'll see it in Chapter 14) to save writing, instead of comparing them one by one. I've forgotten, though, that holes, whose values are undetermined, might be added to align the structure. It means that the comparison of the two structures, even if the corresponding members have identical values, could fail because of different values in the holes. Lucky for me, the holes had the same values. Even luckier, I didn't have to transfer the application in another platform. Imagine the case, the comparison to work in my platform and fail in customer's, because of different holes there. I'm scared just by the thought that I'd have to look for that bug...*Gimme Shelter* from *Rolling Stones*.

## Structure Containing Arrays

Since a structure may contain any type of data, it may contain one or more arrays. For example:

```
#include <stdio.h>
#include <string.h>

struct student
{
    char name[50];
    float grd[2];
};

int main(void)
{
    struct student s1, s2;

    strcpy(s1.name, "somebody");
    s1.grd[0] = 8.5;
    s1.grd[1] = 7.5;
    printf("%s %c %c\n", s1.name, s1.name[0], *s1.name);

    s2 = s1;
    printf("%s %.1f %.1f\n", s2.name, s2.grd[0], s2.grd[1]);
    return 0;
}
```

As you guessed, an array member is treated just like an ordinary array. For example, the statement `strcpy(s1.name, "somebody");` copies the string "somebody" into `name`. The value of `s1.name[0]` becomes 's', `s1.name[1]` becomes 'o', and so on.

When using pointer arithmetic to handle the elements of an array, the `*` operator must precede the name of the structure. For example, since `s1.name` can be used as a pointer to its first character, `*s1.name` is equal to `s1.name[0]`, `*(s1.name+1)` equal to `s1.name[1]`, and so on. As with ordinary arrays, the parentheses must be present for reasons of priority. The assignment `s2 = s1` copies the arrays. Therefore, the program outputs:

```
somebody s s
somebody 8.5 7.5
```

## Structure Containing Pointers

A structure may contain one or more pointer members. A pointer member is treated just like an ordinary pointer. For example:

```
#include <stdio.h>

struct student
{
    char *name;
    float *avg_grd;
};

int main(void)
{
    float grd = 8.5;
    struct student s1, s2;

    s1.name = "somebody";
    s1.avg_grd = &grd;
    printf("%s %.2f\n", s1.name+3, *s1.avg_grd);

    s2 = s1;
    grd = 3.4;
    printf("%s %.2f\n", s2.name, *s2.avg_grd);
    return 0;
}
```

With the statement `s1.name = "somebody";` the compiler allocates memory to store the string "somebody" and then name points to that memory. The statement `s1.avg_grd = &grd;` makes avg\_grd to point to the address of grd. To access the content of the memory pointed to by a pointer member, the `*` operator must precede the name of the structure. Therefore, the program displays: ebody 8.50. The statement `s2 = s1` makes the pointers of s2 to point to the same place as the pointers of s1. Therefore, the program displays somebody 3.40

## Nested Structures

A structure may contain one or more structures. A nested structure must be declared before the declaration of the structure in which it is contained; otherwise, the compiler will produce an error message. For example, in the following program, prod contains the nested structures s\_date and e\_date. s\_date holds the production date and e\_date the expiration date.

```
#include <stdio.h>
#include <string.h>

struct date
{
    int day;
    int month;
    int year;
};
```

```

struct product /* Since the type date is declared, it can be used to
declare nested structures. */
{
    char name[50];
    double price;
    struct date s_date;
    struct date e_date;
};

int main(void)
{
    struct product prod;

    strcpy(prod.name, "product");
    prod.s_date.day = 1;
    prod.s_date.month = 9;
    prod.s_date.year = 2012;

    prod.e_date.day = 1;
    prod.e_date.month = 9;
    prod.e_date.year = 2015;

    prod.price = 7.5;
    printf("The product's life is %d years\n", prod.e_date.year -
prod.s_date.year);
    return 0;
}

```

Notice that in order to access a member of the nested structure, the . operator must be used twice. The program subtracts the respective members and displays the product's life.

Let's see a special case. Suppose that we want to declare two mutually dependent structures, the structure A, which contains a structure of type B, and the structure B, which contains a structure of type A. If we write something like:

```

struct A
{
    struct B b;
    ...
};

struct B
{
    struct A a;
    ...
};

```

the compiler will display an error message for the declaration of b because it has no idea about B and its size. If we reverse the declarations, the compiler will display the same type of error for the member a and the unknown structure A. One way to overcome this situation is to make b pointer (we change both for similarity):

```

struct B; /* Incomplete declaration. */
struct A

```

```
{  
    struct B *b;  
    ...  
};  
  
struct B  
{  
    struct A *a;  
    ...  
};
```

Since the compiler knows how much memory to allocate for a pointer, it is not necessary to know the complete declaration of B, when b is declared. To let the compiler know about the existence of B an incomplete declaration is added. C allows the partial declaration of a structure as long as the complete declaration comes later in the same scope.

## Bit Fields

A structure may contain fields whose length is specified as a number of bits. A bit field is declared like this:

```
data_type field_name : bits_number;
```

The bit fields can be used just like the other members. The type of a bit field may be `int`, `signed int` or `unsigned int`. However, a compiler may support additional types like `char` and `long`. Because some compilers may handle the high order bit of an `int` bit field as a sign bit, while others not, declare the bit fields explicitly as `signed int` or `unsigned int` for portability. Here is an example of a structure with several bit fields:

```
struct person  
{  
    unsigned int sex : 1;  
    unsigned int married : 1;  
    unsigned int children : 4;  
};
```

Since the fields are of the same type, we could write:

`unsigned int sex : 1, married : 1, children : 4;`. Our preference is to use separate lines. Since the size of `sex` and `married` fields is one bit, their values can be either 0 or 1. Since the size of `children` is 4 bits, it can take values between 0 and 15. To store their values,  $1+1+4 = 6$  bits are required. Since their type is `unsigned int`, the compiler allocates four bytes and  $4 \times 8 - 6 = 26$  bits remain unused. To make a note, arrays of bit fields are not allowed.

The main advantage of using bit fields is to save memory space. For example, if we were not using bit fields, the compiler would allocate 12 bytes instead of 4. Therefore, if we were using the `person` structure to store the data of 200.000 persons, we'd save 1.600.000 bytes.

---

For better memory saving, declare all bit fields at the beginning of the structure, not among the other members.

Bit fields are often used for the description and access of information encoded in bit level, such as the hardware registers. For example, the information of the register depicted in Figure 13.1 could be encoded like this:

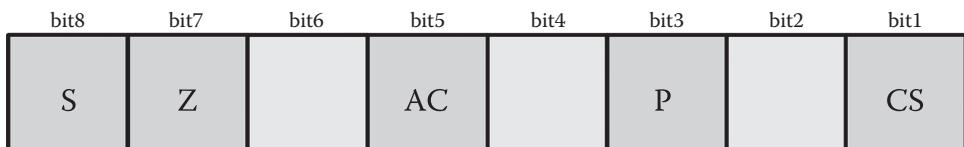
```
struct Flags_Reg
{
    unsigned int cs : 1;
    unsigned int p : 1;
    unsigned int ac : 1;
    unsigned int z : 1;
    unsigned int s : 1;
};
```

An alternative way to handle the register value is to use bit operators as we did in C.4.13. Using the previous structure, it is much easier to handle the register and assign values to its fields. The code becomes clearer and easier to read. However, as we'll see later, if we want our program to be portable, it'd be better to use bit operators, so that it won't depend on how the compiler allocates the bits.

A bit field may be unnamed. Unnamed fields are typically used for padding. For example, to represent the exact form of the previous register, we could add three unnamed fields:

```
struct Flags_Reg
{
    unsigned int cs : 1;
    unsigned int : 1;
    unsigned int p : 1;
    unsigned int : 1;
    unsigned int ac : 1;
    unsigned int : 1;
    unsigned int z : 1;
    unsigned int s : 1;
};
```

The standard says that adjacent bit fields are packed into storage units. The size of a storage unit depends on the implementation. Typical values are 8, 16 and 32 bits. If there is not enough room for a bit field, the compiler may split it between storage units or skip to the beginning of the next unit; it depends on the implementation. To force the compiler to store a bit field at the beginning of the next unit, a bit field of zero length may be used. For example, in the following structure, if the compiler uses a storage unit of 32 bits, it will allocate 3 bits for the f field, skip the next 29 bits to the next unit, and allocate 4 more bits for the k field.



**FIGURE 13.1**

Register layout.

```
struct Test
{
    unsigned int f : 3;
    unsigned int : 0;
    unsigned int k : 4;
};
```

When you assign a value to a bit field, make sure that this value fits in the bit field. For example, suppose that the value 2 is assigned to the married field:

```
struct person p;
p.married = 2; /* Wrong assignment. */
```

If we assume that the bit fields are allocated from right to left, married becomes 0, not 2, because 2 is encoded in two bits (10). Notice that the order in which bit fields are allocated, right to left or left to right, is also implementation dependent.

---

If your program depends on the order the bits are stored into memory and you plan to run it in different platforms, be aware that the allocation order may differ.

Since the memory of a bit field is not allocated like the ordinary variables, you are not allowed to apply the & operator. For example, this statement is wrong:

```
unsigned int *ptr = &p.married;
```

## Pointer to Structure

A pointer to a structure is used like a pointer to an ordinary variable. Consider the following program:

```
#include <stdio.h>
#include <string.h>

struct student
{
    char name[50];
    float grd;
};

int main(void)
{
    struct student *ptr, stud;

    ptr = &stud;
    strcpy((*ptr).name, "somebody");
    (*ptr).grd = 6.7;

    printf("N: %s G: = %.2f\n", stud.name, stud.grd);
    return 0;
}
```

ptr is declared as a pointer to a structure variable of type student. The statement ptr = &stud; makes ptr point to the address of stud. In particular, it points to the address of

its first member. Since `ptr` points to the address of `stud`, `*ptr` is equivalent to `stud`. By using the `.` operator, we can access its members. The expression `*ptr` must be enclosed in parentheses because the precedence of the `.` operator is higher than `*`.

Alternatively, we can use the `->` operator to access the members of a structure. The `->` operator consists of the `-` and `>` characters. Here is another version of the previous program:

```
int main(void)
{
    struct student *ptr, stud;

    ptr = &stud;
    strcpy(ptr->name, "somebody");
    ptr->grd = 6.7;

    printf("N: %s G: %.2f\n", ptr->name, ptr->grd);
    return 0;
}
```

The expression `ptr->name` is equivalent to `(*ptr).name` and the expression `ptr->grd` is equivalent to `(*ptr).grd`. Therefore, both programs output:

```
N: somebody G: 6.70
```

When using a pointer variable to access the members of a structure, we find it simpler to use the `->` operator.

---

Pointer arithmetic works the same as for the ordinary pointers. For example, if a structure pointer is increased by one, its value will be increased by the size of the structure it points to.

## Array of Structures

An array of structures is an array whose elements are structures. Its typical use is to store information about many items, like the data of a company's employees or students' data or the products of a warehouse. In fact, an array of structures may be used as a simple database. For example, with the statement:

```
struct student stud[100];
```

the variable `stud` is declared as an array of 100 structures of type `student`. An array of structures can be initialized when declared. For example, given the declaration:

```
struct student
{
    char name[50];
    int code;
    float grd;
};
```

An initialization example is:

```
struct student stud[3] = { {"nick sterg", 150, 7.3},  
                           {"john theod", 160, 5.8},  
                           {"peter karast", 170, 6.7} };
```

The initialization is done in much the same way as initializing a two-dimensional array. The initial values for the members of each structure are enclosed in braces. For example, stud[0].name becomes "nick sterg", stud[1].code becomes 160 and stud[2].grd becomes 6.7. The inner braces around each structure can be omitted; however, we prefer to use them to make clearer the initialization of each structure. As usual, the compiler will compute the number of elements if the [ ] is left empty.

We can declare and initialize an array of structures together with the structure declaration. For example:

```
struct student  
{  
    char name[50];  
    int code;  
    float grd;  
} stud[] = { {"nick sterg", 150, 7.3},  
            {"john theod", 160, 5.8},  
            {"peter karast", 170, 6.7} };
```

As with the ordinary arrays, the unassigned members are set to 0. For example, if we write:

```
struct student stud[100] = { 0 };
```

the members of all structures are initialized to 0. For example, the following program stores the data of 100 students to an array of students of type student:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
#define SIZE 100  
  
int read_text(char str[], int size, int flag);  
  
struct student  
{  
    char name[50];  
    int code;  
    float grd;  
};  
  
int main(void)  
{  
    int i;  
    struct student stud[SIZE];
```

```

for(i = 0; i < SIZE; i++)
{
    printf("\nEnter name: ");
    read_text(stud[i].name, sizeof(stud[i].name), 1);

    printf("Enter code: ");
    scanf("%d", &stud[i].code);

    printf("Enter grade: ");
    scanf("%f", &stud[i].grd);

    printf("\nN: %s C: %d G: %.2f\n", stud[i].name, stud[i].code,
stud[i].grd);
    getchar(); /* Get the new line character stored in stdin,
after the grade is entered. */
}
return 0;
}

```

As with ordinary arrays, we can use pointer notation to access the elements. For example, `*stud` is equivalent to `stud[0]`, `*(stud+1)` is equivalent to `stud[1]`, and so on. Therefore, to access the `code` member of the third student, both expressions `stud[2].code` and `(*(stud+2)).code` are acceptable. The parentheses in the second case are required for reasons of priority. As with ordinary arrays, our preference is to use array subscripting to get a more readable code.

---

## Exercises

**C.13.1** Modify the previous program and use a pointer to read and display the students' data. Don't declare the `i` variable.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

#define SIZE 100

struct student
{
    char name[50];
    int code;
    float grd;
};

int main(void)
{
    struct student *ptr, stud[SIZE];
}

```

```

    for(ptr = stud; ptr < stud+SIZE; ptr++) /* ptr is increased to
point to the next structure. */
{
    printf("\nEnter name: ");
    read_text(ptr->name, sizeof(ptr->name), 1);

    printf("Enter code: ");
    scanf("%d", &ptr->code);

    printf("Enter grade: ");
    scanf("%f", &ptr->grd);

    printf("\nN: %s C: %d G: %.2f\n", ptr->name, ptr->code,
ptr->grd);
    getchar();
}
return 0;
}

```

**C.13.2** Define the structure type `city` with members: city name, country name, and population. Write a program that uses this type to read the data of 100 cities and store them in an array. Then, the program should read the name of a country and a number and display the cities of that country whose population is greater than the input number.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100

int read_text(char str[], int size, int flag);

struct city
{
    char name[50];
    char cntry[50];
    int pop;
};

int main(void)
{
    char cntry[50];
    int i, pop, flag;
    struct city cities[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("\nCity: ");
        read_text(cities[i].name, sizeof(cities[i].name), 1);

        printf("Country: ");
        read_text(cities[i].cntry, sizeof(cities[i].cntry), 1);
    }
}

```

```

        printf("Population: ");
        scanf("%d", &cities[i].pop);

    }
    printf("\nEnter country to search: ");
    read_text(cntry, sizeof(cntry), 1);

    printf("Population: ");
    scanf("%d", &pop);

    flag = 0;
    for(i = 0; i < SIZE; i++)
    {
        if((strcmp(cities[i].cntry, cntry) == 0) && (cities[i].pop >
pop))
        {
            flag = 1;
            printf("C: %s P: %d\n", cities[i].name, cities[i].
pop);
        }
    }
    if(flag == 0)
        printf("\nNone city is found\n");
    return 0;
}

```

## Structure as Function Argument

A structure variable can be passed to a function just like any other variable, either the variable itself or its address. Consider the following program:

```

#include <stdio.h>
#include <string.h>

struct student
{
    char name[50];
    int code;
    float grd;
};

void test(struct student stud_1);

int main(void)
{
    struct student stud = {"somebody", 20, 5};

    test(stud); /* According to the function declaration, it must be
called with an argument of type student. */
    printf("N:%s C:%d G:%.2f\n", stud.name, stud.code, stud.grd);
    return 0;
}

```

```

void test(struct student stud_1)
{
    strcpy(stud_1.name, "new_name");
    stud_1.code = 30;
    stud_1.grd = 7;
}

```

When `test()` is called, the values of `stud` members are copied to the respective members of `stud_1`. Since `stud_1` and `stud` are stored in different memory locations, any change in the members of `stud_1` don't affect the members of `stud`. As you already know by now, even if we were using the name `stud` instead of `stud_1`, the result would be the same because they are different variables regardless of naming. Therefore, the program displays: N:somebody C:20 G:5.00

On the other hand, when the address is passed the function may change the values of the members. For example, let's change `test()`, in order to modify the members of `stud`:

```

#include <stdio.h>
#include <string.h>

struct student
{
    char name[50];
    int code;
    float grd;
};

void test(struct student *ptr);

int main(void)
{
    struct student stud = {"somebody", 20, 5};

    test(&stud);
    printf("N:%s C:%d G:%.2f\n", stud.name, stud.code, stud.grd);
    return 0;
}

void test(struct student *ptr)
{
    strcpy(ptr->name, "new_name");
    ptr->code = 30;
    ptr->grd = 7;
}

```

When `test()` is called, we have `ptr = &stud`. Therefore, since `ptr` points to the address of `stud`, the function may change the values of its members. As a result, the program displays N:new\_name C:30 Grade:7.00

When a structure is passed to a function, its members are copied to the members of the respective parameter. Notice that if the structure is large or the function is called many times, this copy operation may increase memory requirements and add a time overhead. On the other hand, when the address of the structure is passed to the function, the members are not copied.

For better performance, pass the address of the structure, not the structure itself, even if the function won't change the values of its members. For the same reason, have the function return a pointer to a structure instead of the entire structure.

To prevent the function from changing the values of the members, declare the pointer as `const`. For example:

```
void test(const struct student *ptr);
```

now, `test()` cannot modify the members of the structure pointed to by `ptr`.

## Exercises

C.13.3 What is the output of the following program?

```
#include <stdio.h>

struct student
{
    char name[50];
    int code;
    float grd;
};

struct student *test(void), stud_1;

int main(void)
{
    struct student stud = {"somebody", 1111, 7.5}, *ptr = &stud;

    *ptr = *test();
    printf("%s %d %d\n", ptr->name, ptr->code, ptr->grd);
    return 0;
}

struct student *test(void)
{
    return &stud_1;
}
```

**Answer:** Since `test()` returns the address of `stud_1`, `*test()` is equivalent to `stud_1`. Since `ptr` points to the address of `stud`, `*ptr` is equivalent to `stud`. Therefore, the expression `*ptr = *test();` is equivalent to `stud = stud_1;`. Since the global variable is initialized with zero values, the program outputs 0 0.

C.13.4 Define the structure type `time` with members: hours, minutes, and seconds. Write a function that takes as parameter a pointer to an integer and converts that integer to hours, minutes, and seconds. These values should be stored into the members of a structure of

type time, and the function should return that structure. Write a program that reads an integer, calls the function, and displays the members of the returned structure.

```
#include <stdio.h>

struct time
{
    int hours;
    int mins;
    int secs;
};

struct time mk_time(int *ptr);

int main(void)
{
    int secs;
    struct time t;

    printf("Enter seconds: ");
    scanf("%d", &secs);

    t = mk_time(&secs);
    printf("\nH:%d M:%d S:%d\n", t.hours, t.mins, t.secs);
    return 0;
}

struct time mk_time(int *ptr)
{
    struct time tmp;

    tmp.hours = *ptr/3600;
    tmp.mins = (*ptr%3600)/60;
    tmp.secs = *ptr%60;
    return tmp;
}
```

**Comments:** As said, when you want to save time and space, do not return an entire structure. This exercise is just an example to show you how to return a structure. For example, we could change the return type to **void** and pass the address of t to mk\_time(). Thus, the function modifies a structure of the calling function instead of declaring and returning a new structure.

```
void mk_time(int *ptr, struct time *tmp)
{
    tmp->hours = *ptr/3600;
    tmp->mins = (*ptr%3600)/60;
    tmp->secs = *ptr%60;
}
```

C.13.5 Define the structure type book with members: title, code, and price. Write a function that takes as parameters two pointers to structures of type book and return the structure with the higher price. Examine the case of the same prices and write a

related message. Write a program that reads and stores the data of 10 books into an array of structures. Then, the program should read two numbers in [1,10] that represent two structure indexes in the array. The program should check the validity of the input numbers and use the function to display the data of the structure with the higher price.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 10

struct book
{
    char title[50];
    int code;
    float prc;
};

struct book max_prc(struct book *b1, struct book *b2);
int read_text(char str[], int size, int flag);

int main(void)
{
    int i, j;
    struct book tmp, b[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("\nEnter title: ");
        read_text(b[i].title, sizeof(b[i].title), 1);

        printf("Enter code: ");
        scanf("%d", &b[i].code);

        printf("Enter price: ");
        scanf("%f", &b[i].prc);

        getchar();
    }

    do
    {
        printf("Enter two book numbers [1-%d]: ", SIZE);
        scanf("%d%d", &i, &j);
    } while((i == j) || i > SIZE || j > SIZE || i < 1 || j < 1);

    i--; /* Subtract 1 since indexing starts from 0. */
    j--;
    tmp = max_prc(&b[i], &b[j]);
    if(tmp.prc != 0)
        printf("\nN: %s C: %d P: %.2f\n", tmp.title, tmp.code, tmp.pr
    else
}
```

```

        printf("\nBoth books have the same price: %.2f\n", b[i]);
prc);
    return 0;
}

struct book max_prc(struct book *b1, struct book *b2)
{
    struct book tmp = {0};

    if(b1->prc > b2->prc)
        return *b1;
    else if(b1->prc < b2->prc)
        return *b2;
    else
        return tmp;
}

```

**C.13.6** Define the structure type product with members: name, code, and price. Write a function that takes as parameters an array of such structures, the number of the products and a product code. The function should check if a product's code is equal to that number, and if so, it should return a pointer to the respective structure, otherwise NULL. Write a program that uses the type product to read the data of 100 products. The program should store the products that cost more than \$20 in an array of such structures. Then, the program should read an integer and use the function to display the product's name and price, if found.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100

struct product
{
    char name[50];
    int code;
    float prc;
};

struct product *find_prod(struct product prod[], int num_prods, int code); /* Declare a function that takes as parameters an array of structures and two integers and returns a pointer to a structure of type product. */
int read_text(char str[], int size, int flag);

int main(void)
{
    int i, cnt, code;
    float prc;
    struct product *ptr, prod[SIZE];
    cnt = 0;
    for(i = 0; i < SIZE; i++)

```

```

{
    printf("\nPrice: ");
    scanf("%f", &prc);

    if(prc > 20)
    {
        prod[cnt].prc = prc;

        getchar();

        printf("Name: ");
        read_text(prod[cnt].name, sizeof(prod[cnt].name), 1);

        printf("Code: ");
        scanf("%d", &prod[cnt].code);

        cnt++;
    }
}

if(cnt == 0)
{
    printf("None product is stored\n");
    return 0;
}

printf("\nEnter code to search: ");
scanf("%d", &code);

ptr = find_prod(prod, cnt, code);
if(ptr == NULL)
    printf("\nNo product with code = %d\n", code);
else
    printf("\nN: %s C: %d P: %.2f\n", ptr->name, code,
ptr->prc);
return 0;
}

struct product *find_prod(struct product prod[], int num_prods, int code)
{
    int i;

    for(i = 0; i < num_prods; i++)
    {
        if(prod[i].code == code)
            return &prod[i]; /* If the code is found, the
function terminates and the address of that structure is returned. */
    }
    return NULL; /* If the code reaches that point, it means that the
product is not found and the function returns NULL. */
}

```

**C.13.7** Define the structure type coord with members the coordinates of a point (e.g., x and y). Define the structure type rectangle with members two structures of type coord (e.g., point\_A and point\_B). Write a function that takes as parameters the two endpoints of a

rectangle's diagonal. Each endpoint should be a structure of type coord. The function should calculate and return the area of the rectangle. Write a program that uses the type rectangle to read the coordinates of a rectangle's diagonal and uses the function to display its area.

```
#include <stdio.h>

struct coord
{
    double x;
    double y;
};

struct rectangle
{
    struct coord point_A; /* First diagonal point. */
    struct coord point_B; /* Second diagonal point. */
};

double rect_area(struct coord *c1, struct coord *c2);

int main(void)
{
    struct rectangle rect;

    printf("Enter the x and y coords of the first point: ");
    scanf("%lf%lf", &rect.point_A.x, &rect.point_A.y);

    printf("Enter the x and y coords of the second point: ");
    scanf("%lf%lf", &rect.point_B.x, &rect.point_B.y);

    printf("Area: %f\n",
    rect_area(&rect.point_A, &rect.point_B));
    return 0;
}

double rect_area(struct coord *c1, struct coord *c2)
{
    double base, height;

    if(c1->x > c2->x)
        base = c1->x - c2->x;
    else
        base = c2->x - c1->x;

    if(c1->y > c2->y)
        height = c1->y - c2->y;
    else
        height = c2->y - c1->y;

    return base*height; /* Return the area. */
}
```

C.13.8 For a complex number  $z$ , we have  $z = a + bj$ , where  $j$  is the imaginary unit,  $a$  is the real part, and  $b$  is the imaginary part. Define the structure type `complex` with members the float numbers `re` and `im`, which represent the real and imaginary parts of a complex number. Write a function that takes as parameters two structures of type `complex` and a character, which represents the sign of a math operation. The function should perform the math operation and return the result as a structure of type `complex`. Write a program that uses the structure `complex` to read two complex numbers and a math sign (+, -, \*, /) and uses the function to display the result of the math operation. Remind that if  $z_1 = a + bj$  and  $z_2 = c + dj$ , we have:

$$z = z_1 + z_2 = (a + c) + (b + d)j$$

$$z = z_1 - z_2 = (a - c) + (b - d)j$$

$$z = z_1 \times z_2 = (ac - bd) + (bc + ad)j$$

$$z = \left( \frac{z_1}{z_2} \right) = \left( \frac{ac + bd}{c^2 + d^2} \right) + \left( \frac{bc - ad}{c^2 + d^2} \right) j$$

```
#include <stdio.h>

struct complex
{
    double re; /* The real part of the complex number. */
    double im; /* The imaginary part of the complex number. */
};

struct complex operation(struct complex a1, struct complex a2, char sign);

int main(void)
{
    char sign;
    struct complex z1, z2, z;

    printf("Enter real and imaginary part of the first complex number:\n");
    scanf("%lf%lf", &z1.re, &z1.im);
    printf("z1 = %f%+fj\n", z1.re, z1.im);

    printf("Enter real and imaginary part of the second complex\nnumber: ");
    scanf("%lf%lf", &z2.re, &z2.im);
    printf("z2 = %f%+fj\n", z2.re, z2.im);

    printf("Enter sign (+, -, *, /): ");
    scanf(" %c", &sign); /* We add a space before %c to skip the new
line character stored in stdin. */
    if(sign == '+' || sign == '-' || sign == '*' || sign == '/')
        sign = sign - '0';
}
```

```

    {
        if(sign == '/') && z2.re == 0 && z2.im == 0)
            printf("Division by zero is not allowed\n");
        else
        {
            z = operation(z1, z2, sign);
            printf("z = z1 %c z2 = %f%+fj\n", sign, z.re, z.im);
        }
    }
    else
        printf("Wrong sign\n");
    return 0;
}

struct complex operation(struct complex a1, struct complex a2, char sign)
{
    double div;
    struct complex a;

    switch(sign)
    {
        case '+':
            a.re = a1.re + a2.re;
            a.im = a1.im + a2.im;
        break;

        case '-':
            a.re = a1.re - a2.re;
            a.im = a1.im - a2.im;
        break;

        case '*':
            a.re = (a1.re*a2.re) - (a1.im*a2.im);
            a.im = (a1.im*a2.re) + (a1.re*a2.im);
        break;

        case '/':
            div = (a2.re*a2.re) + (a2.im*a2.im);
            a.re = ((a1.re*a2.re) + (a1.im*a2.im))/div;
            a.im = ((a1.im*a2.re) - (a1.re*a2.im))/div;
        break;
    }
    return a;
}

```

**Comments:** Notice that we're using the flag + in printf() to display the imaginary part of the complex number. Recall from Chapter 2 that the flag + prefixes the positive values with +. The reason we didn't pass the addresses of the structures to operation() is to show you an example of a function with parameters structures.

**C.13.9** Define the structure type student with members: name, code, and grade. Write a program that uses this type to read the data of 100 students and store them in an array of such structures. If the user enters -1 for grade, the insertion of data should terminate. Write a function to sort the structures in grade ascending order and another function to

display the data of the students who got a better grade than the average grade of all students. The program should call those two functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100

struct student
{
    char name[50];
    int code;
    float grd;
};

void sort_by_grade(struct student studs[], int num_studs);
void show_students(struct student studs[], int num_studs, float avg_grd);
int read_text(char str[], int size, int flag);

int main(void)
{
    int i;
    float sum_grd;
    struct student studs[SIZE];

    sum_grd = 0;
    for(i = 0; i < SIZE; i++)
    {
        printf("\nGrade [0-10]: ");
        scanf("%f", &studs[i].grd);
        if(studs[i].grd == -1)
            break;

        sum_grd += studs[i].grd;
        getchar();

        printf("Name: ");
        read_text(studs[i].name, sizeof(studs[i].name), 1);

        printf("Code: ");
        scanf("%d", &studs[i].code);
    }
    if(i == 0)
        return 0;
    sort_by_grade(studs, i); /* Sort the structures in grade ascending
order. The variable i specifies the number of students. */
    show_students(studs, i, sum_grd/i); /* The last argument is the
average grade of all students. */
    return 0;
}

void sort_by_grade(struct student studs[], int num_studs)
{
```

```

int i, j;
struct student temp;

for(i = 0; i < num_studs-1; i++)
{
    /* In each iteration, the grd member is compared with the
others. If it is less, the structures are swapped. */
    for(j = i+1; j < num_studs; j++)
    {
        if(studs[i].grd > studs[j].grd)
        {
            temp = studs[i];
            studs[i] = studs[j];
            studs[j] = temp;
        }
    }
}

void show_students(struct student studs[], int num_studs, float avg_grd)
{
    int i;

    for(i = 0; i < num_studs; i++)
        if(studs[i].grd > avg_grd)
            printf("N: %s C: %d G: %f\n", studs[i].name,
studs[i].code, studs[i].grd);
}

```

**C.13.10** Image-editing programs often need to rotate an image by  $90^\circ$ . An image can be treated as a two-dimensional array whose elements represent the pixels of the image. For example, the rotation of the original image (e.g.,  $p[M][N]$ ) to the right produces a new image (e.g.,  $r[M][N]$ ), as shown here:

$$p = \begin{bmatrix} p_{0,0} & p_{0,1} & \dots & p_{0,N-2} & p_{0,N-1} \\ p_{1,0} & p_{1,1} & \dots & p_{1,N-2} & p_{1,N-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ p_{M-2,0} & p_{M-2,1} & \dots & p_{M-2,N-2} & p_{M-2,N-1} \\ p_{M-1,0} & p_{M-1,1} & \dots & p_{M-1,N-2} & p_{M-1,N-1} \end{bmatrix} \quad r = \begin{bmatrix} p_{M-1,0} & p_{M-2,0} & \dots & p_{1,0} & p_{0,0} \\ p_{M-1,1} & p_{M-2,1} & \dots & p_{1,1} & p_{0,1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ p_{M-1,N-2} & p_{M-2,N-2} & \dots & p_{1,N-2} & p_{0,N-2} \\ p_{M-1,N-1} & p_{M-2,N-1} & \dots & p_{1,N-1} & p_{0,N-1} \end{bmatrix}$$

In particular, the first row of the original image becomes the last column of the new image, the second row becomes the last but one column, up to the last row, which becomes the first column. For example, the image:

$$p = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix} \text{ is transformed to: } r = \begin{bmatrix} 11 & 6 & 1 \\ 12 & 7 & 2 \\ 13 & 8 & 3 \\ 14 & 9 & 4 \\ 15 & 10 & 5 \end{bmatrix}$$

The color of each pixel follows the RGB color model, in which the red, green, and blue colors are mixed together to reproduce a wide range of colors. The color is expressed as an RGB triplet ( $r, g, b$ ), in which each component value varies from 0 to 255.

Define the structure type `pixel` with three integer members named `red`, `green`, and `blue`. Write a program that creates a two-dimensional image (e.g.,  $3 \times 5$ ) whose elements are structures of type `pixel`. Initialize the members of each structure with random values within  $[0, 255]$ . Then, the program should display the original image, rotate the image by  $90^\circ$  right, and display the rotated image (e.g.,  $5 \times 3$ ).

*Hint:* Use a second array to store the rotated image.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS      3
#define COLS      5

struct pixel /* RGB format (Red-Green-Blue). */
{
    unsigned char red; /* Value in [0, 255]. */
    unsigned char green;
    unsigned char blue;
};

void rotate_right_90(struct pixel img[] [COLS], struct pixel tmp[] [ROWS]);

int main(void)
{
    int i, j;
    struct pixel img[ROWS] [COLS], tmp[COLS] [ROWS];

    srand(time(NULL));
    /* Create random colors. */
    for(i = 0; i < ROWS; i++)
    {
        for(j = 0; j < COLS; j++)
        {
            img[i][j].red = rand()%256;
            img[i][j].green = rand()%256;
            img[i][j].blue = rand()%256;
        }
    }
    printf("**** Original Image ****\n\n");
}
```

```

for(i = 0; i < ROWS; i++)
{
    for(j = 0; j < COLS; j++)
    {
        printf("(%3d,%3d,%3d) ", img[i][j].red, img[i][j].
green, img[i][j].blue);
    }
    printf("\n");
}
rotate_right_90(img, tmp);

printf("\n*** Rotated Image ***\n\n");
for(i = 0; i < COLS; i++)
{
    for(j = 0; j < ROWS; j++)
    {
        printf("(%3d,%3d,%3d) ", tmp[i][j].red, tmp[i][j].
green, tmp[i][j].blue);
    }
    printf("\n");
}
return 0;
}

void rotate_right_90(struct pixel img[] [COLS], struct pixel tmp[] [ROWS])
{
    int i, j, k = 0;

    for(i = ROWS-1; i >= 0; i--)
    {
        for(j = 0; j < COLS; j++)
        {
            tmp[j][i] = img[k][j];
        }
        k++;
    }
}

```

---

## Unions

Like a structure, a union contains one or more members, which may be of different types. The properties of unions are almost identical to the properties of structures; the same operations are allowed as on structures. Their difference is that the members of a structure are stored at *different* addresses, while the members of a union are stored at the *same* address.

### Union Declaration

The declaration of a union type resembles that of a structure, with the **union** keyword used instead of **struct**. When a union variable is declared, the compiler allocates only enough space to store its largest member. Therefore, its members are all stored in the same

memory, overlaying each other. For example, in the following program, the `s` variable allocates 8 bytes because its largest member is of type `double`:

```
#include <stdio.h>

union sample
{
    char ch;
    int i;
    double d;
};

int main(void)
{
    union sample s;
    printf("Size: %u\n", sizeof(s));
    return 0;
}
```

Since the compiler does not allocate memory for each union member, the main application of a union is to save memory space. For example, we often used unions in embedded systems where memory is limited.

---

As with structures, to calculate the memory size of a union variable, use the `sizeof` operator.

Since all union members share the same memory, *only* the first member of a union variable can be initialized when it is declared. For example, the compiler would let us write:

```
union sample s = { 'x' };

but not: union sample s = {'x', 10, 1.23};
```

## Access Union Members

The members of a union are accessed in the same way as the members of a structure. However, since they are all stored in the same memory, only the last assigned member has a meaningful value. For example, the following program assigns a value into an `s` member and displays the other members:

```
#include <stdio.h>

union sample
{
    char ch;
    int i;
    double d;
};

int main(void)
{
    union sample s;
```

```

s.ch = 'a';
printf("%c %d %f\n", s.ch, s.i, s.d);

s.i = 64;
printf("%c %d %f\n", s.ch, s.i, s.d);

s.d = 12.48;
printf("%c %d %f\n", s.ch, s.i, s.d);
return 0;
}

```

First, the program outputs a and nonsense values for the s.i and s.d members. Next, the value 64 is assigned to i. Since this value is stored in the common space, the value of s.ch is overwritten. Therefore, the program displays 64 and nonsense values for the s.ch and s.d. Next, the value 12.48 is assigned to d. The value of s.i is overwritten and the program outputs 12.48 and nonsense values for the s.ch and s.i.

When a union member is assigned with a value, any value previously stored in the last assigned member is overwritten.

Let's have a test. What is the output of the following program?

```

#include <stdio.h>

union test
{
    int a, b, c;
};

int main(void)
{
    union test t;

    t.a = 100;
    t.b = 200;
    t.c = 300;

    printf("%d %d %d\n", t.a, t.b, t.c);
    printf("%d %d %d\n", &t.a, &t.b, &t.c);
    return 0;
}

```

Since the members are of the same type and they are all stored in the same memory, once c becomes 300 that would be the value of the other members as well. Therefore, the first printf() displays three times 300. The second printf() displays three times the common memory address.

A typical use of unions is to save memory when data can be stored in two or more forms but never simultaneously. For example, suppose that we want to store in an array of 100 structures the preferences of men and women. The preferences of men concern favorite

game and movie, while for women, television show and book. We could use a structure person like the following:

```
struct person
{
    char game[50];
    char movie[50];

    char tv_show[50];
    char book[50];
};
```

However, this structure wastes space, since for each person, only two of the members are used to store information. To save space, we can use a union with member structures. For example, consider the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100
#define MAN 0
#define WOMAN 1

struct man
{
    char game[50];
    char movie[50];
};

struct woman
{
    char tv_show[50];
    char book[50];
};

union data
{
    struct man m;
    struct woman w;
};

struct person
{
    int type;
    union data d;
};

int read_text(char str[], int size, int flag);

int main(void)
{
    int i, type;
```

```

struct person pers_arr[SIZE] ;

for(i = 0; i < SIZE; i++)
{
    printf("\nSelection: 0 for man - 1 for woman: ");
    scanf("%d", &type);

    pers_arr[i].type = type;
    getchar();
    if(type == MAN)
    {
        printf("Enter favourite game: ");
        read_text(pers_arr[i].d.m.game,
                  sizeof(pers_arr[i].d.m.game), 1);
        printf("Enter favourite movie: ");
        read_text(pers_arr[i].d.m.movie,
                  sizeof(pers_arr[i].d.m.movie), 1);
    }
    else if(type == WOMAN)
    {
        printf("Enter favourite TV show: ");
        read_text(pers_arr[i].d.w.tv_show,
                  sizeof(pers_arr[i].d.w.tv_show), 1);

        printf("Enter favourite book: ");
        read_text(pers_arr[i].d.w.book,
                  sizeof(pers_arr[i].d.w.book), 1);
    }
}
for(i = 0; i < SIZE; i++)
{
    if(pers_arr[i].type == MAN)
    {
        printf("\nGame: %s\n", pers_arr[i].d.m.game);
        printf("Show: %s\n", pers_arr[i].d.m.movie);
    }
    else if(pers_arr[i].type == WOMAN)
    {
        printf("\nMovie:%s\n", pers_arr[i].d.w.tv_show);
        printf("Book: %s\n", pers_arr[i].d.w.book);
    }
}
return 0;
}

```

The program reads the sex type and stores the user's preferences in the respective union members. To know which type of structure, that is, man or woman, is contained in each person we added the member type. In the second loop, the program checks its value and displays the respective preferences.

---

It is the programmer's responsibility to keep track which member of the union contains a meaningful value. A special member may be added (like the type previously) to indicate what is currently stored in the union.

Another application of unions is to provide multiple views of the same data in order to use the most convenient. For example, in embedded applications, it is often needed to view a register in different ways. Look how we can use a 32-bit register value either in bit level or as a whole:

```
struct bit_val
{
    unsigned int CF : 1;
    unsigned int PF : 1;
    /* ... additional bit fields. */
};

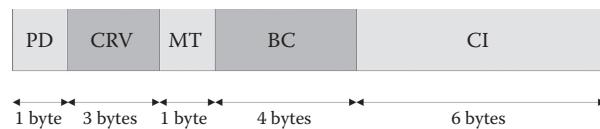
union Flags_Reg
{
    unsigned int i_val;
    struct bit_val b_val;
};
```

Changing the value of `i_val` alters the values of the bit fields and vice versa. If you ever get involved with such applications, don't forget that the order that the bit fields are allocated may differ from system to system.

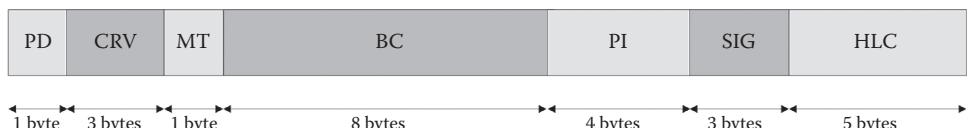
Unions are often used in the development of communication protocols for the encoding of the exchanged messages. The following exercise is a simple example on how we used a union when developing an ISDN protocol. If you ever get involved with that field, it might be useful.



**FIGURE 13.2**  
SETUP message structure.



**FIGURE 13.3**  
ALERTING message structure.



**FIGURE 13.4**  
CONNECT message structure.

## Exercise

C.13.11 In an ISDN network, the SETUP message of the Q.931 signaling protocol is sent by the calling user to the called user to initiate the call establishment procedure. A simplified format of the SETUP message is depicted in Figure 13.2.

The ALERTING message (Figure 13.3) is sent by the ISDN network to the calling user to indicate that the called user is notified for the incoming call (e.g., ring tone).

The CONNECT message (Figure 13.4) is sent by the ISDN network to the calling user to indicate that the called user accepted the call (e.g., picks up the phone).

These messages traverse the ISDN switches along the path between the two users. Each node evaluates the content of the received messages. Let's write a program to simulate this message analysis.

Define the structure type `isdn_msg` with member one union. The members of that union are three structures to represent the SETUP, CONNECT, and ALERTING messages. Write a program that reads a byte stream, uses the previous figures to parse the data, and stores them in the proper union member. To parse the data, the program should check the value of the MT field. Its value in the SETUP, CONNECT, and ALERTING messages is 5, 7, and 1, respectively. To get the byte stream, the program should read 100 positive integers in [0, 255] and store them in an array. If the user enters -1, the insertion of data should terminate.

```
#include <stdio.h>

typedef unsigned char BYTE;

struct header
{
    BYTE pd;
    BYTE crv[3];
    BYTE mt;
};

struct setup
{
    BYTE bc[12];
    BYTE cpn[20];
    BYTE llc[18];
    BYTE hlc[5];
    BYTE dt[8];
};

struct connect
{
    BYTE bc[4];
    BYTE ci[6];
};

struct alerting
{
    BYTE bc[8];
    BYTE pi[4];
};
```

```

    BYTE sig[3];
    BYTE hlc[5];
};

struct isdn_msg
{
    struct header hdr; /* Common header for all messages. */
    union
    {
        struct setup set;
        struct connect con;
        struct alerting alrt;
    };
};

int main(void)
{
    BYTE pkt[100];
    int i;
    struct isdn_msg msg;

    for(i = 0; i < 100; i++)
    {
        printf("Enter octet: ");
        scanf("%d", &pkt[i]);
        if(pkt[i] == -1)
            break;
    }
    msg.hdr.pd = pkt[0];
    for(i = 0; i < 3; i++)
        msg.hdr.crv[i] = pkt[i+1];
    msg.hdr.mt = pkt[4];

    if(msg.hdr.mt == 5) /* SETUP. */
    {
        for(i = 0; i < 12; i++)
            msg.set.bc[i] = pkt[5+i];
        for(i = 0; i < 20; i++)
            msg.set.cpn[i] = pkt[17+i];
        for(i = 0; i < 18; i++)
            msg.set.llc[i] = pkt[37+i];
        for(i = 0; i < 5; i++)
            msg.set.hlc[i] = pkt[55+i];
        for(i = 0; i < 8; i++)
            msg.set.dt[i] = pkt[60+i];
    }
    else if(msg.hdr.mt == 7) /* CONNECT. */
    {
        for(i = 0; i < 4; i++)
            msg.con.bc[i] = pkt[5+i];
        for(i = 0; i < 6; i++)
            msg.con.ci[i] = pkt[9+i];
    }
    else if(msg.hdr.mt == 1) /* ALERTING. */

```

```

    {
        for(i = 0; i < 8; i++)
            msg.alrt.bc[i] = pkt[5+i];
        for(i = 0; i < 4; i++)
            msg.alrt.pi[i] = pkt[13+i];
        for(i = 0; i < 3; i++)
            msg.alrt.sig[i] = pkt[17+i];
        for(i = 0; i < 5; i++)
            msg.alrt.hlc[i] = pkt[20+i];
    }
    return 0;
}

```

---

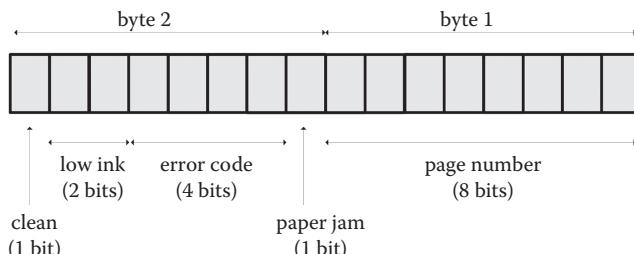
## Unsolved Exercises

**U.13.1** Define the structure type `employee` with members: first name, last name, age, and salary. Write a program that uses this type to read the data of 50 employees and store them in an array. The program should display the last name of the employee with the highest salary and the name of the oldest employee. If more than one employee has the same highest salary or greater age, the program should display the data of the first found.

**U.13.2** Add a `void` function in C.13.7 to return through a pointer parameter to a structure of type `coord` the coordinates of the center of the rectangle. Modify the main program to test the function.

**U.13.3** Figure 13.5 depicts the status register of a printer (16 bits). Use bit fields and define the structure `print_reg` with the five fields of Figure 13.5. Write a program that uses a structure of that type to simulate a printing job of 20 pages, as follows:

1. The low ink field is set to 3, when the 9th page is print and up to the end of the printing job.
2. The error code field is set to 10, only when the 13th page is print.
3. The paper jam field is set to 1, only when the 15th page is print.
4. The clean field is set to 1, only when the last page is print.



**FIGURE 13.5**  
Status register.

Use a loop of 20 iterations to simulate the printing job. Each iteration corresponds to the print of one page. For each printing page, the program should display the value of the status register.

**U.13.4** Define the structure type `circle` with member `the radius` and the type `square` with member `the length`. Write a function that takes as parameters a pointer of type `void*` and an integer parameter. If its value is 0, the pointer points to a structure of type `circle`, otherwise to a structure of type `square`. The function should increase the member of the pointed structure by 5 and return that pointer. Write a program to test the function.

**U.13.5** Define the structure type `student` with members: `name` and `code`. Assume that a classroom can be simulated by a two-dimensional array whose elements are structures of type `student`. Write a program that reads the data of students and stores them in a  $3 \times 5$  array of structures. Then, the program should read the name of a student and his or her code and display his position in the array (i.e., row and column), if registered.

**U.13.6** Define the structure type `student` with members: `name`, `code`, and `grade`. Write a program that uses this type to read the data of 100 students and store them in an array sorted by grade in descending order. The sorting must be done during the data insertion.

**U.13.7** Define the structure type `time` with members: `hours`, `minutes`, and `seconds`. Write a function that takes as parameters two pointers to two structures that represent the start time and the end time of a game and returns the game's duration as a structure. Write a program that uses the type `time` to read the start time of a game and the end time and uses the function to display the game's duration. The user should enter the time in h:m:s format.

**U.13.8** Rename the function `sort_by_grade()` in C.13.9 to `sort()` and add an extra argument. If it is 0, the function should sort the students in ascending order by their codes, if it is 1 by their names, and if it is 2 by their grades. Modify the main program to test the function.

**U.13.9** Define the structure type `publisher` with members: `name`, `address`, and `e-mail`. Define the type `book` with members: `title`, `authors`, `code`, `price`, and a member of type `publisher`. Assume that the texts are less than 100 characters. Write a program that uses the type `book` to read the data of 100 books and store them in an array. Then, the program should read a book's code and display the title of the book and the publisher's name, if registered.

**U.13.10** Define the structure type `student` with members: `name`, `code`, and `grade`. Write a program that declares an array of six such structures and initializes the first five with random data sorted by grade in ascending order (e.g., "A.Smith, 100, 3.2", "B.Jones, 200, 4.7", "K.Lee, 175, 6.4", ...). Then, the program should read the data of the sixth student and store them in the proper position, so that the array remains sorted.

**U.13.11** Define the structure type `car` with members: `model`, `price`, and `manufacture year`. Write a program that uses this type to read the data of 100 cars and store them in an array. Then, the program should provide a menu to perform the following operations:

1. *Show model.* The program should read a model and display the related information, if registered. If the user enters '\*', the program should show the information about all models.
2. *Show prices.* The program should read a price and show all models that cost more.
3. *Program termination.*

**U.13.12** Define the structure type `any_type` with an integer member (e.g., `s_type`) and a member of type `union`. The union's members are a structure of type `time` (e.g., `t`) as defined in U.13.7 and a structure of type `student` (e.g., `s`) as defined in U.13.6. Write a program that reads an integer and stores it into the `s_type` member of a structure variable of type `any_type`. Then, the program should check the value of `s_type` and read data according to that value. If it is 1 the input, data should be stored into the `t` member; otherwise, it should be stored into the `s` member.

**U.13.13** Define the union type `selected_type` with the members named `u_char` (of type `char`), `u_int` (of type `int`), `u_float` (of type `float`), and `u_double` (of type `double`). Define the structure type `var_type` with the members named `type` (of type `int`) and `st` (union of type `selected_type`). Write a function that takes as parameter a structure of type `var_type`. The function should read a number according to the value of the `type` member, store that number in the appropriate `st` member, and display the value of that member. Write a program that prompts the user to enter a number that represents the data type (i.e., 1: `char`, 2: `int`, 3: `float`, 4: `double`) and then uses the function to read a corresponding value and display it.

# Memory Management and Data Structures

Memory management is mainly related to memory allocation and release of allocated memory when it is no longer needed. Memory allocation can be performed either *statically* or *dynamically*. Up to this point, the program variables have been stored in fixed size memory, statically allocated. In this chapter, we'll show you how to allocate memory dynamically during program execution, in order to form flexible data structures, such as linked lists and binary trees.

## Memory Blocks

When a program runs, it asks for memory resources from the operating system. Various operating systems and compilers use their own models to manage the available memory. Typically, the system memory is divided into four parts:

1. The *code* segment, which is used to store the program code.
2. The *data* segment, which is used to store the global and **static** variables. This segment may be also used to store literal strings (typically, in a read-only memory).
3. The *stack* segment, which is used to store a function's data, such as local variables.
4. The *heap* segment, which is used for dynamic memory allocation.

For example, the following program displays the memory addresses of the program data:

```
#include <stdio.h>
#include <stdlib.h>

void test(void);

int global;

int main(void)
{
    int *ptr;
    int i;
    static int st;

    ptr = (int*) malloc(sizeof(int)); /* Allocate memory from the
heap. */
    if(ptr != NULL)
```

```

    {
        printf("Code seg: %p\n", test);
        printf("Data seg: %p %p\n", &global, &st);
        printf("Stack seg: %p\n", &i);
        printf("Heap: %p\n", ptr);
        free(ptr);
    }
    return 0;
}

void test(void)
{
}

```

Notice that this distribution model is a typical choice; each system may define its own. Moreover, each compiler may apply its own optimization policy. For example, the parameters of a function might be stored in system registers for faster access, not in the stack.

---

## Static Memory Allocation

In static allocation, the memory is allocated from the stack. The size of the allocated memory is fixed; we must specify its size when writing the program and it cannot change during program execution. For example, with the statement:

```
float grades[1000];
```

the compiler allocates  $1000 \times 4 = 4000$  bytes to store the grades of 1000 students. The length of the array remains fixed; if we need to store grades for more students, we cannot change it during program execution. If the students proved to be less than 1000, memory is wasted. The only way to change the length of the array is to modify the program and compile it again.

Let's discuss briefly what happens when a function is called. A detailed discussion is beyond the scope of this book. The compiler allocates memory in the stack to store the function's data. As said, system registers might be used; for simplicity, we assume that the allocation occurs in the stack. For example, if the function returns a value, accepts parameters, and uses local variables, the compiler allocates memory to store:

- The values of the parameters.
- The local variables.
- The return value.
- The address of the next statement to be executed, when the function terminates.

When the function terminates, the following actions take place:

- If the return value is assigned into a variable, it is extracted from the stack and stored into that variable.

- b. The address of the next statement is extracted from the stack and the execution of the program continues with that statement.
- c. The memory allocated to store the function's data is released (not that of **static** variables, of course).

For example, in the following program, when `test()` is called, the compiler allocates 808 bytes in the stack to store the values of `i`, `j`, and `arr` elements:

```
#include <stdio.h>

void test(int i, int j);

int main(void)
{
    float a[1000], b[10];
    test(10, 20);
    return 0;
}

void test(int i, int j)
{
    int arr[200];
    ...
}
```

This memory is released when `test()` terminates. Similarly, the memory of 4040 bytes allocated for the local variables of `main()` is released when the program terminates.

---

If the stack has not enough memory to store the data of a function, the execution of the program would terminate abnormally and the message "*Stack overflow*" may appear.

This situation may occur when a function allocates large blocks of memory and calls other nested functions with considerable memory needs as well. For example, a recursive function that calls itself many times may cause the exhaustion of the available stack memory.

---

## Dynamic Memory Allocation

In dynamic allocation, the memory is allocated from the heap during program execution. Unlike static allocation, its size can be dynamically specified. Furthermore, this size may dynamically shrink or grow according to the program's needs. Typically, the default stack size is not very large. For example, the following program might not run because of unavailable stack memory:

```
#include <stdio.h>
int main(void)
{
    int arr[10000000]; /* Static allocation. */
    return 0;
}
```

On the other hand, the size of the heap is usually much larger than the stack size. To use dynamic allocation, we'd write:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int *arr;

    arr = (int*) malloc(10000000* sizeof(int));
    if(arr != NULL)
        free(arr);
    return 0;
}
```

Next, we'll discuss the `malloc()` and `free()` functions. For now, remember that dynamically allocated memory should be released when no longer needed.

## The `malloc()` Function

To allocate memory dynamically, we have to use one of the `realloc()`, `calloc()` or `malloc()` functions declared in `stdlib.h`. The most used is the `malloc()`. It has the following prototype:

```
void *malloc(size_t size);
```

It is reminded that the `size_t` type is usually a synonym of the `unsigned int` type. The `size` parameter declares the number of bytes to be allocated. If the memory is allocated successfully, `malloc()` returns a pointer to that memory, `NULL` otherwise. `malloc()` does not initialize the memory with some value, `calloc()` sets all bits to 0 instead. For example:

```
int *ptr;
ptr = (int*) malloc(100);
```

The cast of the return type from `void*` to `int*` indicates that the allocated memory will be used for the storage of integers. However, no cast is necessary; we could write:

```
ptr = malloc(100);
```

The reason we prefer to cast the return pointer is to make clear what type of data will be stored in the allocated memory, so the reader won't have to look for the pointer declaration to find it. Another reason might be the fact that we often write C programs that have to be compiled with a C++ compiler, and that cast is mandatory in C++. On the other hand, the downside is that if we ever need to change the type of the pointer (e.g., type of `ptr`), we must look for and change all type casts.

If the allocation succeeds, `ptr` will point to the beginning of that memory. If not, its value will be `NULL`.

---

Always check the return value of `malloc()`. If it is `NULL`, an attempt to use a null pointer would have unpredictable results; the program may crash.

In the previous program, the maximum number of stored integers is 25 because each integer needs four bytes. Notice that we could write `25*sizeof(int)` instead of 100. In fact, the best practice is to use the `sizeof` operator, in order to get a platform independent program. For example, an `int` type may reserve two bytes in one system and four in another. If we were not using the `sizeof` operator, `malloc()` would reserve memory for 50 integers in the first system and for 25 in the second one.

---

Always use the `sizeof` operator to specify the number of bytes to be allocated.

However, when allocating memory for a `char` type, you can safely omit the `sizeof` operator because the size of the `char` type is always one. For example, to allocate memory for a string of n characters, we write:

```
char *ptr;  
ptr = (char*) malloc(n+1);
```

The extra place is for the null character.

As another example, to allocate memory for 100 structures of type `student` we write:

```
struct student *ptr;  
ptr = (struct student*) malloc(100*sizeof(student));
```

Alternatively, many prefer to use the pointer. For example:

```
ptr = (struct student*) malloc(100*sizeof(*ptr));
```

---

Always remember that the size of the allocated memory is calculated in bytes.

For example, the following program is wrong because only one integer can be stored in the allocated memory, not four:

```
#include <stdio.h>  
#include <stdlib.h>  
int main(void)  
{  
    int *arr, i;  
  
    arr = (int*) malloc(4);  
    if(arr != NULL)  
    {  
        for(i = 0; i < 4; i++)  
            arr[i] = 10;  
        free(arr);  
    }  
    return 0;  
}
```

The assignment of 10 to arr[0] is correct, but the assignments to arr[1], arr[2], and arr[3] are wrong since no memory is allocated for them. Had we written:

```
arr = (int*) malloc(4*sizeof(int));
```

the program would be executed normally. Hope that you are not wondering if you are allowed to use the pointer as an array. If you are, read again the section “Pointers and Arrays” in Chapter 8.

A typical use of malloc() is to create two-dimensional arrays dynamically. Suppose that the dimensions are not known when the program is compiled. The following program reads the two dimensions of an array of floats, allocates memory, reads values, and stores them into the array.

```
#include <stdio.h>
#include <stdlib.h>

void set_values(double **arr, int rows, int cols);

int main(void)
{
    int i, rows, cols;
    double **arr; /* We use a pointer to pointer variable to handle
the two-dimensional array. */

    do
    {
        printf("Enter dimensions of array[N] [M]: ");
        scanf("%d%d", &rows, &cols);
    } while(rows <= 0 || cols <= 0);

    arr = (double**) malloc(rows * sizeof(double*)); /* The expression
sizeof(double*) is used to allocate memory for 'rows' pointers to
doubles. For example, if rows is 3, we allocate memory for arr[0],
arr[1], and arr[2] pointers.*/
    if(arr == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < rows; i++)
    {
        /* Allocate memory to store the elements of each row. Each
row contains 'cols' elements.*/
        arr[i] = (double*) malloc(cols * sizeof(double));
        if(arr[i] == NULL)
        {
            printf("Error: Not available memory\n");
            exit(EXIT_FAILURE);
        }
    }
    set_values(arr, rows, cols);
    for(i = 0; i < rows; i++)
        free(arr[i]);
```

```

        free(arr);
        return 0;
    }

void set_values(double **arr, int rows, int cols)
{
    int i, j;

    for(i = 0; i < rows; i++)
        for(j = 0; j < cols; j++)
    {
        printf("Set arr[%d] [%d]: ", i, j);
        scanf("%lf", &arr[i][j]);
        printf("arr[%d] [%d] = %f\n", i, j, arr[i][j]);
    }
}

```

As you see, we use each `arr[i]` as a pointer to the row `i` that contains `cols` doubles. The reason we added the function is to show you how we can pass a pointer to a pointer variable and that we can use array subscripting to handle it. Notice also that when memory is allocated in that way, the elements might not reside in successive places as happens with an ordinary declaration, such as `int arr[10][20]`.

Alternatively, we could use `malloc()` to simulate the two-dimensional array as a one-dimensional array. For example:

```
double *arr = (double*) malloc(rows * cols * sizeof(double)); and write
arr[i*cols+j] to access each i, j element.
```

## The `free()` Function

To release a dynamically allocated memory, we have to use the `free()` function declared in `stdlib.h`. It has the following prototype:

```
void free(void *ptr);
```

`ptr` must point to a memory dynamically allocated. If not, the program can have unpredictable behavior. Notice that dynamic memory can be allocated from one function and released from another function, provided that the second function has access to that memory (e.g., through a returned pointer). If `ptr` is `NULL`, `free()` has no effect. For example, the following program may crash because `ptr` does not point to an allocated memory:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int *ptr;
    free(ptr);
    return 0;
}
```

Notice that `free()` is used to release dynamically allocated memory, not static. For example, if we declare `ptr` as an array (i.e., `int ptr[100]`), the program would fail again. Also, don't attempt to release an already released memory. In the following example, the second call to `free()` is wrong because the memory is already released:

```
int *ptr = (int*) malloc(100*sizeof(int));
if(ptr != NULL)
{
    free(ptr);
    free(ptr);
}
```

---

Don't forget to call `free()` to release a dynamically allocated memory when no longer needed, in order to avoid memory leaks.

For example, is the following code well written?

```
char *ptr = (char*) malloc(10);
strcpy(ptr, "test");
ptr = (char*) malloc(20);
printf("%c\n", *ptr);
free(ptr);
```

Since the first memory block was not released before `ptr` points to the second block a memory leak is created; this memory remains unusable. The code outputs the value of the first character in that new memory. Be careful with memory leaks, otherwise you may end up with a program crash due to an out-of-memory error. Find the errors in the following program:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char *p1, *p2;

    p1 = (char*) malloc(10);
    p2 = (char*) malloc(10);
    if(p1 != NULL && p2 != NULL)
    {
        p1 = p2;
        free(p1);

        p2[0] = 'a';
        free(p2);
    }
    return 0;
}
```

- a. The statement `p1 = p2;` makes `p1` and `p2` point to the same memory. The statement `free(p1);` releases this memory, so the next statement assigns a value to a nonallocated memory.

- b. The statement `free(p2);` releases memory already released.
- c. The first block of memory pointed to by `p1` was never released.

Need more practice? Find the errors in the following program:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    int *i;
    char *c;
} test;

int main(void)
{
    test *t;
    int i;

    t = (test*) malloc(sizeof(test));
    t->i = (int*) malloc(10*sizeof(int));
    t->c = (char*) malloc(10);
    t->i[0] = 'a';
    t->c[5] = 50;
    for(i = 0; i < 10; i++)
        printf("%d %d\n", t->i[i], t->c[i]);
    free(t);
    free(t->i);
    free(t->c);
    return 0;
}
```

Are `t->i[0] = 'a'` or `t->c[5] = 50` wrong? Certainly not, we just assign two integers. What about the loop? Nothing wrong with that; it uses array subscripting to display the existing values. What about the memory release? Well, here is the bug. First we have to release the memory allocated for the members of the structure and then the memory for the entire structure, not the other way around. Therefore, we must move `free(t)` to the end.

In large programs, where the same pointer may be used in different parts of the program, our advice is to make it `NULL` each time you free it for more safety. This practice helped us a lot when debugging programs with memory problems. See what happened to me once.

One time, I used an array of some thousands of pointers to structures. I initialized them with `NULL` and memory was allocated for some of them during program execution. So far, so good. Once in a while, memory was released in random order. After that, the program was crashing. And the worse part was that the program was crashing occasionally, not always. It was really a nightmare. After days of stress and hard debugging, we concluded that the crash was due to some pointer misuse (... what else?). Here is the buggy code:

```
if(arr[i] != NULL)
    x = arr[i]->member;
```

The `if` condition didn't let a null pointer access the member. That's fine, but what about a pointer that used to point to a memory and that memory is later released? For example,

suppose that the memory pointed to by some `arr[i]` is released. The condition is still true, because the fact that the memory is released does not mean that the value of `arr[i]` becomes `NULL`. Therefore, when using `arr[i]` to access the member, the program was crashing. The `NULL` value must be explicitly assigned, so when I added the statement `arr[i] = NULL` after releasing the memory, the problem was solved and we went straight for beers!!!

---

## Memory Management Functions

This section discusses briefly the `memcpy()`, `memmove()`, and `memcmp()` functions that are often used for memory management. `memcpy()` is used to copy any type of data from one memory region to another. It is declared in `string.h`:

```
void *memcpy(void *dest, const void *src, size_t size);
```

`memcpy()` copies `size` bytes from the memory pointed to by `src` to the address pointed to by `dest`. If the source and destination addresses overlap, its behavior is undefined. For example, the following program allocates memory to store the string "ABCDE" and displays its content:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    char *arr;
    /* Allocate one extra byte for the null character. */
    arr = (char*) malloc(6);
    if(arr != NULL)
    {
        memcpy(arr, "ABCDE", 6);
        printf("%s\n", arr);
        free(arr);
    }
    return 0;
}
```

The main difference with `strcpy()` is that `strcpy()` stops copying when the null character is met. On the other hand, `memcpy()` does not stop copying until it copies the number of specified bytes. For example, if we have:

```
char str2[6], str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
```

and write:

```
memcpy(str2, str1, 6);
```

the content of `str2` would be the same as of `str1`.

On the other hand, if we write:

```
strcpy(str2, str1);
```

the content of str2 would be: {'a', 'b', 'c', '\0'}.

memmove() is similar to memcmp(), except that memmove() guarantees that the data will be copied correctly even if the source and destination memory overlap. Because memcpy() does not check if the two memory regions overlap, it is executed faster than memmove(). Notice that the size of the destination memory should be size bytes, at least. If it is not, the extra bytes would be written in a nonallocated memory, meaning that the existing data will be overwritten. For example, the next copy is wrong because the size of the destination memory is 3 bytes, while the copied bytes are 6:

```
char str1[3], str2[] = "abcde";
memcpy(str1, str2, sizeof(str2));
```

---

memcpy() is often very useful because it is usually implemented in a way that copying large blocks of data is probably accomplished faster than an iteration loop.

For example, the following program declares two arrays of 100000 integers, sets the values 1 to 100000 into the elements of the first array, and uses memcpy() to copy them into the second array:

```
#include <stdio.h>
#include <string.h>

#define SIZE 100000

int main(void)
{
    int i, arr1[SIZE], arr2[SIZE];

    for(i = 0; i < SIZE; i++)
        arr1[i] = i+1;

    memcpy(arr2, arr1, sizeof(arr1));
    /* We use memcpy() instead of an iteration loop such as:
    for(i = 0; i < SIZE; i++)
        arr2[i] = arr1[i]; */
    for(i = 0; i < SIZE; i++)
        printf("%d\n", arr2[i]);
    return 0;
}
```

memcmp() is used to compare the data stored in one memory region with the data stored in another. It is declared in string.h:

```
int memcmp(const void *ptr1, const void *ptr2, size_t size);
```

memcmp() compares size bytes of the memory regions pointed to by ptr1 and ptr2. The return value of memcmp() is as that of strcmp(). The main difference with strcmp()

is that `strcmp()` stops comparing when it encounters the null character in either string. `memcmp()`, on the other hand, stops comparing only when `size` bytes are compared; it does not look for the null character. For example:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
    char str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
    char str2[] = {'a', 'b', 'c', '\0', 'd', 'f'};

    if(strcmp(str1, str2) == 0)
        printf("Same\n");
    else
        printf("Different\n");

    if(memcmp(str1, str2, sizeof(str1)) == 0)
        printf("Same\n");
    else
        printf("Different\n");
    return 0;
}
```

Because `strcmp()` stops comparing when the null character is met, the program displays Same. On the contrary, `memcmp()` compares all bytes, and the program displays Different.

---

## Exercises

C.14.1 How many bytes does the following `malloc()` allocate?

```
double *ptr;
ptr = (double*) malloc(100*sizeof(*ptr));
```

**Answer:** Since `ptr` is a pointer to `double`, `sizeof(*ptr)` calculates the size of the `double` type, that is 8. Therefore, `malloc()` allocates 800 bytes.

C.14.2 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void test(char *p);

int main(void)
{
    char *p1, *p2;
    p1 = p2 = (char*) malloc(5);
```

```

    if(p1 != NULL)
    {
        strcpy(p1, "example");
        test(p2);
        printf("%s\n", p1);
    }
    return 0;
}

void test(char *p)
{
    printf("%s\n", p);
    free(p);
}

```

**Answer:** When malloc() returns, p1 and p2 point to the same memory. When test() is called, we have p = p2 and test() displays example. Since free() releases the memory pointed to by p (that's the same pointed to by p1 and p2), main() displays random characters.

**C.14.3** Write a function that resizes a dynamically allocated memory that stores integers. The function takes as parameters a pointer to the original memory, the initial memory size, and the new size and returns a pointer to the new memory. The existing data should be copied in the new memory. Write a program that allocates memory dynamically to store an array of 10 integers and sets the values 100 up to 109 to its elements. Then, the program should call the function to reallocate new memory to store 20 integers and display its content.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int *realloc_mem(int *ptr, int old_size, int new_size);

int main(void)
{
    int *arr, i;

    /* Allocate memory for 10 integers. */
    arr = (int*) malloc(10 * sizeof(int));
    if(arr == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < 10; i++)
        arr[i] = i+100;

    arr = realloc_mem(arr, 10, 20); /* arr points to the new memory.
*/
    printf("\n***** Array elements *****\n");
    for(i = 0; i < 20; i++)
        printf("%d\n", arr[i]);
}

```

```

        free(arr); /* Release new memory. */
        return 0;
    }

int *realloc_mem(int *old_mem, int old_size, int new_size)
{
    int *new_mem;

    /* Allocate memory for new_size integers. */
    new_mem = (int*) malloc(new_size * sizeof(int));
    if(new_mem == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    /* Copy the existing data to the new memory. */
    memcpy(new_mem, old_mem, old_size * sizeof(int));
    free(old_mem); /* Release old memory. */
    return new_mem; /* Return the pointer to the new memory. */
}

```

**Comments:** The program displays the values 100–109 for the first 10 elements and random values for the next ten, since they are not initialized. C library provides a function similar to `realloc_mem()`, called `realloc()`. `realloc()` is declared in `stdlib.h` and it is used to modify the size of a dynamically allocated memory. A short description is provided in Appendix C.

**C.14.4** Write a function similar to `memcmp()`. The program should read two strings of less than 100 characters and the number of the characters to be compared and use the function to display the result of the comparison.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int mem_cmp(const void *ptr1, const void *ptr2, size_t size);
int read_text(char str[], int size, int flag);

int main(void)
{
    char str1[100], str2[100];
    int num;

    printf("Enter first text: ");
    read_text(str1, sizeof(str1), 1);

    printf("Enter second text: ");
    read_text(str2, sizeof(str2), 1);

    printf("Enter characters to compare: ");
    scanf("%d", &num);

    printf("%d\n", mem_cmp(str1, str2, num));
}

```

```

        return 0;
    }

int mem_cmp(const void *ptr1, const void *ptr2, size_t size)
{
    char *p1, *p2;

    p1 = (char*)ptr1;
    p2 = (char*)ptr2;
    while(size != 0)
    {
        if(*p1 != *p2)
            return *p1 - *p2;
        p1++;
        p2++;
        size--;
    }
    return 0;
}

```

**Comments:** The loop compares the characters pointed to by p1 and p2. Since characters are compared, we cast the type `void*` to `char*`. If all characters are the same, `mem_cmp()` returns 0, otherwise the difference of the first two nonmatching characters.

**C.14.5** Write a function similar to `strcpy()`. The function should take as parameters two pointers and copy the string pointed to by the second pointer into the memory pointed to by the first pointer. The memory pointed to by the first pointer should have been allocated dynamically and its size should be equal to the length of the copied string. Write a program that reads a string of less than 100 characters, calls the function to copy it in the dynamically allocated memory, and displays the content of that memory.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *str_cpy(char *trg, const char *src);
int read_text(char str[], int size, int flag);

int main(void)
{
    char *trg, src[100];
    int len;

    printf("Enter text: ");
    len = read_text(src, sizeof(src), 1);
    /* Allocate memory to store the input string and the null
character. */
    trg = (char*) malloc(len+1);
    if(trg == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }

```

```

printf("Copied text: %s\n", str_cpy(trg, src));
free(trg);
return 0;
}

char *str_cpy(char *trg, const char *src)
{
    int i = 0;
    while(*(src+i) != '\0') /* Or else, while(src[i] != '\0')*/
    {
        *(trg+i) = *(src+i); /* Or else, trg[i] = src[i]; */
        i++;
    }
    *(trg+i) = '\0'; /* Add the null character. */
    return trg;
}

```

**Comments:** We could omit i, we'd get a more complicated code though. Here is an example:

```

char *str_cpy(char *trg, const char *src)
{
    char *ptr = trg;
    while(*trg++ = *src++)
    ;
    return ptr;
}

```

The loop executes until the null character is copied into the memory pointed to by trg. For example, in the first iteration, we have `trg[0] = src[0]`; and the pointers are increased to point to the next elements. In fact, the statement `*trg++ = *src++;` is equivalent to:

```

*trg = *src;
trg++;
src++;

```

**C.14.6** Write a function that takes as parameters three pointers to strings and stores the last two strings into the first one. The memory for the first string should have been allocated dynamically. Write a program that reads two strings of less than 100 characters and uses the function to store them into a dynamically allocated memory.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void str_cat(char *fin, char *str1, char *str2);
int read_text(char str[], int size, int flag);

int main(void)
{
    char *fin, str1[100], str2[100];
    int len1, len2;

```

```

printf("Enter first text: ");
len1 = read_text(str1, sizeof(str1), 1);

printf("Enter second text: ");
len2 = read_text(str2, sizeof(str2), 1);
/* Allocate memory to store both strings and the null character.
*/
fin = (char*) malloc(len1+len2+1);
if(fin == NULL)
{
    printf("Error: Not available memory\n");
    exit(EXIT_FAILURE);
}
str_cat(fin, str1, str2);
printf("Merged text: %s\n", fin);
free(fin);
return 0;
}

void str_cat(char *fin, char *str1, char *str2)
{
    while(*str1 != '\0') /* Or else, while(*str1) */
        *fin++ = *str1++;

    /* Copy the second string after the first one. */
    while(*str2 != '\0') /* Or else, while(*str2) */
        *fin++ = *str2++;
*fin = '\0'; /* Add the null character. */
}

```

**Comments:** The first loop copies the characters of the string pointed to by str1 into the memory pointed to by fin. Similarly, the next loop adds the characters of the second string. Instead of **while** loops, we could use **for** loops, as follows:

```

void str_cat(char *fin, char *str1, char *str2)
{
    for(; *str1; *fin++ = *str1++);
    for(; *str2; *fin++ = *str2++);
    *fin = '\0';
}

```

Notice the ; at the end of both **for** statements. How about a more complex solution with a single loop?

```

void str_cat(char *fin, char *str1, char *str2)
{
    for(; *str2; *str1 ? *fin++ = *str1++ : *fin++ = *str2++);
    *fin = '\0';
}

```

That's another example of what we consider bad coding style. Don't forget our advice. Write code having simplicity in mind.

C.14.7 Write a function that takes as parameters two arrays of doubles (e.g., a1 and a2) of the same size and their number of elements and allocates memory to store the elements of a1 that are not contained in a2. The function should return a pointer to that memory. Write a program that reads pairs of doubles and stores them into two arrays of 100 elements (e.g., p1 and p2). If either input value is -1, the insertion of numbers should end. The program should use the function to display the elements of p1 that are not contained in p2.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 100

double *find_diff(double a1[], double a2[], int size, int *items); /* items indicates how many elements are stored in the memory. A pointer is passed, so that the function may change its value. */

int main(void)
{
    int i, elems;
    double *p3, j, k, p1[SIZE], p2[SIZE];

    for(i = 0; i < SIZE; i++)
    {
        printf("Enter numbers: ");
        scanf("%lf%lf", &j, &k);
        if((j == -1) || (k == -1))
            break;
        p1[i] = j;
        p2[i] = k;
    }
    elems = 0;
    p3 = find_diff(p1, p2, i, &elems);
    if(elems == 0)
        printf("\n***** No different elements *****\n");
    else
    {
        for(i = 0; i < elems; i++)
            printf("%f\n", p3[i]);
    }
    free(p3);
    return 0;
}

double *find_diff(double a1[], double a2[], int size, int *items)
{
    int i, j, found;
    double *mem;

    mem = (double*) malloc(size * sizeof(double));
    if(mem == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
```

```

    for(i = 0; i < size; i++)
    {
        found = 0; /* This variable indicates whether an element of
the first array exists in the second, or not. The value 0 means that it
does not exist. */
        for(j = 0; j < size; j++)
        {
            if(a2[j] == a1[i])
            {
                found = 1;
                break; /* Since this element exists, we stop
searching. */
            }
        }
        /* If it does not exist, it is stored in the memory. */
        if(found == 0)
        {
            mem[*items] = a1[i];
            (*items)++;
        }
    }
    return mem;
}

```

**C.14.8** Write a program that declares an array of 20 pointers to strings and allocates the exact memory needed to store strings of less than 100 characters. The program should read 20 strings and display the longest one. If more than one string is the longest, the program should display the one found first.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 20

int read_text(char str[], int size, int flag);

int main(void)
{
    char *ptr[SIZE], str[100];
    int i, pos, len, max_len; /* pos is used to indicate the ptr
element, which points to the longest string. max_len holds its length. */
    pos = max_len = 0;
    for(i = 0; i < SIZE; i++)
    {
        printf("Enter text: ");
        len = read_text(str, sizeof(str), 1);
        /* Allocate the required memory. */
        ptr[i] = (char*) malloc(len+1);
        if(ptr[i] == NULL)
        {
            printf("Error: Not available memory\n");
            exit(EXIT_FAILURE);
        }
    }
}

```

```

/* Store the string in the allocated memory. */
strcpy(ptr[i], str);
/* We compare the length of each string against max_len and
if a longer string is found, we store its position and length. */
    if(len > max_len)
    {
        pos = i;
        max_len = len;
    }
}
printf("Longer string: %s\n", ptr[pos]);
for(i = 0; i < SIZE; i++)
    free(ptr[i]);
return 0;
}

```

**Comments:** This method of handling the strings is more efficient than the one presented in C.10.37. There is no waste of memory.

**C.14.9** Write a program that reads its command line arguments and allocates memory to store their characters in reverse order. For example, if the arguments are next and time, the program should store into the memory txeN and emit.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *rvs_str;
    int i, j, len;

    if(argc == 1)
    {
        printf("Missing string arguments ... \n");
        exit(EXIT_FAILURE);
    }
    for(i = 1; i < argc; i++)
    {
        len = strlen(argv[i]);
        rvs_str = (char*) malloc(len+1); /* Allocate one extra place
for the null character. */
        if(rvs_str == NULL)
        {
            printf("Error: Not available memory\n");
            exit(EXIT_FAILURE);
        }
        for(j = 0; j < len; j++)
            rvs_str[j] = argv[i][len-1-j]; /* The last character
is stored in position len-1. */

        rvs_str[j] = '\0'; /* Terminate the string. */
        printf("Reverse of %s is: %s\n", argv[i], rvs_str);
    }
}

```

```
        free(rvs_str);
    }
    return 0;
}
```

C.14.10 Define the structure type publisher with members: name, address, and phone number. Then, define the structure type book with members: title, author, field, code, price, and a pointer to a structure of type publisher. Besides price, all other members must be pointers. Assume that the maximum text length is 100 characters. Write a program that reads the number of books and allocates memory to store their data and the data of the publishers as well. Then, the program should read a book's code and display its title and the name of its publisher, if registered.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int read_text(char str[], int size, int flag);

typedef struct /* Let's remember typedef. */
{
    char *name;
    char *addr;
    char *phone;
} pub;

typedef struct
{
    char *title;
    char *auth;
    char *code;
    pub *pub_ptr;
    float prc;
} book;

/* To save space, we assume that all malloc() calls are successful. */
int main(void)
{
    book *books_ptr;
    char str[100];
    int i, num, len;

    printf("Enter number of books: ");
    scanf("%d", &num);
    getchar();

    books_ptr = (book*) malloc(num * sizeof(book));
    for(i = 0; i < num; i++)
    {
        printf("\nTitle: ");
        len = read_text(str, sizeof(str), 1);
        books_ptr[i].title = (char*) malloc(len+1);
        strcpy(books_ptr[i].title, str);
```

```

printf("Authors: ");
len = read_text(str, sizeof(str), 1);
books_ptr[i].auth = (char*) malloc(len+1);
strcpy(books_ptr[i].auth, str);

printf("Code: ");
len = read_text(str, sizeof(str), 1);
books_ptr[i].code = (char*) malloc(len+1);
strcpy(books_ptr[i].code, str);

printf("Price: ");
scanf("%f", &books_ptr[i].prc);

getchar();
/* Allocate memory to store the data of the publishing
firm. */
books_ptr[i].pub_ptr = (pub*) malloc(sizeof(pub));

printf("Name: ");
len = read_text(str, sizeof(str), 1);
(books_ptr[i].pub_ptr)->name = (char*)malloc(len+1);
strcpy((books_ptr[i].pub_ptr)->name, str);

printf("Address: ");
len = read_text(str, sizeof(str), 1);
(books_ptr[i].pub_ptr)->addr = (char*)malloc(len+1);
strcpy((books_ptr[i].pub_ptr)->addr, str);

printf("Phone: ");
len = read_text(str, sizeof(str), 1);
(books_ptr[i].pub_ptr)->phone = (char*)malloc(len+1);
strcpy((books_ptr[i].pub_ptr)->phone, str);
}

printf("\nEnter code to search: ");
read_text(str, sizeof(str), 1);

for(i = 0; i < num; i++)
{
    if(strcmp(books_ptr[i].code, str) == 0)
    {
        printf("\nTitle: %s\tPublisher: %s\n\n", books_ptr[i].title,
               (books_ptr[i].pub_ptr)->name);
        break;
    }
}
if(i == num)
    printf("\nCode '%s' isn't registered\n", str);

for(i = 0; i < num; i++)
{
    free((books_ptr[i].pub_ptr)->name);
    free((books_ptr[i].pub_ptr)->addr);
    free((books_ptr[i].pub_ptr)->phone);
}

```

```

        free(books_ptr[i].title);
        free(books_ptr[i].auth);
        free(books_ptr[i].code);
        free(books_ptr[i].pub_ptr);
    }
    free(books_ptr);
    return 0;
}

```

---

## Dynamic Data Structures

The data structures we've met so far are used for data storage and processing in an easy and fast way. For example, arrays are data structures, which are used for the storage of the same type of data. Structures and unions are also data structures, which can be used for the storage of any type of data. These data structures are *static*, in the sense that the allocated memory is *fixed* and cannot be modified during program execution. However, in many applications, it'd be more efficient to use a dynamic data structure, a structure whose size may grow or shrink as needed. The next sections introduce you to some simple forms of dynamic data structures and show you implementation examples. For more complex variants of data structures you should read a book focusing on that subject.

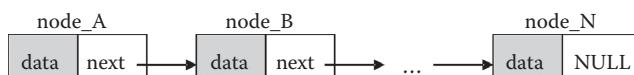
---

## Linked List

A linked list consists of a chain of linked elements, called *nodes*. Each node is typically represented by a structure, which contains its data and a pointer to the next node in the chain, as depicted in Figure 14.1.

The first node is the head of the list and the last one its tail. The value of the pointer member in the last node must be `NULL` to indicate the end of the list.

Unlike arrays whose size remains fixed, a linked list is more flexible because its size can be dynamically adjusted to the program's needs. Because we can easily insert and delete nodes at any point of the list, it is much easier to create and maintain a sorted order. On the other hand, to access an array element is very fast; we just use its position as an index. Typically, to access a list node, we should start from the head node (leave out the case of storing the nodes' addresses for direct access) and traverse the list serially, whereas arrays provide random access. Therefore, the time to access a node depends on its position in the list. It is fast if the node is close to the beginning, slow if it is near the end. This can make a huge difference to performance.



**FIGURE 14.1**

Linked list.

## Insert a Node

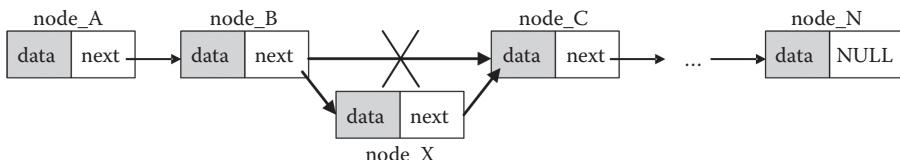
A new node can be inserted at any point in the list. To insert a new node, we check the following cases:

1. If the list is empty, the node is inserted at the beginning and becomes the head and the tail of the list. The value of its pointer member is set to NULL since there is no next node in the list.
2. If the list is not empty, we check the following subcases:
  - a. If the node is inserted at the beginning of the list, it becomes the new head of the list and its pointer member points to the previous head, which now becomes the second node of the list.
  - b. If the node is inserted at the end of the list, it becomes the new tail of the list and the value of its pointer member is set to NULL. The old tail becomes the second to last node and the value of its pointer member changes from NULL to point to the new node.
  - c. If the node is inserted after an existing node, the pointer member of the new node must point to the node after the current node and the pointer member of the current node must point to the new node. Figure 14.2 depicts how the new node X is inserted between the nodes B and C.

## Delete a Node

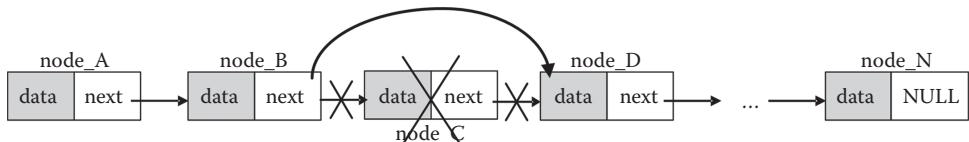
To delete a node from the list and release the memory it allocates, we check the following cases:

1. If it is the head of the list, we check the following subcases:
  - a. If there is a next node, this node becomes the new head of the list.
  - b. If there is no next node, the list becomes empty.
2. If it is the tail of the list, the previous node becomes the new tail and its pointer member is set to NULL.
3. If it is an intermediate node, the pointer member of the previous node must point to the node after the one to be deleted. This operation is shown here with the deletion of node C (Figure 14.3).



**FIGURE 14.2**

Node insertion.



**FIGURE 14.3**  
Node deletion.

## Implementation Examples

Before creating a linked list, we'll implement two special cases of a linked list, a stack and a queue. Notice that there are various ways to implement these data structures; we have tried to implement them in a simple and comprehensible way. In the following examples, each node is a structure of type `node`. If you ever need to develop similar data structures, the majority of the code can be reused as is.

### Implementing a Stack

In this section, we'll create a LIFO (Last In First Out) stack, where, as its name indicates, the last inserted node it is the first to get extracted. It is a special case of a linked list with the following restrictions:

- a. A new node can be inserted only at the beginning of the stack and becomes its new head.
- b. Only the head can be deleted.

A stack like this resembles a stack of dirty dishes. Each new dish is put at the top of the stack and it is the top dish we get to wash.

### Exercise

C.14.11 Define the structure type `node` with members: `code`, `name`, `grade`, and a pointer to a structure of type `node`. Create a stack whose nodes are structures of that type. Write a program that displays a menu to perform the following operations:

1. Insert a student. The program should read the student's data and store them in a node, which becomes the new head of the stack.
2. Display the data of the stored students.
3. Display the data of the last inserted student.
4. Delete the last inserted student.
5. Display the total number of the stored students.
6. Program termination.

To handle the stack, we use a global pointer. This pointer always points to the head of the stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node
{
    char name[100];
    int code;
    float grd;
    struct node *next; /* Pointer to the next node. Notice that the
existence of the structure tag enables us to declare its type. */
} node;

node *head; /* Global pointer that always points to the head of the
stack. */

void add_stack(const node *p);
void show_stack(void);
void pop(void);
int size_stack(void);
void free_stack(void);

int read_text(char str[], int size, int flag);

int main(void)
{
    int sel;
    node n;

    head = NULL; /* The NULL value indicates an empty stack. */
    while(1)
    {
        printf("\nMenu selections\n");
        printf("-----\n");

        printf("1. Add student\n");
        printf("2. View all students\n");
        printf("3. View top student\n");
        printf("4. Delete top student\n");
        printf("5. Number of students\n");
        printf("6. Exit\n");

        printf("\nEnter choice: ");
        scanf("%d", &sel);

        switch(sel)
        {
            case 1:
                getchar();
                printf("Name: ");
```

```

        read_text(n.name, sizeof(n.name), 1);

        printf("Code: ");
        scanf("%d", &n.code);
        printf("Grade: ");
        scanf("%f", &n.grd);

        add_stack(&n);
break;

case 2:
    if(head != NULL)
        show_stack();
    else
        printf("\nThe stack is empty\n");
break;

case 3:
    if(head != NULL)
        printf("\nC:%d N:%s G:%.2f\n\n",
               head->code, head->name, head->grd);
    else
        printf("\nThe stack is empty\n");
break;

case 4:
    if(head != NULL)
        pop();
    else
        printf("\nThe stack is empty\n");
break;

case 5:
    if(head != NULL)
        printf("\n%d students exist in stack\n",
               size_stack());
    else
        printf("\nThe stack is empty\n");
break;

case 6:
    if(head != NULL)
        free_stack();
return 0;

default:
    printf("\nWrong choice\n");
break;
}

return 0;
}

void add_stack(const node *p)
{
    node n;
    scanf( "%s %d %.2f", n.name, &n.code, &n.grd );
    n.next = head;
    head = &n;
}
}

```

```

{
    node *new_node;

    /* Allocate memory to create a new node. */
    new_node = (node*) malloc(sizeof(node));
    if(new_node == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    *new_node = *p; /* Copy the student's data into the new node. */
    new_node->next = head; /* The new node is inserted at the
beginning of the stack. For example, when the first node is inserted the
value of new_node->next becomes equal to the initial value of head, which
is NULL. */
    head = new_node; /* head points to the new node, therefore the new
node becomes the new head of the stack. */
}

void show_stack(void)
{
    node *p;

    p = head;
    printf("\n***** Student Data *****\n\n");
    while(p != NULL)
    {
        printf("C:%d N:%s G:%.2f\n", p->code, p->name, p->grd);
        p = p->next; /* In each iteration, p points to the next
node. Once it becomes NULL it means that there is no other node left and
the loop terminates. */
    }
}

void pop(void)
{
    node *p;

    p = head->next; /* p points to the next node after the head. This
node will become the new head of the stack. */
    printf("\nStudent with code =%d is deleted\n", head->code);
    free(head); /* Release the allocated memory. The information about
the next node is not lost, because it was saved in p. */
    head = p; /* head points to the new head of the stack. */
}

int size_stack(void)
{
    node *p;
    int num;

    num = 0;
    p = head;
    while(p != NULL)

```

```

    {
        p = p->next;
        num++; /* This variable counts the nodes. */
    }
    return num;
}

void free_stack(void)
{
    node *p, *next_node;

    p = head;
    while(p != NULL)
    {
        next_node = p->next; /* next_node always points to the node
after the one to be deleted. */
        free(p); /* Release the allocated memory. The information
about the next node is not lost, because it was saved in next_node. */
        p = next_node; /* p points to the next node. */
    }
}

```

### Comments:

- In `add_stack()` we pass a pointer and not the structure itself, in order to avoid the creation of a structure's copy and make faster the execution of the function. We declare the pointer as `const`, just to remember how to prevent modifications.
- To display immediately the number of the stored students, without traversing the stack, we could remove the `size_stack()` function and declare a global variable that would be incremented by one each time a student is inserted and decremented when a student is deleted. Once the user selects that menu choice, `printf()` would display its value. It is much simpler and faster. The reason we are using `size_stack()` is to show you how to traverse the nodes of the stack.
- If the variable `head` had been declared locally in `main()`, we should have passed its address to the functions that need it. For example, `add_stack()` would change to:

```

void add_stack(const node *p, node **head_ptr)
{
    node *new_node;

    new_node = (node*) malloc(sizeof(node));
    if(new_node == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    *new_node = *p;
    new_node->next = *head_ptr;
    *head_ptr = new_node;
}

```

To call it we'd write: `add_stack(&n, &head);`

Because we think that this code is more complicated, at least for a beginner, we preferred to declare `head` as a global variable and use it directly wherever needed.

## Implementing a Queue

In this section, we'll create a FIFO (First In First Out) queue, where, as its name indicates, the first inserted node it is the first to get extracted. It is a special case of a linked list with the following restrictions:

- a. A new node can be inserted only at the end of the queue and becomes its new tail.
- b. Only the head can be deleted.

---

### Exercise

C.14.12 Define the structure type `node` with members: `code`, `name`, `grade`, and a pointer to a structure of type `node`. Create a queue whose nodes are structures of that type. Write a program that displays a menu to perform the following operations:

1. Insert a student. The program should read the student's data and store them in a node, which becomes the new tail of the queue.
2. Display the data of the stored students.
3. Display the data of the last inserted student.
4. Delete the first inserted student.
5. Program termination.

To handle the queue, we use two global pointers. The first one always points to the head of the queue and the second one to its tail.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node
{
    char name[100];
    int code;
    float grd;
    struct node *next;
} node;

node *head; /* Global pointer that always points to the head of the
queue. */
node *tail; /* Global pointer that always points to the tail of the
queue. */
```

```
void add_queue(const node *p);
void show_queue(void);
void pop(void);
void free_queue(void);

int read_text(char str[], int size, int flag);

int main(void)
{
    int sel;
    node n;

    head = NULL;
    while(1)
    {
        printf("\nMenu selections\n");
        printf("-----\n");
        printf("1. Add student\n");
        printf("2. View all students\n");
        printf("3. View last student\n");
        printf("4. Delete top student\n");
        printf("5. Exit\n");

        printf("\nEnter choice: ");
        scanf("%d", &sel);

        switch(sel)
        {
            case 1:
                getchar();

                printf("Name: ");
                read_text(n.name, sizeof(n.name), 1);

                printf("Code: ");
                scanf("%d", &n.code);

                printf("Grade: ");
                scanf("%f", &n.grd);

                add_queue(&n);
                break;

            case 2:
                if(head != NULL)
                    show_queue();
                else
                    printf("\nThe queue is empty\n");
                break;

            case 3:
                if(head != NULL)
                    printf("\nC:%d N:%s G:%.2f\n\n",
                           tail->code,tail->name,tail->grd);
        }
    }
}
```

```

        else
            printf("\nThe queue is empty\n");
        break;

    case 4:
        if(head != NULL)
            pop();
        else
            printf("\nThe queue is empty\n");
        break;

    case 5:
        if(head != NULL)
            free_queue();
        return 0;

    default:
        printf("\nWrong choice\n");
        break;
    }
}

return 0;
}

void add_queue(const node *p)
{
    node *new_node;

    new_node = (node*) malloc(sizeof(node));
    if(new_node == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    *new_node = *p;
    new_node->next = NULL;

    if(head == NULL)
        head = tail = new_node; /* If the queue is empty, both head
and tail pointers point to the new node. */
    else
    {
        tail->next = new_node; /* The new node is inserted at the
end of the queue. */
        tail = new_node; /* tail points to the last node. */
    }
}

```

**Comments:** We omit the code of the `show_queue()`, `pop()`, and `free_queue()` functions because it is the same as the code of `show_stack()`, `pop()`, and `free_stack()` functions in the stack program.

## Implementing a Linked List

The following program implements a linked list whose nodes correspond to students. Like before, we use one global pointer to point to the head of the list and another one to point to its tail.

### Exercises

C.14.13 Define the structure type node with members: code, name, grade, and a pointer to a structure of type node. Create a list whose nodes are structures of that type. Write a program that displays a menu to perform the following operations:

1. Insert a student at the end of the list. The program should read the student's data and store them in a node, which becomes the new tail of the list.
2. Insert a student in a specific place. The program should read the code of a student, locate the corresponding node with that code, and create a new node after that to insert the data of the new student.
3. Display the data of the stored students.
4. Find a student. The program should read the code of a student and display its data, if registered.
5. Modify the grade of a student. The program should read the code of a student and the new grade and modify the existing grade.
6. Delete a student. The program should read the code of a student and remove the node that corresponds to that student, if registered.
7. Program termination.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node
{
    char name[100];
    int code;
    float grd;
    struct node *next;
} node;

node *head; /* Global pointer that always points to the head of the
list. */
node *tail; /* Global pointer that always points to the tail of the
list. */

void add_list_end(const node *p);
void add_list(const node *p, int code);
```

```
void show_list(void);
node *find_node(int code);
int del_node(int code);
void free_list(void);

int read_text(char str[], int size, int flag);

int main(void)
{
    int k, sel, code;
    float grd;
    node *p, n;

    head = NULL;
    while(1)
    {
        printf("\nMenu selections\n");
        printf("-----\n");

        printf("1. Add student at the end\n");
        printf("2. Add student\n");
        printf("3. View all students\n");
        printf("4. View student\n");
        printf("5. Modify student\n");
        printf("6. Delete student\n");
        printf("7. Exit\n");

        printf("\nEnter choice: ");
        scanf("%d", &sel);

        switch(sel)
        {
            case 1:
            case 2: /* To avoid the repetition of the same code
we use the same case to insert data. Then, the if statement checks the
user's choice and calls the respective function. */
                getchar();

                printf("Name: ");
                read_text(n.name, sizeof(n.name), 1);

                printf("Code: ");
                scanf("%d", &n.code);

                printf("Grade: ");
                scanf("%f", &n.grd);

                if(sel == 1)
                    add_list_end(&n);
                else
                {
                    printf("\nEnter student code after which
the new student will be added: ");

```

```
        scanf ("%d", &code);
        add_list (&n, code);
    }
break;

case 3:
    if (head == NULL)
        printf ("\nThe list is empty\n");
    else
        show_list();
break;

case 4:
    if (head == NULL)
        printf ("\nThe list is empty\n");
    else
    {
        printf ("\nEnter student code to search:
");
        scanf ("%d", &code);
        p = find_node (code);
        if (p != NULL)
            printf ("\nN:%s G:%.2f\n\n",
p->name, p->grd);
        else
            printf ("\nStudent with code = %d
does not exist\n", code);
    }
break;

case 5:
    if (head == NULL)
        printf ("\nThe list is empty\n");
    else
    {
        printf ("\nEnter student code to modify:
");
        scanf ("%d", &code);

        printf ("Enter new grade: ");
        scanf ("%f", &grd);
        p = find_node (code);
        if (p != NULL)
            p->grd = grd;
        else
            printf ("\nStudent with code = %d
does not exist\n", code);
    }
break;

case 6:
    if (head == NULL)
        printf ("\nThe list is empty\n");
```

```

        else
        {
            printf("\nEnter student code to delete:
");
            scanf("%d", &code);
            k = del_node(code);
            if(k == 0)
                printf("\nStudent with code = %d
is deleted\n", code);
            else
                printf("\nStudent with code = %d
does not exist\n", code);
        }
        break;

    case 7:
        if(head != NULL)
            free_list();
        return 0;

    default:
        printf("\nWrong choice\n");
        break;
    }
}
return 0;
}

```

```

/* For a better understanding of add_list_end(), read the comments of
add_queue() in the previous exercise. */
void add_list_end(const node *p)
{
    node *new_node;

    new_node = (node*) malloc(sizeof(node));
    if(new_node == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    *new_node = *p;
    new_node->next = NULL;

    if(head == NULL)
        head = tail = new_node;
    else
    {
        tail->next = new_node;
        tail = new_node;
    }
}

```

```

void add_list(const node *p, int code)
{
    node *new_node, *ptr;

    ptr = head;
    /* We traverse the list, until the node with the indicated code is
found. If found, the new node is added after that and the function
terminates. */
    while(ptr != NULL)
    {
        if(ptr->code == code)
        {
            new_node = (node*) malloc(sizeof(node));
            if(new_node == NULL)
            {
                printf("Error: Not available memory\n");
                exit(EXIT_FAILURE);
            }
            *new_node = *p; /* Copy the student's data. */
            new_node->next = ptr->next; /* The new node is linked
to the node after the current node. */
            ptr->next = new_node; /* The current node is linked
to the new node. */
            if(ptr == tail) /* Check if the new node is added at
the end of the list. If it is, it becomes the new tail. */
            {
                tail = new_node;
                return;
            }
            ptr = ptr->next; /* Check the next node. */
        }
        /* If this point is reached it means that the input code does not
correspond to an existing student. */
        printf("\nStudent with code = %d does not exist\n", code);
    }

void show_list(void)
{
    node *p;

    p = head;
    printf("\n***** Student Data *****\n\n");
    while(p != NULL)
    {
        printf("C:%d N:%s G:%.2f\n\n", p->code, p->name, p->grd);
        p = p->next;
    }
}

node *find_node(int code)
{
    node *p;
}

```

```

p = head;
while(p != NULL)
{
    if(p->code == code)
        return p;
    p = p->next;
}
return NULL;
}

int del_node(int code)
{
    node *p, *prev_node; /* prev_node always points to the previous
node from the one that is going to be deleted. */

    p = prev_node = head;
    while(p != NULL)
    {
        if(p->code == code)
        {
            if(p == head)
                head = p->next; /* If the node is the head of
the list, the next node becomes the new head. If there is no other node,
the list becomes empty and head becomes NULL. */
            else
            {
                /* p points to the node that will be deleted
and prev_node points to the previous one. This statement links the
previous node with the node after the one to be deleted. */
                prev_node->next = p->next;
                if(p == tail) /* Check if the deleted node is
the tail of the list. If it is, the previous node becomes the new tail. */
                    tail = prev_node;
                }
            free(p); /* Delete the node. */
            return 0;
        }
        prev_node = p; /* prev_node points to the node that was just
checked and found that it has other code than the input code. */
        p = p->next; /* p points to the next node. Notice that prev_
node always points to the previous node from the one to be examined. */
    }
    return -1;
}

void free_list(void)
{
    node *p, *next_node;

    p = head;
    while(p != NULL)

```

```

    {
        next_node = p->next;
        free(p);
        p = next_node;
    }
}

```

**Comments:** If you want to change the first menu choice and insert the data of the new student at the beginning of the list and not at its end, replace the `add_list_end()` with the `add_stack()` presented in the stack program. In that case, the `tail` pointer is not needed.

**C.14.14** Consider the previous linked list. Write a function that takes as parameters the codes of two students and, if they are stored in the list, the function should swap their grades and return 0. If not, the function should return -1.

```

int swap(int code_a, int code_b)
{
    node *p, *tmp1, *tmp2;
    float grd;

    p = head;
    tmp1 = tmp2 = NULL;
    while(p != NULL)
    {
        if(p->code == code_a)
            tmp1 = p;
        else if(p->code == code_b)
            tmp2 = p;

        if(tmp1 && tmp2)
        {
            grd = tmp1->grd;
            tmp1->grd = tmp2->grd;
            tmp2->grd = grd;
            return 0;
        }
        p = p->next;
    }
    return -1;
}

```

**C.14.15** What is the output of the following program?

```

#include <stdio.h>

typedef struct node
{
    int i;
    struct node *next;
    struct node *prev;
} node;

```

```

int main(void)
{
    node a = {10}, b = {20}, c = {30}, d = {40};

    a.next = &b;
    a.prev = &d;

    b.next = &c;
    b.prev = &a;

    c.next = &d;
    c.prev = &b;

    d.next = &a;
    d.prev = &c;
    printf("%d\n", a.prev->prev->prev->next->i);
    return 0;
}

```

**Comments:** This program is a simple implementation example of another type of list, a circular double linked list, where each node, except the pointer to the next node, contains another pointer to the previous node. Notice that the next pointer of the last node points to the head of the list, while head's prev pointer points to the last node. `printf()` begins with node a, traverses sequentially the d, c, and b nodes, returns back to c and outputs 30.

---

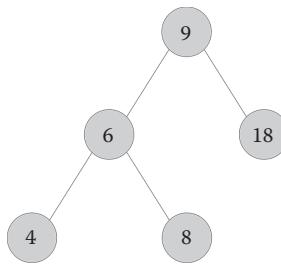
## Binary Search Tree

To organize data in sorted order, a flexible data structure named *tree* is typically used. This type of structure is met in several real-life examples, like a family tree, an organizational chart, or a computer's folder system. There are many types of trees; we are going to describe the basic operations in a binary search tree.

Let's present in short some basic concepts. A tree with *root* is a collection of nodes connected with edges. There is only one node with no parent, the top node. That node is named *root*. Each node, except the root, is connected with just one upper node, the *parent* node. The nodes directly connected one level below are its *children*. A tree in which each node has at most two children is called *binary tree*. The two parts are referred to as the left and right subtree. A binary tree where each node has a key and that key:

- a. Is greater than or equal to the keys in all nodes in its left subtree and
- b. Is less than or equal to the keys in all nodes in its right subtree

is called *binary search tree*. The main advantage of using that structure is its high performance in critical operations like inserting and searching for a value. Related sorting and search algorithms can be very efficient. Figure 14.4 shows an example of a binary search tree.



**FIGURE 14.4**

Binary search tree.

Let's see some implementation examples of the most basic operations in a binary search tree. Suppose that each node is represented by the following structure:

```

struct node
{
    int key;
    struct node *left;
    struct node *right;
};
  
```

We are going to store different keys. The pointers point to the node's children. If a child is missing, the pointer is NULL.

```

#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    int key;
    struct node *left;
    struct node *right;
} node;

node* add_node(node *p, int key);
void show_inorder(node *p);
void delete_tree(node *p);

int main(void)
{
    int sel, key;
    node *root;

    root = NULL;
    while(1)
    {
        printf("\nMenu Selections\n");
        printf("-----\n");
        printf("1. Add Node\n");
        printf("2. Show Tree\n");
        printf("3. Exit\n");
  
```

```

printf("\nEnter choice: ");
scanf("%d", &sel);

switch(sel)
{
    case 1:
        printf("Key: ");
        scanf("%d", &key);
        root = add_node(root, key);
    break;

    case 2:
        printf("Added Keys: ");
        show_inorder(root);
    break;

    case 3:
        printf("\nDeleted Keys: ");
        delete_tree(root);
    return 0;

    default:
        printf("\nWrong choice\n");
    break;
}
}

return 0;
}

node* add_node(node *p, int key)
{
    if(p == NULL)
    {
        p = (node*) malloc(sizeof(node));
        if(p == NULL)
        {
            printf("Error: Not available memory\n");
            exit(EXIT_FAILURE);
        }
        p->left = p->right = NULL;
        p->key = key;
    }
    else
    {
        if(key < p->key)
            p->left = add_node(p->left, key);
        else if(key > p->key)
            p->right = add_node(p->right, key);
        else
            printf("Error: Key(%d) exists\n", key);
    }
    return p;
}
}

```

```

void show_inorder(node *p)
{
    if (p == NULL)
        return;
    show_inorder(p->left);
    printf("%d ", p->key);
    show_inorder(p->right);
}

void delete_tree(node *p)
{
    if (p == NULL)
        return;

    delete_tree(p->left);
    delete_tree(p->right);
    printf("%d ", p->key);
    free(p);
}

```

`add_node()` creates and inserts a new node in the tree, provided that no other node has the same key value. The function compares the inserted key with the key of the examined node and uses recursion to head towards the proper subtree. For example, suppose that the user enters the numbers: 9, 6, 18, 4, and 8.

Insertion of 9: Since `root` is `NULL`, the function creates the root of the tree and assigns the value `NULL` to its pointers. The address of the root node is returned to the main program.

Insertion of 6: Because 6 is less than 9, the function is called recursively with argument the `p->left` pointer. Since it is `NULL`, the recursive call creates a new node (at the left of the root) and assigns the value `NULL` to its pointers. The address of that new node is returned and `p->left` becomes equal to that value. The function terminates with the return of the root pointer.

Insertion of 18: Similar to node 6, with the difference that `p->right` pointer is used.

Insertion of 4: Because 4 is less than 9, the function is called recursively with argument the `p->left` pointer. Since it is `NULL` and 4 is less than 6, a second recursive call is made with argument the `left` pointer of node 6. Since it is `NULL`, that second call creates a new node and returns its address at the point of the first recursive call. This value is stored in `p->left`, where `p` points to node 6. The first call terminates with the return of `p`. Next, the function terminates with the return of the root pointer.

Insertion of 8: Similar to node 4, with the difference that the argument of the second recursive call is the `right` pointer of node 6.

In this way, the tree gets the form of Figure 14.4.

The procedure to insert a key in the tree resembles the procedure to search for a key. Try U.14.15 to write a recursive search function. Alternatively, here is a nonrecursive implementation. In fact, this is our preference; remember what we've discussed about recursion in Chapter 11.

```

node* find_node(node *p, int key)
{
    if (p == NULL) /* Check if the tree is empty. We could use the
while condition, we just prefer to make it clear. */
        return NULL;

```

```
while(p != NULL)
{
    if(key < p->key)
        p = p->left;
    else if(key > p->key)
        p = p->right;
    else
        return p;
}
return NULL;
```

The code compares the key and the proper subtree is selected. If the lower level is reached, it means that the key is not found. Now that you've seen this method, try U.14.17 to implement `add_node()` without using recursion. To help you, use the previous code to find the node under which the new node must be connected, that is, the parent node. Then, create the new node and connect it with the parent node.

As a new exercise, suppose that the `show_inorder()` is called. Try to figure out what it is doing and tell us how it works. Spend some time, don't rush to look at the answer on the next page.

You looked right away, didn't you? Go back and give a try. It's worth testing your skills, to see if you can decode multiple recursions.

Let's see how the function works. In the first recursive call, `p->left` points to node 6. Since it is not `NULL`, a second recursive call is made and `p->left` points to node 4. A third call is made (now `p` points to 4), and since `p->left` is `NULL` the function returns.

The second recursive call outputs 4 and a new call is made with argument `p->right`. Since it is `NULL`, the function returns. The second recursive call ends and we continue with the first one.

The first call outputs 6 and a new recursive call is made with argument `p->right`. The recursion is executed where `p` points to 8. A new recursion, which returns because `p->left` is `NULL`. The program outputs 8 and the next recursion returns, because `p->right` is `NULL`.

Where are we? ... We got lost. If we are not wrong, we were checking the first recursion where `p->left` points to node 6, and that recursion is just ended. The program outputs 9 and a new recursive call is made with argument `p->right`. Like before, the program outputs 18 and the function ends, ... at last.

Essentially, `show_inorder()` performs an in-order tree traversal. In particular, for each traversed node, it first visits the nodes of its left subtree, then the current node, and then the nodes of its right subtree. As a result, the function displays the key values in ascending order.

In the same sense, `show_preorder()` performs a preorder tree traversal. In particular, for each traversed node, it first visits the node itself, then the nodes of its left subtree, and then the nodes of its right subtree.

```
void show_preorder(node *p)
{
    if(p == NULL)
        return;
    printf("%d ", p->key);
    show_preorder(p->left);
    show_preorder(p->right);
}
```

`show_postorder()` performs a postorder tree traversal. For each traversed node, it first visits the nodes of its left subtree, then the nodes of its right subtree, and then the current node.

```
void show_postorder(node *p)
{
    if(p == NULL)
        return;
    show_postorder(p->left);
    show_postorder(p->right);
    printf("%d ", p->key);
}
```

`delete_tree()` uses the postorder method to release the allocated memory. For each traversed node, first the memory of the nodes of its two subtrees is released, and then the memory of the node itself. In the previous example, the nodes are released in that order: 4, 8, 6, 18, and 9.

Another method to traverse a tree is to visit the nodes starting from the root and going down in level order. In each level, the nodes are visited from left to right. In the previous example, the nodes are visited in that order: 9, 6, 18, 4, and 8. To implement the level order method, we'll write a nonrecursive function that uses an array of pointers as a FIFO queue to store the visited nodes. For simplicity, we assume that the tree contains a maximum numbers of nodes.

```
#define MAX_NODES 200

void show_levelorder(node *p)
{
    node *arr[MAX_NODES] = {0}; /* To simulate a FIFO queue, we use an
array of pointers to the nodes. */
    int start, end; /* start indicates the node to display its key and
end indicates the last node stored in the queue. */

    if(p == NULL) /* Check if the tree is empty. */
        return;

    start = end = 0;
    add_queue(arr, p, &end); /* Store the root. */

    while((p = pop(arr, &start)) != NULL)
    {
        printf("%d ", p->key);

        if(p->left)
            add_queue(arr, p->left, &end);
        if(p->right)
            add_queue(arr, p->right, &end);
    }
}

void add_queue(node *arr[], node *p, int *cnt)
{
    arr[*cnt] = p;
    (*cnt)++;
}

node* pop(node *arr[], int *cnt)
{
    node *tmp;

    tmp = arr[*cnt];
    (*cnt)++;
    return tmp;
}

/* Alternatively, without using tmp:
(*cnt)++;
return arr[*cnt-1]; */
}
```

Consult that code and try U.14.18. Our approach to solve it was first to visit all left nodes. We also changed the queue to LIFO. For example, consider the nodes of the first program. Initially, the nodes 9, 6, and 4 are stored. We pop the last one and display its key, that is, 4. We pop the last but one, that is, 6, and ... to be continued.

---

## Exercises

C.14.16 Add a recursive function that displays the smallest key value in the tree.

```
void find_min(node *p)
{
    if(p == NULL)
        return;
    if(p->left == NULL)
    {
        printf("Min:%d\n", p->key);
        return;
    }
    find_min(p->left);
}
```

**Comments:** In a binary search tree, the smallest value is stored in the leftmost node. Then, the recursion ends. To find the largest value, just change the pointer to `p->right`, to find the rightmost node. Call the function with argument the `root` pointer, to test its operation. For an alternative nonrecursive solution, try U.14.16.

C.14.17 Write a recursive function that counts the nodes of the tree.

```
void tree_nodes(node *p)
{
    if(p == NULL)
        return;
    cnt++;
    tree_nodes(p->left);
    tree_nodes(p->right);
}
```

**Comments:** `cnt` could be a global variable initialized with 0 and incremented by one each time a node is visited.

C.14.18 Modify the structure that represents the node, in order to store different strings in the tree. For example:

```
typedef struct node
{
    char *key;
    struct node *left;
    struct node *right;
} node;
```

Modify add\_node() and delete\_tree().

### Answer:

```
node* add_node(node *p, char key[])
{
    int tmp;

    if(p == NULL)
    {
        p = (node*) malloc(sizeof(node));
        if(p == NULL)
        {
            printf("Error: Not available memory\n");
            exit(EXIT_FAILURE);
        }
        p->key = (char*) malloc(strlen(key)+1);
        if(p->key == NULL)
        {
            free(p);
            printf("Error: Not available memory\n");
            exit(EXIT_FAILURE);
        }
        strcpy(p->key, key);
        p->left = p->right = NULL;
    }
    else
    {
        tmp = strcmp(key, p->key);
        if(tmp < 0)
            p->left = add_node(p->left, key);
        else if(tmp > 0)
            p->right = add_node(p->right, key);
        else
            printf("Error: Key(%s) exists\n", key);
    }
    return p;
}

void delete_tree(node *p)
{
    if(p == NULL)
        return;

    delete_tree(p->left);
    delete_tree(p->right);
    printf("%s\n", p->key);
    free(p->key);
    free(p);
}
```

## Hashing

As discussed in Chapter 12, one way to search for a value in a nonsorted array is to compare each element with that value. Because the search time with that linear method might be long, efficient algorithms based on hashing have been developed. In short, hashing refers to the transformation of the searching value into a usually shorter value that represents the original value. That shorter value is used to index the array and speed up access to an element. Hashing is quite popular in a wide range of applications when it comes to search for a value (i.e., databases).

The development of an algorithm that uses hashing is done in two steps. The first is the implementation of a hash function that transforms the searching value (or else key) into an array position. Because two or more different keys may be hashed in the same array position, the second step is to implement a collision resolution function. The function we are going to implement is based on linked lists. Let's see an application example where hashing proved really useful.

In the company I used to work for, we had to develop an application for a network operator. The application should simulate a large number of calls (e.g., 500000). Each call is identified by a unique 32-bit number, the Call Reference Value (CRV). Each transmitted message contains the CRV of the call it refers to. When a message is received from the network, the application should check fast whether it corresponds to an active call. An efficient solution could be the allocation of a large array (e.g., `calls`) and use the CRV to index it. For example, when a call between two users is initiated, we'd store its CRV:

```
void insert_key(int crv)
{
    calls[crv] = crv;
}
```

and when a message is received, we'd check whether it corresponds to an active call:

```
int check_key(int crv)
{
    if(calls[crv] == -1) /* Suppose that the array is initialized with
a non valid value, such as -1. */
        return NON_EXISTING_CALL;
    return EXISTING_CALL;
}
```

Because the size of the required memory was prohibitively high, I used hashing. Here is an implementation example:

- a. Suppose that the array contains  $N$  elements. We need a function that transforms the keys into integers in  $[0, N-1]$ . A simple hashing method uses a prime number for array size and the function  $h(k) = k \% N$  to calculate the position of the  $k$  key in the array.
- b. If two or more keys are hashed in the same array position, that is, we have a collision, those keys are stored in the nodes of a linked list created in that position.

For example, suppose that  $N = 5$  and the keys are 7, 23, 10, 43, 84, 90, and 60. The keys would be stored in the array like this:

```
[0] : 10 -> 90 -> 60 -> NULL
[1] : NULL
[2] : 7 -> NULL
[3] : 23 -> 43 -> NULL
[4] : 84 -> NULL
```

In position [0], a linked list of three nodes is created. To explain, each  $k$  key is examined. If  $k \% 5$  equals 0, a node with key value equal to  $k$  is created. As a result, three nodes are created with keys 10, 90, and 60, respectively. In position [1] no list is created because there is no key  $k$  so that  $k \% 5$  equals 1. The remaining positions are configured in the same way.

To search for a key  $k$ , we use the hash function to get the position and start searching from that point. For example, suppose that we want to check if the key 53 is stored in the array. The position to look for is  $53 \% 5 = 3$ . Next, we traverse the linked list in position [3], to see if the key is stored.

The following program provides an implementation example of the insert and search functions. The array used for storing the keys is an array of pointers.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

typedef struct node
{
    int key;
    struct node *next;
} node;

int hash_func(int key);
void insert_key(node **cur, int key);
node* find_key(node *p, int key);
void free_mem(node *hash_arr[]);

int main(void)
{
    node *p, *hash_arr[SIZE] = {NULL};
    int i, pos, key;

    for(i = 0; i < 10; i++) /* Insert 10 keys. */
    {
        printf("Enter key: ");
        scanf("%d", &key);

        pos = hash_func(key);
        insert_key(&hash_arr[pos], key);
    }
    printf("Enter key to search: ");
    scanf("%d", &key);

    pos = hash_func(key);
```

```

p = find_key(hash_arr[pos], key);
if(p == NULL)
    printf("Key_%d doesn't exist\n", key);
else
    printf("Key_%d exists\n", key);

free_mem(hash_arr);
return 0;
}

int hash_func(int key) /* We write it separately from the rest of the
code to make it clear. If we want to use another hashing function, we
just change that line. */
{
    return key % SIZE;
}

void insert_key(node **cur, int key)
{
    node *prev, *tmp, *p;

    tmp = (node*) malloc(sizeof(node));
    if(tmp == NULL)
    {
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    tmp->key = key;
    tmp->next = NULL;

    if(*cur == NULL) /* We check if it is the first time to insert a
key in that position. If it isn't, we add the new node at the end of the
list. */
        *cur = tmp;
    else
    {
        p = prev = *cur;
        while(p != NULL)
        {
            prev = p;
            p = p->next;
        }
        p = tmp;
        prev->next = p;
    }
}

node* find_key(node *p, int key)
{
    while(p != NULL)
    {
        if(p->key == key)
            return p;
        p = p->next;
    }
}

```

```

    return NULL;
}

void free_mem(node *hash_arr[])
{
    int i;
    node *p, *tmp;

    for(i = 0; i < SIZE; i++)
    {
        p = hash_arr[i];
        printf("\nKeys_%d: ", i); /* Display the key values. */
        while(p != NULL)
        {
            printf("%d ", p->key);
            tmp = p->next;
            free(p);
            p = tmp;
        }
    }
}

```

Let's try the following exercise.

## Exercise

**C.14.19** Write the following function to delete the key from the array.

```
int del_key(node **cur, int key);
```

Here is an example how to call it:

```
pos = hash_func(key);
del_key(&hash_arr[pos], key);
```

## Answer:

```
int del_key(node **cur, int key)
{
    node *p, *prev;

    p = prev = *cur;
    while(p != NULL)
    {
        if(p->key == key)
        {
            if(p == *cur)
                *cur = p->next;
            else
                prev->next = p->next;
        }
    }
}
```

```
        free(p);
        return 0;
    }
    prev = p;
    p = p->next;
}
return -1;
}
```

**Comments:** The implementation is similar to that of `del_node()` in C.14.13.

Before we finish this chapter, we'd like to underline how important it is to have a basic knowledge of algorithms and data structures. To become an expert programmer in any language, not only in C, you'll definitely need it. With the data structures presented in this chapter and the algorithms in Chapter 12, we tried to introduce you to these concepts and give you an idea. Because there is still a lot to learn, our suggestion is to read a book that focuses on these subjects. It'd be for your own benefit.

---

## Unsolved Exercises

**U.14.1** Complete the following program to read an integer and a `double` number and display their sum. Don't use another variable.

```
int main(void)
{
    int *p1;
    double *p2;
    ...
}
```

**U.14.2** Write a program that uses a pointer variable to read three integers and display the greatest. Don't use any other variable.

**U.14.3** Use the two pointer variables and complete the following function to return how many products cost less than \$20 and how many more than \$100.

```
void find(double *arr, int size, int *und20, int *ov100);
```

Write a program that prompts the user to enter the number of the products (i.e., `size`) and their prices and stores them in a dynamically allocated memory (i.e., `arr`). Then, the program should use the function to display how many products cost less than \$20 and how many more than \$100.

**U.14.4** Write a program that does the following:

- Allocates memory to store a number of integers. The program should prompt the user to enter that number.
- The program should read those integers and store them in the allocated memory.

- c. If the user enters -5, the program should release the memory and continue from the first step.
- d. If all integers are entered, the program should display the memory content and terminate.

**U.14.5** Use `ptr` and complete the following program to read and display the data of one student.

```
struct student
{
    char *name;
    int code;
    float grd;
};

int main(void)
{
    struct student *ptr;
    ...
}
```

Then, the program should read a number of students (use extra variables) and use `ptr` to allocate memory and store the data of those students whose name begin with an 'A'. The program should display their data before it ends. Assume that the names are less than 100 characters.

**U.14.6** Write a program that reads 10 `double` numbers and stores them in a dynamically allocated memory. Then, the program should allocate extra memory of the same size and prompt the user to enter a number, as follows:

- a. If it is 0, the program should store the numbers in that new memory in reverse order.
- b. If it is 1, the program should store first the negatives and then the positives.

Use pointer arithmetic to handle the memory regions. For example, assume that the first memory region contains the numbers:

-3.2 4 3 -9.1 7 6 -2 15 9 -37. If the user enters 0, the numbers should be stored in the second memory in reverse order: -37 9 15 -2 ... If the user enters 1, it should be stored in that order: -3.2 -9.1 -2 -37 4 3 ...

**U.14.7** Use `ptr` and complete the following program to read and display the data of one book. Don't use any other variable. Assume that the length of the input strings is less than 100 characters.

```
struct book
{
    char *title;
    char *authors;
```

```

    int *code;
    double *prc;
};

int main(void)
{
    struct book *ptr;
    ...
}

```

**U.14.8** An array in which a large number of elements has zero values is called sparse. An efficient way to store its elements is to put the non-zero elements and its positions in groups of three, in a one-dimensional array. For example, the elements of the array A are stored into B, as follows:

$$A = \begin{bmatrix} 0 & 0 & 6 & 8 \\ 0 & 0 & 0 & -1 \\ -3 & 0 & 0 & 0 \end{bmatrix} \quad B = [0 \ 2 \ 6 \ 0 \ 3 \ 8 \ 1 \ 3 \ -1 \ 2 \ 0 \ -3]$$

Because the element A[0, 2] is 6, the triplet 0 2 6 is stored in B. The remaining triplets are generated in the same way. Write a program that reads integers and stores them in a  $3 \times 4$  array A. Then, the program should allocate memory to create array B and store the triplets in it. The size of the allocated memory should be the exact size needed. The program should display the elements of B, before it ends.

**U.14.9** Write a program that reads integers continuously until the users enters -1. The program should display only the numbers entered for first time. Here is an example of program execution:

```

Enter number: -20
Output: -20
Enter number: 345
Output: 345
Enter number: -20
Enter number: 432
Output: 432

```

The number -20 does not appear twice.

**U.14.10** Write a program that provides a menu to perform the following operations:

1. Intersection. The program reads the common size of two arrays (e.g., A and B) and allocates memory dynamically to store their elements. Then, it reads integers and stores into A those that are not already stored. Then, it does the same for the array B. The program should store the common elements of the two arrays into a third array and display them.
2. Union. The program should store the elements of both arrays A and B into a third array. Notice that a common element should be stored once.
3. Program termination.

**U.14.11** Consider the linked list of C.14.13. Write a function that takes as a parameter the code of a student and, if it is stored in the list, the function should return how many nodes are left from this point up to the end of the list. If not, the function should return -1. Don't use the tail pointer.

**U.14.12** Consider the linked list of C.14.13. Write a **void** function that returns a pointer to the node with the best grade in the list and a pointer to the node with the worst grade. Write a sample program to show how to test the function. *Hint:* Use pointer to pointer function parameters.

**U.14.13** A double linked list is a list where each node, except the pointer to the next node, contains another pointer to the previous node. Write a program that generates 100 random integers and creates a double linked list with those integers. Use the following struct to represent the node:

```
typedef struct node
{
    int num;
    struct node *next;
    struct node *prev;
} node;
```

Write the following functions:

- add\_list (**int** num); it creates a new node with value equal to num.
- dup\_list (**void**); if all numbers in the list are different, it returns NULL, otherwise a pointer to the first node whose num value is the same as a previous one.
- del\_node (**int** num); it deletes the node with value equal to num.
- show\_list (**void**); it displays the num values of the nodes backwards, starting from the end of the list to the beginning.
- free\_list (**void**); it releases the allocated memory starting from the end of the list to the beginning.

The program should call all these functions in that order, to test their operation.

**U.14.14** Write a program that reads the names and the score of some candidates and inserts them sorted in a linked list. The sorting should be performed when data are entered. The data should be stored in score descending order or in alphabetical order. The program should prompt the user to enter the sorting type (e.g., 1 for alphabetical). If the user enters end for name, the data insertion should terminate and the program should display the list data. Use the following structure to represent the node:

```
typedef struct node
{
    char name[100];
    int score;
    struct node *next;
} node;
```

For example, if the user has selected sorting in alphabetical order and the list nodes are:

Arnesen	350
Santer	280

and the user enters the new data Melk and 450, the three list nodes should be in that order:

Arnesen	350
Melk	450
Santer	280

**U.14.15** In the binary search tree program, add a recursive function that takes as parameters a pointer to the root and a key value. If the key is found in the tree, the function should return a pointer to that node, NULL otherwise.

**U.14.16** In the binary search tree program, add a nonrecursive function that takes as parameter a pointer to the root and displays the smallest and the largest key value of the tree.

**U.14.17** In the binary search tree program, implement `add_node()` without using recursion. The function should return a pointer to the root.

**U.14.18** In the binary search tree program, implement `show_inorder()` without using recursion. Read the hint in that section.

Real-world programs often need to perform access operations on files. This chapter introduces C's most important library functions dealing with files. We won't describe them in full detail, but we'll give you enough of what you need in order to perform file operations. In particular, you'll learn how to open and close a file, as well as how to read and write data in *text* and *binary* files.

---

## Binary Files and Text Files

A *text* file consists of one or more lines that contain readable characters according to a standard format, like the American Standard Code for Information Interchange (ASCII) code. Each line ends with the special character(s) that the operating system uses to indicate the end of line. In *Windows* systems, for example, the pair of '\r' (Carriage Return) and '\n' (Line Feed) characters, that is, CR/LF, with ASCII codes 13 and 10 respectively, indicate the end of line. Therefore, the new line character '\n' is replaced by '\r' and '\n' when written in the text file. The reverse replacement happens when the file is read. On the other hand, this replacement does not take place in *Unix* systems because the '\n' character indicates the end of line.

Unlike the *text* files, the bytes of a *binary* file don't necessarily represent readable characters. For example, an executable C program is stored in a *binary* file. If you open it, you'll probably see some unintelligible characters. A *binary* file is not divided into lines and no character translation takes place. In *Windows*, for example, the new line character is not expanded to \r\n when written in the binary file.

Another difference between *text* and *binary* files is that the operating system may add a special character at the end of a *text* file to mark its end. In *Windows*, for example, the Ctrl+Z character marks the end of a *text* file. On the other hand, no character has a special significance in a *binary* file. They are all treated the same.

Storing data in a *binary* file can save space compared to a *text* file. For example, suppose that we are using the ASCII character set to write the number 47654 in a *text* file. Since this number is represented with five characters, the size of the file would be 5 bytes, as shown in Figure 15.1.

On the other hand, if this number is stored in *binary*, the size of the file would be 2 bytes, as shown in Figure 15.2.

*Binary* files might seem a better choice, but they have some disadvantages. When a *binary* file is transferred from one system to another, the data might not be represented the same, because different systems might store them in different ways. For example, a system may store first the high byte of a number, while some other system, the lower byte. Furthermore, because the sizes of the data types may differ from system to system,

00110100	00110111	00110110	00110101	00110100
'4'	'7'	'6'	'5'	'4'

**FIGURE 15.1**

Text file content.

10111010	00100110
$47654_{10} = 1011101000100110_2$	

**FIGURE 15.2**

Binary file content.

different number of bytes might be written. For example, a function using `sizeof(int)` to save an integer into a file may write two bytes in one system, four in another.

## Open a File

In C, the term *stream* refers to a source of data or a destination for output. For example, in all our programs so far, the input stream is associated with the keyboard, and the output stream, with the screen. Larger programs may need additional streams, such as streams associated with disk files or other devices (e.g., printer, modem). We'll talk about the simpler and most common case, streams associated with files. The `fopen()` function is used to open a file. It is declared in `stdio.h`:

```
FILE *fopen(const char *filename, const char *mode);
```

The first argument points to the name of the file to be opened. A file name may include path information. The second argument specifies the operations we intend to perform on the file, according to Table 15.1.

**TABLE 15.1**

Open Modes

Mode	Operation
r	Open file for reading.
w	Open file for writing. If the file exists, it will be truncated and its data will be lost. If it does not exist, it will be created.
a	Open file for appending. If the file exists, the existing data will be preserved and the new data will be added at its end. If it does not exist, it will be created.
r+	Open file for reading and writing.
w+	Open file for reading and writing. If the file exists, it will be truncated and its data will be lost. If it does not exist, it will be created.
a+	Open file for reading and appending. If the file exists, the existing data will be preserved and the new data will be added at its end. If it does not exist, it will be created.

To open a *text* file, we'd choose one of these modes. To open a *binary* file, we add the letter b. For example, the "rb" mode opens a binary file for reading, while the "w+b" mode opens a binary file for reading and writing.

If the file is opened successfully, `fopen()` returns a pointer to a structure of type FILE. This pointer identifies the file and it can be used later to perform file operations. The FILE structure is defined in `stdio.h` and holds information about the opened file. If the file cannot be opened, `fopen()` returns NULL. For example, `fopen()` fails when the mode is set to "r" and the specified file does not exist or we don't have permission rights to open it.

---

Always check the return value of `fopen()` against NULL, to see if the file was opened successfully or not.

---

If the file is stored in the same folder with the executable, just put the file name in double quotes. Here are some examples of typical `fopen()` calls:

- `fopen("test.txt", "r");` opens the `test.txt` text file for reading.
- `fopen("test.dat", "a+b");` opens the `test.dat` binary file for reading and appending. If it does not exist, it will be created in the same folder with the executable file.

If the file is stored somewhere else, the path must be specified.

---

If the operating system uses the character \ to specify the path, write \\ because C treats \ as the beginning of an escape sequence.

---

For example, if your program runs in Windows and you intend to open the `test.txt` file in `d:\dir1\dir2`, you should write:

```
fopen("d:\\dir1\\dir2\\test.txt", "r");
```

However, if the program obtains the file name from the command line, it is not needed to add an extra \. For example, we type `d:\dir1\dir2\test.txt`.

---

If the file is opened for both reading and writing, to switch from reading to writing a file positioning function (e.g., `fseek()`) must be first called. To switch from writing to reading a file positioning function or `fflush()` must be first called.

---

For example, the following program reads a file name and displays a message to indicate if it is opened successfully or not:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    char fname[100];

    printf("Enter file name: ");
```

```
read_text(fname, sizeof(fname), 1);
fp = fopen(fname, "r"); /* Open file for reading. */
if(fp == NULL)
{
    printf("Error: File can not be opened\n");
    exit(EXIT_FAILURE); /* Program termination. */
}
printf("File is opened successfully\n");
fclose(fp); /* Close file. */
return 0;
}
```

We could check the return value of `fopen()` in one line, like this:

```
if((fp = fopen(fname, "r")) == NULL)
```

The inner parentheses are necessary for reasons of priority.

---

In applications where many files are opened, it is advisable to close the files you need no more, because the operating system sets a limit to the number of files that can be opened simultaneously.

## Close a File

The `fclose()` function is used to close an open file. It is declared in `stdio.h`:

```
int fclose(FILE *fp);
```

The argument is a pointer that has been associated with an open file. If the file is closed successfully, `fclose()` returns 0, EOF otherwise.

---

Although an open file is closed automatically when a program terminates normally, our suggestion is to close it when you no longer use it. A good reason is that if the file is opened for writing and the program crashes after the file is closed, the file won't be affected.

When a program writes data to a file, the data are normally stored in a buffer area in memory. Without going into further details, `fflush()` can be used to flush the buffer as often as we wish. When it is full or the file is closed, the buffer is flushed and the data are written to the file. Buffering increases the performance, since one large transfer to the disk is much more efficient than many individual transfers. For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    int *p;

    if((fp = fopen("test.txt", "w")) != NULL)
    {
```

```
    fprintf(fp, "example\n");
    *p = 20; /* Wrong action. */
    fclose(fp);
}
return 0;
}
```

We use `fprintf()`, we'll see it in a moment, to store some data in a file. Because `p` does not point to a valid address, the program may crash. In that case, the data stored in the buffer will be lost. Had we called `fclose()` right after `fprintf()`, the buffer would have been flushed and the data stored in the file. Just remember to close the file when you no longer need it, to be sure that the data won't be lost. Another reason is that the operating system sets a limit on the number of files that a program may have open simultaneously, so, it'd be better to close files no longer in use.

---

## Process a File

As discussed, the pointer returned from a successful call to `fopen()` is associated with the opened file and points to a structure of type `FILE`. This structure holds information about the file. For example, it keeps the file position where the next read or write operation can be performed.

When a file is opened for reading or writing, the file position is set at the beginning of the file. If it is opened for appending, it is set at the end of the file. When a write or read operation is performed, the file position advances automatically. For example, if a file is opened for reading and the program reads 50 characters, the file position advances 50 bytes from the beginning of the file. Similarly, in a write operation, the file position advances a number of places equal to the number of the written bytes. Subsequent reading or writing will be done at that new position.

Besides sequential access, C provides the ability of random access. As we'll see, the `fseek()` function can be used to set the file position anywhere in the file.

---

## Write Data in a Text File

The most common functions to write data in a text file are: `fputs()`, `fprintf()`, and `fputc()`. These functions are mostly used with text files; however, we can use them to write data in binary files as well.

*For simplicity, we are not going to check the return values. We assume that the functions are executed successfully.*

### The `fputs()` Function

The `fputs()` function is used to write a string in an output file. It is declared in `stdio.h`:

```
int fputs(const char *str, FILE *fp);
```

`str` points to the string to be written in the file indicated by `fp`. If it is successful, `fputs()` returns a nonnegative value, `EOF` otherwise.

## The `fprintf()` Function

The `fprintf()` function is more generic than `fputs()` because it can write a variable number of different data items to an output file. Like `printf()`, `fprintf()` uses a format string to define the order and the type of data written in the file. It is declared in `stdio.h`:

```
int fprintf(FILE* fp, const char *format, ...);
```

The only difference between them is that `printf()` always writes data to `stdout`, while `fprintf()` writes to the file indicated by `fp`. In fact, if `fp` is replaced by `stdout`, `fprintf()` call is equivalent to `printf()` call. If it is successful, `fprintf()` returns the number of bytes written in the file, a negative value otherwise.

The `stdin` (*standard input*), `stderr` (*standard error*), and `stdout` (*standard output*) streams are declared in `stdio.h` as pointers of type `FILE*` and they are ready to use; it is not needed to open or close them. By default, the `stdout` and `stderr` output streams are associated with the screen, while the `stdin` input stream is associated with the keyboard. An operating system (e.g., *Unix*) may allow their redirection. For example, if the name of the executable is `test` and we write in the command line:

```
test >output.txt
```

the data written in `stdout` (e.g., with `printf()`) won't be displayed in the screen, but it will be written in the `output.txt` file. Notice that the program does not realize the redirection. In particular, the `>output.txt` is not interpreted as an argument of `main()`.

The following program uses `fputs()` and `fprintf()` to write some data in a text file:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char str[] = "Hello_2";
    int i;

    fp = fopen("test.txt", "w"); /* Open file for writing. */
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    fputs("Hello_1\n", fp);
    for(i = 0; i < 3; i++)
        fprintf(fp, "%d. %s\n", i+1, str); /* Use fprintf() to write
a string along with an increasing number. */
    fclose(fp);
    return 0;
}
```

Since `stdout` and `stderr` are associated with the screen by default, we could use `printf()` to output messages. For example:

```
fprintf(stdout, "Error: fopen() failed\n");
```

```
or: fprintf(stderr, "Error: fopen() failed\n");
```

For consistency with the rest programs, we'll continue using `printf()` to output all kind of messages. However, in case of error messages, it'd be better to use `stderr` instead of `stdout`. If `stdout` is redirected, you cannot see the messages until the file is opened. On the other hand, error messages written in `stderr` appear real time on the screen, even if `stdout` is redirected.

---

## Exercise

C.15.1 What does the following program do? Assume that the file opens successfully. Easy, think for a moment and see the answer on the next page.

```
#include <stdio.h>

void open_file(char fname[], FILE *fp_w);

int main(void)
{
    FILE *fp = NULL;

    open_file("test.txt", fp);
    fprintf(fp, "Everything OK\n");
    fclose(fp);
    return 0;
}

void open_file(char fname[], FILE *fp_w)
{
    fp_w = fopen(fname, "w");
}
```

**Answer:** What is it doing? Boom. Fell in the trap? *Thunderstruck* from AC/DC. For `open_file()` to be able to change the value of `fp`, its address should be passed, otherwise its value remains `NULL`. For example:

```
void open_file(char fname[], FILE **fp_w)
{
    *fp_w = fopen(fname, "r");
}
```

and change the call to: `open_file("test.txt", &fp);`

## The `fputc()` Function

The `fputc()` function is used to write a character in an output file. It is declared in `stdio.h`:

```
int fputc(int ch, FILE *fp);
```

`ch` specifies the character to be written in the file indicated by `fp`. Notice that any integer can be passed to `fputc()`; however, only the lower 8 bits will be written. If it is successful, `fputc()` returns the written character, `EOF` otherwise.

`fputc()` is similar to `putc()`. Their difference is that `putc()` is usually implemented as a macro, while `fputc()` is a function. Because the overhead of a function call is avoided, macros tend to be executed faster than functions, so, it'd be preferable to use `putc()`. For example, the following program writes one by one the characters of a string in a text file:

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char str[] = "This text will be written in the file";
    int i;
    fp = fopen("test.txt", "w");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; str[i] != '\0'; i++)
        putc(str[i], fp);

    fclose(fp);
    return 0;
}
```

## Exercises

C.15.2 Write a program that reads products' prices continuously and stores in a text file those that cost more than \$10 and less than \$20. If the user enters -1, the insertion of prices should terminate.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    double prc;

    fp = fopen("test.txt", "w");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    while(1)
    {
        printf("Enter price: ");
        scanf("%lf", &prc);
        if(prc == -1)
            break;
        if(prc > 10 && prc < 20)
            fprintf(fp, "%f\n", prc);
    }
    fclose(fp);
    return 0;
}
```

C.15.3 Write a program that reads strings continuously (less than 100 characters each) and appends in a user selected file those with less than 10 characters and begin with 'a'. If the user enters end, the input of the strings should terminate and the program should display how many strings were written in the file.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    char str[100];
    int len, cnt;

    printf("Enter file name: ");
    read_text(str, sizeof(str), 1);
    fp = fopen(str, "a"); /* Open file for appending. */

```

```

if(fp == NULL)
{
    printf("Error: fopen() failed\n");
    exit(EXIT_FAILURE);
}
cnt = 0;
while(1)
{
    printf("Enter text: ");
    len = read_text(str, sizeof(str), 1);
    if(strcmp(str, "end") == 0)
        break;
    if((str[0] == 'a') && (len < 10))
    {
        cnt++;
        fputs(str, fp);
    }
}
printf("\n%d strings were written\n", cnt);
fclose(fp);
return 0;
}

```

**C.15.4** What is written in the file and what is displayed on the screen?

```

#include <stdio.h>
int main(void)
{
    FILE *fp[2];

    if((fp[0] = fp[1] = fopen("test.txt", "w")) != NULL)
    {
        fputs("One", fp[0]);
        fclose(fp[0]);
        printf("%d\n", fputs("Two", fp[1]));
        fclose(fp[1]);
    }
    return 0;
}

```

**Answer:** The fp[0] and fp[1] pointers become equal to the return value of fopen() and they are both associated with the same file. The first call of fputs() writes One in the file, while the second call fails because the file is closed with the first fclose(). Indeed, the program displays the return value of the second fputs(), that is EOF. Therefore, only one One is written in the file. Had we checked the return value of the second fclose(), we would have seen that it fails too.

**C.15.5** Write a program that intends to write some data into a file. The program should read a file name and, if the file does not exist, the program should create it and write the string One into it. If it exists, the program should ask the user to overwrite it. If the answer is positive, the program should write the string One into it; otherwise, the

program should prompt the user to enter another file name in order to repeat the same procedure.

```
#include <stdio.h>
#include <stdlib.h>

FILE *open_file(char name[], int *f);

int main(void)
{
    FILE *fp;
    char name[100];
    int flag, sel;

    flag = 0;
    do
    {
        printf("Enter file name: ");
        scanf("%s", name);

        fp = fopen(name, "r"); /* Check whether the file exists or
not. If not, we open it for writing. Otherwise, we close the file and ask
the user. */
        if(fp == NULL)
            fp = open_file(name, &flag);
        else
        {
            fclose(fp);
            printf("Would you like to overwrite existing file
(Yes:1 - No:0)? ");
            scanf("%d", &sel);
            if(sel == 1) /* Overwrite the file. */
                fp = open_file(name, &flag);
        }
    } while(flag == 0);

    fputs("One", fp);
    fclose(fp);
    return 0;
}

FILE *open_file(char name[], int *f)
{
    FILE *fp;

    fp = fopen(name, "w");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    *f = 1;
    return fp;
}
```

**Comments:** This code is reliable with the assumption that the failure reason of `fopen()` is that the file does not exist, not some other reason. Also, we assume that no other running application will create the file in the time interval between checking and creating the file.

## Read Data from a Text File

The most common functions to read data from a text file are: `fscanf()`, `fgets()`, and `fgetc()`. These functions are mostly used with text files; however, we can use them to read data from binary files as well.

*For simplicity, we assume that if the function we used to read data returns EOF, it is because the end of file is reached, not because of a read error. However, a robust program should check the reason for failure.*

### The `fscanf()` Function

The `fscanf()` function is used to read a variable number of different data items from an input file. Like `scanf()`, `fscanf()` uses a format string to define the order and the type of data that will be read from the file. It is declared in `stdio.h`:

```
int fscanf(FILE* fp, const char *format, ...);
```

The only difference between them is that `scanf()` always reads from `stdin`, while `fscanf()` reads from the file indicated by `fp`.

As discussed, the operating system (e.g., *Unix*) may allow the redirection of `stdin`. For example, if the name of the executable is `test` and we write in the command line:

```
test <input.txt
```

the data read from `stdin` (e.g., with `scanf()`) won't be read from the keyboard, but from the `input.txt` file. Also, we can combine both redirections. For example:

```
test <input.txt >output.txt
```

`fscanf()` returns the number of data items successfully read from the file and assigned to program variables. If either the end of file is reached or the read operation fails, it returns `EOF`. Later, we'll see that we can use `feof()` to determine which one happened. The following program assumes that the `test.txt` text file contains the annual temperatures in an area. It reads them and displays those within  $[-5, 5]$ .

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
```

```

int k;
double temp;

fp = fopen("test.txt", "r");
if(fp == NULL)
{
    printf("Error: fopen() failed\n");
    exit(EXIT_FAILURE);
}
while(1)
{
    k = fscanf(fp, "%lf", &temp); /* Since fscanf() reads one
item, the return value 1 indicates that the value was successfully read
and assigned to temp. */
    if(k != 1)
        break;
    if(temp >= -5 && temp <= 5)
        printf("%f\n", temp);
}
fclose(fp);
return 0;
}

```

We could omit the declaration of `k` and write:

```
if(fscanf(fp, "%lf", &temp) != 1).
```

Also, we could put `fscanf()` inside the `while` statement and write it like this:

```
while(fscanf(fp, "%lf", &temp) == 1).
```

As a matter of style, we prefer the `while(1)` statement to make clearer the terminating condition of the loop.

We also suggest to test the return value of `fscanf()` against the number of the assigned items, rather than EOF. For example, suppose that the following code reads the floats contained in a text file:

```

float i;
while(fscanf(fp, "%d", &i) != EOF)
{
/* Do some work with i */
}

```

Although `i` is not assigned with a correct value due to the wrong specifier `%d`, `fscanf()` does not return EOF and the loop continues. Had we written:

```
while(fscanf(fp, "%d", &i) != 1)
```

the wrong assignment would have been traced and the loop would terminate.

Before using `fscanf()`, the programmer is supposed to know the type of the stored data and their order in the file. For example, in the following program, it is assumed that

the programmer knows that each line of the `test.txt` file contains a string of less than 100 characters, an integer and a **double** number, in order to pass the right arguments in `fscanf()`.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char str[100];
    int i;
    double d;

    fp = fopen("test.txt", "r");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    while(1)
    {
        if(fscanf(fp, "%s%d%lf", str, &i, &d) != 3)
            break;
        printf("%s %d %f\n", str, i, d);
    }
    fclose(fp);
    return 0;
}
```

Like `scanf()`, `fscanf()` uses the space character to distinguish the read values.

---

As with `scanf()`, be careful when using `%s` in `fscanf()` to read strings. If the string contains many words, only the first will be read.

Since `stdin` is associated with the keyboard by default, if we replace the first argument of `fscanf()` with `stdin`, a call of `fscanf()` is equivalent to a call of `scanf()`. For example, the following program uses `fscanf()` and `fprintf()`, instead of `scanf()` and `printf()`, to read an integer and a double number and display them.

```
#include <stdio.h>
int main(void)
{
    int i;
    double d;

    fprintf(stdout, "Enter an integer and a double: ");
    if(fscanf(stdin, "%d%lf", &i, &d) == 2)
        fprintf(stdout, "%d %f\n", i, d);
    return 0;
}
```

## Exercises

C.15.6 Write a program that reads products' codes (less than 20 characters each) and their prices and stores them in a text file, as follows:

```
C101 17.5  
C102 32.8  
...
```

If the user enters -1 for price, the insertion of products should terminate. Then, the program should read a product's code and search the file to find and display its price.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int read_text(char str[], int size, int flag);  
  
int main(void)  
{  
    FILE *fp;  
    char flag, str[20], prod[20];  
    double prc;  
  
    fp = fopen("test.txt", "w+"); /* Open file for reading and  
writing. */  
    if(fp == NULL)  
    {  
        printf("Error: fopen() failed\n");  
        exit(EXIT_FAILURE);  
    }  
    while(1)  
    {  
        printf("Enter price: ");  
        scanf("%lf", &prc);  
        if(prc == -1)  
            break;  
        getchar();  
        printf("Enter product code: ");  
        read_text(str, sizeof(str), 1);  
        fprintf(fp, "%s %f\n", str, prc);  
    }  
    getchar();  
    printf("Enter product code to search for: ");  
    read_text(prod, sizeof(prod), 1);  
  
    flag = 0;  
    fseek(fp, 0, SEEK_SET);  
    while(1)  
    {  
        if(fscanf(fp, "%s%lf", str, &prc) != 2)  
            break;
```

```

        if(strcmp(str, prod) == 0)
    {
        flag = 1;
        break; /* Since the product is found exit from the
loop. */
    }
if(flag == 0)
    printf("The %s product is not listed\n", prod);
else
    printf("The price for product %s is %f\n", prod, prc);
fclose(fp);
return 0;
}

```

**Comments:** We could use first the w mode to open the file to write data, then close it, and reopen it with the r mode to read data. To avoid that, we used the r+w mode and the fseek() function to rewind the file pointer at the beginning of the file and start reading from there. We'll talk about fseek() in a short.

**C.15.7** Suppose that each line of the students.txt text file contains the names of the students and their grades in two lessons, as follows:

John	Morne	7	8.12
Sahil	Nehrud	4.5	9
Koon	Lee	2	5.75

Write a program that reads each line of students.txt and stores in suc.txt the names and grades of the students with average grade greater than or equal to 5, while in fail.txt the students with average grade less than 5. To read the grades, use double variables and assume that the names are less than 100 characters. The program should display the number of students written in each file.

```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp_in, *fp_suc, *fp_fail;
    char fnm[100], lnm[100];
    int suc_stud, fail_stud;
    double grd1, grd2;

    fp_in = fopen("students.txt", "r");
    if(fp_in == NULL)
    {
        printf("Error: students.txt can't be loaded\n");
        exit(EXIT_FAILURE);
    }
    fp_suc = fopen("suc.txt", "w");
    if(fp_suc == NULL)
    {
        fclose(fp_in);
        printf("Error: suc.txt can't be created\n");
    }
    else
    {
        double sum = 0;
        int count = 0;
        while(fscanf(fp_in, "%s %s %d %d", fnm, lnm, &grd1, &grd2) != EOF)
        {
            double avg = (grd1 + grd2) / 2;
            if(avg >= 5)
                fprintf(fp_suc, "%s %s %d %.2f\n", fnm, lnm, count, avg);
            else
                fprintf(fp_fail, "%s %s %d %.2f\n", fnm, lnm, count, avg);
            count++;
        }
    }
}

```

```

        exit(EXIT_FAILURE);
    }
fp_fail = fopen("fail.txt", "w");
if(fp_fail == NULL)
{
    fclose(fp_in);
    fclose(fp_suc);
    printf("Error: fail.txt can't be created\n");
    exit(EXIT_FAILURE);
}
suc_stud = fail_stud = 0;
while(1)
{
    if(fscanf(fp_in, "%s%s%lf%lf", fnm, lnm, &grd1, &grd2) != 4)
        break;
    if((grd1+grd2)/2 >= 5)
    {
        fprintf(fp_suc, "%s %s %f %f\n", fnm, lnm, grd1, grd2);
        suc_stud++;
    }
    else
    {
        fprintf(fp_fail, "%s %s %f %f\n", fnm, lnm, grd1,
grd2);
        fail_stud++;
    }
}
printf("Failed: %d Succeeded: %d\n", fail_stud, suc_stud);
fclose(fp_suc);
fclose(fp_fail);
fclose(fp_in);
return 0;
}

```

## The fgets() Function

The fgets() function is used to read a number of characters from an input file. It is declared in stdio.h:

```
char *fgets(char *str, int size, FILE *fp);
```

The first argument points to the memory allocated to store the read characters. The second argument declares the maximum number of characters that will be read from the file indicated by fp. Its value should not be greater than the size of the allocated memory, in order to avoid a memory overflow.

fgets() stops reading characters once a new line character is read or size-1 characters have been read, whichever comes first. fgets() adds a null character at the end. If fgets() is executed successfully, the read characters are stored in the memory pointed to by str and this pointer is returned. If either the end of file is reached or a read error occurs, fgets() returns NULL.

The following program reads 50 names (less than 100 characters each) and stores them in different lines of a text file. Then, the program reads a character, reads the names from the file, and displays those that begin with the input character.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    char ch, str[100];
    int i, times;

    fp = fopen("test.txt", "w+");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < 50; i++)
    {
        printf("Enter name: ");
        read_text(str, sizeof(str), 1);
        fprintf(fp, "%s\n", str);
    }
    printf("Enter char: ");
    ch = getchar();

    fseek(fp, 0, SEEK_SET);
    times = 0;
    while(1)
    {
        if(fgets(str, sizeof(str), fp) == NULL)
            break;
        if(str[0] == ch)
        {
            printf("Name: %s\n", str);
            times++;
        }
    }
    printf("Total occurrences = %d\n", times);
    fclose(fp);
    return 0;
}
```

## Exercise

C.15.8 Write a program that reads the names of two files, compares them line by line (assume that each line contains less than 100 characters), and displays their first common line. If the two files have no common line, the program should display a related message.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp1, *fp2;
    char flag, str1[100], str2[100];

    printf("Enter first file: ");
    read_text(str1, sizeof(str1), 1);

    fp1 = fopen(str1, "r");
    if(fp1 == NULL)
    {
        printf("Error: fopen() for %s failed\n", str1);
        exit(EXIT_FAILURE);
    }
    printf("Enter second file: ");
    read_text(str2, sizeof(str2), 1);

    fp2 = fopen(str2, "r");
    if(fp2 == NULL)
    {
        fclose(fp1);
        printf("Error: fopen() for %s failed\n", str2);
        exit(EXIT_FAILURE);
    }
    flag = 0;
    while(1)
    {
        if((fgets(str1, sizeof(str1), fp1) == NULL)
           || (fgets(str2, sizeof(str2), fp2) == NULL))
            break; /* We check if a read error occurred or the
end of a file is reached. In either case, the loop terminates. */
        if(strcmp(str1, str2) == 0)
        {
            printf("The same line is: %s\n", str1);
            flag = 1;
            break; /* Since a common line is found, exit from the
loop. */
        }
    }
}
```

```
    if(flag == 0)
        printf("There is no common line\n");

    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

## The `fgetc()` Function

The `fgetc()` function is used to read a character from an input file. It is declared in `stdio.h`:

```
int fgetc(FILE *fp);
```

If `fgetc()` is executed successfully, it returns the read character as `unsigned char` cast to `int`. If either the end of file is reached or the read operation fails, it returns `EOF`. `getc()` is similar to `fgetc()`. Like `putc()` and `fputc()`, their difference is that `getc()` is usually implemented as a macro; therefore, it'd be executed faster.

---

As with `getchar()`, always store the return value of `fgetc()` and `getc()` into an `int` variable, not `char`.

---

For example, suppose that we use `getc()` to read characters from a file that contains the value 255. If the `char` type is signed, the value that will be stored in a `char` variable is -1 when 255 is read. As a result, a program that compares its value against `EOF` to determine if there are more characters to read, would stop reading.

---

## Exercises

C.15.9 Write a program that reads the name of a file and displays its second line.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    char str[100];
    int ch, lines;

    printf("Enter file name: ");
    read_text(str, sizeof(str), 1);

    fp = fopen(str, "r");
```

```

if(fp == NULL)
{
    printf("Error: fopen() failed\n");
    exit(EXIT_FAILURE);
}
printf("\nLine contents\n");
lines = 1;
while(1)
{
    ch = getc(fp);
    if((ch == EOF) || (lines > 2))
        break;
    if(ch == '\n') /* Once the new line character is read,
increment the variable that counts the lines. */
        lines++;
    if(lines == 2) /* Only the characters of the second line are
displayed. */
        printf("%c", ch);
}
fclose(fp);
return 0;
}

```

**Comments:** Once the program displays the second line, the loop terminates.

**C.15.10** Assume that the `test.txt` is a text file. What would be the output of the following program?

```

#include <stdio.h>
int main(void)
{
    FILE *fp;
    char ch;

    if((fp = fopen("test.txt", "r")) != NULL)
    {
        while(ch = getc(fp) != EOF)
            putc(ch, stdout);
        fclose(fp);
    }
    return 0;
}

```

**Answer:** If the `while` statement had been written correctly, the program would have displayed the file's characters. Its correct syntax is:

```
while((ch = getc(fp)) != EOF)
```

The inner parentheses are needed for reasons of priority. Since they are missing, the expression `getc(fp) != EOF` is first executed and its value is 1, as long as the end of file is not reached. Therefore, `ch` becomes 1 and the program displays continuously the character with that ASCII value. When `getc()` returns EOF, `ch` becomes 0 and the loop terminates.

C.15.11 A simple way to encrypt data is to XOR them with a secret key. Write a program that reads a key character and the name of a text file and encrypts its content by XORing each character with the key. The encrypted characters should be stored in a second file selected by the user.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp_in, *fp_out;
    char str[100];
    int ch, key_ch;

    printf("Enter input file: ");
    read_text(str, sizeof(str), 1);

    fp_in = fopen(str, "r");
    if(fp_in == NULL)
    {
        printf("Error: Input file can't be loaded\n");
        exit(EXIT_FAILURE);
    }
    printf("Enter output file: ");
    read_text(str, sizeof(str), 1);

    fp_out = fopen(str, "w");
    if(fp_out == NULL)
    {
        fclose(fp_in);
        printf("Error: Output file can't be created\n");
        exit(EXIT_FAILURE);
    }
    printf("Enter key char: ");
    key_ch = getchar();
    while(1)
    {
        ch = getc(fp_in);
        if(ch == EOF)
            break;
        putc(ch ^ key_ch, fp_out);
    }
    fclose(fp_in);
    fclose(fp_out);
    return 0;
}
```

**Comments:** If you rerun the program and give as an input the encrypted file and the same key, the output file would be the same as the original file, because according to the Boolean algebra we have:  $(a \wedge b) \wedge b = a$ .

C.15.12 Define a structure of type country with members: name, capital, and population. Assume that the names are less than 100 characters. Suppose that a text file contains the data of several countries. The first line contains the number of countries and the following lines store the data of each country in the form:

```
name    capital    population
```

Write a program that reads the name of the file and uses the country type to store the countries' data in a dynamically allocated memory. Then, the program should read a number and display the countries with higher population than this number.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

typedef struct
{
    char name[100];
    char capital[100];
    int pop;
} country;

int main(void)
{
    FILE *fp;
    country *cntr;
    char str[100];
    int i, num_cntr, pop;

    printf("Enter file name: ");
    read_text(str, sizeof(str), 1);

    fp = fopen(str, "r");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    if(fscanf(fp, "%d", &num_cntr) != 1)
    {
        fclose(fp);
        printf("Error: fscanf() failed\n");
        exit(EXIT_FAILURE);
    }
    /* Dynamic memory allocation to store the countries' data. */
    cntr = (country*) malloc(num_cntr * sizeof(country));
    if(cntr == NULL)
    {
        fclose(fp);
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
```

```

    for(i = 0; i < num_cntr; i++)
        if(fscanf(fp, "%s%s%d", cntr[i].name, cntr[i].capital,
&cntr[i].pop) != 3)
    {
        fclose(fp);
        printf("Error: fscanf() read error\n");
        exit(EXIT_FAILURE);
    }
    fclose(fp);

    printf("Enter population: ");
    scanf("%d", &pop);
    for(i = 0; i < num_cntr; i++)
        if(cntr[i].pop >= pop)
            printf("%s %s\t%d\n", cntr[i].name, cntr[i].capital,
cntr[i].pop);

    free(cntr);
    return 0;
}

```

## End of File

As discussed, the operating system may add a special character at the end of a text file to mark its end, whereas none character marks the end of a *binary* file. In *Dos/Windows* applications, for example, the Ctrl+Z character with ASCII value 26 is typically used to mark the end of a *text* file. For example, suppose that the following program runs in such a system:

```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    int ch;

    fp = fopen("test.txt", "w+");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    fprintf(fp, "%c%c%c%c%c\n", 'a', 'b', 26, 'c', 'd');
    fseek(fp, 0, SEEK_SET);
    while(1)
    {
        ch = getc(fp);
        if(ch == EOF)

```

```
        break;
        printf("%c", ch);
    }
fclose(fp);
return 0;
}
```

The loop displays only the characters a and b and the next call of `getc()` ends it, since the next character is interpreted as the end of file. On the other hand, had we used the "w+b" mode, the program would have displayed all the characters since none character has a special significance in a *binary* file.

---

## The `fseek()` and `ftell()` Functions

Each read or write operation is normally performed sequentially. When necessary, however, a file can be read or written in any order. The `fseek()` function is used for random access within a file. It is declared in `stdio.h`:

```
int fseek(FILE *fp, long int offset, int origin);
```

`fseek()` moves the file pointer associated with `fp` to a new location `offset` bytes from the point indicated by `origin`. If `offset` is negative, the file pointer moves back. The value of `origin` must be one of the following constants, defined in `stdio.h`:

- `SEEK_SET`. The file pointer is moved `offset` bytes from the beginning of the file.
- `SEEK_CUR`. The file pointer is moved `offset` bytes from its current position.
- `SEEK_END`. The file pointer is moved `offset` bytes from the end of the file.

For example, to move to the end of the file, we would write:

```
fseek(fp, 0, SEEK_END);
```

To move 20 bytes from the beginning of the file, we would write:

```
fseek(fp, 20, SEEK_SET);
```

If `fseek()` is executed successfully, it returns 0. Otherwise, it returns a nonzero value.

When `fseek()` is used with text files, care is required with the new line character(s). For example, suppose that the following code writes some text in the first two lines of a text file. If the operating system expands '\n' to '\r' and '\n', the value of `offset` should be 6 (not 5) in order to move to the beginning of the second line.

```
fputs("text\n", fp);
fputs("another text\n", fp);
/* Move to the beginning of the second line. */
fseek(fp, 6, SEEK_SET);
```

The `ftell()` function is used to find the current position of a file pointer. It is declared in `stdio.h`:

```
long int ftell(FILE *fp);
```

`ftell()` returns the current position of the file pointer associated with `fp`. The position is expressed as the number of bytes from the beginning of the file. If an error occurs, `ftell()` returns `-1`. `ftell()` may be used together with `fseek()` to return to a previous file position, like this:

```
long int old_pos;  
/* ... open file and so some work with it. */  
old_pos = ftell(fp);  
/* ... do some other work. */  
fseek(fp, old_pos, SEEK_SET); /* return back to the old position. */
```

---

It is safer to use `fseek()` and `ftell()` with binary rather than text files because potential translations of the new line character can produce unexpected results. `fseek()` is guaranteed to work with text files only if:

- a. offset is 0 or
  - b. offset is obtained from a previous call to `ftell()` and origin is set to `SEEK_SET`.
- 

## Write and Read Data from a Binary File

The `fwrite()` and `fread()` functions are used to write and read data from a binary file. These functions are used primarily with binary files, although we can use them, with some care, with text files as well.

### The `fwrite()` Function

The `fwrite()` function is very useful for writing large blocks of data in a single step. It is declared in `stdio.h`:

```
int fwrite(const void *buf, size_t size, size_t count, FILE *fp);
```

The first argument points to the memory that holds the data to be written in the file indicated by `fp`. The third argument specifies how many elements will be written in the file, while the second one specifies the size of one element in bytes. The product of the second and third arguments specifies the number of the bytes to be written in the file. For example, to write an array of 1000 integers, the second argument should be equal to the size of one integer, that is, 4, while the third argument should be equal to 1000.

```
int arr[1000];  
fwrite(arr, 4, 1000, fp);
```

The best practice is to use the `sizeof` operator to specify the size of one element, in order to make your program platform independent. For example, it'd be better to write:

```
fwrite(arr, sizeof(int), 1000, fp);
```

As another example, the following code writes one `double` number:

```
double a = 1.2345;
fwrite(&a, sizeof(double), 1, fp);
```

`fwrite()` returns the number of the elements actually written in the file. If the return value equals the third argument, it is implied that the `fwrite()` was executed successfully. If not, a write error occurred.

Some programmers prefer to declare the total memory size and make the number of elements equal to 1. For example:

```
fwrite(arr, 1000*sizeof(int), 1, fp);
```

Don't do that. If `fwrite()` fails, it will return 0 and you'll have no idea if any and how many integers were stored in the file. The same applies for `fread()`.

## The `fread()` Function

Like `fwrite()`, `fread()` is very useful for reading large blocks of data in a single step. It is declared in `stdio.h`:

```
int fread(void *buf, size_t size, size_t count, FILE *fp);
```

The first argument points to the memory in which the read data will be stored. Like `fwrite()` arguments, the second argument specifies the size of one element in bytes, while the third one specifies how many elements will be read from the file.

---

Because two systems may differ (e.g., the size of types), there is no guarantee that a file written with the `fwrite()` function will be read correctly with the `fread()` function in another system.

---

`fread()` returns the number of the elements actually read from the file. As with `fwrite()`, this value should be tested against the third argument. If they are equal, `fread()` was executed successfully. Otherwise, either the end of file is reached or a read error occurred. In the following example, a `double` number is read and stored into `a`:

```
double a;
fread(&a, sizeof(double), 1, fp);
```

As another example, 1000 integers are read and stored into array `arr`:

```
int arr[1000];
fread(arr, sizeof(int), 1000, fp);
```

---

It is safer to use `fwrite()` and `fread()` with binary rather than text files because potential translations of the new line character can produce unexpected results.

For example, suppose that we use `fwrite()` to write a string of 50 characters in a text file and the value of `count` is set to 50. If the program runs in *Windows* and the string contains new line character(s), the replacement(s) with the `\r\n` pair would make its size more than 50; therefore, `fwrite()` won't write the entire string.

---

## Exercises

C.15.13 Write a program that declares an array of 5 integers with values 10, 20, 30, 40, and 50 and writes them in a binary file. Then, the program should read an integer and replace the third stored integer with the input number. The program should read and display the file's content before it ends.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

int main(void)
{
    FILE *fp;
    int i, arr[SIZE] = {10, 20, 30, 40, 50};

    fp = fopen("test.bin", "w+b"); /* Open binary file for reading and
writing. */
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    fwrite(arr, sizeof(int), SIZE, fp);

    printf("Enter new value: ");
    scanf("%d", &i);
    fseek(fp, 2*sizeof(int), SEEK_SET); /* Since each integer is 4
bytes, fseek() moves the file pointer 2*sizeof(int) = 8 bytes from the
beginning of the file to get to the third integer. */
    fwrite(&i, sizeof(int), 1, fp);

    fseek(fp, 0, SEEK_SET);
    if(fread(arr, sizeof(int), SIZE, fp) != SIZE)
    {
        fclose(fp);
        printf("Error: fread() failed\n");
        exit(EXIT_FAILURE);
    }
}
```

```

printf("\n***** File contents *****\n");
for(i = 0; i < SIZE; i++)
    printf("%d\n", arr[i]);

fclose(fp);
return 0;
}

```

**C.15.14** Write a program that reads the titles of 10 books (less than 100 characters each) and writes them in a user selected binary file. First write the size of the title, then the title. Next, the program should read a title and display a message to indicate if it is contained in the file or not.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    char found, str[100], tmp_str[100];
    int i, len;

    printf("Enter file name: ");
    read_text(str, sizeof(str), 1);

    fp = fopen(str, "w+b");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < 10; i++)
    {
        printf("Enter text: ");
        len = read_text(str, sizeof(str), 1);

        fwrite(&len, sizeof(int), 1, fp);
        fwrite(str, 1, len, fp);
    }
    printf("Enter title to search: ");
    read_text(tmp_str, sizeof(tmp_str), 1);

    found = 0;
    fseek(fp, 0, SEEK_SET);
    while(1)
    {
        if(fread(&len, sizeof(int), 1, fp) != 1)
            break;
        if(fread(str, 1, len, fp) != len)
            break;
        str[len] = '\0';

```

```

        if(strcmp(str, tmp_str) == 0)
    {
        found = 1;
        break;
    }
}
if(found == 0)
    printf("\n%s isn't found\n", tmp_str);
else
    printf("\n%s is found\n", tmp_str);

fclose(fp);
return 0;
}

```

**C.15.15** Suppose that the purpose of the following program is to write a string into a binary file, read it from the file, and display it. Is this code error-free?

```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE* fp;
    char str1[5], str2[] = "test";

    if((fp = fopen("text.bin", "w+b")) != NULL)
    {
        fwrite(str2, 1, 4, fp);
        fread(str1, 1, 4, fp);
        printf("%s\n", str1);
        fclose(fp);
    }
    return 0;
}

```

**Answer:** As you guess, there are several errors. The first bug is that the file pointer is not rewinded to the beginning of the file before calling `fread()`. This bug is eliminated by adding:

```
fseek(fp, 0, SEEK_SET);
```

between `fwrite()` and `fread()`.

Now, `fread()` reads successfully the four characters and stores them into `str1`.

The second bug is that `str1` is not null terminated; therefore, `printf()` won't work properly. To eliminate this bug, add this statement before `printf()`:

```
str1[4] = '\0';
```

or initialize `str1` like: `char str1[5] = {0};`

**C.15.16** Suppose that the `test.bin` binary file contains a student's grades. The number of grades is declared in the beginning of the file. Write a program that reads the grades

from the binary file (use the `float` type) and stores them in a dynamically allocated memory. Then, the program should read a number and display the grades greater than that number.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    int i, grd_num;
    float *grd_arr, grd;

    fp = fopen("test.bin", "rb");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    if(fread(&grd_num, sizeof(int), 1, fp) != 1)
    {
        fclose(fp);
        printf("Error: fread() failed\n");
        exit(EXIT_FAILURE);
    }
    grd_arr = (float*) malloc(grd_num * sizeof(float)); /* Allocate
memory to store the grades. */
    if(grd_arr == NULL)
    {
        fclose(fp);
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
    /* Read all grades and check if they are read successfully. */
    if(fread(grd_arr, sizeof(float), grd_num, fp) == grd_num)
    {
        printf("Enter grade: ");
        scanf("%f", &grd);
        for(i = 0; i < grd_num; i++)
            if(grd_arr[i] > grd)
                printf("%f\n", grd_arr[i]);
    }
    else
        printf("Error: fread() failed to read grades\n");

    free(grd_arr);
    fclose(fp);
    return 0;
}
```

**C.15.17** A common method that antivirus software uses to identify viruses is signature-based detection. The signature is a sequence of bytes (e.g., F3 BA 20 63 7A 1B) that identify a particular virus. When a file is scanned, the antivirus software searches the

file for signatures that identify the presence of viruses. Write a program that reads a virus signature (e.g., 5 integers) and checks if it is contained in the binary file test.dat.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 5

int main(void)
{
    FILE *fp;
    int i, found, len, back, buf[SIZE], pat[SIZE];

    fp = fopen("test.dat", "rb");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Enter virus signature (%d integers)\n", SIZE);
    for(i = 0; i < SIZE; i++)
    {
        printf("Enter number: ");
        scanf("%d", &pat[i]);
    }
    len = sizeof(pat);
    found = 0;
    back = len(sizeof(int));
    while(1)
    {
        if(fread(buf, sizeof(int), SIZE, fp) != SIZE)
            break;
        if(memcmp(buf, pat, len) == 0)
        {
            found = 1;
            break;
        }
        else
            fseek(fp, -back, SEEK_CUR); /* Go back to check the
next group of five. */
    }
    if(found == 1)
        printf("SOS: Virus found\n");
    else
        printf("That virus signature isn't found\n");
    fclose(fp);
    return 0;
}
```

C.15.18 Define a structure of type employee with members: name (less than 100 characters), tax number, and salary. Write a program that uses this type to read the data of 100

employees and store them in an array of such structures. If the user enters fin for name, the data insertion should terminate and the program should write the structures in the test.bin binary file.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100

struct employee
{
    char name[100];
    int tax_num;
    int salary;
};

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    int i, num_empl;
    struct employee empl[SIZE];

    fp = fopen("test.bin", "wb");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    num_empl = 0;
    for(i = 0; i < SIZE; i++)
    {
        printf("\nEnter full name: ");
        read_text(empl[i].name, sizeof(empl[i].name), 1);
        if(strcmp(empl[i].name, "fin") == 0)
            break;

        printf("Enter tax number: ");
        scanf("%d", &empl[i].tax_num);

        printf("Enter salary: ");
        scanf("%d", &empl[i].salary);

        num_empl++;
        getchar();
    }
    /* Write all structures in a single step. */
    fwrite(empl, sizeof(struct employee), num_empl, fp);
    fclose(fp);
    return 0;
}
```

C.15.19 Suppose that the test.bin binary file contains structures of the type employee defined in the previous exercise. Write a program that reads them and copies the employees' data whose salary is more than an input amount in the data.bin binary file. The program should also display the average salary of the employees stored in data.bin.

```
#include <stdio.h>
#include <stdlib.h>

struct employee
{
    char name[100];
    int tax_num;
    int salary;
};

int main(void)
{
    FILE *fp_in, *fp_out;
    int count, amount, sum_sal;
    struct employee tmp_emp;

    fp_in = fopen("test.bin", "rb");
    if(fp_in == NULL)
    {
        printf("Error: Input file can't be loaded\n");
        exit(EXIT_FAILURE);
    }
    fp_out = fopen("data.bin", "wb");
    if(fp_out == NULL)
    {
        fclose(fp_in);
        printf("Error: Output file can't be created\n");
        exit(EXIT_FAILURE);
    }
    printf("Enter amount: ");
    scanf("%d", &amount);
    count = sum_sal = 0;
    while(1)
    {
        if(fread(&tmp_emp, sizeof(employee), 1, fp_in) != 1)
            break;
        if(tmp_emp.salary > amount)
        {
            fwrite(&tmp_emp, sizeof(employee), 1, fp_out);
            sum_sal += tmp_emp.salary;
            count++;
        }
    }
    if(count)
        printf("Avg = %.2f\n", (float)sum_sal/count);
    else
        printf("No employee gets more than %d\n", amount);
```

```
fclose(fp_in);
fclose(fp_out);
return 0;
}
```

C.15.20 Define a structure of type band with members: name, category, singer (all less than 100 characters), and records. Suppose that the test.bin binary file contains such structures. The number of bands is declared in the beginning of the file. Write a program that reads the file and uses the structure type to store the data in a dynamically allocated memory. Then, the program should read the name of a band, a new singer, and replace the existing singer with the new one.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LEN 100

typedef struct
{
    char name[LEN];
    char category[LEN];
    char singer[LEN];
    int records;
} band;

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    band *band_arr;
    char found, name[LEN], singer[LEN];
    int i, band_num;

    fp = fopen("test.bin", "r+b");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    if(fread(&band_num, sizeof(int), 1, fp) != 1)
    {
        fclose(fp);
        printf("Error: fread() failed\n");
        exit(EXIT_FAILURE);
    }
    band_arr = (band*) malloc(sizeof(band) * band_num);
    if(band_arr == NULL)
    {
        fclose(fp);
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
```

```

if(fread(band_arr, sizeof(band), band_num, fp) == band_num)
{
    printf("Enter band name: ");
    read_text(name, sizeof(name), 1);

    printf("Enter new singer: ");
    read_text(singer, sizeof(singer), 1);

    found = 0;
    for(i = 0; i < band_num; i++)
        if(strcmp(band_arr[i].name, name) == 0)
        {
            fseek(fp, i*sizeof(band), SEEK_SET); /* If the
band is found, move the file pointer to the beginning of the structure. */
            strcpy(band_arr[i].singer, singer); /* Change
the singer and write the structure in the current position. */
            fwrite(&band_arr[i], sizeof(band), 1, fp);
            printf("\n%s Singer of band %s is changed to
%s\n", name, singer);
            found = 1;
            break;
        }
    }
    else
        printf("Error: fread() failed to read bands\n");
    if(found == 0)
        printf("\n%s band isn't found\n\n", name);

    free(band_arr);
    fclose(fp);
    return 0;
}

```

**C.15.21** Suppose that the number of students is stored in the beginning of the test. bin binary file. The next group of fives declares the grades of the first student in five courses. The next group declares the five grades of the second student in the same courses, and so on. Write a program that reads the file and displays the average grade of each student.

```

#include <stdio.h>
#include <stdlib.h>

#define LESSONS 5

int main(void)
{
    FILE *fp;
    int i, j, stud_num;
    float **grd_arr, sum_grd;

    fp = fopen("test.bin", "rb");
    if(fp == NULL)

```

```

{
    printf("Error: fopen() failed\n");
    exit(EXIT_FAILURE);
}
if(fread(&stud_num, sizeof(int), 1, fp) != 1)
{
    fclose(fp);
    printf("Error: fread() failed to read number\n");
    exit(EXIT_FAILURE);
}
/* We use grd_arr as a two-dimensional array of stud_num rows and
LESSONS columns. First, we allocate memory for the rows. Next, we
allocate memory for each row. Each row holds the grades of the student in
LESSONS courses. */
grd_arr = (float**) malloc(stud_num * sizeof(float*));
if(grd_arr == NULL)
{
    fclose(fp);
    printf("Error: Not available memory\n");
    exit(EXIT_FAILURE);
}
for(i = 0; i < stud_num; i++)
{
    grd_arr[i] = (float*) malloc(LESSONS * sizeof(float));
    if(grd_arr[i] == NULL)
    {
        fclose(fp);
        printf("Error: Not available memory\n");
        exit(EXIT_FAILURE);
    }
}
for(i = 0; i < stud_num; i++)
{
    sum_grd = 0;
    if(fread(grd_arr[i], sizeof(float), LESSONS, fp) == LESSONS)
    {
        for(j = 0; j < LESSONS; j++)
            sum_grd += grd_arr[i][j];

        printf("%d. %f\n", i+1, sum_grd/LESSONS);
    }
    else
    {
        printf("Error: fread() failed\n");
        break; /* Stop reading. */
    }
}
for(i = 0; i < stud_num; i++)
    free(grd_arr[i]);

free(grd_arr);
fclose(fp);
return 0;
}

```

**Comments:** We could present a simpler solution similar to C.15.16. That is, we could declare `grd_arr` as an ordinary pointer, store the grades in the allocated memory, and read the grades from this memory in groups of five, in order to calculate the average grade of each student. The reason we choose this solution is to remember how to handle the allocated memory as a two-dimensional array.

---

## The `feof()` Function

The `feof()` function is used to determine whether the end of file is reached. It is declared in `stdio.h`:

```
int feof(FILE *fp);
```

If a read operation attempts to read beyond the end of file indicated by `fp`, `feof()` returns a nonzero value, 0 otherwise. As we've seen, when we are using a read function we examine its return value to determine if it is executed successfully or not. If the operation is unsuccessful, we can use `feof()` to determine whether the failure was due to an end of file condition or for another reason. For example, the following program reads the contents of a text file (assume that each line has less than 100 characters) and if `fgets()` fails, we use `feof()` to see why.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char str[100];

    fp = fopen("test.txt", "r");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    while(1)
    {
        if(fgets(str, sizeof(str), fp) != NULL)
            printf("%s", str);
        else
        {
            if(feof(fp))
                printf("End of file\n");
            else
                printf("Failed for another reason\n");
            break;
        }
    }
    fclose(fp);
    return 0;
}
```

## Exercise

C.15.22 Assume that each line of the `test.c` text file contains less than 100 characters. What does the following “badly written” program do?

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    char str[100];

    for(fp = fopen("test.c", "r"); fp && !feof(fp); fgets(str,
sizeof(str), fp) ? printf("%s", str) : 1);
    return fp ? fclose(fp) : 0;
}
```

**Answer:** The program uses `fgets()` to read and display each line of `test.c`, while the loop is executed as long as the end of file is not reached. If `fopen()` fails, `fp` would be equal to `NULL`, the loop won't be executed and the program returns `0`. Otherwise, `fclose()` closes the file and the program returns its return value.

## Unsolved Exercises

U.15.1 Suppose that the `grades.txt` text file contains the grades of a number of students. Write a program that reads the file and displays the best and the worst grades, the average grade of those who failed (`grade < 5`), and the average grade of those who succeeded (`grade ≥ 5`). Suppose that the grades are within  $[0, 10]$ .

U.15.2 Write a program that finds the sequential doubled words (e.g., “In this this chapter we we present...”) in a user selected text file and writes them into another text file. Assume that each word has less than 100 characters.

U.15.3 Suppose that each line of the `students.txt` text file contains the grades of the students in three courses. Write a program that reads the file and displays the average grade of each course. For example, if the file content is:

5	3.5	9
9	6	4.5

the program should display: 7 4.75 6.75

U.15.4 Write a program that checks if two user selected text files have the same content.

U.15.5 Write a program that converts the uppercase letters of a user selected text file to lowercase letters and vice versa. *Note:* Don't forget to use `fseek()` between successive read and write operations.

**U.15.6** Write a program that reads the names of two text files from the command line and appends the content of the second file into the first one. Then, the program should display the content of the first file. Assume that each line contains less than 100 characters.

**U.15.7** Write a program that reads a name of a text file and copies its content in reverse order, that is, from the last character up to the first one, into a second user selected text file.  
*Hint:* Open the first file as binary.

**U.15.8** Suppose that each line of the `students.txt` text file contains the name of a student (less than 100 characters) and his grades in three courses. Write a program that reads the file and displays the name(s) of the student(s) with the best average grade. Assume that there are fewer than 300 students and more than one student may have the same best average grade.

**U.15.9** Write a program that reads the name of a text file from the command line and displays its longest line. Assume that there is no more than one line with the same longer length. The program should also display the occurrences of each digit [0, 9] in the file. Furthermore, the program should read an integer (e.g., `x`) and display the last `x` lines of the file. The program should check that the input number is not greater than the number of the total lines. For example, if the user enters 3 the program should display the last three lines of the file. There is one restriction; the program should read each line only once. Assume that there are fewer than 1000 lines and each line contains fewer than 100 characters.

**U.15.10** Write a program that reads the name of a text file and the number of a line and replaces that line with a new input text. If the line has more characters than the new text, the extra characters until the end of line should be replaced with the space character. If it has fewer, the replacement should be done until the end of line. It is mandatory to use `getc()` and `putc()` to read and write to the file and `getchar()` to read the new text. For example, if the file content is:

First  
Second  
Third

and the user enters as line number 2 and the text New, the file becomes:

First  
New  
Third

if the users enters 1 and Introduction, it becomes:

Intro  
Second  
Third

**U.15.11** Define the structure type `book` with members: title, author, and price. Suppose that the `book.dat` binary file contains 100 of those structures. Write a program that reads the sequence number of a book entry (e.g., 25), the new data, and replace the existing data

with the new data. Then, the program should read from the file the data of that entry and display them, in order to verify that it was written correctly.

**U.15.12** Consider the file book.dat of the previous exercise. Write a program that reads the existing entries and writes them in the book\_rvs.dat binary file in reverse order, that is, the last entry should be written first, the last but one written second, and so on.

**U.15.13** Suppose that the grades.dat binary file contains the grades of a number of students. The number of students is stored in the beginning of the file. Write a program that reads the file and writes the grades sorted in ascending order in the grd\_sort.dat binary file.

## The Preprocessor

Although the preprocessor can be a separate program, it is typically integrated with the compiler to improve compilation speed. It is a software program that processes a C program before it is compiled. Its major capabilities involve file inclusion, macro substitution, and conditional compilation. First, we'll discuss how to define and use macros in a C program. We'll devote the rest of the chapter to describe preprocessor directives and operators.

### Simple Macros

The behavior of the preprocessor is controlled by directives. The directives are special commands that begin with a # character and instruct the preprocessor to do something. For example, the `#include` directive instructs the preprocessor to open a specified file and include its contents into the program. Directives can appear anywhere in the program. A line that contains a single # is legal; it has no effect, though.

Another directive we've been using is the `#define` directive. We've used it to define a simple macro, that is, a symbolic name associated with a constant value. To define a simple macro we write:

```
#define macro_name replacement_characters
```

The `macro_name` has the same form as a variable name. The `replacement_characters` are arbitrary. Typically, most programmers choose capital letters to name macros (we do the same), in order to distinguish them from program variables. The scope of the name begins from the point of its definition until the end of the file. For example, in the following program:

```
#include <stdio.h>

#define LEN 200

int main(void)
{
    int i, arr[LEN];

    for(i = 0; i < LEN; i++)
        arr[i] = i+LEN;
    return 0;
}
```

the preprocessor replaces each occurrence of LEN with its defined value, that is, 200. A common error is to add a = in the definition. For example, if we write:

```
#define LEN = 200
```

the definition of arr expands to arr[= 200] and the compilation fails. Another common error is to add a semicolon at the end. For example:

```
#define LEN 200;
```

the definition expands to arr[200;] and the compilation fails.

---

If a macro contains operators, enclose it in parentheses.

For example, see what happens if we omit the parentheses:

```
#define NUM 2*5 /* Instead of (2*5). */
```

a statement like this:

```
double j = 3.0/NUM;
```

is expanded to: j = 3.0/2\*5 = 7.5; which assigns the value 7.5 to j, not 0.3, because the division is performed first.

A macro can be defined anywhere in a program, for example, inside a function. The usual practice is to define all macros with global scope at the top of the program, or in a header file, and use the `#include` directive to include it when needed. For example, suppose that we've created the test.h file with content:

```
#include <stdio.h>
#define LEN 200
```

could we write the previous program as follows?

```
#include "test.h"
int main(void)
{
    ...
}
```

Sure, no problem. It is not mandatory to include explicitly stdio.h. We can include another file, as we did here with test.h, and inside there to include stdio.h.

If the name of a macro is part of the name of a variable or contained in a string literal, it is not replaced. For example:

```
#include <stdio.h>

#define LEN 200

int main(void)
{
    int BUF_LEN; /* No replacement takes place. */
```

```
    printf("LEN is not used\n"); /* No replacement takes place. */
    return 0;
}
```

Although simple macros are primarily used to define names for numbers, they can be used for other purposes as well. For example, in the following program:

```
#include <stdio.h>

#define test printf("example")

int main(void)
{
    test;
    return 0;
}
```

the preprocessor replaces the `test` macro with `printf("example")` and the program displays `example`.

Notice that it is legal to define a macro with no replacement value. As we'll see later, this kind of macros is typically used for controlling conditional compilation. For example:

```
#define LABEL
```

The name of a macro can be used in the definition of another macro. For example:

```
#define SIZE 200
#define NUM SIZE
```

---

Remember that first the preprocessor replaces the macro names with the corresponding values and then the program is compiled.

To extend a macro in several lines, add the backslash character (\) at the end of each line. For example, the following program displays 60:

```
#include <stdio.h>

#define NUM \
    10 + \
    20 + \
    30

int main(void)
{
    printf("%d\n", NUM);
    return 0;
}
```

As discussed in Chapter 2, when a value appears a lot, it is advisable to use a macro name instead because it is much easier to change that value if needed. We just change the macro definition. Furthermore, well chosen names in place of hard coded values improve

the code readability. It is much easier for the reader to understand the meaning of the constant.

Notice also that several compilers allow defining a macro name, when the program is compiled. For example:

```
gcc -DVERSION=3 test.c
```

The option `-D` defines the `VERSION` macro with value 3. If no value is specified, the default is 1. This ability is very useful because it is not needed to modify a file to define or change a value.

In C, the macros in Table 16.1 are predefined. Their names are enclosed in double underscores. Their main purpose is to provide special information about the current compilation.

For example, consider the following program:

```
#include <stdio.h>
int main(void)
{
    printf("Line %d of file %s compiled on %s at %s\n",
__FILE__, __DATE__, __TIME__);
    return 0;
}
```

The program outputs

```
Line 4 of file c:\edu\projects\test.c compiled on Jun 16 2016 at 17:58:14
```

This output is shown each time the program runs. If we recompile and run the program, the new compile date and time will be displayed. In case of various versions of the same program, this information might be useful to distinguish the running version. Furthermore, we can use the `__LINE__` and `__FILE__` macros to locate errors. For example:

```
if(error)
    printf("Error in line %d of file %s\n", __LINE__, __FILE__);
```

For the same reason, we can use the `assert()` macro. It is declared in `assert.h`:

```
void assert(int exp);
```

**TABLE 16.1**

Predefined Macros

Name	Description
<code>__DATE__</code>	Date of compilation.
<code>__FILE__</code>	Name of compiled file.
<code>__LINE__</code>	Line number of the compiled file.
<code>__TIME__</code>	Time of compilation.
<code>__STDC__</code>	It is 1, if the compiler supports ANSI C.

If `exp` is true, nothing is displayed. Otherwise, `assert()` outputs an error message to `stderr` and calls `abort()` to terminate the program. The message contains `exp` in text form, the file name, and the line of `assert()`. For example:

```
assert(a != 0); /* Before division, the value of denominator is checked.  
*/  
c = b/a;
```

`assert()`, `if` conditions, and directives coming later (e.g., `#ifdef`) can be used for the same reason, that is, to detect potential bugs during the development phase. Once the program is tested, those checks should be removed, in order not to increase the execution time, particularly in time-critical applications. `if` conditions are the less flexible way, because once removed and the program fails again, it'd take some time to put them back. Using directives (e.g., `#ifdef`), things are much simpler, we just define or not a macro. The same with `assert()`. To disable `assert()` we just define the macro `NDEBUG` before including the `assert.h`. If we need to reactivate `assert()` we remove the definition of `NDEBUG`. For example:

```
#define NDEBUG  
#include <assert.h>
```

One more thing, don't put a function call inside `assert()`. For example, don't write something like this:

```
assert((len = strlen(str)) > 10);  
/* Use len. */
```

if `NDEBUG` is defined, `assert()` will be ignored and `strlen()` won't be called.

Before continuing, can you find the errors in the following program?

```
#include <stdio.h>  
  
#define LEN20  
#define test printf("example\n");  
  
int main(void)  
{  
    int arr[LEN] = {10};  
    if(arr[0] == 10)  
        test;  
    else  
        printf("Not 10\n");  
    return 0;  
}
```

Because there is no space between `LEN` and `20`, the compiler will produce an error message for the undeclared variable `LEN`. The second compilation error is due to the `;` at the end of `test`. When `test` is replaced, the second `;` does not let `else` to be associated with the `if` statement.

## Macros with Parameters

Besides its simple form, a macro can take parameters and behave like a function. For example, in the following program:

```
#include <stdio.h>

#define MIN(a, b) ((a) < (b) ? (a) : (b))

int main(void)
{
    int i, j, min;

    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);
    min = MIN(i, j);
    printf("Min = %d\n", min);
    return 0;
}
```

the MIN macro takes two parameters. Once it is met, the preprocessor replaces a with i, b with j, and expands this line to:

```
min = ((i) < (j) ? (i) : (j));
```

Because a macro can take parameters of any data type, it can be more generic than a function. For example, we could use the MIN macro to find the minimum of `int`, `char`, `double`, ... and other data types.

---

Don't leave a whitespace after the name of a parameterized macro because the preprocessor will handle the left parenthesis as the beginning of a simple macro definition.

For example, if we write:

```
#define MIN (a, b) ((a) < (b) ? (a) : (b))
```

the compilation will fail because the preprocessor starts the replacement of MIN from the first left parenthesis.

---

Because of operators' precedence, always enclose each macro parameter and the macro definition in parentheses.

For example, suppose that we omit the parentheses in the macro definition:

```
#define MIN(a, b) (a) < (b) ? (a) : (b)
```

What would be the output of

```
printf("%d\n", 3*MIN(4, 10));
```

When the macro is expanded, because multiplication is performed first, the compared values would be 12 and 10, not 4 and 10. As a result, `printf()` outputs 10.

See what happens if we omit the parentheses around the parameters:

```
#define MUL(a, b) (a*b) /* Instead of ((a)*(b)). */
```

the statement:

```
int j = MUL(9+1, 8+2); is expanded to: int j = 9+1*8+2 = 19;
```

and `j` becomes 19, not 100, because the multiplication is performed first. Try both U.16.2 and U.16.3. Now, not later.

Using a parameterized macro, instead of a function that does the same job, may improve performance, because a function call imposes some run-time overhead due to the storage of context information (e.g., the memory address of the function) and copy of the arguments. On the other hand, a macro invocation does not impose any delay because the preprocessor expands it before the execution of the program. For example, `getchar()`, `putchar()`, and functions declared in `ctype.h` are typically implemented as macros for better performance.

However, using a parameterized macro instead of a function has several disadvantages. For example, because macros rely on text substitution, you can get into serious trouble when the macro contains complex code, like control-flow constructs. Needless to say, it is much harder to read, maintain, and debug a macro that contains multiple statements. Also, since macros are removed during preprocessing, it is not possible to declare a pointer to a macro.

When a function is called, the compiler checks the types of the arguments. If an argument has a wrong type and the compiler cannot convert it to the proper type, it produces an error message. On the other hand, the preprocessor does not check the types of the macro arguments, so undesirable values may be passed.

Typically, a parameterized macro is used in place of a function, when its code is not complicated and it does not extend to many lines.

---

Don't use macro arguments with side effects.

For example, the statement: `int x = MIN(i++, j);` is expanded to: `int x = ((i++) < (j) ? (i++) : (j));`

As a result, if `i` is less than `j`, `i` will be incremented twice and a wrong value will be assigned to `x`.

---

Pay attention when defining and using a macro to avoid unexpected results. If it is not simple, use a function.

## # and ## Preprocessor Operators

The # operator in front of an argument instructs the preprocessor to create a string literal having the name of that argument. For example:

```
#include <stdio.h>

#define f(s) printf("%s = %s\n", #s, s);

int main(void)
{
    char *str = "text";
    f(str);
    return 0;
}
```

When the preprocessor expands the f macro, it replaces #s with the name of the argument, that is, str. Therefore, the program displays: str = text.

A more difficult example is the expansion of sum in the following program:

```
#include <stdio.h>

#define sum(a, b) printf(#a "+" #b " = %d\n", a+b)

int main(void)
{
    int i, j;

    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);
    sum(i, j);
    sum(i*j, i*j);
    return 0;
}
```

The preprocessor replaces the first sum with:

```
printf("i "+"j" " = %d\n", i+j);
```

Since the compiler merges consecutive string literals, printf() is equivalent to:

```
printf("i+j = %d\n", i+j);
```

Similarly, the preprocessor replaces the second sum with:

```
printf("i*j "+"i*j" " = %d\n", i*j+i*j);
```

and after the string concatenation, it becomes:

```
printf("i*j+i*j = %d\n", i*j+i*j);
```

For example, if the user enters 2 and 5, the program displays:

```
i+j = 7  
i*j+i*j = 20
```

The ## operator is used to merge identifiers together. It is not allowed to appear in the beginning or at the end of the sequence. If the identifier is a macro parameter, the pre-processor first replaces it with the value of the argument and then pasting occurs. For example:

```
#include <stdio.h>  
  
#define f(a) s##u##m##a  
  
int sum1(int a, int b);  
  
int main(void)  
{  
    int i, j;  
  
    printf("Enter numbers: ");  
    scanf("%d%d", &i, &j);  
    printf("%d\n", f(1)(i, j));  
    return 0;  
}  
  
int sum1(int a, int b)  
{  
    return a+b;  
}
```

When the preprocessor expands the f macro, because the character a has the same name as the argument, the preprocessor replaces a with the value of the argument, that is, 1, and then merges that value with the s, u, and m characters. Therefore, f(1)(i, j) is expanded to sum1(i, j) and the program calls sum1() to display the sum of the two input numbers.

---

## Preprocessor Directives and Conditional Compilation

This section discusses the preprocessor directives that allow the conditional compilation of a part of the program. The conditional compilation can be very useful in many situations, for example, to debug the program, to monitor the program execution, or to maintain multiple versions of the same program.

## The `#if`, `#else`, `#elif`, and `#endif` Directives

The `#if` and `#endif` directives are used to define which parts of a program will be compiled, depending on the value of an expression. The general syntax is:

```
#if expression
    ... /* block of statements */
#endif
```

The expression is a constant integer expression which is not allowed to include `sizeof`, casts or `enum` constants. If its value is true, the preprocessor keeps the block of statements to be compiled. If not, the block will be removed and the compiler won't see it. Notice that if the expression is an undefined identifier, the `#if` directive evaluates to false. For example:

```
#if NUM
```

If NUM is not defined the outcome is false, while the value of `#if !NUM` is true. If NUM is defined with value 0, `#if NUM` evaluates to false and `#if !NUM` to true.

The `#else` directive is used in conjunction with the `#if` directive to define a block of statements that will be compiled if the value of the expression is false.

```
#if expression
    ... /* block of statements A */
#else
    ... /* block of statements B */
#endif
```

For example, in the following program, the `#else` part will be compiled because NUM is greater than 0:

```
#include <stdio.h>

#define NUM 10

int main(void)
{
    #if NUM < 0
        printf("Seg_1\n");
    #else
        printf("Seg_2\n");
    #endif
    return 0;
}
```

Therefore, the program displays Seg\_2. Notice that if the definition of NUM was missing, the program would display Seg\_2, too.

Since nested comments are not allowed, it is not possible to put in comments a part of the program that contains comments. Instead, we can use the `#if` directive. For example:

```
#if 0
    /* Comment. */
```

```
... /* Code. */
/* Another comment. */
#endif
```

The **#elif** directive can be used together with **#if**, **#ifdef**, and **#ifndef** directives to define multiple compilation paths. For example:

```
#if expression_A
    ... /* block of statements A */
#elif expression_B
    ... /* block of statements B */
.

.

#else
    ... /* block of statements N */
#endif
```

The second block of statements will be compiled only if the value of expression\_A is false and the value of expression\_B is true. Once an **#if** or **#elif** expression is found true, next **#elif** and **#else** directives are discarded. The last **#else** is optional and its block of statements will be compiled only if the previous expressions are all false. In practice, we've often used this syntax to support multiple versions of the same program. For example:

```
#include <stdio.h>

#define VER_2 1

int main(void)
{
    int cnt = 0;

    #if VER_1
        cnt = 1;
        printf("Version_1\n");
    #elif VER_2
        cnt = 2;
        printf("Version_2\n");
    #else
        cnt = 3;
        printf("Version_3\n");
    #endif

    printf("Cnt = %d\n", cnt);
    return 0;
}
```

Since VER\_1 is not defined and the value of VER\_2 is true, the preprocessor keeps those lines and discards the rest, cnt becomes 2 and the program displays:

```
Version_2
Cnt = 2
```

If we change the value of VER\_2 to 0, the **#elif** expression becomes false and the pre-processor keeps the lines of the **#else** directive. Therefore, the program displays:

```
Version_3  
Cnt = 3
```

If the **#else** directive was missing, the program would display: Cnt = 0

### The **#ifdef**, **#ifndef**, and **#undef** Directives

The **#ifdef** directive is used to check if an identifier is defined as a macro. The general syntax is:

```
#ifdef name  
    ... /* block of statements */  
#endif
```

The difference with the **#if** directive is that the **#ifdef** directive only checks if the identifier is defined as a macro; it does not evaluate its value. For example:

```
#include <stdio.h>  
  
#define VER_1 0  
  
int main(void)  
{  
    #ifdef VER_1  
        printf("Version_1\n");  
    #else  
        printf("Version_2\n");  
    #endif  
    return 0;  
}
```

Since VER\_1 is defined, the program displays Version\_1.

The **#ifndef** directive is used to check whether an identifier is not defined as a macro. A typical use of the **#ifndef** directive is to prevent multiple inclusions of the same file. It is not only to avoid unnecessary recompilations and save some time, but also for another reason. Suppose that a program consists of several source files and some of them include the same file (e.g., test.h). If test.h contains type definitions (e.g., structures), the compilation will fail because redefinition of the same types is not allowed. To avoid that, we add the following lines at its beginning:

```
#ifndef SOME_TAG /* Use a name not defined somewhere else. */  
#define SOME_TAG  
/* Contents of test.h */  
#endif
```

Once the first file that includes test.h is compiled, the preprocessor will include the contents of the test.h because the SOME\_TAG macro is not defined yet. In next inclusions

of test.h, the preprocessor skips its content because SOME\_TAG has been defined and **#ifndef** evaluates to false. The common practice is to select a name for the SOME\_TAG similar to the name of the header file.

Furthermore, the use of **#ifndef** prevents endless inclusions. For example, this might happen if one file (e.g., one.h) includes a second one (e.g., sec.h) and that second one includes the first.

The **#undef** directive cancels the definition of a macro. If the macro has not been defined, **#undef** has no effect. For example:

```
#include <stdio.h>

#define NUM 100

int main(void)
{
    int arr[NUM];
#undef NUM

    printf("Array contains %d elements\n", NUM);
    return 0;
}
```

Since the directive **#undef** NUM cancels the definition of NUM, the compiler will produce an error message in printf().

## The **defined** Operator

An alternative way to check if an identifier is defined as a macro is to use the **defined** operator. The **defined** operator is usually used together with the **#if** or **#elif** directives. For example:

```
#if defined(VER_1) /* Equivalent to #ifdef VER_1 */
...
#endif
```

The parentheses are not necessary; it is a matter of preference. Similarly, the expression **#if !defined(VER\_1)** is equivalent to **#ifndef VER\_1**. The advantage over the **#ifdef** and **#ifndef** is that these can be used to check if only one macro is defined or not, while the **defined** operator and the **#if** directive can be used to check multiple macros. See how we are using the **&&** operator to check multiple conditions:

```
#if defined(VER_1) && !defined(VER_2) && defined(VER_3)
...
#endif
```

or use the **||** operator to check if a macro is defined:

```
#if defined(VER_1) || defined(VER_2)
...
#endif
```

Here is another common application of the conditional compilation. Suppose that we are writing a program that depends on some external factor, for example, the operating system or the compiler. Instead of writing different versions, we could use conditional compilation. For example:

```
#if defined(UNIX)
...
#elif defined(LINUX)
...
#elif defined(WINDOWS)
...
#endif
```

At the beginning of the program, the proper macro is defined and the preprocessor keeps the corresponding code.

---

## Miscellaneous Directives

Before continuing with the exercises, let's discuss briefly the `#error`, `#line`, and `#pragma` directives.

The syntax of the `#error` directive is `#error message` and instructs the preprocessor to display an error message that contains the `message`. The compilation fails. Typically, it is used in the `#else` part of an `#if-#elif-#else` series to indicate an undesirable situation. For example:

```
#if defined(VER_1)
...
#elif defined(VER_2)
...
#else
    #error Unspecified version
#endif
```

The `#line` directive has two forms. The first one is `#line n`, where `n` is an integer. It changes the numbering of the next lines in the program to `n, n+1, n+2, ...` The second form is `#line n "file"`, where the compiler assumes that the lines that follow this directive are coming from `file`, with line numbering starting from `n`.

The `#pragma` directive is used to request special behavior from the compiler. The set of commands that can appear in the `#pragma` directive is different for each compiler. If the command is not supported, the compiler ignores the directive. For example, if a compiler supports the `warning` command, the directive `#pragma warning(disable : 4244)` may be used to disable the appearance of the warning message with code 4244.

## Exercises

C.16.1 Write a macro that calculates the absolute value of a number. Write a program that reads an integer and uses the macro to display its absolute value.

```
#include <stdio.h>

#define abs(a) ((a) >= 0 ? (a) : -(a))
int main(void)
{
    int i;

    printf("Enter number: ");
    scanf("%d", &i);
    printf("abs = %d\n", abs(i));
    return 0;
}
```

C.16.2 Write a macro that checks whether a number is odd or even. Write a program that reads an integer and uses the macro to display whether it is odd or even.

```
#include <stdio.h>

#define odd_even(a) (((a) & 1) == 0)

int main(void)
{
    int i;

    printf("Enter number: ");
    scanf("%d", &i);
    if(odd_even(i))
        printf("Even\n");
    else
        printf("Odd\n");
    return 0;
}
```

C.16.3 Suppose that the content of test.h is

```
#include <stdio.h>

#define TEST

#define TEST
#define f() printf("One ")
#undef TEST
#endif
```

What is the output of the following program?

```
#include "test.h"
int main(void)
{
    f();
#define TEST
    f();
#endif
    f();
    return 0;
}
```

**Answer:** The preprocessor replaces the first occurrence of `f()` with `printf()` and then cancels the definition of `TEST`. Since `TEST` is not defined, the preprocessor does not expand the second `f()` and continues with the third one. Therefore, the program displays: One One.

C.16.4 Write a program that reads double numbers continuously and counts either the positives or the negatives depending on the definition of a macro. For example, if the `CNT_POS` macro is defined, the program should count the positives, otherwise the negatives. If the user enters 0, the insertion of numbers should terminate.

```
#include <stdio.h>

#define CNT_POS

int main(void)
{
    int cnt = 0;
    double num = 1;

    while(num != 0)
    {
        printf("Enter number: ");
        scanf("%lf", &num);
        #ifdef CNT_POS
            if(num > 0)
                cnt++;
        #else
            if(num < 0)
                cnt++;
        #endif
    }
    printf("Cnt = %d\n", cnt);
    return 0;
}
```

C.16.5 Write a program that displays One if both VER\_1 and VER\_2 macros are not defined. Otherwise, if either VER\_3 or VER\_4 macro is defined, the program should display Two. If nothing from the above happens, it should display Three.

```
#include <stdio.h>

int main(void)
{
#if !defined(VER_1) && !defined(VER_2)
    printf("One\n");
#elif defined(VER_3) || defined(VER_4)
    printf("Two\n");
#else
    printf("Three\n");
#endif
    return 0;
}
```

C.16.6 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>

#define f(text) printf(text); if(strlen(text) < 5) return 0;

int main(void)
{
    f("One");
    f("Two");
    f("Three");
    return 0;
}
```

**Answer:** The preprocessor expands the first f() to the following lines:

```
printf("One");
if(strlen("One") < 5)
    return 0;
```

When the program runs, the `return` statement terminates the program because the length of One is 3, less than 5. Therefore, the program displays One.

C.16.7 Write a macro that calculates the greatest of three numbers. Write a program that reads three double numbers and uses the macro to display the greatest.

```
#include <stdio.h>

#define max(a, b, c) ((a) >= (b) && (a) >= (c) ? (a) : \
(b) > (a) && (b) > (c) ? (b) : (c))
```

```

int main(void)
{
    double i, j, k;

    printf("Enter numbers: ");
    scanf("%lf%lf%lf", &i, &j, &k);
    printf("Max = %f\n", max(i, j, k));
    return 0;
}

```

**C.16.8** What is the output of the following program?

```

#include <stdio.h>

#define hide(t, r, a, p, i, n) p##a##r##t(i, n)

double show(int a, int b);

int main(void)
{
    printf("%d\n", (int)hide(w, o, h, s, 1, 2));
    return 0;
}

double show(int a, int b)
{
    return (a+b)/2.0;
}

```

**Answer:** The preprocessor replaces one by one the arguments of the `hide()` macro. Therefore, the preprocessor substitutes `t` with `w`, `r` with `o`, and so on. Since the `p, a, r, t` are replaced by the `s, h, o, w` and the `##` operator merges the operands together, the macro is expanded to `show(i, n)`. Therefore, the program calls `show()` with arguments `1` and `2`, which returns their average, that is, `1.5`. Since the return value casts to `int`, the program displays `1`.

**C.16.9** Write a macro that calculates the greater of two numbers. Write a program that reads four integers and uses the macro to display the greatest.

```

#include <stdio.h>

#define max(a, b) ((a) > (b) ? (a) : (b))

int main(void)
{
    int i, j, k, l;

    printf("Enter numbers: ");
    scanf("%d%d%d%d", &i, &j, &k, &l);
    printf("Max = %d\n", max(max(max(i, j), k), l));
    return 0;
}

```

**Comments:** The preprocessor expands the macros from the inner to the outer. Notice, that there are other alternatives, for example, we could write `max(max(i, j), max(k, l))`.

**C.16.10** Write a macro that can read an integer, a float, or a character and display it. Write a program that uses this macro to read an integer, a float, and a character in that order and display their values.

```
#include <stdio.h>

#define f(type, text, a) printf(text); scanf(type, &a); printf(type"\n", a);

int main(void)
{
    char ch;
    int i;
    float fl;

    f("%d", "Enter integer: ", i);
    f("%f", "Enter float: ", fl);
    f(" %c", "Enter character: ", ch); /* Insert a space, so that
scanf() ignores white space. */
    return 0;
}
```

**C.16.11** What is the output of the following program?

```
#include <stdio.h>

#define no_main(type, name, text, num) type name(void) {printf(text);
return num; }

no_main(int, main, "No main()", 0)
```

**Answer:** Leave the best for the end. “Silver alert,” `main()` is lost. The program works, though. The preprocessor replaces `type` with `int`, `name` with `main`, `text` with “`No main()`” and `num` with 0. Therefore, the preprocessor expands `no_main()` to:

```
int main(void) {printf("No main()"); return 0;}
```

and the program displays: `No main()`

Notice that if we were using `void` instead of `int` the compilation would fail, because a `void` function cannot return a value.

---

## Unsolved Exercises

**U.16.1** Write a macro that checks whether a number is between two other numbers. Write a program that reads three numbers (e.g., `x, a, b`) and uses the macro to check if `x` is within  $[a, b]$ .

**U.16.2** The following macro calculates the absolute value. Find the error(s) in its definition, by giving examples to prove it. Fix the definition.

```
#define abs (a) a > 0 ? a : -a
```

**U.16.3** The following macro calculates the minimum of two numbers. It is not correct, though. Pass an expression with the & operator as an argument, to prove the error. Fix the definition.

```
#define MIN(a, b) (a < b ? a : b)
```

**U.16.4** Write a macro that takes two arguments and calculates their average. Write a program that reads four integers and uses the macro to display their average.

**U.16.5** Write a macro that takes as an argument one character and if it is a lowercase letter it expands to the respective uppercase letter. Write a program that reads a string of less than 100 characters and uses the macro to display its lowercase letters in uppercase.

**U.16.6** Write a macro that takes as an argument one character, and if it is digit, the macro should evaluate to true, otherwise false. Write a program that reads a string of less than 100 characters and uses the macro to display how many digits it contains.

**U.16.7** What would be the output of the following program when the macro ONE:

- a. is defined?
- b. it is not defined?

Explain why.

```
#include <stdio.h>

#define ONE 1
#define myprintf(a) printf("x" #a " = %d\n", x##a)

int main(void)
{
#define ONE 1
#define TWO ONE+ONE
#else
#define ONE 2
#define TWO (ONE+ONE)
#endif

    int x1 = 3*(ONE+ONE), x2 = 3*TWO;

    myprintf(1);
    myprintf(2);
    return 0;
}
```

**U.16.8** Complete the following macro to display the result of the math operation specified by sign.

```
#define f(a, sign, b) ...
```

Assume that the a and b arguments are integers. Write a program that reads two integers and a math sign and uses the macro to display the result of the math operation.

**U.16.9** Modify the macro of the previous exercise in order to display the result of math operations applied on both integers and floats.

```
#define f(a, sign, b, fmt) ...
```

The fmt argument should be either %d or %f. Write a program that reads an integer and a double number (not 0 in case of division) and a math sign and uses the macro to display the result of the math operation.

**U.16.10** Write a program that reads a string of less than 100 characters, and based on the definition of a macro, it should display either the number of its lowercase letters or the number of its uppercase letters or the number of its digits. For example, if the macro UL is defined, the program should display the number of the uppercase letters.

**U.16.11** Complete the macro SET\_BIT to set the bit in position pos, the macro CLEAR\_BIT to clear the bit in position pos, and the macro CHECK\_BIT to display the value of the bit in position pos. Write a program that reads an integer and a bit position and tests the operation of these macros.

```
#define SET_BIT(a, pos) ...
#define CLEAR_BIT(a, pos) ...
#define CHECK_BIT(a, pos) ...
```

## Building Large Programs

A non-small program is usually divided into several files, so tasks like development, testing, maintenance, and upgrade become easier to perform. Because in our working experience, we were deeply involved with the development of network protocols, we decided to implement a simple protocol to show you an example of a larger application. If you ever decide to work professionally on that subject, this example might help you.

First, what is a network protocol? Simply put, it is a set of rules that enables system communication. A protocol may be implemented in hardware, software, or both. A protocol example is the famous *Internet Protocol (IP)*. It is specified by the Internet Engineering Task Force (IETF) organization in the *RFC 791* standard. The heart of a protocol standard is the information exchanged between systems, the format of the exchanging messages, their significance, and the message handling procedures. Typically, the guideline for the implementation of a protocol is a finite state machine that models the protocol operation in terms of states and events.

Let's describe the protocol we are going to implement. It is an oversimplified version of the *ITU-T Q.931* signaling protocol. The *Q.931* protocol is designed for establishment, maintenance, and release of calls in an Integrated Services Digital Network (ISDN) network. When the calling user dials the number of the called user, a message named **SETUP** is created. The **SETUP** message is routed through the ISDN network to reach the called user. If the called user accepts the call (e.g., picks up the phone), a message named **CONNECT** is transmitted back to the calling user. The called user responds with **CONNECT\_ACK** and the call establishment procedures ends. A circuit is reserved along the path between the two parties to serve the data transfer. Now, the two parties may communicate. If either party closes the connection (e.g., hangs up), a message named **RELEASE** is transmitted to the other party and the circuit is released.

Our protocol will simulate the message exchange from the side of the called user. Each message is identified by a unique code and contains specific information. For example, the code of the **SETUP** message is 5 and it contains several information elements, such as the number of the calling user. For simplicity, we assume that each message contains only its code.

Before starting the implementation, let's have a brief discussion about how to write a larger program. First, we break down the application requirements into separate modules. This partitioning facilitates project management and good understanding of the program structure. The functionality of each module should be well defined, not overlapping with other modules so that different programmers may work in parallel to implement them. Next, we define the interactions between modules, their dependencies, and how these modules communicate with each other. A communication example is the *callback* mechanism we saw in C.8. The operations of each module are mapped to functions. For each module, we choose the functions that may be shared by other modules and put their declarations in a header file, used as the module's interface file. For example, a module might be implemented as a library exporting to the external environment an interface file, such as `stdio.h`, that provides information for the implemented functions.

The usual choice for the name of the file that contains `main()` is the name of the application, for example, `prtcl.c`. The common practice is to put function definitions and declarations of external variables into source files, while function declarations, type definitions, constants, and macros are put into header files. Typically, a source file comes together with a header file. The header file facilitates one who reads the program to get quickly an idea what part of the program the source file implements. Alternatively, we could declare everything into one header file and all source files include that file. Although we'd have a single file to edit, it is not our preference. One reason is that whenever its content changes, all source files must be recompiled. That might be time consuming for a large application. Common information used in more than one source file is put into one header file and that file is included when needed. The advantage to have a single header file is that if we ever need to make a change, we don't have to search in several files to make that change. Looking at just one place is quick and safe.

Let's start the implementation. We split the functionality of the program into several modules. Each source file represents a module:

`prtcl.c`. It contains functions to communicate with the external environment and forward the received messages to the inner modules. An example of the external environment might be the network card or the subscriber who uses the numpad to communicate with the protocol.

`send.c`. It contains functions to create messages and send them to the upper and lower layers. The concept of layers is described in the *Open Systems Interconnection (OSI)* model, a reference model that describes how applications can communicate over a network. A layer may be implemented in software or hardware. For example, the lower layer might be the driver of the network card or the card itself, while the upper layer might be the application which presents the data to the user (e.g., web browser).

`recv.c`. It contains functions to parse the received data.

`fsm.c`. It contains functions to implement the protocol finite state machine.

As discussed, we put common information in one header file. We name it `general.h`. To avoid multiple compilation, we use the `#ifndef` directive.

```
/* general.h */

#ifndef general_h
#define general_h

#include <stdio.h>

typedef unsigned char BYTE;

#define LOOPBACK_MODE

/* Message codes. */
#define SETUP          0x5
#define CONNECT        0x7
#define CONNECT_ACK    0xF
```

```

#define RELEASE          0x4D

typedef struct
{
    BYTE code;
    /* Typically, the format of the structure resembles that of
C.13.11. */
} Msg;

int fsm(Msg *p);
int forward_msg(Msg *p);
int rcv_msg(BYTE data[]);
int handle_rcvd_msg(BYTE data[]);

#endif

```

Let's describe `prtcl.c`. It creates the `SETUP` message and forwards that message to `fsm.c`. In real conditions, this message is created when the called user initiates communication with the calling user (e.g., dials the number) and traverses the ISDN network to reach the destination. The function `rcv_msg()` is called when the application receives a message from the lower layer. For example, the driver gets a message from the network card and uploads that message to the application. In a real implementation, we put that function inside a thread; think of a thread as a separate program executed in parallel with the main program. It is called asynchronously whenever a message is received. Don't worry about thread programming, we just give some hints in case someone is involved with something similar in the future. The message is contained in the `data` array. The variable `cur_state` holds the current state of the protocol. We declare it with global scope, to show you how to use it as `extern` in other files.

```

#include "general.h"
#include "fsm.h"

int cur_state;

int main(void)
{
    Msg msg;

    cur_state = IDLE;

    msg.code = SETUP;
    fsm(&msg);
    return 0;
}

int rcv_msg(BYTE data[])
{
    handle_rcvd_msg(data);
    return 0;
}

```

The next files are the `fsm.c` and `fsm.h`. The protocol state machine is implemented here. For example, if the `SETUP` message is received in `IDLE` state, the state changes to `CALL_INIT` and the message is forwarded for transmission. For simplicity, we'll use the `switch` statement. However, our typical choice is to use an array of pointers to functions, as in the example of Chapter 8, in order to improve performance.

```
#include "general.h"
#include "fsm.h"

extern int cur_state;

int fsm(Msg *p)
{
    switch(cur_state)
    {
        case IDLE:
            if(p->code == SETUP)
            {
                cur_state = CALL_INIT;
                forward_msg(p);
            }
            else
            {
                printf("Unexpected message_%d in state_%d\n",
p->code, cur_state);
                return -1;
            }
            break;

        case CALL_INIT:
            if(p->code == CONNECT)
            {
                printf("Call activated\n");
                cur_state = ACTIVE;

                p->code = CONNECT_ACK;
                forward_msg(p);
            }
            else
            {
                printf("Unexpected message_%d in state_%d\n",
p->code, cur_state);
                return -1;
            }
            break;

        case ACTIVE:
            if(p->code == RELEASE)
            {
                printf("Call released\n");
                cur_state = IDLE;
            }
            else
```

```

        {
            printf("Unexpected message_%d in state_%d\n",
p->code, cur_state);
            return -1;
        }
    break;

default:
    printf("Not supported state\n");
break;
}
return 0;
}

/* fsm.h */

#ifndef fsm_h
#define fsm_h

/* States */
#define IDLE      1
#define CALL_INIT 2
#define ACTIVE    3
#endif

```

The next file is the `send.c`. All messages are created according to the protocol specification and passed to the lower layer. As discussed, this layer might be the driver of the network card that forwards the message to the card for transmission in the network. For simplicity, we assume that the message does not contain any other information element besides its code.

```

#include "general.h"

int forward_msg(Msg *p)
{
    BYTE data[1]; /* Since the message contains only its code, its
size is set to 1. The information stored into Msg is used to create the
message. C.11.38 shows how a message might be created. */
    data[0] = p->code;
    switch(data[0])
    {
        case SETUP:
        break;

        case CONNECT_ACK:
        break;

default:
        printf("Unexpected message_%d for transmission\n",
data[0]);
        return -1;
    }
#ifndef LOOPBACK_MODE

```

```

/* Normally, here is the place where a function of the lower layer
is called in order to pass the data array and the data length. */
printf("Only loopback tests are supported\n");
#else
    switch(data[0])
    {
        case SETUP:
            data[0] = CONNECT; /* We implement the scenario that
the other party accepted the call and the message CONNECT is received. */
            break;

        case CONNECT_ACK:
            data[0] = RELEASE; /* We implement the scenario that
the other party closed the connection and the message RELEASE is
received. */
            break;
    }
    rcv_msg(data);
#endif
    return 0;
}

```

If you are ever assigned to implement a protocol, pay attention to that point. Suppose that you simulate side A. You create and send messages to side B. The question is, how could you verify that your code properly handles the responses coming from side B? If you can find an application in the market that simulates side B, it is fine, get it and use it to test your code. But what if you cannot find one? For example, I was involved with the development of a protocol where side B was a switching node in a cellular network and I could not find an application to simulate that side. The company I used to work for was a small one and could not afford the cost to buy a real node. So, what did I do?

You just saw it. I defined the macro LOOPBACK\_MODE and added code to simulate side B into the same application. As a result, I managed to test the most part of the code and fix bugs locally, before going to the customer's premises to deliver the product. Of course, the product was thoroughly tested when integrated into the customer's network platform, but most tests had already been performed. Thanks to the loopback testing, the integration phase completed earlier than expected saving time and money for the company. In a similar situation, the second author decided to develop the application that simulates side B in another system and use a network service (e.g., sockets) to communicate with that system and test his A side implementation. Of course, the simulation code should be written with special care, in order to avoid inserting bugs that may mislead you into incorrectly thinking that your implementation is not correct. The simulation code does not have to be fully functional right from the beginning. At first, write code to perform the basic tests, add functionality gradually for more complicated tests.

The last file to edit is the `rcv.c`. The message is analyzed and its fields are validated. Then the message is forwarded to `fsm.c`.

```

#include "general.h"

int handle_rcvd_msg(BYTE data[])
{
    Msg msg;

```

```

msg.code = data[0];
switch(msg.code)
{
    case CONNECT:
        fsm(&msg);
        break;

    case RELEASE:
        fsm(&msg);
        break;

    default:
        printf("Received message_%d not supported\n",
data[0]);
        return -1;
}
return 0;
}

```

So far, all our programs consist of a single source file. How can we create a program that consists of several files? It depends on the compiler you use. For example, integrated development environments, apart from the compiler also provide a graphical environment, which usually presents a menu command such as *File->New->Project* to create the program and *Project->Add\_To\_Project->Files* to add files, or something similar. Source files can be compiled separately or all together with the *Build* command. The program outputs:

```

Call activated
Call released

```

To compile all source files with the `gcc` compiler, we write the following in the command line:

```
gcc -o call_sim prtcl.c fsm.c send.c recv.c
```

Four object files are created and the linker combines them into the executable file. The option `-o` is used to specify the name of the executable. The default is `a.out`. To run the program, we write `call_sim`, or `./call_sim` if not executed, and the program displays the same output.

A flexible way to build large programs is to use a special file named *makefile*. The concept of the makefile is originated in *Unix* system, and that file contains the necessary information to build the program. In particular, it describes the dependencies among the program files and sets the rules to build the program. An advantage of using a makefile is that if something changes, only those files affected by that change are recompiled, not all. Here is a makefile example for our program:

```

call_sim: prtcl.o fsm.o send.o recv.o
        gcc -o call_sim prtcl.o fsm.o send.o recv.o

prtcl.o: prtcl.c general.h fsm.h
        gcc -c prtcl.c

```

```
fsm.o: fsm.c general.h fsm.h  
        gcc -c fsm.c  
  
send.o: send.c general.h  
        gcc -c send.c  
  
rcv.o: rcv.c general.h  
        gcc -c rcv.c
```

There are five pairs of lines. The first line in each group defines the target file that will be created, followed by the files it depends on. The second line defines the command to be executed if one of the dependent files changes. Let's explain the first two pairs, the rest are similar.

The first line states that `call_sim` is the target file. That is the executable file. It depends on the next four object files. If any of these four files has changed since the time the program was last built, `call_sim` will be rebuilt. The command on the second line specifies how the rebuilding will be done. In the second pair, the first line states that `prtcl.o` is the target file and will be rebuilt if any of the following three files changes. The second line specifies how to rebuilt `prtcl.o`. The `-c` option instructs the compiler only to compile `prtcl.c`, not attempt to link it.

After creating the makefile, we can use the *make* tool in a command shell to build the program and get the executable. Once *make* runs, it reads the makefile and examines which files have changed since the program was last built. Then, it recompiles these files and all files that depend on them to produce the new executable. Notice that makefiles and the *make* program can be very complicated, we just presented a simple example.

For your information, the implemented system was one of the company's products and supported the parallel activation of thousands of calls. In fact, that was what the customer was asking for. The customer was an equipment manufacturer searching for testing systems to evaluate the equipment performance, before selling it to the network operators. And how could our system manage all these calls in parallel? Easy. I've implemented a *class* and each call was managed by an *object* of that class. *Class* and *object*, still new things to learn? Yes, learning never ends. Time to talk about object-oriented programming. Go on to the next chapter to get an idea.

## Introduction to C++

C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories during the 1980s. With a few exceptions, C++ is a superset of ANSI C. A C++ program may use C libraries. C++ is a rather large language, much more complex than C. The most important difference from C is the support of object-oriented features. This chapter introduces you in C++ and highlights some of the most important object-oriented key concepts, such as encapsulation, polymorphism, and inheritance. Although their description won't be complete, you'll get a basic understanding of what C++ programming is like.

### Classes and Objects

Before we talk about classes and objects, let's write our first C++ program. The usual extension of a source file is .cpp

```
#include <iostream>
using namespace std;
// This is my first C++ program
int main()
{
    cout << "The Clash: Should I stay or should I go\n";
    return 0;
}
```

Play the song loud and decide. We'd say `stay`, you won't regret it. However, if you feel tired and prefer `go` to the next chapter, we understand, we won't hold bad feelings.

Let's explain the program sentences. The `iostream` file contains information related to data input and output. Next, we inform the compiler that the namespace `std` is used. C++ allows the grouping of classes and objects in a common namespace. For example, the `cin` and `cout` objects are defined in the `std` namespace. C++ supports `//` to insert a single line comment. The C style `/* */` is still available.

Like C, the existence of `main()` is mandatory. A difference with C is that use of the word `void` is not needed when declaring or defining a C++ function with no arguments. By default, the `cout` object is associated with the screen. In this syntax, the `<<` operator does not perform the shift operation; it is overloaded to write output to `cout`. The compiler checks the types of the operands to decide whether to make a shift or write data. We'll talk about overloading in a while. Notice that we can still use C input/output functions (e.g., `scanf()`/`printf()`), but it is more convenient to use C++ object-oriented methods.

Let's move to the main course. The concept of encapsulation refers to the grouping of data and functions and the ability to define their accessibility. In C++, the mechanism that supports encapsulation is called *class*. In fact, the most significant difference between C

and C++ is the support of classes. To underscore the importance of classes, the original name of C++ was C with classes. The **class** data type is an extension of the **struct** type. A class may contain data and functions that operate on that data. An advantage of the class type is that it allows us to model real-world objects. For example, if we are developing an e-mail application, we might define a class named Mail. The Mail class could define data like an array to hold text messages and functions like send and receive. A class like this makes the program easier to read and write, since both data and operations look more natural. The general form to define a class is:

```
class class_name
{
    Accessibility: Declarations;
    ...
    Accessibility: Declarations;
};
```

As with a structure, once a class is defined, we can declare variables of that type. Such a variable is said to be an *object* or *instance* of that class. Every object is built from a class. For example, to declare two objects of a class named A we write:

```
class A a1, a2; // The word class is not needed.
```

A class tag serves as a type name, so the word **class** is not necessary. As with structures, to access the members of an object, we can use the . and -> operators. For example, the following program reads the data of a student and uses an object of the class Student to store them:

```
#include <iostream>
#include <string>
using namespace std;

class Student
{
public:
    string name; /* name is an object of the class string. */
    int code;
    float grd;

    void show();
};

void Student::show()
{
    cout << "N:" << name << " C:" << code << " G:" << grd << endl;
/* Each time << is used, printing continues. endl adds a new line. */
}

int main()
{
    Student s; /* The variable s is an object of the class Student. */

    cout << "Name: ";
```

```

getline(cin, s.name); /* We use the getline() function to read
characters and store them in name. */
cout << "Code: ";
cin >> s.code;

cout << "Grade: ";
cin >> s.grd;

s.show();
return 0;
}

```

By default, the `cin` object is associated with the keyboard. Similar to `cout` and `<<`, the `>>` does not perform a shift operation, it is overloaded to get data input from `cin`. The `string` class is one of the many available C++ classes and it is used for string manipulation. The important thing to notice is the existence of a function declaration inside the class. Functions declared inside a class are said to be member functions. A member function may be defined inside or outside its class. The first approach is typically used for very short functions. If it is not short, it is usually defined outside the class so that its code is clearly shown, not mixed together with other declarations. Although the code of the function `show()` is short, we implemented it outside the class to show you how to do it. We use the scope resolution operator `::` preceded by the name of the class that it belongs to. `show()` outputs the values of the members of the calling object.

Let's talk about accessibility. The accessibility defines the access to the class data. It is a feature missing from C. In C, the data of a structure or union are all accessible. We cannot make some of them inaccessible. In C++, the members of a class can be `public`, `private`, or `protected`. There is no restriction in the access of the public members. The private members are accessible by the member functions of the same class. The protected members are like the private members for the same class, but they are accessible from functions in classes derived from this class. By default, the class data are private. The main advantage to restrict access in class data is to secure the code and avoid the case of unwanted modifications. Restricted access is a major feature of object-oriented programming. Let's find the errors in the following program:

```

#include <iostream>
using namespace std;

class Test
{
private: // Not needed.
    int a;
    void f1();
protected:
    int b;
    void f2();
public:
    int c;
    void f3();
};

```

```

void Test::f1()
{
    cout << a << ' ' << b << ' ' << c << ' ' << endl;
}

void Test::f2()
{
    a = 100;
}

void Test::f3()
{
    f1();
}

int main()
{
    Test t;

    t.a = 10;
    t.b = 20;
    t.c = 30;

    t.f1();
    t.f2();
    t.f3();
    return 0;
}

```

Since the `a` and `f1()` members are declared as **private**, the `t` object is not allowed to access them. Therefore, the statements `t.a = 10;` and `t.f1();` are illegal. The same applies for the **protected** members. Therefore, the statements `t.b = 20;` and `t.f2();` are illegal. However, the member functions may access them. Therefore, the `f1()`, `f2()`, and `f3()` functions are correct.

## Constructors and Destructors

A class may contain one or more special functions called constructors. A constructor is called automatically whenever an object of that class is created. The constructor has the same name as the class, it has no return value, it can accept arguments, and it can be overloaded. The main reason to use a constructor is to initialize the object's members when it is created.

A class may also contain another special function called destructor. A destructor is invoked automatically when an object of that class is destroyed. The destructor has the same name as the class preceded by the character `~`, it has no arguments, no return type and cannot be overloaded. Destructors are less common than constructors, and they are typically used to release memory dynamically allocated, when the object ceases to exist. For example, let's see the following program:

```

#include <iostream>
using namespace std;

class Test
{
public:
    int a, b;

    Test();
    Test(int i, int j);
    ~Test();
    void check();
};

Test::Test()
{
    cout << "no-param constructor called\n";
}

Test::Test(int i, int j)
{
    cout << "param constructor called\n";
    a = i;
    b = j;
}

Test::~Test()
{
    cout << "destructor called\n";
}

void Test::check()
{
    Test t;
}

int main()
{
    Test t1, t2(5, 10);

    cout << t1.a << ' ' << t1.b << endl;
    cout << t2.a << ' ' << t2.b << endl;

    t1.check();
    return 0;
}

```

Two constructors and one destructor are defined. Whenever an object is created, the compiler checks the arguments and selects the appropriate constructor. A constructor with no arguments is called *default* constructor. Notice that if no constructor is defined, the compiler creates itself the *default* constructor of the class, which does nothing. This happened in the second program, when the `s` object of the class `Student` was created. However, if a nondefault constructor is defined, the compiler does not create the default constructor. If you need a default constructor, you have to define it. Therefore, when the `t1`

object is created, the first constructor is automatically invoked and the program displays no-param constructor called. When the t2 object is created, the second constructor is called and the program displays param constructor called. Next, the program displays the arbitrary values of the variables a and b of the t1 object and the values 5 and 10 for the t2 variables.

When check() is called, the t object is created and the first constructor is called. Next, the function ends and the destructor of the t object is automatically invoked. Therefore, the program displays destructor called. When the program ends, the destructors for the t1 and t2 objects are also called and the program displays destructor called two more times.

---

## Overloading of Functions and Operators

Simply put, overloading refers to the ability to reuse the existing name of a function or an operator to operate on different data types. Overloading is part of the polymorphism in the sense that it allows programmers to reuse existing function names and operators and give them additional meanings.

When a function is overloaded, each version must have the same name but a different number of parameters or types. We have already seen an example of function overloading. It is the presence of two constructors in the previous program. Polymorphism is resolved at compile time (*static binding*). It means that when an overloaded function is called, the compiler checks the function's arguments and determines which version to call. The advantage of function overloading is that functions performing the same operation may be given the same name. There is no need to search for new names. For example, in the following program, the function abs() is overloaded three times and the compiler selects the right version:

```
#include <iostream>
using namespace std;

int abs(int a);
double abs(double a);
void abs(int a, int b);

int main()
{
    int i = -100;
    double j = 1.2345;

    cout << abs(i) << ' ';
    cout << abs(j) << ' ';
    abs(i, 30);
    return 0;
}

int abs(int a)
{
    if(a < 0)
        return -a;
    else
```

```

        return a;
    }

double abs(double a)
{
    if(a < 0)
        return -a;
    else
        return a;
}

void abs(int a, int b)
{
    int sum;

    sum = a+b;
    if(sum < 0)
        cout << -sum << endl;
    else
        cout << sum << endl;
}

```

The program displays: 100 1.2345 70

Notice that only the return type does not enable function overloading. The functions must have different *signatures*, that is, they must differ in the number of parameters or in their types, or both. For example, if we add this `abs()` the compiler will raise an error message:

```
int abs(int a, int b);
```

Like function overloading, an operator may be overloaded within user-defined types and perform different operations. Operator overloading is usually applied in objects of user defined classes, to make them operate in a more natural way. The result is an easier-to-read program. To overload an operator and give it a special meaning inside a class, we must define a member function like this:

```
return_type class_name::operator<operator>(parameters)
```

The operator can be any of the C++ operators with a few exceptions. To show you some examples, we'll use binary (e.g., +, -, \*, >, ...) and unary (e.g., ++, !, ...) operators. For example, the following program defines the class `Rect`, which contains a function to overload the `+` operator and another one for the `>` operator.

```
#include <iostream>
using namespace std;

class Rect
{
private:
    float len;
    float hei;
public:
    Rect(float l, float h);
```

```

Rect operator+(Rect r);
bool operator>(Rect r);
float area();
void show();

};

Rect::Rect(float l, float h)
{
    len = l;
    hei = h;
}

void Rect::show()
{
    cout << "L:" << len << " H:" << hei << endl;
}

float Rect::area()
{
    return len * hei;
}

Rect Rect::operator+(Rect r)
{
    Rect tmp(0,0);

    tmp.len = len + r.len;
    tmp.hei = hei + r.hei;
    return tmp;
}

bool Rect::operator>(Rect r)
{
    if(area() > r.area())
        return true;
    else
        return false;
}

```

Now, what is the output of the following program?

```

int main()
{
    Rect r1(10, 20), r2(30, 40), r3(0, 0);

    r3 = r1+r2;
    r3.show();

    if(r1 > r2)
        cout << "One\n";
    else
        cout << "Two\n";
    return 0;
}

```

When the compiler meets the statement `r3 = r1+r2;`, and since `r1` is an object of the class `Rect`, it looks inside the class to find a function named `operator+`. If that function was missing, the compiler would display an error message for invalid operation. Next, the compiler translates the statement to `r3 = r1.operator+(r2);` and the `operator+` function of the `r1` object with parameter the `r2` object is called. `r1+r2` is just an abbreviation of `r1.operator+(r2)`, much easier to read and use it. The function returns a new object. Its dimensions are the sum of the respective dimensions of the two objects.

Similarly, the compiler translates the expression `r1 > r2` to `r1.operator>(r2)`. C++ declares a new type called `bool` and a bool variable may be either `true` or `false`. Default values are `1` and `0`, respectively. Therefore, the program displays:

```
L:40  H:60  
Two
```

Notice that when an operator is overloaded, it gets a different meaning only for the objects of the class that overloads it. For example, the statement `c = a+b` in the following code works as usual, it adds the operands:

```
int main()  
{  
    int a = 10, b = 20, c;  
    Rect r1(10, 20), r2(30, 40), r3(0, 0);  
    ...  
    c = a+b;  
    return 0;  
}
```

Another example of operator overloading is the `<<` operator when used with the `cout` object. In particular, `cout` is an object of the `ostream` class and the `operator<<` function is called to write data. Similarly, `cin` is an object of the `istream` class and the `operator>>` is called to get data.

Try this exercise. Add a function to overload the `==` operator. If the dimensions of the two `Rect` objects are the same, the function should return `10`, `-10` otherwise. Then, add another function to overload the `!` operator. The function should swap the dimensions of the calling object. Modify `main()` to test the operation of the two functions.

```
class Rect  
{  
public:  
    int operator==(Rect r);  
    void operator!();  
    ...  
};  
  
int Rect::operator==(Rect r)  
{  
    if(len == r.len && hei == r.hei)  
        return 10;  
    else  
        return -10;  
}
```

```

void Rect::operator!()
{
    int tmp;

    tmp = len;
    len = hei;
    hei = tmp;
}

int main()
{
    int a;
    Rect r1(10, 20), r2(10, 0);

    a = (r1 == r2);
    if(a == 10) /* Alternatively, without using a, if((r1 == r2) ==
10) */
        cout << "Equal\n";
    else
        cout << "Unequal\n";

    !r1;
    r1.show();
    return 0;
}

```

The expression `(r1 == r2)` is translated to `r1.operator==(r2)`. Next, the return value is checked. Similarly, the statement `!r1;` is translated to `r1.operator!()`; and the dimensions of the `r1` object are swapped.

Try this one. Find the errors in the following program:

```

#include <iostream>
using namespace std;

class A
{
public:
    int var;
    A(int i);
    void operator+(int i);
};

A::A(int i)
{
    var = i;
}

void A::operator+(int i)
{
    var += i;
}

```

```

class B
{
public:
    int var;
    B(int i);
};

B::B(int i)
{
    var = i;
}

int main()
{
    int i;
    A a1(10), a2(20, 30);
    B b1(a1.var);

    a2 = a1+10;
    a1+10;
    b1+10;
    10+a1;

    i = a1.var + a2.var;
    return 0;
}

```

- Since class A does not define a constructor with two arguments, the declaration of the a2 object is wrong. Since the a1 object has been created, the declaration b1(a1.var) is fine. It has the same effect as b1(10).
- The statement a2 = a1+10; is wrong because the overloading function of the + operator does not return an object. The statement a1+10; is acceptable and adds 10 to the variable var of the a1 object.
- Since the + operator is not overloaded in class B, the statement b1+10; is wrong.
- The statement 10+a1; is translated to 10.operator+(a1); which is wrong. a1 must be at left.

## Inheritance

Inheritance is one of the most important features of an object-oriented programming language. It is a concept that does not really exist in C. C++ allows us to derive a new class from an existing one, called *base* or *parent* class, instead of writing it from scratch. The derived class inherits the properties of the base class and it can customize them, if necessary. It can also add new data members and member functions. To derive a new class we write

```
class derived_class_name:accessiblity_type base_class_name
```

The accessibility\_type specifies the accessibility to the members of the base class. To give you an idea how inheritance works, we'll use the type **public**. It is one of several possibilities, too detailed to describe all options. When the **public** type is used, the functions and the objects of the derived class can access the public members of the base class and cannot access the private members. The functions of the derived class can access the protected members of the base class; the objects of the derived class cannot do the same though.

Inheritance usually provides increasing specialization from a general class (e.g., Product) to a more specific derived class (e.g., Book). For example:

```
#include <iostream>
#include <string>
using namespace std;

class Product
{
public:
    string code;
    float prc;

    void init(string c, float p);
    void show();
};

class Book : public Product
{
public:
    string title;
    string auth;
    int pages;

    void display();
};

void Product::init(string c, float p)
{
    code = c;
    prc = p;
}

void Product::show()
{
    cout << "\nP:" << prc << " C:" << code << endl;
}

void Book::display()
{
    cout << "T:" << title << " A:" << auth << " P:" << pages << endl;
}
```

```

int main()
{
    Book b;

    cout << "Price: ";
    cin >> b.prc;

    cout << "Code: ";
    cin >> b.code;

    cout << "Title: ";
    cin >> b.title;

    cout << "Auth: ";
    cin >> b.auth;

    cout << "Pages: ";
    cin >> b.pages;

    b.show();
    b.display();
    return 0;
}

```

The main advantage of inheritance is that it allows us to reuse existing code. For example, the class `Book` inherits the data members and functions of the class `Product`. It means that the members of `Product` become members of `Book`, without the need to rewrite any code. The program gets input, stores it into `Product` and `Book` members, and calls the proper functions to display it. As you guess, we may use `Product` to derive additional classes. Besides flexibility, inheritance simplifies program maintenance and upgrade. For example, if we need later to add a new member, which is relevant to all classes, we just add it in the base class and the derived classes will inherit it. If the base class has a bug, we fix it only there; there is no need to modify the code of the derived classes. Given the previous classes, what will be the output of the following program?

```

int main()
{
    Product p;
    Book b;

    p.init("123-456-789", 40.38);
    b.show();
    return 0;
}

```

The object `p` is not related to the object `b`. We called on purpose the `show()` function of `b`, instead of `p`, to mislead you. Since the members of `b` have not been initialized, the program displays arbitrary values.

Inheritance allows a programmer to create multiple hierarchy levels. To give you an example, we'll use `Book` as base class:

```
class Details : public Book
{
public:
    string field;
    string publisher;
    int year;
};
```

A Details object inherits the data members and functions of both Book and Product classes. Try to find the errors in the following program:

```
#include <iostream>
using namespace std;

class A
{
private:
    int a;
    void show();
protected:
    int x;
public:
    void check_B();
};

class B : public A
{
public:
    int k;
    void display();
};

void A::show()
{
    cout << a << endl;
}

void A::check_B()
{
    cout << k << endl;
}

void B::display()
{
    cout << a << ' ' << k << endl;
}

int main()
{
    B b;

    cin >> b.k;
    cin >> b.a;
    cin >> b.x;
```

```
b.show();
b.display();
return 0;
}
```

Since the members `a` and `show()` are declared as `private`, the objects and the functions of class `B` cannot access them. Since `x` is declared as `protected`, access from an object of a derived class is not allowed. As a result, the following statements are illegal:

```
cout << a << ' ' << k << endl; // in display
cin >> b.a;
cin >> b.x;
b.show();
```

Since `k` is declared in the derived class `B`, not in the base class `A`, the following statement is wrong:

```
cout << k << endl; // in check_B()
```

Besides single inheritance, C++ supports multiple inheritance. In multiple inheritance, a new class may be derived from two or more base classes. For example:

```
#include <iostream>
using namespace std;

class A
{
public:
    int a;
    void show_A();
};

void A::show_A()
{
    cout << a << endl;
}

class B
{
public:
    int b;
    void show_B();
};

void B::show_B()
{
    cout << b << endl;
}

class C : public A, public B
{
public:
    void show_C();
};
```

```
void C::show_C()
{
    show_A();
    show_B();
}

int main()
{
    C c;

    c.a = 100;
    c.b = 200;

    c.show_C();
    return 0;
}
```

Class C is derived from classes A and B with public access. Notice that different access types may be used. A C object behaves as an object of either class. It inherits the data members and functions of both A and B classes.

---

## Polymorphism

As said, polymorphism is closely related to overloading. With ordinary overloading, function prototypes must differ so that the compiler can choose the appropriate function. In that case, polymorphism, that is, the selection of the right function, is resolved at compile time (*static binding*). In this section, we'll talk about virtual functions, that is, functions declared in a base class and implemented differently in each derived class. The prototypes of the virtual functions must be identical and polymorphism is resolved at runtime (*dynamic binding*). Let's see the following program:

```
#include <iostream>
#include <string>
using namespace std;

class A
{
public:
    void show();
};

class B : public A
{
public:
    void show();
};
```

```

class C : public B
{
public:
    void show();
};

void A::show()
{
    cout << "A show()\n";
}

void B::show()
{
    cout << "B show()\n";
}

void C::show()
{
    cout << "C show()\n";
}

int main()
{
    A *p;
    B b;
    C c;

    p = &b;
    p->show();

    p = &c;
    p->show();
    return 0;
}

```

Although p points to objects of derived classes, it is the `show()` of the base class that is called. Therefore, the program displays the unexpected result:

```
A show()
A show()
```

In order to call the `show()` of the respective object, the function should be declared as `virtual` in the base class. To declare a function as virtual, we use the `virtual` keyword. For example:

```

class A
{
public:
    virtual void show();
};

```

Now, the program displays:

- B show()
- C show()

When a virtual function is called through a pointer (or reference) to a base class, the runtime system examines the object that the pointer points to and calls the function of that object. In this case, the polymorphism is a runtime effect.

## Templates and Standard Template Library

The templates enable us to write general purpose functions and classes, which can be later instantiated with different data types. We could think of templates as another method of compile time polymorphism.

### Function Templates

Suppose that we want to write a function that takes as parameter a number and returns its absolute value. Since we do not know the type of the number, we should overload the function and create different versions. For example:

```
int abs(int a);
double abs(double a);
```

When using a function template, we can write a generic code and reuse it for different data types. It is not necessary to rewrite the code. Here is an example how to define a function template:

```
template <class type1, class type2, ...> return_type function_name
(parameter_list)
```

The template parameters `type1`, `type2`, ..., are used to pass data types to the function. Despite the word `class`, the arguments don't have to be classes. We'll show you some examples with the same data type passed to the function. The following program uses the function template `abs()` to find the absolute values of numbers with different data types.

```
#include <iostream>
using namespace std;

template <class T> T abs(T a);

int main()
{
    int a = -6;
    double b = 7.89;

    cout << abs(a) << endl;
    cout << abs(b) << endl;
    return 0;
}
```

```
template <class T> T abs(T a)
{
    if(a < 0)
        return -a;
    else
        return a;
}
```

T is the usual choice for the parameter name; any name would do though. Look how things work. When a template function is called, the compiler checks the types of its arguments and replaces the template parameters with the actual data types. For example, in the first call of `abs()`, since the type of a is `int`, the type `int` is transferred to the template parameter T and the compiler translates the function to:

```
int abs(int a);
```

In the next call, the type `double` is transferred and the function is translated to:

```
double abs(double a);
```

Here is another example. The following program defines a template function that takes two parameters of the same type and displays the lesser.

```
#include <iostream>
using namespace std;

template <class T> void test(T a, T b);

int main()
{
    int i = 10, j = 20;
    double k = 5.64, m = 1.23;

    test(i, j);
    test(k, m);
    return 0;
}

template <class T> void test(T a, T b)
{
    if(a < b)
        cout << a << endl;
    else
        cout << b << endl;
}
```

In the first call of `test()`, the type of i and j, that is, `int`, is passed to T. The function is translated to:

```
void test(int a, int b);
```

In the second call, the type of k and m, that is, `double`, is passed. Had we written `test(i, k);` the compiler would have produced an error message because the types of i and k should be the same, not different.

Notice that we can use the template parameter inside the function just like an ordinary data type. For example, the following code swaps the values of the parameters:

```
template <class T> void test(T a, T b)
{
    T tmp; /* We use the template parameter T just like a type. */
    tmp = a;
    a = b;
    b = tmp;
    cout << a << ' ' << b << endl;
}
```

And as last example, the following is a template function with parameters of different types:

```
#include <iostream>
using namespace std;

template <class T1, class T2> void test(T1 a, T2 b);

int main()
{
    int i = 10, j = 20;
    double k = 5.64, m = 1.23;

    test(i, k);
    test(m, j);
    return 0;
}

template <class T1, class T2> void test(T1 a, T2 b)
{
    if(a < b)
        cout << a << endl;
    else
        cout << b << endl;
}
```

In the first call of `test()`, the types of `i` and `k`, that is, `int` and `double`, are transferred to `T1` and `T2`. Therefore, the compiler translates the function to:

```
void test(int a, double b)
{
    ...
}
```

In the second call, the types of `m` and `j`, that is, `double` and `int`, are transferred. Therefore, the function is translated to

```
void test(double a, int b)
{
    ...
}
```

Function templates are usually applied in algorithms. For example, we have implemented the `bubble_sort()` function in C.12.10 to sort an array of integers. If we later need to sort an array of `double` numbers, we should rewrite the function and change `int` to `double`. If we use a template parameter (e.g., `bubble_sort(T arr[])`), we've got a generic sorting function that works for any data type.

## Class Templates

Like the function templates, we may use class templates to define generic classes for different data types. Here is an example how to define a class template:

```
template <class type1, class type2, ...>
class class_name
{
    ...
};
```

To declare an object of that class, we write

```
class_name<type1, type2, ...> object_name;
```

For example, the following program declares the `Test` class template. The function `set()` takes as parameter a number and stores it into the private member `a`. `get()` returns its value.

```
#include <iostream>
using namespace std;

template <class T> class Test
{
    T a;
public:
    void set(T val);
    T get();
};

template <class T> void Test<T>::set(T val) /* Yes, you are right, the
syntax is quite complicated. */
{
    a = val;
}

template <class T> T Test<T>::get()
{
    return a;
}

int main()
{
    Test<int> t1;
    Test<double> t2;
```

```
t1.set(10);
t2.set(5.82);
cout << t1.get() << ' ' << t2.get() << endl;
return 0;
}
```

To create a `Test` object and pass to it the `int` type, we write `Test<int>`. `Test` is translated to:

```
class Test
{
    int a;
public:
    void set(int val);
    int get();
};
```

Similarly, when the `t2` object is created, the type `double` replaces `T` and `Test` is translated to:

```
class Test
{
    double a;
public:
    void set(double val);
    double get();
};
```

The program displays: 10 5.82

Class templates are very helpful when we want to create a generic data structure. For example, if we have created a list of integers and we later need a list that stores another type of data (e.g., `double`), we have to create a new list, to copy the code and change `int` to `double` in many places. Using a class template is much more flexible; the same code works for any desired type.

## Standard Template Library

The Standard Template Library provides a rich set of ready-to-use classes and algorithms that may be used in a wide range of applications. For example, the class `string` provides a rich set of functions to handle strings in an object-oriented way. String operations, like adding a string to another, modifying a string, and dynamic resizing, are very easy to perform. For example, the following program reads two strings, merges them and stores the result into a third string, appends the word `end`, and displays its length and the position of the first occurrence of `a`. Next, the program doubles its size and sets the additional characters equal to the first character of the first string. Last, the program erases the first three characters and displays the string. Take a look at how easy is to perform all these operations:

```
#include <iostream>
#include <string>
using namespace std;

int main()
```

```

{
    int i, len;
    string s1, s2, s3;

    cout << "Enter strings: ";
    cin >> s1 >> s2;

    s3 = s1+s2;
    s3.append("end");
    len = s3.length();
    cout << "L:" << len << " S:" << s3 << endl;

    i = s3.find('a');
    if(i == -1)
        cout << "Not found\n";
    else
        cout << "Found in position:" << i+1 << endl;

    s3.resize(2*len, s1[0]);
    s3.erase(0, 3);
    cout << s3 << endl;
    return 0;
}

```

Another example of an often used standard class is the class template `vector`. It is used as an array. The main advantage over an ordinary array is that its size may be modified at runtime. For example:

```

#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int i;
    vector<int> arr(3); /* Create a vector of integers with three elements. In another example, we could use a class instead of int and create a vector of objects of that class. */
    for(i = 0; i < arr.size(); i++)
    {
        arr[i] = 10+i;
        cout << arr[i] << ' ';
    }
    arr.push_back(20); /* Add elements at the vector end and allocate memory if necessary. */
    arr.push_back(30);
    for(i = 0; i < arr.size(); i++)
        cout << arr[i] << ' ';
    return 0;
}

```

First, the program displays the values of three elements: 10 11 12. Next, it displays the values of five elements: 10 11 12 20 30.

## More Features

So far, we've made a brief review of how C++ supports the basic concepts of object-oriented programming. Certainly, there are plenty of extra C++ features not covered in this introductory chapter. When you feel confident working with C, go to the next level, buy a C++ book, and discover them. In this section we'll introduce you to some of them, the `new` and `delete` operators, the reference variables, and the exceptions.

### The `new` and `delete` Operators

Although we may still use standard C functions (e.g., `malloc()`) to allocate memory dynamically, a better practice is to use the `new` operator. The `delete` operator is used to free memory. Notice that `new` and `delete` are operators, not functions. On success, `new` returns a pointer to the allocated memory. We may handle that pointer as we did in the past. For example, the following program allocates memory to read an integer and displays that integer:

```
#include <iostream>
using namespace std;

int main()
{
    int *ptr;

    try
    {
        ptr = new int; // Allocate memory.
    }
    catch(bad_alloc)
    {
        cout << "Unavailable memory\n";
        exit(1);
    }
    cout << "Input: ";
    cin >> *ptr;
    cout << "Output: " << *ptr << endl;

    delete ptr; // Free memory.
    return 0;
}
```

If `new` fails, it throws the `bad_alloc` standard exception. Otherwise, the input number will be stored into the memory that `ptr` points to. We'll talk about exceptions and the `try` and `catch` keywords in a short.

To allocate memory for an array, we use the operator `new[]` and write the number of the elements inside the brackets. To free that memory, we use the operator `delete[]`. For example, to allocate and free memory for 100 `float` numbers, we write:

```
float *ptr = new float[100];
delete[] ptr;
```

Let's see an example of how to allocate memory for some objects. The following program reads a number of students and allocates memory to read and store their data. Also, the

user enters a number and the program displays how many students got a grade better than that number.

```
#include <iostream>
using namespace std;

class Student
{
public:
    int code;
    float grd;
};

int main()
{
    Student *ptr;
    int i, num, cnt;
    float grd;

    cout << "Enter number of students: ";
    cin >> num;

    try
    {
        ptr = new Student[num];
    }
    catch(bad_alloc)
    {
        cout << "Unavailable memory\n";
        exit(1);
    }
    cout << "Enter base: ";
    cin >> grd;
    cnt = 0;
    for(i = 0; i < num; i++)
    {
        cout << "Code: ";
        cin >> ptr[i].code;

        cout << "Grade: ";
        cin >> ptr[i].grd;

        if(ptr[i].grd > grd)
            cnt++;
    }
    cout << cnt << " students got more than " << grd << endl;

    delete[] ptr;
    return 0;
}
```

As you see, we use the pointer `ptr` as array.

## Reference Variables

A reference variable is an alias for an existing variable. A reference variable must be initialized when it is declared. To declare a reference variable, we add the & before its name and initialize it with an existing variable. For example:

```
int i = 10;
int &j = i;
j = 200;
```

The reference variable j is an alias of i. Both refer to the same memory location. Therefore, i becomes 200.

References are typically used in function calls. As you already know, when a function needs to modify the value of an argument, we pass a pointer to its address. An alternative way is to pass a reference. For example:

```
#include <iostream>
using namespace std;

void test(int &a, int b);

int main()
{
    int i = 10, j = 50;

    test(i, j);
    cout << i << " " << j << endl;
    return 0;
}

void test(int &a, int b)
{
    a = 100;
    b = 200;
}
```

When a reference to a variable is passed to a function, it is the address of the variable that is actually passed. When test() is called, i passes by reference. Notice that the & operator is added only in the function declaration, not in function call. Inside test(), a is understood to be an alias of i. Therefore, i changes from 10 to 100 and the program displays: 100 50.

Because a pointer, like a reference, may be used to change the value of an argument, this type of call is sometimes referred to as *call by reference*. However, to be precise, C supports only *call by value* (as discussed, except for arrays). C++ brings references and supports both types of call.

Someone could probably say that references are simpler and easier to use compared to pointers. However, our preference is to use pointers in order to make clear the arguments whose values might change. For example, when using references, a function call such as f(i, j, k) does not provide any indication to the reader whether f may change or not their values. On the other hand, when using pointers, the call f(&i, j, k) would tell the reader that f may change the value of i, not the others. Also, the use of the indirection operator \* inside the function makes it clearer and faster to see when the value of an argument changes.

Try this exercise. Write a function that takes as parameters two pointers to integer variables and swaps the contents of those two variables. Overload the function to take two references. Write a program that reads two integers, stores them in two variables, and uses the functions to swap their contents.

```
#include <iostream>
using namespace std;

void swap(int *ptr1, int *ptr2);
void swap(int &ptr1, int &ptr2);

int main()
{
    int i, j;

    cout << "Enter numbers: ";
    cin >> i >> j;

    swap(&i, &j);
    cout << i << ' ' << j << endl;

    swap(i, j);
    cout << i << ' ' << j << endl;
    return 0;
}

void swap(int *ptr1, int *ptr2)
{
    int m;

    m = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = m;
}

void swap(int &ptr1, int &ptr2)
{
    int m;

    m = ptr1;
    ptr1 = ptr2;
    ptr2 = m;
}
```

The program first uses pointers to swap their contents, then references.

Since we referred to functions, let's see another C++ feature. C++ allows us to set default values to function parameters. If the function is called with missing arguments, the default values are used. For example:

```
#include <iostream>
using namespace std;

void add(int i, int j, int k = 30, int m = 40);
```

```

int main()
{
    add(10, 20);
    return 0;
}

void add(int i, int j, int k, int m)
{
    cout << i+j+k+m << endl;
}

```

When `add()` is called, since the last two arguments are missing, `k` and `m` will be initialized with the default values, that is, 30 and 40. Therefore, the program displays 100. Notice that if a parameter is initialized, all the following parameters should be also initialized. For example, the following declaration is wrong because `k` is not initialized:

```
void add(int i, int j = 20, int k, int m = 40);
```

## Exceptions

During the execution of a program, unhandled error conditions may cause its abnormal termination. To avoid that case, a robust program should always check for possible errors. For example, when a function is called, the program should check if it is executed successfully. If it fails, the program should determine how to continue.

C++ allows the separation of detecting and handling special situations. For example, when an error occurs, a function can use the `throw` statement to create an exception and report that error. The code that might throw an exception is placed in a `try` block. The code that handles exceptions is placed in `catch` blocks at the end of the `try` block. Each `catch` block specifies the type of the exception it handles. The type can be a standard type or a user-defined type. For example:

```

try
{
    f();
}
catch(type exc_1)
{
    ...
}
catch(type exc_2)
{
    ...
}
catch(...)
{
    ...
}

```

```

void f()
{
    ...
    if(error_occurs)
        throw exception;
    ...
}

```

The main concept of exceptions refers to the ability of a part of a program to inform another part that something *exceptional* occurred. The **throw** statement creates an exception of a particular type. For example, the statement **throw** 20; creates an exception of type **int**. Next, the catch handlers are processed to find a matching type. If there is a catch handler for exceptions of type **int**, i.e., **catch(int a)**, the exception is caught and the program executes the block code. In that example, a becomes 20. Notice that it is not necessary to use parameter names; that is, we could write **catch(int)**. Typically, the **throw** argument is used to provide information about the exception. The catch block **catch(...)** handles every type of exception. If the exception is not caught, it is propagated from the current function to the calling function, to its caller, and so on, all the way back to **main()**, to find a handler. If **main()** cannot handle the exception, the program terminates, probably abnormally. If no exception is thrown, the **catch** statements are skipped and the execution of the program continues with the first statement after the last **catch** block. For example, the following program reads a student's name and grade and checks that value:

```

#include <iostream>
#include <string>
using namespace std;

void check_grd(float grd);

int main()
{
    float grd;
    string name;

    try
    {
        cout << "Enter name: ";
        getline(cin, name);

        cout << "Enter grade: ";
        cin >> grd;

        check_grd(grd);
        cout << "N:" << name << ' ' << "G:" << grd << endl;
    }
    catch(float g)
    {
        cout << "Error: " << g << " is out of range\n";
    }
    return 0;
}

```

```
void check_grd(float grd)
{
    if(grd < 0 || grd > 10)
        throw grd;
}
```

If the user enters a value out of [0, 10], the `throw` statement creates an exception with argument the value of `grd`. The type of the exception is the type of `grd`, that is, `float`. The `catch` statement catches the exception, `g` becomes equal to `grd`, a message is displayed, and the program terminates. Notice that the `catch` statement can catch the exception because it is defined to catch exceptions of type `float`. Had we written `catch(int g)`, the exception wouldn't have been caught, because of different types.

Besides exceptions of basic data types, the type of an exception can be a class. When discussing the `new` operator we saw such an example, the `bad_alloc` exception. Let's add another exception in our example. If the user enters a negative value, an exception of type `string` is thrown. In addition, we write a function that takes as parameter the student's name, and if the name:

- a. Contains a number, it creates an exception of type `int` and argument the value 20.
- b. Contains either the character + or -, it creates an exception of type `int` and argument the value 30.

```
#include <iostream>
#include <string>
using namespace std;

void check_grd(float grd);
void check_name(string n);

int main()
{
    float grd;
    string name;

    try
    {
        cout << "Enter name: ";
        getline(cin, name);
        check_name(name);

        cout << "Enter grade: ";
        cin >> grd;
        check_grd(grd);

        cout << "N:" << name << ' ' << "G:" << grd << endl;
    }
    catch(float g)
    {
        cout << "Error: " << g << " is out of range\n";
    }
}
```

```

catch(string msg)
{
    cout << msg;
}
catch(int code)
{
    if(code == 20)
        cout << "Error: " << name << " contains number\n";
    else if(code == 30)
        cout << "Error: " << name << " contains sign\n";
}
catch(...) /* We added this block, just to see it. */
{
    cout << "Error: Another exception\n";
}
return 0;
}

void check_grd(float grd)
{
    string msg;

    if(grd > 10)
        throw grd;
    else if(grd < 0)
    {
        msg = "Error: negative number is entered\n";
        throw msg;
    }
}

void check_name(string n)
{
    for(int i = 0; i < n.length(); i++)
    {
        if(n[i] >= '0' && n[i] <= '9')
            throw 20;
        else if(n[i] == '+' || n[i] == '-')
            throw 30;
    }
}

```

Well, how did you like this quick tour? Hope we didn't tire you. You've got a C++ feeling, go on to next chapter to visit planet Java.

---

## Introduction to Java

---

This chapter introduces you to Java and discusses some of its most important features. If you skipped the C++ introduction it'd be better to go back and read it, in order to get familiar with the basic object-oriented concepts. The implementation examples we are using are quite similar, in order to become easier for you to see how the same features are supported in both languages and compare the code.

James Gosling and his colleagues, Mike Sheridan and Patrick Naughton, initiated the Java language project in Sun Microsystems labs in 1991, in the framework of Stealth Project, which was then renamed to Green Project. The aim of Green Project was the creation of a revolutionary object-oriented programming language. The language was initially called Greentalk and it was then renamed to Oak, because of an oak tree that stood outside Gosling's office, while in 1994, it was finally renamed to Java, inspired by Java coffee and the love of the development team for the coffee.

Recall that in C/C++, the source code is translated into machine language, which depends on the computer architecture. For example, a C executable program compiled in *Unix* environment cannot run as is in *Windows* environment. The source code has to be recompiled in *Windows* environment, in order to generate the new executable.

On the other hand, Java is a portable and cross-platform programming language, meaning that a Java program may run on any system without having to recompile the program. The only requirement set for the system is to support the *Java Virtual Machine* (JVM). The source code is compiled into an "intermediate" specialized format called *bytecode*, which is the machine language of a virtual architecture and specifically that of JVM. The JVM is responsible to translate real time, that is, during the execution of the Java program, the bytecode into the system's machine language.

---

## Classes and Objects

The extension of Java source files is .java, while, after successful compilation, class files (one per each defined class) with .class extension are produced. Let's see our first Java program:

```
class MainClass {  
    // This is my first Java program  
    public static void main(String[] args) {  
        System.out.println("D. Bowie: Space Oddity");  
    }  
}
```

Listen to *D. Bowie* and let's start our short tour of the Java planet. Suppose that our program resides in the *MainClass.java* file. Notice that the name of the file has to be the

same with the name of the class. If using the javac compiler, we write in the command line:

```
javac MainClass.java
```

The compilation produces the bytecode file MainClass.class, which is executed by writing in the command line:

```
java MainClass
```

Let's explain the program sentences. Since Java is an object-oriented language, it allows us to define our own classes. In our program, we defined the class MainClass. The general form to define a class is as follows:

```
A_M  N_A_M  class  class_name {
    A_M  N_A_M  class_field;
    A_M  N_A_M  class_field;
    ...
    A_M  class_constructor;
    A_M  class_constructor;
    ...
    A_M  N_A_M  class_method {
        ...
    }
    A_M  N_A_M  class_method {
        ...
    }
    ...
}
```

A\_M and N\_A\_M stand for Access Modifier and Non Access Modifier, respectively, and they are optional. We can omit either one of them or both, as we did in our MainClass. We'll talk later about modifiers and constructors, so forget them for now.

As in C/C++, the existence of main() is mandatory. However, unlike C/C++, main() should be implemented as a method of a class. Just for your information, a Java program can have more than one main() method. In Java, every function defined within a class is called a *method* of the class. In contrast to C++, Java methods must be implemented within the class. The member variables defined within a class are called *fields* of the class.

The **public** keyword is one of the available Access Modifiers and declares that this class can be accessed by another class. The **static** keyword is one of the available Non Access Modifiers and declares that main() can be invoked even if no instance of the class in which it is contained has been created, that is, an instance of the MainClass class in our example. We'll discuss more about the **public** and **static** keywords later.

As in C/C++, the **void** keyword before the method's name declares that no value is returned. In contrast to C/C++, you are not allowed to use the **void** keyword inside the parentheses to declare that the method does not accept parameters. In Java, we just leave the parentheses empty. To end with the first line of the program, the args[] parameter is an array of instances of the class String, which is one of the native classes provided by Java and it is used for string manipulation. The role of args[] is similar to the argc and argv[] parameters in C/C++; that is, it serves to pass data to main() from the command line. The class System is another one Java native class, while out is one of its static fields

(of type `PrintStream`), which is associated with the screen by default, and `println()` is one of the methods of the `PrintStream` native class, used to display a message. Therefore, we write altogether `System.out.println()` to display the message on the screen.

Because you might feel a little dizzy, we left the simplest part for the end. For multiple line comments, we use the known notation `/*...*/`, while for one line comments, we can also use `//`. Java also supports documentation comments (*Java doc comments*) by using the `/**...*/` notation. This kind of comments is very useful because they briefly describe each class, its fields, the operation of its methods, and how to invoke them. For your information, the JDK `javadoc` tool can produce documentation comments in HTML format.

Once a class is defined, we can declare variables of that type. As in C++, these variables are called *objects* or *instances* of the class. Each instance has the properties defined for the class. In order to create an instance, we have to use the `new` operator to call a constructor of the class, as follows:

```
A a1 = new A(); /* We declare and create a1, which is an instance of  
class A. */  
A a2 = new A(); /* We declare and create a2, which is an instance of class  
A. */
```

In order to access the members of an object, we use the dot operator `(.)`. Pay attention that pointers in Java are hidden and they are not exposed to the programmer, so you cannot use pointers to your programs, as you do in C/C++. For example, the following program defines the class `Student`, reads the data of a student, and creates an instance of the class `Student` to store them:

```
import java.util.Scanner; /* We import the class Scanner, which provides  
the appropriate methods for reading data from the keyboard. */  
  
class Student {  
    // Fields  
    public String name; /* The field name is an instance of the class  
String. */  
    public int code;  
    public float grd;  
    // Methods  
    public void show() {  
        System.out.println("N:"+name+" C:"+code+" G:"+grd);  
    }  
}  
  
class MainClass {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in); /* We create the scan  
instance of the class Scanner, in order to read data from the keyboard.  
in is a static field of the native class System, which is associated with  
the keyboard by default. */  
        Student s = new Student(); /* We create the s instance of  
the class Student to store the input data. */  
  
        System.out.print("Name: ");  
        s.name = scan.next(); /* We use the next() method of the  
scan instance to read characters from the keyboard and store them in the  
field name. */
```

```
        System.out.print("Code: ");
        s.code = scan.nextInt(); /* We use the nextInt() method to
read an integer and store it in the field code. */
        System.out.print("Grade: ");
        s.grd = scan.nextFloat(); /* We use the nextFloat() method
to read a floating-point number and store it in the field grd. */
        s.show();
    }
}
```

Notice that although we allocated memory for the `s` instance, we did not call any method to release this memory, as we'd do in C/C++. Java provides a native *garbage collector* mechanism, which is used to release the memory allocated by objects that are no longer used.

---

## Packages

A Java *package* is a namespace that may contain classes, interfaces, and other packages, called *subpackages*. Conceptually, we could parallelize a package with a file folder, but a Java package can only have .class files, that is, files containing classes and interfaces or other subpackages. Typically, in a package, we find or put classes whose methods perform specific tasks.

To import to our program classes contained in a package, we use the `import` keyword. In the previous example, we imported the class `Scanner`, which is contained in the `util` package, which is a subpackage of the `java` package; that's why we wrote:

```
import java.util.Scanner;
```

If we had not imported the class `Scanner`, we would have to write explicitly the full path of the class `Scanner`, in order to create the `scan` instance. For example:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

As you see, it is much more convenient to import the class.

We can import all classes contained in a package and/or subpackage by using `*` after the name of the package. For example, the statement:

```
import java.util.*;
```

imports all classes contained in the `util` package and not only the class `Scanner`.

---

## Constructors

In Java, as in C++, the constructor of a class is a special function, which is automatically invoked whenever an instance of that class is created. Constructors are similar to methods,

but they are not methods. Their main difference is that a method can return a value, whereas a constructor cannot. Furthermore, the constructor must have the same name as the class. You are allowed to apply access modifiers to constructors. As in C++, a constructor can be overloaded. For example, in the following code, we declare a constructor with no parameters:

```
public class A {  
    public A() {  
        System.out.println("A constructor for class A");  
    }  
}
```

To create an instance of class A, we use the `new` operator. For example:

```
A a = new A();
```

A constructor can take parameters. For example:

```
class B {  
    B(float x, int y) {  
        System.out.println("A constructor for class B");  
    }  
}
```

To create an instance of class B, we have to pass two arguments. For example:

```
B b = new B(1.23, 50);
```

If no constructor is declared in a class, the compiler creates the *default* constructor of the class, which always has no parameters. This happened in the previous program, where the class Student had no constructor available. However, if a constructor with parameter(s) is declared in the class, the compiler does not create the default constructor. For example, if we write:

```
B b = new B();
```

the compiler will display an error message indicating that the constructor of the class B requires two arguments.

As said, a constructor can be overloaded. For example:

```
class Test {  
    public int a, b;  
  
    Test() {  
        System.out.println("no-param constructor called");  
    }  
    Test(int a, int b) {  
        System.out.println("param constructor called");  
        this.a = a;  
        this.b = b;  
    }  
}
```

```
class MainClass {
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = new Test(5, 10);

        System.out.println(t1.a + " " + t1.b);
        System.out.println(t2.a + " " + t2.b);
    }
}
```

We'll talk about overloading later, but it is easy to observe that there are two overloaded constructors of class `Test`, with exactly the same name, but with different parameters. The compiler checks the arguments and invokes the appropriate constructor. Therefore, when the `t1` instance is created, the first constructor is invoked and the program displays no-param constructor called. When `t2` is created, the second constructor is invoked and the program displays param constructor called.

Uninitialized fields of a class are initialized with default values (e.g., 0 for arithmetic types, ...), while uninitialized local variables of a method are assigned random values. Therefore, the program displays the values 0 and 0 for the `t1` fields and the values 5 and 10 for the `t2` fields.

The `this` keyword is used in methods and constructors of a class and refers to the current instance, whose method or constructor has been invoked. Because the parameters of the second constructor have the same names as the `a` and `b` fields, we are using `this` to refer to them. For example, `this.a` refers to the field `a` of the `t2` instance. Of course, if we were using different names, `this` would be unnecessary. For example:

```
Test(int i, int j) {
    System.out.println("param constructor called");
    a = i;
    b = j;
}
```

Although it is perfectly legal to use different names, the common practice is to use the same names and use `this` to refer to the class fields.

You may wonder when we'll talk about destructors in Java. The answer is never! As said, the garbage collector is the native Java mechanism that automatically undertakes to release the memory that is not used anymore. Therefore, unlike C++, a Java programmer is not able to define destructors to destroy objects on demand, since the Java garbage collector is responsible for doing this work by itself.

---

## Access Modifiers and Nonaccess Modifiers

Java access modifiers are applied to classes, interfaces, and class members, that is, fields, methods, and constructors, in order to define the access to them.

The `public` access modifier can be applied to a class or interface and declares that it can be accessed by any other class or interface. If no access modifier is used, the class or the interface can be accessed only by classes or interfaces that are contained in the same package. This is the reason that the absence of an access modifier is also known as *package-private* or *friendly*.

Similarly, there are four access modifiers that can be applied to the members of a class. The **public** modifier makes the member accessible by any other class, while the absence of an access modifier makes the member accessible only by classes of the same package. The **private** modifier declares that the member can be accessed only within its own class. Finally, the **protected** modifier declares that the member can be accessed only by classes of the same package or by its subclasses, that is, classes that inherit the class in which the member is contained. Table 19.1 depicts the access to the members of a class, according to the applied access modifier.

The members of an interface can be accessed by any class even if the **public** modifier is not applied. We'll talk about interfaces later.

The aim of the nonaccess modifiers is to define extra properties, besides accessibility, to classes, interfaces, fields, methods, and constructors. The nonaccess modifiers are the following: **static**, **final**, **abstract**, **synchronized**, **native**, **strictfp**, **transient**, and **volatile**, which have to follow the access modifier, if present. We'll provide a short description for the first three. The rest are rather specialized and beyond the scope of an introductory chapter.

The **static** modifier can only be applied to fields and methods of a class or to classes that have to be nested into other classes. A **static** field belongs to the class and not to an instance of the class. Therefore, even if we create multiple instances of the same class, the **static** field is created once and all instances share the same field.

---

Since a **static** field belongs to the class and not to an instance of the class, it can be accessed even if no instance has been created. On the other hand, in order to have access to a nonstatic field, an instance must be created. This is the reason that nonstatic fields are also known as instance fields.

For example, in the following program, it is not necessary to create an instance of A to access the **static** field x:

```
class A {  
    public static int x = 5;  
    public int y = 55;  
}  
  
class MainClass {  
    public static void main(String[] args) {  
        System.out.println(A.x);  
        System.out.println(A.y);  
    }  
}
```

**TABLE 19.1**

#### Access Modifiers

Access Modifier	Within the Class	Within the Package	By Subclasses	By Any Class
<b>public</b>	✓	✓	✓	✓
<b>protected</b>	✓	✓	✓	✗
(absence)	✓	✓	✗	✗
<b>private</b>	✓	✗	✗	✗

However, this is not true for the nonstatic field `y`. Therefore, the compiler will produce an error message.

In another example, what is the output of the following program?

```
class A {
    public static int x = 5;
    public int y = 55;
}

class MainClass {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new A();

        System.out.println(A.x + " " + a1.x + " " + a2.x);
        System.out.println(a1.y + " " + a2.y);

        a2.x = 2;
        a2.y = 22;
        System.out.println(A.x + " " + a1.x + " " + a2.x);
        System.out.println(a1.y + " " + a2.y);
    }
}
```

At first, the program displays 5 5 5 and 55 55. Then, the `a2` instance changes the value of the `static` field `x`, which is a common field for both class `A` and all of its instances. Since `y` is a nonstatic field, any change in its value does not affect `y` fields in other instances. As a result, the program outputs: 2 2 2 and 55 22.

The `static` modifier is also used to declare a `static` method, also called class method, because it belongs to the class and not to an instance of the class. As with `static` fields, it is not needed to create an instance of the class to call a `static` method. The `main()` method is an example of a `static` method, since it can be invoked without having created an instance of `MainClass`.

The `final` modifier can be applied to classes, fields, and methods of a class or to local variables of a method. Although we've not discussed yet about inheritance and polymorphism, it is not so hard to understand how it works. A `final` class is considered final; that is, it cannot be extended and have subclasses. In other words, a `final` class cannot be inherited. For example:

```
final class A {...}

class B extends A {...} /* Not allowed. */
```

Overriding of `final` methods is not allowed. For example:

```
public class A {
    public int m1(int x, int y) {
        return x+y;
    }
    public final void m2(String s) {
        System.out.println("Method m2 in class A prints: "+s);
    }
}
```

```
public class B extends A { /* Class B is a subclass of class A. */
    public int m1(int x, int y) {
        return x-y;
    }
    public void m2(String s) {
        System.out.println("Method m2 in class B prints: "+s);
    }
}
```

Class B inherits class A; that is, B is a subclass of A. Since m1() of class A is not declared as **final**, it is allowed to be overridden by the m1() method of subclass B. On the other hand, since m2() is **final**, it is not allowed to be overridden by m2() of subclass B. Therefore, the compiler will produce an error message.

A **final** field or a **final** local variable in a method can be assigned a value only once and this value cannot change later. In the following code, the value of the field C\_STR will be constant and equal to "Constant\_String". Notice that this value will be the same in all instances of class A.

```
public class A {
    public final String C_STR = "Constant_String";
}
```

Since the value of a **final** field cannot change, it is a common practice to be declared as **static** as well, in order to belong to the class and not to the instances of the class. **final** fields are usually named with all letters in uppercase. If a field is declared as **final** but not initialized, it has to be initialized in the constructor of the class; otherwise, the compiler will raise an error message.

The **abstract** modifier can be applied to methods, classes, and interfaces. An **abstract** method has no implementation but only a *signature* (prototype in C/C++ terms). Since it is not implemented, it cannot be declared as **final**. In the following example, two methods are declared as **abstract**:

```
public abstract class A {
    public int m1(int x, int y) {
        return x*y;
    }
    public abstract float m2(float f1, float f2);
    public abstract void m3();
}
```

Notice that m1() is implemented, whereas only the signatures of m2() and m3() are provided.

---

A class with one **abstract** method or more must be declared as **abstract**.

---

Each subclass of an **abstract** class has to override the **abstract** methods of the parent class by implementing them. If not, the subclass must also be declared as **abstract** and let its own subclasses to override them. With this in mind, could class B in the following code be a subclass of the previous class A?

```
public abstract class B extends A {
    public float m2(float f1, float f2) {
```

```
        return f1+f2;
    }
    public abstract void m3();
}
```

Of course, since the `m2()` method of class B is implemented and overrides the `m2()` of class A. Furthermore, since `m3()` is not implemented, `m3()` and B are correctly declared as **abstract**. And what about the following class C? Could it be a subclass of class B?

```
public class C extends B {
    /* m2() of class C overrides m2() of class B. */
    public float m2(float f1, float f2) {
        return 2*(f1+f2);
    }
    public final void m3() {
        System.out.println("Hello from Class C!");
    }
}
```

Since method `m3()` is implemented, the answer is yes. It is also declared as **final**, in order to prevent further overriding.

---

## Inheritance

Inheritance is one of the most important features of both Java and C++ and of any object-oriented programming language. In the C++ section, we talked about the advantages of inheritance. In short, inheritance provides the ability to create a new class, which inherits the properties of an existing class and extra features might be added. Inheritance allows us to create multiple levels of hierarchy. The new class is called *derived* class or *subclass* of the existing one, while the existing class is called *base* or *parent* or *super* class for the derived one. A subclass can use fields or methods of its base class without the need to rewrite any code. The general form to define a subclass is:

```
A_M N_A_M class subclass_name extends base_class_name {
    ...
    // Constructors, fields, and methods of subclass.
}
```

`A_M` and `N_A_M` represent the Access Modifiers and Non-Access Modifiers, respectively. For example, consider the following program:

```
public class Product {
    public String code;
    public double prc;

    public void init(String code, double prc) {
        this.code = code;
        this.prc = prc;
    }
    public void show() {
```

```

        System.out.println("\nP:"+ prc + " C:" + code);
    }

public class Book extends Product {
    public String title;
    public String auth;
    public int pages;

    public void display() {
        System.out.println("T:"+ title + " A:"+ auth + " P:"+ pages);
    }
}

import java.util.Scanner;

public class MainClass {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        Book b = new Book();

        System.out.print("Price: ");
        b.prc = scan.nextFloat();

        System.out.print("Code: ");
        b.code = scan.nextInt();

        System.out.print("Title: ");
        b.title = scan.next();

        System.out.print("Auth: ");
        b.auth = scan.next();

        System.out.print("Pages: ");
        b.pages = scan.nextInt();

        b.show();
        b.display();
    }
}

```

The Book class inherits all members of Product. Therefore, we don't have to copy the fields and the methods of the Product class into Book. The program reads the input data, stores them into the respective fields, and calls the proper methods to display the data. Given the previous classes, what is the output of the following program?

```

public class MainClass {
    public static void main(String[] args) {
        Product p = new Product();
        Book b = new Book();

        p.init("123-456-789", 40.38);
        b.show();
    }
}

```

Instance p is not related to instance b. We invoked on purpose first the `init()` method of p and then the `show()` of b to mislead you. Therefore, the program displays the default values of the b fields, that is:

```
P:0.0 C:null
```

Unlike C++, Java supports only *single* inheritance and not *multiple* inheritance. In other words, each class has only one *direct* super class. The only exception is the native class `Object` in `java.lang` package, which is the only class without any super class. All classes with no explicit super class are derived from class `Object`. Like C++, Java supports multilevel inheritance. This means that a class can be a subclass of another class, which is in its turn a subclass of another class and so on. In this case, a class inherits all fields and methods of all of its super classes. For example:

```
public class Details extends Book {  
    public String field;  
    public String publisher;  
    public int year;  
}
```

An instance of class `Details` inherits all fields and methods of its super classes `Book` and `Product`.

Notice that a class does not inherit the constructors of its super class(es), but it can invoke them. A class inherits all `public` and `protected` fields and methods of its super class(es), regardless of whether they are contained in the same package. If they are contained, it also inherits their *package-private* fields and methods as well, that is, the fields and methods with no access modifier. The only fields and methods that cannot be inherited, and therefore cannot be accessed, are the ones declared as `private`. Now, try to find the errors in the following program:

```
import java.util.Scanner;  
  
class A {  
    private int a;  
    protected int x;  
  
    private void show() {  
        System.out.println(a);  
    }  
    public void check_B() {  
        System.out.println(k);  
    }  
}  
  
class B extends A {  
    public int k;  
  
    public void display() {  
        System.out.println(a + " " + k);  
    }  
}
```

```

class MainClass {
    public static void main(String[] args) {
        B b = new B();
        Scanner scan = new Scanner(System.in);

        System.out.print("Enter number: ");
        b.k = scan.nextInt();

        System.out.print("Enter number: ");
        b.a = scan.nextInt();

        System.out.print("Enter number: ");
        b.x = scan.nextInt();

        b.show();
        b.display();
    }
}

```

Since the field `a` and the method `show()` are declared as **private** in class `A`, the instances and the methods of class `B` are not allowed to access them. As a result, the following statements are illegal:

```

System.out.println(a + " " + k); // In method display() of class B.
b.a = scan.nextInt();
b.show();

```

On the other hand, the access to both **public** method `check_B()` and to **protected** field `x` is allowed. But since the field `k` has been declared in subclass `B` and not in the base class `A`, the following statement is not acceptable:

```
System.out.println(k);
```

As said, Java does not support multiple inheritance in the way that C++ does. We'll see later that multiple inheritance is supported implicitly in Java, through Java interfaces.

## Polymorphism

As in C++, polymorphism in Java is the ability of an object to take many forms and it is coupled with the ability to reuse existing method names and give them additional meanings. Java, like C++, provides two types of polymorphism: *static polymorphism*, in which the proper method is selected during the compilation of the program (*static binding*), and *dynamic polymorphism*, in which the proper method is selected during the execution of the program (*runtime binding*). Indicative examples of static polymorphism are *method overloading* and *constructor overloading*. An indicative example of dynamic polymorphism is *method overriding*.

## Method Overloading

Java, like C++, supports method overloading, that is, the definition of different versions of methods having the same name. Each version must have the same name but a different number of parameters or types. An example of method overloading is the constructor overloading we met when discussing constructors. Once an overloaded method is invoked, the compiler checks the arguments and selects the appropriate version. Therefore, polymorphism is resolved at compile time. The advantage of method overloading is that there is no need to use different names for methods having similar functionality. For example, in the next program, the method `abs()` in class `Absolute` is overloaded three times. Each time `abs()` is called, the compiler checks the arguments and invokes the appropriate method.

```
class Absolute {
    public int abs(int a) {
        if(a < 0)
            return -a;
        else
            return a;
    }
    public double abs(double a) {
        if(a < 0)
            return -a;
        else
            return a;
    }
    public void abs(int a, int b) {
        int sum;

        sum = a+b;
        if(sum < 0)
            System.out.print(-sum);
        else
            System.out.print(sum);
    }
}

class MainClass {
    public static void main(String[] args) {
        int i = -100;
        double j = 1.2345;
        Absolute a = new Absolute();

        System.out.print(a.abs(i) + " ");
        System.out.print(a.abs(j) + " ");
        a.abs(i, 30);
    }
}
```

The program displays: 100 1.2345 70

Notice that having the same number of parameters and types is not allowed, even if the return types of the overloading methods are different. For example, if we add a fourth `abs()` method in class `Absolute`:

```
public int abs(int a, int b) {
    if(a-b < 0)
        return b-a;
    else
        return a-b;
}
```

the compiler will raise an error message, although it has a different return type.

Unlike C++, Java does not support operator overloading.

## Method Overriding

Java, like C++, supports method overriding; that is, different, but related, classes can declare methods with the same signature, but with different implementations. Unlike method overloading, polymorphism is resolved at runtime (*runtime binding*), that is, the appropriated method is determined during program execution.

In C++ runtime polymorphism is achieved by using virtual functions and the `virtual` keyword. This is not required in Java, since all nonstatic methods are virtual functions by default. `final` methods cannot be overridden, as well as `private` methods, which cannot be extended, and therefore cannot be overridden. For example, consider the following program:

```
class A {
    public void show() {
        System.out.println("A show() ");
    }
}

class B extends A {
    public void show() {
        System.out.println("B show() ");
    }
}

class C extends B {
    public void show() {
        System.out.println("C show() ");
    }
}

class MainClass {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        C c = new C();
```

```
a.show();  
b.show();  
c.show();  
}  
}
```

The `show()` method of class A is overridden by the corresponding method of class B, which, in its turn, is overridden by the corresponding method of class C. The runtime system checks the instance and calls the method of the respective class. As a result, the program outputs:

```
A show()  
B show()  
C show()
```

Notice that a derived class may invoke an overridden method of its super class by using the `super` keyword. For example, if we write `super.show()` in class C, the method `show()` of class B is invoked.

---

## Interfaces

A Java *interface* is a reference type in Java. Interfaces are used to define an abstract framework, and then, this framework has to be implemented by classes. An interface resembles a class that contains only constant declarations and method's signatures. We say constants and not variables because all variables declared in an interface are `public`, `static`, and `final` by default, so they are constants. We also refer to signatures and not the method's implementations because all methods declared in an interface are `public` and `abstract` by default, so they are not implemented. An interface is a kind of "contract" about what has to be provided by the class that is going to implement it. It defines "what a class has to do" without saying "how to do it." The programmer can have a high-level overview of the methods that have to be implemented by reading their signatures.

For all versions up to Java 8, an interface could not implement any of its methods, while this is now possible starting with Java 8.

An interface is declared by using the `interface` keyword and a class can implement an interface by using the `implements` keyword. A class that implements an interface has to implement all its methods; otherwise, it has to be declared as `abstract`. For example:

```
public interface SportMatch {  
    String MATCH = "### The match of the day ###";  
  
    void setHomeClub(String home);  
    String getHomeClub();  
    void setVisitorClub(String visitor);  
    String getVisitorClub();  
    void printMatch();  
}
```

The SportMatch interface declares the constant MATCH and the signatures of five methods, without implementing any of them. These methods should be implemented by any class that is going to use this interface. For example:

```
import java.util.Scanner;

public class SportMatchImpl implements SportMatch {
    private String h;
    private String v;

    public void setHomeClub(String home) {
        this.h = home;
    }
    public String getHomeClub() {
        return h;
    }
    public void setVisitorClub(String visitor) {
        this.v = visitor;
    }
    public String getVisitorClub() {
        return v;
    }
    public void printMatch() {
        System.out.println(h + " vs. " + v);
    }
    public static void main (String[] args) {
        Scanner scan = new Scanner(System.in);
        SportMatchImpl smi = new SportMatchImpl();

        System.out.println("Home Club: ");
        smi.setHomeClub(scan.nextLine());

        System.out.println("Visitor Club: ");
        smi.setVisitorClub(scan.nextLine());

        System.out.println(smi.MATCH);
        smi.printMatch();
    }
}
```

Since the SportMatchImpl class implements the SportMatch interface, it has to implement all its methods, regardless whether it is going to use them. For example, it uses three out of the five methods. If we did not implement these two unused methods, we would have to declare both of them as well as the SportMatchImpl class as **abstract**.

Besides the methods of the interface, the SportMatchImpl class can also implement other methods as well. For example, it implements the main() method. Notice the last but one line of main(). Anything strange? What about the smi.MATCH? Is this a field of the smi instance? No, it is not contained in the SportMatchImpl class. So, what is it? It is a constant of the SportMatch interface, and since SportMatchImpl implements this interface, it also inherits all constants declared within it.

We declared on purpose both h and v fields as **private**, in order to describe a common technique of accessing **private** fields through proper methods. Methods that set values

to the fields of a class are called *setters*, while methods that read these values are called *getters*. In SportMatchImpl, the methods setHomeClub() and setVisitorClub() are the setters, while getHomeClub() and getVisitorClub() are the getters. Because the names chosen for the setter parameters (e.g., h) are usually the same with the names of the corresponding fields, we followed this convention and used the **this** keyword. Of course, we could have chosen different names and avoid its use.

Since the methods of an interface are implemented by other classes, it is quite plausible for someone to wonder why do we need interfaces? Isn't the same functionality provided by **abstract** classes? The reason for using interfaces is to support multiple inheritance.

---

In Java, a class can inherit only one class, but it is allowed to implement more than one interface.

For example, consider the following program:

```
public interface A {
    String HELLO = "Hello interface A!!";
    String STR1 = "str1 from interface A";

    public void m1();
}

public interface B {
    String HELLO = "Hello interface B!!";
    String STR2 = "str2 from interface B";

    public void m1();
    public void m2();
}

public class X {
    public static int m3(int i){
        return i*i;
    }
}

public class MainClass extends X implements A,B {
    public void m1() {
        System.out.println("Implementation of m1()");
    }
    public void m2() {
        System.out.println("Implementation of m2()");
    }
    public static void main(String[] args) {
        MainClass mc = new MainClass();

        mc.m1();
        mc.m2();
        System.out.println(m3(4));
        System.out.println(A.HELLO);
    }
}
```

```
    /* System.out.println(HELLO); Compilation error, because the
HELLO constant has been declared in both interfaces and the compiler does
not know to which one to refer to. */
    System.out.println(STR1);
    System.out.println(B.HELLO);
    System.out.println(STR2);
}
}
```

MainClass inherits class X, therefore, it inherits its **static** method m3() and can invoke it directly, without requiring to create an instance of X. Furthermore, MainClass implements both A and B interfaces, which means that it has access to all of their declared constants (i.e., HELLO, STR1 and HELLO, STR2, respectively) and it has to implement all their methods (i.e., m1() and m2(), respectively).

Notice that both interfaces declare a method with the same signature, that is, the m1() method. Since it is the same, MainClass implements m1() once. If these two methods had the same name, the same return type, but different parameters, MainClass would have to implement both of these different methods. However, if they had the same name but different return types, MainClass could not implement both interfaces. Furthermore, notice that both A and B interfaces declare a constant with the same name (i.e., HELLO). In order to have access to these constants, we have to say explicitly to which one we refer to (e.g., A.HELLO); otherwise, a compilation error will occur. As a result, the program displays:

```
Implementation of m1()
Implementation of m2()
16
Hello interface A!!
str1 from interface A
Hello interface B!!
str2 from interface B
```

We think that you've got an idea of Java. Let's return to C and continue with the review exercises.



# Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

## Review Exercises

C.20.1 Write a program that reads its command line arguments and allocates memory to concatenate them into one string and display that string.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char *tot_str;
    int i, tot_chars;

    if(argc == 1) /* Check if the command line contains only the name
of the program. */
    {
        printf("Missing arguments ...\\n");
        exit(EXIT_FAILURE);
    }
    tot_chars = 0; /* It counts the characters of all arguments. */
    for(i = 1; i < argc; i++) /* Remember that argv[1] points to the
first argument, argv[2] to the second one, and so on. argv[0] points to
the name of the program. */
        tot_chars += strlen(argv[i]);

    tot_str = (char*) malloc(tot_chars+1); /* Allocate an extra place
for the null character. */
    if(tot_str == NULL)
    {
        printf("Error: Not available memory\\n");
        exit(EXIT_FAILURE);
    }
    *tot_str = '\\0'; /* Store the null character. */
    for(i = 1; i < argc; i++)
        strcat(tot_str, argv[i]);

    printf("The merged string is: %s\\n", tot_str);
    free(tot_str);
    return 0;
}
```

C.20.2 What is the output of the following program?

```
#include <stdio.h>

void test(int **tmp, int i);

int main(void)
{
    int **tmp, *ptr, i, arr[] = {10, 20, 30};
```

```

ptr = arr;
tmp = &ptr;
for(i = 0; i < 3; i++)
{
    test(tmp, i);
    printf("%d ", arr[i]);
}
return 0;
}

void test(int **tmp, int i)
{
    *(*tmp + i) += 50;
}

```

**Answer:** tmp is declared as a pointer to a pointer to an integer. The statement tmp = &ptr; makes tmp point to the address of ptr, which points to the address of the first element of the array arr. Since \*tmp is equal to ptr, we have \*(\*tmp+i) = \*(ptr+i) = \*(arr+i) = arr[i]. Therefore, each call to test() increases the respective element of the array arr by 50, and the program displays: 60 70 80.

C.20.3 Write a program that reads the order of preference that 100 tourists answered in the question: "What did you enjoy the most in Greece?"

1. The natural diversity
2. The climate
3. The islands
4. The night life
5. The monuments
6. The people
7. The food

Each answer takes 1–7 points according to its rank order. The first answer takes 7 points, the second one takes 6 points, and the last one takes 1 point. For example, if two tourists answered the following:

First Tourist	Second Tourist
5 (7p.)	3 (7p.)
4 (6p.)	7 (6p.)
6 (5p.)	2 (5p.)
3 (4p.)	1 (4p.)
2 (3p.)	5 (3p.)
1 (2p.)	4 (2p.)
7 (1p.)	6 (1p.)

the program should display:

First answer gets 6 points

Second answer gets 8 points

Third answer gets 11 points  
Fourth answer gets 8 points  
Fifth answer gets 10 points  
Sixth answer gets 6 points  
Seventh answer gets 7 points

The program should read valid answers in [1, 7] and check if the answer is already given. If it is, the program should display a message and prompt the user to enter a new one.

```
#include <stdio.h>
#include <string.h>

#define TOURISTS      100
#define ANSWERS       7

int main(void)
{
    int i, j, sel, pnts[ANSWERS] = {0}; /* This array holds the points of each answer. For example, pnts[0] holds the points of the first answer, pnts[1] holds the points of the second answer and so on. */
    int given_ans[ANSWERS] = {0}; /* This array is used to check whether an answer is already given or not. If an element's value is 1, it means that the respective answer is chosen. For example, if the user chooses the third answer, given_ans[2] becomes 1. */
    for(i = 0; i < TOURISTS; i++)
    {
        printf("\nEnter answers of tourist_%d:\n", i+1);

        memset(given_ans, 0, sizeof(given_ans)); /* The values must be zeroed before reading the answers of a new tourist. See memset() in Appendix C. */
        for(j = 0; j < ANSWERS; j++)
        {
            while(1) /* Infinite loop until the user enters a valid answer, not already given. */
            {
                printf("Answer_%d [1-%d]: ", j+1, ANSWERS);
                scanf("%d", &sel);

                if(sel < 1 || sel > ANSWERS)
                    printf("Wrong answer ...\\n");
                else if(given_ans[sel-1] == 1)
                    printf("Error: This answer is already given ...\\n");
                else
                    break;
            }
            pnts[sel-1] += ANSWERS - j; /* For example, if the first answer (j = 0) is the fifth one, then pnts[sel-1] = pnts[5-1] = pnts[4] += 7-0 = 7; that is, 7 more points are added to the points of the fifth choice. */
        }
    }
}
```

```

        given_ans[sel-1] = 1;
    }
}
printf("\n***** Answer Results *****\n");
for(i = 0; i < ANSWERS; i++)
    printf("Answer_%d gets %d points\n", i+1, pnts[i]);
return 0;
}

```

**C.20.4** Write a program that requires two command line arguments that correspond to the dimensions (rows and columns) of a two-dimensional array of doubles. The program should allocate memory to create the array and assign random values to its elements so that the sum of the elements in each row is equal to 1. The program should display the elements of the array before it terminates.

**NOTE:** To convert the command line arguments to integers, use the atoi() function. See its prototype in Appendix C.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[])
{
    int i, j, rows, cols;
    double **arr, sum_line;

    if(argc < 3)
        printf("Error: missing arguments...\n");
    else if(argc == 3)
    {
        rows = atoi(argv[1]); /* Convert the second argument to
integer. */
        cols = atoi(argv[2]); /* Do the same for the third argument. */
        srand(time(NULL));
        arr = (double**) malloc(rows * sizeof(double*));
        if(arr == NULL)
        {
            printf("Error: Not available memory\n");
            exit(EXIT_FAILURE);
        }
        for(j = 0; j < rows; j++)
        {
            arr[j] = (double*) malloc(cols * sizeof(double));
            if(arr[j] == NULL)
            {
                printf("Error: Not available memory\n");
                exit(EXIT_FAILURE);
            }
        }
        for(i = 0; i < rows; i++)
        {
            sum_line = 0;

```

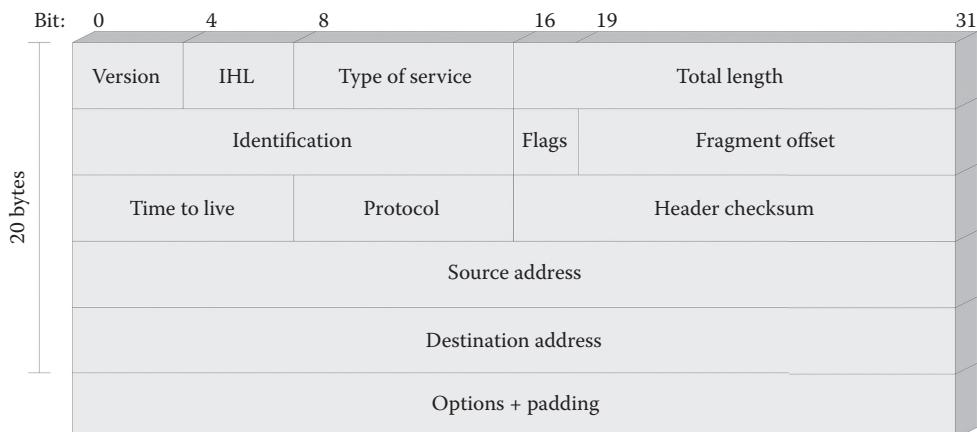
```

        for(j = 0; j < cols-1; j++)
    {
        arr[i][j] = (rand()%101/(cols-1))/100.0;
        sum_line += arr[i][j]; /* As an example,
assume that the number of columns is 5. For the first four rows' elements
rand()%101 generates an integer in [0, 100]. Dividing it by the number of
the columns minus 1, that is, 4, we get an integer in [0, 25] and
dividing that integer with 100.0 we get a number in [0, 0.25]. In this
way, the sum of the first four elements is constrained in [0, 1]. */
    }
    arr[i][j] = 1-sum_line; /* To satisfy the requirement
that the sum of the elements of each row must be equal to 1, the value of
the last element is calculated by subtracting from 1 the sum of the
remaining elements. */
}
for(i = 0; i < rows; i++)
{
    for(j = 0; j < cols; j++)
        printf("%6.2f", arr[i][j]);

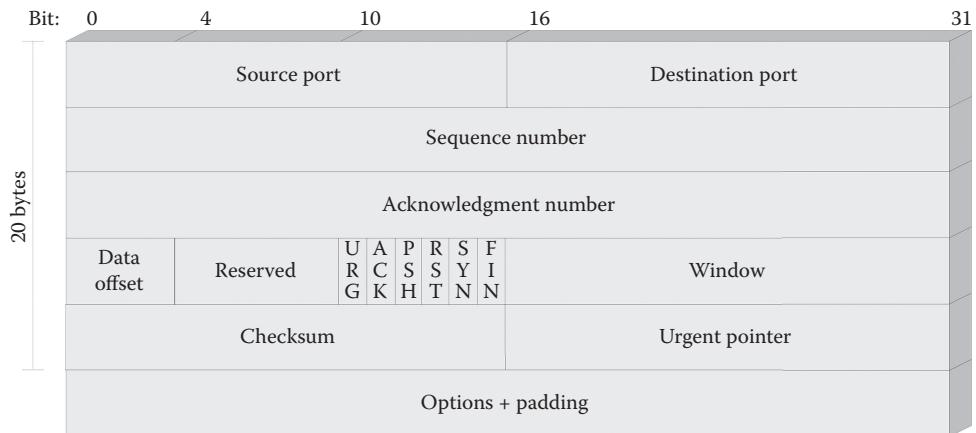
    printf("\n");
}
for(j = 0; j < rows; j++)
    free(arr[j]);
free(arr);
}
else
    printf("Error: too many arguments...\n");
return 0;
}

```

**C.20.5** For a system to connect to an Internet address, its network card must transmit an Internet Protocol (IP) packet that encapsulates a special TCP segment. The IPv4 header format is depicted in Figure 20.1.



**FIGURE 20.1**  
IPv4 header.



**FIGURE 20.2**

TCP header.

The TCP header format is depicted in Figure 20.2.

Write a program that reads the source IP address in  $x.x.x.x$  format (each  $x$  is an integer in  $[0, 255]$ ), the destination IP address in  $x.x.x.x$  format, and the TCP destination port (integer in  $[1, 65535]$ ) and creates an IP packet that encapsulates the proper TCP segment. The program must store the content of the IP packet in hexadecimal format in a user-selected text file. Each line must contain 16 bytes. Set the following values in the IPv4 header:

- Version = 4.
- IHL = 5.
- Total Length = total length of the IP packet, including the TCP data.
- Protocol = 6.
- Time to Live = 255.
- Destination Address = destination IP address.
- Source Address = source IP address.

Set the following values in the TCP header:

- Destination Port = destination TCP port.
- Source Port = 1500.
- Window = the maximum allowed value.
- SYN bit = 1.

Set the remaining fields to 0 and assume that there are no Options fields.

The program has one restriction; if the destination IP address starts from 130.140 or 160.170 and the TCP destination port is 80, do not create the IP packet and display a related message.

```
#include <stdio.h>
#include <stdlib.h>
```

```
typedef unsigned char BYTE;
```

```

void Build_Pkt(int IP_src[], int IP_dst[], int port);
void Save_Pkt(BYTE pkt[], int len);
int read_text(char str[], int size, int flag);

int main(void)
{
    int IP_src[4], IP_dst[4], TCP_dst_port;

    do
    {
        printf("Enter dst port [1-65535]: ");
        scanf("%d", &TCP_dst_port);
    } while(TCP_dst_port < 1 || TCP_dst_port > 65535);

    printf("Enter dst IP (x.x.x.x): ");
    scanf("%d.%d.%d.%d", &IP_dst[0], &IP_dst[1], &IP_dst[2], &IP_dst[3]);

    if(TCP_dst_port == 80)
    {
        if(IP_dst[0] == 130 && IP_dst[1] == 140)
        {
            printf("It isn't allowed to connect to network
130.140.x.x\n");
            return 0;
        }
        else if(IP_dst[0] == 160 && IP_dst[1] == 170)
        {
            printf("It isn't allowed to connect to network
160.170.x.x\n");
            return 0;
        }
    }
    printf("Enter src IP (x.x.x.x): ");
    scanf("%d.%d.%d.%d", &IP_src[0], &IP_src[1], &IP_src[2], &IP_src[3]);

    Build_Pkt(IP_src, IP_dst, TCP_dst_port);
    return 0;
}

void Build_Pkt(int IP_src[], int IP_dst[], int port)
{
    BYTE pkt[40] = {0}; /* Initialize all fields to 0. */
    int i, j;

    pkt[0] = 0x45; /* Version, IHL. */
    pkt[8] = 255; /* Time to Live. */
    pkt[9] = 6; /* Protocol = TCP. */
    for(i = 12, j = 0; i < 16; i++, j++)
        pkt[i] = IP_src[j]; /* IP Source. */
    for(i = 16, j = 0; i < 20; i++, j++)
        pkt[i] = IP_dst[j]; /* IP Destination. */
    pkt[20] = 1500 >> 8; /* TCP Source Port. */
    pkt[21] = 1500 & 0xFF;
    pkt[22] = port >> 8; /* TCP Dest Port. */
}

```

```

pkt[23] = port & 0xFF;
pkt[33] = 2; /* SYN bit. */
pkt[34] = 0xFF; /* The maximum value for the Window field is all
16 bits set to 1. */
pkt[35] = 0xFF;
/* The values of the CheckSum and Urgent Pointer are set in positions
36-40, therefore the total length of the IP packet is 40 bytes. */
pkt[2] = 0; /* IP Total Length. */
pkt[3] = 40;

Save_Pkt(pkt, 40);
}

void Save_Pkt(BYTE pkt[], int len)
{
    FILE *fp;
    char name[100];
    int i;

    printf("Enter file name: ");
    read_text(name, sizeof(name), 1);

    fp = fopen(name, "w");
    if(fp == NULL)
    {
        printf("Error: fopen() failed\n");
        exit(EXIT_FAILURE);
    }
    for(i = 0; i < len; i++)
    {
        if(i%16 == 0)
            putc('\n', fp);
        fprintf(fp, "%02X ", pkt[i]);
    }
    fclose(fp);
}

```

**Comments:** Do you have any idea about what this program really does? This program is an oversimplified version of a popular application, almost certainly installed in your computer, the *firewall*. Like this program, a firewall may prevent communication to specific IP addresses and specific applications (e.g., web servers listen to TCP port 80). In fact, the main part of a firewall is nothing more than a sequence of **if-else** statements.

In a real networking application, this IP packet is encapsulated in a MAC frame (see C.11.38), the hardware of the network card encodes the bits of the frame in digital signals (e.g., two different voltage levels to represent 0 and 1) and forwards the frame to the connected router. The router checks the IP destination address, finds the best path to that destination, and forwards the frame to the next router. Yes, we understand, it is all Greek to you, we just tried to introduce you to the fascinating world of computer networking.

**C.20.6** Suppose that 20 firms participate in a contest. Each firm rates the other firms with a grade in [1, 20]. Write a program that reads their names (less than 100 characters) and the

grades each firm gives to the others. The program should check that each given grade is in [1, 20] and that it is not already given. Then, the program should calculate the total score for each firm and display the rank list in score descending order.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_FIRMS 20

int read_text(char str[], int size, int flag);

int main(void)
{
    char firm_name[MAX_FIRMS][100], tmp[100];
    int i, j, k, found, tot_grd[MAX_FIRMS] = {0}, firm_grd[MAX_FIRMS]
[MAX_FIRMS] = {0};

    for(i = 0; i < MAX_FIRMS; i++)
    {
        printf("\nEnter name of firm_%d: ", i+1);
        read_text(firm_name[i], sizeof(firm_name[i]), 1);

        for(j = 0; j < MAX_FIRMS; j++)
        {
            /* A firm does not grade itself. */
            if(i == j)
                continue;
            while(1)
            {
                do
                {
                    printf("Enter grade for firm_%d [1-%d]: ",
", j+1, MAX_FIRMS);
                    scanf("%d", &firm_grd[i][j]);
                } while(firm_grd[i][j] < 1 || firm_grd[i][j] >
MAX_FIRMS);

                found = 0;
                /* Check if the grade is already given to
another firm. */
                for(k = 0; k < j; k++)
                {
                    if(firm_grd[i][k] == firm_grd[i][j])
                    {
                        printf("Error: The grade %d is
given\n", firm_grd[i][j]);
                        found = 1;
                        break;
                    }
                }
                if(found == 0)
                    break;
            }
        }
    }
}

```

```

        tot_grd[j] += firm_grd[i][j]; /* Each tot_grd[j]
holds the total score of the respective firm. */
    }
    getchar();
}
for(i = 0; i < MAX_FIRMS-1; i++)
{
    for(j = i+1; j < MAX_FIRMS; j++)
    {
        if(tot_grd[i] < tot_grd[j])
        {
            /* Swap grades and the corresponding names. */
            k = tot_grd[i];
            tot_grd[i] = tot_grd[j];
            tot_grd[j] = k;

            strcpy(tmp, firm_name[j]);
            strcpy(firm_name[j], firm_name[i]);
            strcpy(firm_name[i], tmp);
        }
    }
}
for(i = 0; i < MAX_FIRMS; i++)
    printf("%s: %d\n", firm_name[i], tot_grd[i]);
return 0;
}

```

C.20.7 What is the output of the following program?

```

#include <stdio.h>

void test(int **tmp);

int main(void)
{
    int *ptr, arr[] = {5, 10, 15};

    ptr = arr;
    test(&ptr);

    printf("%d ", *ptr);
    return 0;
}

void test(int **tmp)
{
    int i;

    i = *(*tmp)++;
    printf("%d ", i);

    i = ++(**tmp);
    printf("%d ", i);
}

```

**Answer:** When `test()` is called, we have `tmp = &ptr`, so `*tmp` is equivalent to `ptr`. Therefore, the expression `i = *(*tmp)++` is equivalent to `i = *ptr++`. This statement first assigns to `i` the value of `*ptr`, that is, `*ptr = *arr = arr[0] = 5`, and then `ptr` is incremented to point to `arr[1]`. Therefore, the program displays 5.

Then, we have `i = ++(**tmp) = ++(*ptr) = ++arr[1]`. Now, `arr[1]` is first incremented and becomes 11, then, that value is assigned to `i`. Therefore, the program displays 11. After the execution of `test()`, since `ptr` points to `arr[1]`, the program displays 11.

**C.20.8** Write a program that simulates a simple electronic roulette that allows the player to bet on whether the winning number of the next spin of the roulette ball will be odd or even. The numbers of the roulette are from 0 up to 36. If the winning number is 0, the player loses because 0 is counted as neither odd nor even. The program should display a menu to perform the following operations:

1. *Bet on odd numbers*
2. *Bet on even numbers*
3. *Spin the ball*
4. *Statistics*
5. *Termination*

After selecting the kind of bet (i.e., odd or even), the player should specify the bet. When the third option is chosen, the program should generate a random integer in  $[0, 36]$  and display a message to indicate if the player won or lost. The fourth option should display some statistics, that is, how many times the player won/lost and how much money the player wins or loses.

We left the best for the end. The program should be written in such a way that the player should have definitely *lost*, at the end of the game. In particular, the player must lose the first bet and when he or she bets more than the amount he or she loses.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define LOSS 0
#define WIN 1

void unfair_play(int sel, int bet, int *lost, int *times);
int fair_play(int sel, int bet, int *lost);

int main(void)
{
    int sel, last_sel, flag, tmp, bet, sum_lost, win_times,
lost_times;

    flag = 1;
    sum_lost = win_times = lost_times = 0;

    while(1)
    {
        printf("\nRoulette Game\n");
        if(flag)
            unfair_play(sel, bet, &sum_lost, &win_times);
        else
            fair_play(sel, bet, &sum_lost);
        if(sum_lost > bet)
            break;
    }
}
```

```

printf("-----\n");

printf("1. Odd\n");
printf("2. Even\n");
printf("3. Play\n");
printf("4. Stats\n");
printf("5. Exit\n");

printf("\nEnter choice: ");
scanf("%d", &sel);

srand(time(NULL));

switch(sel)
{
    case 1:
    case 2:
        last_sel = sel; /* last_sel holds the last
choice of the player, before the third choice is selected. */
        do
        {
            printf("\nPlace your bet: ");
            scanf("%d", &bet);
        } while(bet <= 0);
        break;

    case 3:
        if(bet == 0)
        {
            printf("No bet is placed\n");
            break;
        }
        if(flag == 1) /* We force the program to play
unfairly and make the player to lose the first bet. */
        {
            flag = 0;
            unfair_play(last_sel, bet,
                        &sum_lost, &lost_times);
        }
        else
        {
            if(bet >= sum_lost) /* If the player
bets a larger amount than the money he loses, the program behaves
unfairly and the player loses. */
                unfair_play(last_sel, bet,
                            &sum_lost, &lost_times);
            else
            {
                tmp = fair_play(last_sel, bet,
                                &sum_lost);
                if(tmp == LOSS)
                {
                    printf("Sorry, you lost
...\\n");
                }
            }
        }
    }
}

```

```

    lost_times++;
}
else
{
    printf("Yaaaaah, you won
...\\n");
    win_times++;
}
}
}
bet = 0;
break;

case 4:
printf("\nWin_Times: %d\tLost_Times: %d\tLost_
Money: %d euro\\n", win_times, lost_times, sum_lost);
break;

case 5:
return 0;

default:
printf("\nWrong choice\\n");
break;
}
}
return 0;
}

```

```

/* The program generates a number, so that the player loses. For example,
if the player bets on odd numbers the program generates an even number,
and vice versa. */
void unfair_play(int sel, int bet, int *lost, int *times)
{
    int num;

*lost += bet;
(*times)++;

while(1)
{
    num = rand() % 37;
    if(sel == 1) /* The player bets on odd numbers. */
    {
        if((num & 1) == 0)
        {
            printf("\\nThe ball goes to %d. Sorry, you lost
...\\n", num);
            return;
        }
    }
    else /* The player bets on even numbers. */
    {
        if((num & 1) == 1)

```

```

    {
        printf("\nThe ball goes to %d. Sorry, you lost
...\\n", num);
        return;
    }
}

int fair_play(int sel, int bet, int *lost)
{
    int num;

    num = rand() % 37;
    printf("\nThe ball goes to %d. ", num);

    if(num == 0) /* If zero comes out, the player loses. */
    {
        *lost += bet;
        return LOSS;
    }
    if(sel == 1) /* The player bets on odd numbers. */
    {
        if((num & 1) == 1)
        {
            *lost -= bet; /* If the player wins, the total lost
amount is reduced. */
            return WIN;
        }
        else
            *lost += bet; /* If the player loses, the total lost
amount is increased. */
    }
    else /* The player bets on even numbers. */
    {
        if((num & 1) == 0)
        {
            *lost -= bet;
            return WIN;
        }
        else
            *lost += bet;
    }
    return LOSS;
}

```

**Comments:** Since the program controls the bets, the player eventually *loses* even if he or she won more times than he or she lost. In other words, the program creates the feeling of “misfortune” in the player, whereas it actually cheats him or her.

However, the main purpose of this exercise is instructive. With this simple simulation example, we want to show you that gambling games’ software can be written in a way that creates the feeling of “bad luck,” whereas the player is just a victim of fraud.

Therefore, stay away from online gaming and any kind of electronic “lucky machines” (e.g., slot machines, fruit machines, and so on). The big profits they promise are not for you, but for their owners.

C.20.9 What is the output of the following program?

```
#include <stdio.h>

void test(int val, int *tmp);

int main(void)
{
    int *ptr, i, arr[] = {5, 10, 15};

    ptr = arr;
    for(i = 0; i < 2; i++)
    {
        test(*ptr, ptr);
        ptr++;
    }
    while(ptr >= arr)
    {
        printf("%d ", *ptr);
        ptr--;
    }
    return 0;
}

void test(int val, int *tmp)
{
    printf("%d %d\n", ++val, (*tmp)++);
}
```

### Answer:

*First iteration of the for loop:* When `test()` is called, we have `val = *ptr = *arr = arr[0] = 5`. The expression `++val` first increments `val` by one, and the program displays 6. Also, we have `tmp = ptr = arr`. Therefore, the expression `(*tmp)++` is equivalent to `(*arr)++ = arr[0]++`. Now, the program first displays the value of `arr[0]`, that is, 5, and then it is incremented by one.

*Second iteration of the for loop:* `ptr` has been increased, so it points to the second element. When `test()` is called, we have `val = *ptr = *(arr+1) = arr[1] = 10`. The expression `++val` increments `val` and the program displays 11. Also, we have `tmp = ptr = arr+1 = &arr[1]`. Therefore, the expression `(*tmp)++` is equivalent to `arr[1]++`. The program first displays the value of `arr[1]`, that is, 10, and then it is incremented by one.

*Execution of the while loop:* `ptr` has been increased, so it points to the third element. Therefore, the first iteration displays the value of `arr[2]`, that is, 15, and `ptr` is decremented and points to `arr[1]`. Therefore, the second iteration displays the value of `arr[1]`, that is, 11, and the last one the value of `arr[0]`, that is, 6.

To sum up, the program displays:

```
6      5  
11    10  
15   11   6
```

C.20.10 Because a C program executes rather quickly, the C language is often used for implementing cipher algorithms. As an example, we'll describe one of the most simple and famous cipher algorithms.

The Caesar algorithm is one of the oldest cipher methods used by Julius Caesar to encrypt his messages. According to this algorithm, each character is substituted by the one located in the next three places. For example, if we apply the Caesar algorithm in the English alphabet, the message "Watch out for Ovelix !!!" is encrypted as "Zdwfk rxw iru Ryhola !!!". Notice that the character 'x' is encrypted as 'a' because the substitution continues from the beginning of the alphabet. Similarly, 'y' is replaced by 'b' and 'z' by 'c'. The recipient decrypts the message by substituting each character with the one located three places before it.

Write a program that provides a menu to perform the following operations:

1. File Encryption. The program should read the name of a file and the key that will be used to encrypt its content. For example, in the case of Caesar algorithm, the key is 3. The program should encrypt only the lowercase and uppercase characters.
2. File Decryption. The program should read the name of a file and the key that will be used to decrypt its content.
3. Program Termination.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
void cipher(FILE *fp_in, FILE *fp_out, int key);  
void decipher(FILE *fp_in, FILE *fp_out, int key);  
int read_text(char str[], int size, int flag);  
  
int main(void)  
{  
    FILE *fp_in, *fp_out;  
    char str[100];  
    int sel, key;  
  
    while(1)  
    {  
        printf("\nMenu selections\n");  
        printf("-----\n");  
        printf("1. Cipher\n");  
        printf("2. Decipher\n");  
        printf("3. Exit\n");  
  
        printf("\nEnter choice: ");  
        scanf("%d", &sel);
```

```
switch(sel)
{
    case 1:
    case 2:
        /* Check whether the input key is valid or
not. Since we are using the English alphabet, the valid values are
between 1 and 25. */
        do
        {
            printf("Enter key size: ");
            scanf("%d", &key);
        } while(key < 1 || key > 25);

        getchar();
        printf("Input file: ");
        read_text(str, sizeof(str), 1);
        fp_in = fopen(str, "r");
        if(fp_in == NULL)
        {
            printf("Error: fopen() for %s failed\n",
str);
            exit(EXIT_FAILURE);
        }
        printf("Output file: ");
        read_text(str, sizeof(str), 1);
        fp_out = fopen(str, "w");
        if(fp_out == NULL)
        {
            printf("Error: fopen() for %s failed\n",
str);
            exit(EXIT_FAILURE);
        }
        if(sel == 1)
            cipher(fp_in, fp_out, key);
        else
            decipher(fp_in, fp_out, key);

        fclose(fp_in);
        fclose(fp_out);
        break;

    case 3:
        return 0;

    default:
        printf("\nWrong choice\n");
        break;
}
return 0;
}
```

```
void cipher(FILE *fp_in, FILE *fp_out, int key)
{
    int ch;

    while(1)
    {
        ch = getc(fp_in);
        if(ch == EOF)
            return;
        /* Only the lower and upper case characters are encrypted. */
        if(ch >= 'A' && ch <= 'Z')
        {
            ch += key;
            if(ch > 'Z')
                ch -= 26;
        }
        else if(ch >= 'a' && ch <= 'z')
        {
            ch += key;
            if(ch > 'z')
                ch -= 26;
        }
        putc(ch, fp_out);
    }
}

void decipher(FILE *fp_in, FILE *fp_out, int key)
{
    int ch;

    while(1)
    {
        ch = getc(fp_in);
        if(ch == EOF)
            return;
        /* Only the lower and upper case characters are decrypted. */
        if(ch >= 'A' && ch <= 'Z')
        {
            ch -= key;
            if(ch < 'A')
                ch += 26;
        }
        else if(ch >= 'a' && ch <= 'z')
        {
            ch -= key;
            if(ch < 'a')
                ch += 26;
        }
        putc(ch, fp_out);
    }
}
```

**Comments:** In cipher(), the key is added to each character. If the new value exceeds the last character of the alphabet ('Z' or 'z'), we subtract 26 to go back to the beginning of the alphabet. In decipher(), the reverse actions take place.

C.20.11 What is the output of the following program?

```
#include <stdio.h>
#include <string.h>

void test(char a, char str[], char *ptr);

int main(void)
{
    char *tmp, txt[20] = "abcde";

    tmp = txt;
    test(*(tmp+1), txt+3, &txt[1]);
    printf("%s\n", txt);
    return 0;
}

void test(char a, char str[], char *ptr)
{
    strcpy(str, "1234");
    str[0] = a;
    ptr[2] = *str + 5;
}
```

**Answer:** When test() is called, we have:

- a.  $a = *(tmp+1) = *(txt+1) = txt[1] = b$ .
- b.  $str = txt+3$ , so,  $str[0]$  is equal to  $txt[3]$ .
- c.  $ptr = &txt[1] = txt+1$ . Since  $ptr$  points to  $txt[1]$ ,  $ptr[0]$  is equal to  $txt[1]$ ,  $ptr[1]$  is equal to  $txt[2]$ , and  $ptr[2]$  is equal to  $txt[3]$ .

strcpy() copies the string "1234" into the memory that str points to. Since str points to  $txt[3]$ , the content of  $txt$  becomes "abc1234".

The statement  $str[0] = a$ ; is equivalent to  $txt[3] = a$ . Since the value of  $a$  is equal to 'b',  $txt[3]$  becomes 'b' and the content of  $txt$  changes to "abcb234".

As said, the values of  $str[0]$  and  $ptr[2]$  are equal to  $txt[3]$ . Therefore,  $*str$  is equal to 'b' and the statement  $ptr[2] = *str+5$ ; is equivalent to  $txt[3] = 'b'+5$ ; meaning that  $txt[3]$  becomes equal to the character located five places after 'b', that is, 'g'.

Therefore, the program displays abcg234.

C.20.12 Suppose that we have created subnets in a Class C IP network. An IP network is specified as Class C when the first octet (byte) of its IP address is within [192, 223]. Write a program that reads the last octet of an IP address and the subnet mask and displays the addresses of all subnets, the broadcast address of each subnet, and the subnet in which the IP address belongs to.

For example, suppose that we have a Class C IP network (e.g., x.x.x.x) and the user enters 74. Therefore, we have to find out in which subnet the IP address x.x.x.74 belongs to. To find the subnet, the user must enter the subnet mask in one of the two following ways:

- a. either enter the last octet of the subnet mask (valid values are 252, 248, 240, 224, 192, 128) or,
- b. enter the number of the network bits, which must be an integer in [25, 30].

To find out the subnets, we calculate the distance between them:

- a. In the first case, the distance is equal to  $256-x$ , where x is the input number.
- b. In the second case, the distance is equal to  $2^{32-x}$ , where x is the input number of bits. For example, if the user enters 26, the distance is equal to  $2^{32-26} = 2^6 = 64$ .

Starting from the IP address x.x.x.0, the IP address of each subnet starts from a number multiple of the distance, while the broadcast IP address is the last address of each subnet. For example, if the distance is 64, four subnets (i.e., 256/64) can be created and each subnet starts from an IP address multiple of 64. The broadcast address is always one less than the address of the next subnet. The program should display the last octet of each IP address, as follows:

Network:	.0	.64	.128	.192
Broadcast:	.63	.127	.191	.255

The valid IP range that may be assigned to a system is the numbers between the subnet address and the broadcast address. Therefore, a system with IP address x.x.x.74 is a member of the second subnet because its last octet falls in [64, 127].

Although the length of this exercise may discourage you to deal with it, it'd be useful for those who take "computer networking" courses to learn in both theoretical and programming level a simple method to create IP subnets fast. This method works the same for the other two classes, A and B. Of course there are several other ways for subnetting; however, this method helped me a lot to configure easily and quickly IP addresses in a router and detect configuration errors in an IP network. I'd definitely recommend it.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i, j, flag, sel, num, dist, host_byte;

    do
    {
        printf("Enter last host byte [0-255]: ");
        scanf("%d", &host_byte);
    } while(host_byte < 0 || host_byte > 255);

    do
    {
        flag = 0;
```

```

printf("Enter mask (0: 255.255.255.x form or 1: /bits form): ");
scanf("%d", &sel);
if(sel == 0)
{
    printf("Enter last mask octet 255.255.255.");
    scanf("%d", &num);
    if(num != 252 && num != 248 && num != 240
        && num != 224 && num != 192 && num != 128)
    {
        printf("Last octet should be one of {128, 192,
224, 240, 248, 252}\n");
        flag = 1;
    }
    else
        dist = 256-num;
}
else if(sel == 1)
{
    printf("Enter network bits: /");
    scanf("%d", &num);

    if(num < 25 || num > 30)
    {
        printf("Enter valid mask /25-/30\n");
        flag = 1;
    }
    else
    {
        num = 32-num;
        dist = 1;
        for(i = 0; i < num; i++)
            dist = dist*2; /* For faster
implementation we could replace the loop with dist = 1 << num; */
    }
}
} while(flag == 1);

```

printf("\nThe mask 255.255.255.%d produces %d subnets, each with  
%d hosts\n", 256-dist, 256/dist, dist-2);

```

printf("\nNetwork : ");
for(i = 0; i < 256; i+=dist)
    printf("%.d\t", i);

printf("\nBroadcast: ");
for(i = dist-1; i < 256; i+=dist)
    printf("%.d\t", i);

for(i=j=0; i < 256; i+=dist, j++)
{
    if(host_byte >= i && host_byte < i+dist)
    {
        if(host_byte == i)

```

```

        printf("\n\nThe x.x.x.%d address is the
network address of subnet_%d\n", host_byte, j+1);
    else if(host_byte == i+dist-1)
        printf("\n\nThe x.x.x.%d address is the
broadcast address of subnet_%d\n", host_byte, j+1);
    else
        printf("\n\nThe x.x.x.%d address belongs in
subnet %d\n", host_byte, j+1);
        break;
    }
}
return 0;
}

```

**C.20.13** What is the output of the following program?

```

#include <stdio.h>
int main(void)
{
    char *arr[] = {"TEXT", "SHOW", "OPTIM", "DAY"};
    char **ptr1;

    ptr1 = arr;
    printf("%s ", *++ptr1);
    printf("%s", *++ptr1+2);
    printf("%c\n", *++ptr1+1);
    return 0;
}

```

**Answer:** The arr array is declared as an array of pointers to strings. In particular, arr[0] points to "TEXT", arr[1] points to "SHOW", and so on. The statement ptr1 = arr; makes ptr1 to point to the address of arr[0].

The expression ++ptr1 makes ptr1 point to arr[1]. Since \*++ptr1 is equivalent to arr[1], the program displays SHOW.

Similarly, the expression ++ptr1+2 makes first ptr1 to point to arr[2]. Since \*++ptr1 is equivalent to arr[2], printf() is equivalent to printf("%s", ptr[2]+2); and the program skips the first two characters of "OPTIM" and displays TIM.

Like before, the expression ++ptr1+1 makes first ptr1 to point to arr[3]. Therefore, \*++ptr1 is equivalent to \*arr[3]. What is the value of \*arr[3]? Since arr[3] points to the first character of "DAY", \*arr[3] is equal to 'D'. Therefore, the value of \*arr[3]+1 is 'E' and the program displays E.

As a result, the program displays: SHOW TIME

**C.20.14** Write a program that simulates the popular game master-mind. The program should generate a random sequence of four different colors out of six available colors (e.g., red:0, green:1, blue:2, white:3, black:4, yellow:5). This is the secret sequence. Then, the program should prompt the user to enter sequences of colors in order to find the secret one. For each given sequence, the program should inform the user of the number of the right colors in right places and the number of right colors in wrong places.

For example, if the secret sequence to be found is: 0 1 2 3 and the user enters the sequence: 5 1 4 2, the program should display: "one right color in the right place and one right color in wrong place." If the user makes 10 unsuccessful attempts, the program terminates and the user loses.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define DEBUG

void guess_code(int guess[]);

int main(void)
{
    int i, j, k, found, corr_guess, col_guess, code[4], guess[4];

    srand(time(NULL));
    i = 0;
    /* Generate a random sequence with four different colors. */
    while(i < 4)
    {
        code[i] = rand()%6;
        found = 0;
        for(j = 0; j < i; j++)
        {
            if(code[j] == code[i])
            {
                found = 1;
                break;
            }
        }
        if(found == 0)
            i++;
    }
#ifdef DEBUG
    printf("\n**** Secret code is: %d %d %d %d\n\n", code[0], code[1],
code[2], code[3]);
#endif
    for(i = 0; i < 10; i++)
    {
        guess_code(guess);

        corr_guess = col_guess = 0;
        /* Calculate the number of right colors in right places. */
        for(j = 0; j < 4; j++)
        {
            if(guess[j] == code[j])
                corr_guess++;
        }
    }
}
```

```

    if(corr_guess == 4)
    {
        printf("\nCongratulations !!! You did it in %d
attempts\n", i+1);
        return 0;
    }
    /* Calculate the number of right colors in wrong places. */
    for(j = 0; j < 4; j++)
    {
        for(k = 0; k < 4; k++)
        {
            /* The second term is needed, in order not to
add again the number of right colors in right places. */
            if((guess[k] == code[j]) && (k != j))
            {
                col_guess++;
                break;
            }
        }
    }
    printf("Colors in right places = %d\nMatching colors in
wrong places = %d\n\n", corr_guess, col_guess);
}
printf("\nFail: The secret code is %d %d %d %d\n", code[0],
code[1], code[2], code[3]);
return 0;
}

/* For simplicity, we assume that the user enters different colors. */
void guess_code(int guess[])
{
    int i, col;

    i = 0;
    while(i < 4)
    {
        printf("Enter color (RED:0, GREEN:1, BLUE:2, WHITE:3, BLACK:4,
YEL:5): ");
        scanf("%d", &col);

        switch(col)
        {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
            case 5:
                guess[i] = col;
                i++;
            break;

            default:
                printf("Wrong choice\n");
        }
    }
}

```

```
        break;
    }
}
```

**Comments:** The reason we didn't remove the DEBUG macro is to show you how we can use a macro to check the operation of the program.

**C.20.15** Write a program that can be used as a book library management application. Define the structure type book with members: title, authors, and code. Suppose that the test.bin binary file exists and contains structures of that type. Write a program that provides a menu to perform the following operations:

1. Add a new book. The program should read the data of a new book and add it at the end of the file.
2. Search for a book. The program should read the title of a book and display its details. If the user enters \*, the program should display the details of all books.
3. Modify a book. The program should read the title of a book and its new data and it should replace the existing data with the new one.
4. Delete a book. The program should read the title of a book and set its code equal to -1.
5. Program termination.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LEN 50

typedef struct
{
    char title[LEN];
    char auth[LEN];
    int code;
} book;

void read_data(book *ptr);
void find_book(FILE *fp, char title[]);
void modify_book(FILE *fp, book *ptr, int flag);
void show_books(FILE *fp);

int read_text(char str[], int size, int flag);

int main(void)
{
    FILE *fp;
    char title[LEN];
    int sel;
    book b;

    fp = fopen("test.bin", "r+b");
```

```
if(fp == NULL)
{
    printf("Error: fopen() failed\n");
    exit(EXIT_FAILURE);
}
while(1)
{
    printf("\n\nMenu selections\n");
    printf("-----\n");

    printf("1. Add Book\n");
    printf("2. Find Book\n");
    printf("3. Modify Book \n");
    printf("4. Erase Book\n");
    printf("5. Exit\n");

    printf("\nEnter choice: ");
    scanf("%d", &sel);
    getchar();

    switch(sel)
    {
        case 1:
            read_data(&b);
            fseek(fp, 0, SEEK_END); /* Add the book at the
end of the file. */
            fwrite(&b, sizeof(book), 1, fp);
            break;

        case 2:
            printf("\nTitle to search: ");
            read_text(title, sizeof(title), 1);

            if(strcmp(title, "*") != 0)
                find_book(fp, title);
            else
                show_books(fp);
            break;

        case 3:
            read_data(&b);
            modify_book(fp, &b, 0);
            break;

        case 4:
            printf("\nTitle to search: ");
            read_text(b.title, sizeof(b.title), 1);
            modify_book(fp, &b, 1);
            break;

        case 5:
            fclose(fp);
    }
    return 0;
}
```

```
        default:
            printf("\nWrong choice\n");
            break;
    }
}
return 0;
}

void read_data(book *ptr)
{
    printf("\nTitle: ");
    read_text(ptr->title, sizeof(ptr->title), 1);

    printf("Authors: ");
    read_text(ptr->auth, sizeof(ptr->auth), 1);

    do
    {
        printf("Code [> 0]: ");
        scanf("%d", &ptr->code);
    }
    while(ptr->code <= 0);
}

void find_book(FILE *fp, char title[])
{
    book b;

    fseek(fp, 0, SEEK_SET);
    while(1)
    {
        if(fread(&b, sizeof(book), 1, fp) != 1)
            break;
        else
        {
            if(strcmp(b.title, title) == 0)
            {
                printf("\nT:%s A:%s C:%d", b.title, b.auth,
b.code);
                return;
            }
        }
    }
    printf("\n%s doesn't exist", title);
}

void modify_book(FILE *fp, book *ptr, int flag)
{
    book b;

    fseek(fp, 0, SEEK_SET);
    while(1)
    {
        if(fread(&b, sizeof(book), 1, fp) != 1)
            break;
        else
```

```

    {
        if(strcmp(b.title, ptr->title) == 0)
        {
            /* Now, fp points to the next entry. We call
fseek(), to make it point at the current entry. */
            fseek(fp, -sizeof(book), SEEK_CUR);
            if(flag == 0)
                fwrite(ptr, sizeof(book), 1, fp);
            else
            {
                b.code = -1; /* Set the code to -1. */
                fwrite(&b, sizeof(book), 1, fp);
                printf("\n%s title is erased",
ptr->title);
            }
        }
    }
    printf("\n%s doesn't exist", ptr->title);
}

void show_books(FILE *fp)
{
    book b;

    fseek(fp, 0, SEEK_SET);
    while(1)
    {
        if(fread(&b, sizeof(book), 1, fp) != 1)
            return;
        else
            printf("T:%s A:%s C:%d\n", b.title, b.auth, b.code);
    }
}

```

**C.20.16** The permutation method is an example of symmetric cryptography, in which the sender and the receiver share a common key. The encryption key is an integer of  $n$  digits ( $n \leq 9$ ). Each digit must be between 1 and  $n$  and appear only once. In the encryption process, the message is divided into segments of size  $n$ . For example, since this is the last exercise (yes, in a minute we'll set you free), suppose that we are using the key 25413 to encrypt the message This is the end!! from the classical song of *Doors: The End*. Since the key size is 5, the message is divided into four segments of 5 characters each. If the size of the last segment is less than the key size, padding characters are added. Suppose that the padding character is the \*, as shown in Figure 20.3.

T	h	i	s		i	s		t	h	e		e	n	d	!	!	*	*	*
h		s	T	i	s	h	t	i			d	n	e	e	!	*	*	!	*

**FIGURE 20.3**

Permutation encryption algorithm.

The characters of each segment are rearranged according to the key digits. For example, the second character of the original message is the first one in the encrypted segment, the fifth character goes to the second position, the fourth character in the third position, and so on, as shown in Figure 20.3. This process is repeated for each segment.

The recipient uses the same key to decrypt the message. The reverse process takes place. For example, the first character of the first encrypted segment corresponds to the second character of the original message, the second character corresponds to the fifth one, the third character to the fourth one, and so on.

Write a program that reads a string of less than 100 characters, the encryption key, and the padding character and uses that method to encrypt the string. Then, the program should decrypt the message and display it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int read_text(char str[], int size, int flag);

int main(void)
{
    char pad_ch, key_str[10], in_str[110] = {0}, out_str[110] = {0};
/* The size of the arrays is more than 100 characters, to cover the case
of padding characters in the last segment. */
    int i, j, tmp, seg, key_len, max_key_dig, msg_len;

    tmp = 1;
    while(tmp)
    {
        tmp = 0; /* The loop ends only if tmp remains 0. */
        printf("Enter 1 up to 9 different key digits: ");
        key_len = read_text(key_str, sizeof(key_str), 1);
        if(key_len < 1 || key_len > 9)
        {
            printf("Error: Length should be 1 to 9 different
digits\n");
            tmp = 1;
            continue;
        }
        max_key_dig = '0'; /* This variable holds the key digit with
the highest value. */
        for(i = 0; (tmp != 1) && i < key_len; i++)
        {
            if(key_str[i] < '1' || key_str[i] > '9')
            {
                printf("Error: Only digits are allowed\n");
                tmp = 1;
                break;
            }
            if(key_str[i] > max_key_dig)
                max_key_dig = key_str[i];
/* Check if each digit appears once. */
            for(j = i+1; j < key_len; j++)
                if(key_str[j] == key_str[i])
                    tmp = 1;
        }
    }
}
```

```

    {
        if(key_str[i] == key_str[j])
        {
            printf("Error: Digits should be
different\n");
            tmp = 1;
            break;
        }
    }
    if(tmp == 0)
    {
        max_key_dig -= '0';
        if(key_len != max_key_dig) /* For example, the key
value 125 is not acceptable, because the value of the highest digit
(i.e., 5) is not equal to the key length, which is 3. */
        {
            printf("Error: Digits should be from 1 to
%d\n", key_len);
            tmp = 1;
        }
    }
    printf("Enter padding character: ");
    pad_ch = getchar();

    getchar();
    while(1)
    {
        printf("Enter text: ");
        msg_len = read_text(in_str, sizeof(in_str), 1);
        if(msg_len >= key_len)
            break;
        else
            printf("Error: Text length must be greater than the
key size\n");
    }
    seg = msg_len/key_len;
    tmp = msg_len - (seg*key_len);

    if(tmp != 0) /* If it is not 0, it means that the message length
is not divided exactly by the length of the key and padding characters
must be added. Notice that the replacement starts from the position of
the null character. */
    {
        seg++;
        for(i = 0; i < key_len-tmp; i++)
            in_str[msg_len+i] = pad_ch;
    }
    for(i = 0; i < seg; i++)
    {
        for(j = 0; j < key_len; j++)

```

```
{  
    tmp = key_str[j]-1; /* We subtract the ASCII value of  
1, in order to use tmp as an index to the original message. */  
    out_str[i*key_len+j] = in_str[i*key_len+tmp];  
}  
}  
printf("Encoded text: %s\n", out_str);  
for(i = 0; i < seg; i++)  
{  
    for(j = 0; j < key_len; j++)  
    {  
        tmp = key_str[j]-1;  
        in_str[i*key_len+tmp] = out_str[i*key_len+j];  
    }  
}  
printf("Decoded text: %s\n", in_str); /* Any padding characters  
appear at the end of the original message. */  
return 0;  
}
```

# Appendix 1: Precedence Table

Table A.1.1 lists C operators from the highest to the lowest order of precedence. Operators listed in the same line have the same precedence. The last column indicates the order in which operators of the same precedence are evaluated.

**TABLE A.1.1**

Precedence Table

Precedence	Operators	Associativity
1	( ) (parentheses-function call) [] -> . ++(postfix) --(postfix)	Left to right
2	++(prefix) --(prefix) ! ~ * (dereference) &(address) +(unary) -(unary) (cast) <b>sizeof</b>	Right to left
3	* (multiplication) / %	Left to right
4	+ (binary) - (binary)	Left to right
5	<< >>	Left to right
6	< <= > >=	Left to right
7	== !=	Left to right
8	&	Left to right
9	^	Left to right
10		Left to right
11	&&	Left to right
12		Left to right
13	? :	Right to left
14	= += -= *= /= %= &= ^=  = <<= >>=	Right to left
15	,	Left to right

## *Appendix 2: ASCII Table*

This appendix presents the standard (0–127) and extended (128–255) ASCII character sets.

TABLE A.2.1

## ASCII Character Set

Dec	Hex	Name	Char	Ctrl-Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	
0	0	Null		NUL	CTRL-@	32	20	Space	64	40	@	96	60	`	128	80		160	A0	á	192	C0	l	224	E0	α
1	1	Start of heading		SOH	CTRL-A	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	l	225	E1	β
2	2	Start of text		STX	CTRL-B	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	t	226	E2	Γ
3	3	End of text		ETX	CTRL-C	35	23	#	67	43	C	99	63	c	131	83	à	163	A3	ú	195	C3	l	227	E3	π
4	4	End of xmit		EOT	CTRL-D	36	24	\$	68	44	D	100	64	d	132	84	à	164	A4	ñ	196	C4	-	228	E4	Σ
5	5	Enquiry		ENQ	CTRL-E	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	ñ	197	C5	+	229	E5	σ
6	6	Acknowledge		ACK	CTRL-F	38	26	&	70	46	F	102	66	f	134	86	à	166	A6	á	198	C6	ƒ	230	E6	μ
7	7	Bell		BEL	CTRL-G	39	27	'	71	47	G	103	67	g	135	87	ſ	167	A7	o	199	C7	॥	231	E7	τ
8	8	Backspace		BS	CTRL-H	40	28	(	72	48	H	104	68	h	136	88	é	168	A8	ž	200	C8	ł	232	E8	Φ
9	9	Horizontal tab		HT	CTRL-I	41	29	)	73	49	I	105	69	i	137	89	ë	169	A9	‐	201	C9	ſ	233	E9	Θ
10	0A	Line feed		LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j	138	8A	ë	170	AA	‑	202	CA	ł	234	EA	Ω
11	0B	Vertical tab		VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k	139	8B	í	171	AB	½	203	CB	ſ	235	EB	δ
12	0C	Form feed		FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l	140	8C	í	172	AC	¼	204	CC	ł	236	EC	∞
13	0D	Carriage feed		CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m	141	8D	í	173	AD	i	205	CD	=	237	ED	φ
14	0E	Shift out		SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n	142	8E	Á	174	AE	«	206	CE	¾	238	EE	ε
15	0F	Shift in		SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o	143	8F	Á	175	AF	»	207	CF	±	239	EF	∩
16	10	Data line escape		DLE	CTRL-P	48	30	0	80	50	P	112	70	p	144	90	É	176	B0	■	208	D0	⊥	240	F0	≡
17	11	Device control 1		DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	■■	209	D1	⊤	241	F1	±
18	12	Device control 2		DC2	CTRL-R	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	■■	210	D2	⊤	242	F2	≥
19	13	Device control 3		DC3	CTRL-S	51	33	3	83	53	S	115	73	s	147	93	ð	179	B3	—	211	D3	⊤	243	F3	≤
20	14	Device control 4		DC4	CTRL-T	52	34	4	84	54	T	116	74	t	148	94	ö	180	B4	—	212	D4	⊤	244	F4	ƒ
21	15	Neg. acknowledge		NAK	CTRL-U	53	35	5	85	55	U	117	75	u	149	95	ð	181	B5	—	213	D5	ƒ	245	F5	•
22	16	Synchronous idle		SYN	CTRL-V	54	36	6	86	56	V	118	76	v	150	96	ü	182	B6	—	214	D6	†	246	F6	÷
23	17	End of xmit block		ETB	CTRL-W	55	37	7	87	57	W	119	77	w	151	97	í	183	B7	—	215	D7	†	247	F7	≈
24	18	Cancel		CAN	CTRL-X	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	—	216	D8	†	248	F8	≈
25	19	End of medium		EM	CTRL-Y	57	39	9	89	59	Y	121	79	y	153	99	ó	185	B9	—	217	D9	—	249	F9	•
26	1A	Substitute		SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	—	218	DA	—	250	FA	·
27	1B	Escape		ESC	CTRL-I	59	3B	;	91	5B	I	123	7B	i	155	9B	€	187	BB	—	219	DB	■	251	FB	√
28	1C	File separator		FS	CTRL-\	60	3C	<	92	5C	\	124	7C	\	156	9C	£	188	BC	—	220	DC	—	252	FC	„
29	1D	Group separator		G5	CTRL-J	61	3D	=	93	5D	J	125	7D	j	157	9D	¥	189	BD	—	221	DD	—	253	FD	2
30	1E	Record separator		RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~	158	9E	₱	190	BE	—	222	DE	—	254	FE	■
31	1F	Unit separator		US	CTRL_-	63	3F	?	95	5F	—	127	7F	DEL	159	9F	f	191	BF	—	223	DF	—	255	FF	—

# Appendix 3: Library Functions

This appendix provides a brief description of the C standard library functions.

## <assert.h>

**void assert(int exp):** If the value of `exp` is 0, a diagnostic message is displayed and the program terminates. Otherwise, it does nothing.

## <ctype.h>

**int isalnum(int ch):** It returns a nonzero value if `ch` is alphanumeric (e.g., a-z, A-Z, or 0-9), 0 otherwise.

**int isalpha(int ch):** It returns a nonzero value if `ch` is alphabetic (e.g., a-z or A-Z), 0 otherwise.

**int iscntrl(int ch):** It returns a nonzero value if `ch` is a control character, 0 otherwise.

**int isdigit(int ch):** It returns a nonzero value if `ch` is a decimal digit, 0 otherwise.

**int isgraph(int ch):** It returns a nonzero value if `ch` is a printing character (except space), 0 otherwise.

**int islower(int ch):** It returns a nonzero value if `ch` is a lowercase letter, 0 otherwise.

**int isprint(int ch):** It returns a nonzero value if `ch` is a printing character (including space), 0 otherwise.

**int ispunct(int ch):** It returns a nonzero value if `ch` is a punctuation character, 0 otherwise.

**int isspace(int ch):** It returns a nonzero value if `ch` is a white-space character, 0 otherwise.

**int isupper(int ch):** It returns a nonzero value if `ch` is an uppercase letter, 0 otherwise.

**int isxdigit(int ch):** It returns a nonzero value if `ch` is a hexadecimal digit (a-f, A-F, 0-9), 0 otherwise.

**int tolower(int ch):** If `ch` is an uppercase letter, it returns the corresponding lowercase letter. Otherwise, it returns `ch` unchanged.

**int toupper(int ch):** If `ch` is a lowercase letter, it returns the corresponding uppercase letter. Otherwise, it returns `ch` unchanged.

## <locale.h>

**struct lconv \*localeconv():** It returns a pointer to a structure of type `lconv`, which contains information about the local settings.

**char \*setlocale(int category, char \*locale):** Sets the portion of the program's locale settings specified by `category` and `locale`.

## <math.h>

**int** abs(**int** a): It returns the absolute value of a.

**double** acos(**double** a): It returns the arc cosine of a in the range 0 to  $\pi$ .

**double** asin(**double** a): It returns the arc sine of a in the range  $-\pi/2$  to  $\pi/2$ .

**double** atan(**double** a): It returns the arc tangent of a in the range  $-\pi/2$  to  $\pi/2$ .

**double** atan2(**double** a, **double** b): It returns the arc tangent of a/b in the range  $-\pi$  to  $\pi$ .

**double** ceil(**double** a): It returns the smallest integer that is greater than or equal to a.

**double** cos(**double** a): It returns the cosine of a.

**double** cosh(**double** a): It returns the hyperbolic cosine of a.

**double** exp(**double** a): It returns the result of  $e^a$ , where e is the logarithmic base, that is, 2.7182....

**double** fabs(**double** a): It returns the absolute value of a.

**double** floor(**double** a): It returns the largest integer that is less than or equal to a.

**double** fmod(**double** a, **double** b): It returns the floating-point remainder of a/b.

**double** frexp(**double** a, **int** \*ptr): Splits a into a fractional part m and an exponent n, so that  $a = m \times 2^n$ . It returns m.

**long** labs(**long** a): It returns the absolute value of a.

**double** ldexp(**double** a, **int** n): It returns the result of  $a \times 2^n$ .

**double** log(**double** a): It returns the result of  $\ln(a)$ .

**double** log10(**double** a): It returns the result of  $\log_{10}(a)$ .

**double** modf(**double** a, **double** \*ptr): Splits a into integer and fractional parts. It returns the fractional part.

**double** pow(**double** a, **double** b): It returns the result of  $a^b$ .

**double** sin(**double** a): It returns the sine of a.

**double** sinh(**double** a): It returns the hyperbolic sine of a.

**double** sqrt(**double** a): It returns the square root of a.

**double** tan(**double** a): It returns the tangent of a.

**double** tanh(**double** a): It returns the hyperbolic tangent of a.

## <setjmp.h>

**int** setjmp(**jmp\_buf** env): Stores the current stack environment in env.

**void** longjmp(**jmp\_buf** env, **int** val): Restores the environment stored in env.

## <signal.h>

**int** raise(**int** sig): Sends the sig signal to the program. It returns 0 if successful, a nonzero value otherwise.

**void** (\*signal(**int** sig, **void** (\*func)(**int**)))(**int**): It takes as parameters an integer and a pointer to another function, which takes an integer parameter and

returns nothing. It returns a pointer to a function, which takes an integer parameter and returns nothing. `signal()` establishes the function that `func` points to as the handler of the `sig` signal.

`<stdarg.h>`

`void va_start(va_list arg_ptr, type)`: Initializes `arg_ptr` to point to the beginning of the variable argument list.

`type va_arg(va_list arg_ptr, type)`: Gets the value of the argument indicated by `arg_ptr`.

`void va_end(va_list arg_ptr)`: Terminates the processing of the variable argument list.

`<stdio.h>`

`void clearerr(FILE *fp)`: Clears the error indicator of the file pointed to by `fp`.

`int fclose(FILE *fp)`: Closes the file pointed to by `fp`. It returns 0 if successful, EOF otherwise.

`int feof(FILE *fp)`: It returns a nonzero value if the end of file pointed to by `fp` is reached, 0 otherwise.

`int ferror(FILE *fp)`: It returns a nonzero value if an error in the file pointed to by `fp` occurred, 0 otherwise.

`int fflush(FILE *fp)`: Writes any unwritten data in the output file pointed to by `fp`. If it is an input file, the behavior is undefined. It returns 0 if successful, EOF otherwise.

`int fgetc(FILE *fp)`: Reads a character from the file pointed to by `fp` as `unsigned char` cast to `int`. It returns the character read if successful, EOF if the end of file is reached or a read error occurs.

`int fgetpos(FILE *fp, fpos_t* pos)`: Stores the current position of the file pointed to by `fp` into a `fpos_t` variable pointed to by `pos`. It returns 0 if successful, a nonzero value otherwise.

`char *fgets(char *str, int n, FILE *fp)`: Reads characters from the file pointed to by `fp` until the new line character is met or `n-1` characters are read or the end of file is reached, whichever comes first. The characters are written into the array pointed to by `str`. It returns `str` or `NULL` if the end of file is reached or a read error occurs.

`FILE *fopen(const char *filename, const char *mode)`: Opens the file specified by `filename` according to the `mode` argument. It returns a pointer associated with the file or `NULL` if the file can't be opened.

`int fprintf(FILE *fp, char *mode, ...)`: Writes data to the file pointed to by `fp` according to the format specified by `mode`. It returns the number of the characters written or a negative value if an error occurs.

`int fputc(int ch, FILE *fp)`: Writes the character `ch` to the file pointed to by `fp`. It returns the character written or EOF if an error occurs.

`int fputs(const char *str, FILE *fp)`: Writes the string pointed to by `str` in the file pointed to by `fp`. It returns a nonnegative value if successful, EOF otherwise.

**int** **fread**(**void** \*buf, **int** size, **int** count, **FILE** \*fp): Reads from the file pointed to by fp, count elements of size bytes each, and stores them into the array pointed to by buf. It returns the number of the elements successfully read.

**FILE** \***freopen**(**const char** \*filename, **const char** \*mode, **FILE** \*fp): Closes the file pointed to by fp and opens a new file specified by filename according to the mode argument. It returns a pointer associated with the new file or NULL if the file can't be opened.

**int** **fscanf**(**FILE** \*fp, **const char** \*mode, ...): Reads data from the file pointed to by fp according to the format specified by mode. It returns the number of the data items successfully read and stored, EOF if end of file is reached or a read error occurs.

**int** **fseek**(**FILE** \*fp, **long** offset, **int** origin): Moves the file pointer indicated by fp to a new position that is offset bytes from the position specified by origin. It returns 0 if successful, a nonzero value otherwise.

**int** **fsetpos**(**FILE** \*fp, **const** **fpos\_t** \*pos): Moves the file pointer indicated by fp to the position specified by the pos parameter. It returns 0 if successful, a nonzero value otherwise.

**long** **ftell**(**FILE** \*fp): It returns the current position of the file pointer indicated by fp from the beginning of the file.

**int** **fwrite**(**const void** \*buf, **int** size, **int** count, **FILE** \*fp): Writes count elements of size bytes each from the array pointed to by buf into the file pointed to by fp. It returns the number of elements written.

**int** **getc**(**FILE** \*fp): Similar to **fgetc**().

**int** **getchar**(): Reads a character from **stdin**. It returns the character read if successful, EOF if the end of file is reached or a read error occurs.

**char** \***gets**(**char** \*str): Reads characters from **stdin** and stores them into the array pointed to by str. It replaces the new line character with the null character. It returns str or NULL if the end of file is reached or a read error occurs.

**void** **perror**(**const char** \*str): Displays a diagnostic error message to the **stderr** stream.

**int** **printf**(**const char** \*mode, ...): Similar to **fprintf**(), except that the data are written to **stdout**.

**int** **putc**(**int** ch, **FILE** \*fp): Similar to **fputc**().

**int** **putchar**(**int** ch): Writes the ch character to **stdout**. It returns the character written if successful, EOF otherwise.

**int** **puts**(**const char** \*str): Writes the string pointed to by str and a new line character to **stdout**. It returns a nonnegative value if successful, EOF otherwise.

**int** **remove**(**const char** \*filename): Deletes the file specified by filename. It returns 0 if successful, a nonzero value otherwise.

**int** **rename**(**const char** \*oldname, **const char** \*newname): Renames the file specified by oldname to the name pointed to by newname. It returns 0 if successful, a nonzero value otherwise.

**void** rewind(FILE \*fp): Sets the file pointer indicated by fp to the beginning of the file.

**int** scanf(**const char** \*mode, ...): Similar to fscanf(), except that the data are read from stdin.

**void** setbuf(FILE \*fp, **char** \*ptr): The memory pointed to by ptr will be used as intermediate buffer to store the data before being written in the file pointed to by fp.

**int** setvbuf(FILE \*fp, **char** \*ptr, **int** mode, **size\_t** size): Similar to setbuf(), except that the memory size is specified by size.

**int** sprintf(**char** \*ptr, **const char** \*mode, ...): Similar to fprintf(), except that the data are written into the array pointed to by ptr. It appends the null character.

**int** sscanf(**const char** \*ptr, **const char** \*mode, ...): Similar to fscanf(), except that the data are read from the memory pointed to by ptr.

FILE \*tmpfile(): Creates a temporary file that will be automatically deleted when the file is closed or the program terminates. It returns a file pointer associated with the file or NULL if the file can't be created.

**char** \*tmpnam(**char** \*str): Creates a string that is not the name of an existing file. It returns a pointer to the created name.

**int** ungetc(**int** ch, FILE \*fp): Pushes the character ch back to the file pointed to by fp. It returns ch if successful, EOF otherwise.

**int** vfprintf(FILE \*fp, **const char** \*mode, va\_list arg): Similar to fprintf(), except that a variable argument list indicated by arg will be written into the file.

**int** vprintf(**const char** \*mode, va\_list arg): Similar to vfprintf(), except that the data are displayed to stdout.

**int** vsprintf(**char** \*ptr, **const char** \*mode, va\_list arg): Similar to vfprintf(), except that the data are written into the memory pointed to by ptr.

## <stdlib.h>

**void** abort(): Creates abnormal program termination.

**int** atexit(**void** (\*ptr)()): Registers the function pointed to by ptr to be called if the program terminates normally. It returns 0 if successful, a nonzero value otherwise.

**double** atof(**const char** \*str): Converts the string pointed to by str to a floating-point number. It returns the number if successful, otherwise the behavior is undefined.

**int** atoi(**const char** \*str): Similar to atof() except that the string is converted to **int**.

**long** atol(**const char** \*str): Similar to atof() except that the string is converted to **long**.

**void \*bsearch(const void \*key, const void \*base, size\_t num, size\_t width, int (\*cmp)(const void \*elem1, const void \*elem2)):** Searches for the value pointed to by key in a sorted array pointed to by base, which has num elements, each of width bytes. cmp is a pointer to a function that takes as parameters two pointers and returns a value as follows:

<0	if *elem1 < *elem2
=0	if *elem1 = *elem2
>0	if *elem1 > *elem2

If the \*key value is found, bsearch() returns a pointer to the respective array element. Otherwise, it returns NULL.

**void \*calloc(size\_t num, size\_t size):** Allocates memory for an array of num elements, each of size bytes. The memory is initialized to 0. It returns a pointer to the beginning of the memory block or NULL if the memory can't be allocated.

**div\_t div(int a, int b):** It calculates the quotient and the remainder of a/b and stores them in the respective members of the returned div\_t structure.

**void exit(int status):** Causes program termination. The value of status indicates the termination status. The value 0 indicates successful termination.

**void free(void \*ptr):** Releases the memory pointed to by ptr.

**char \*getenv(const char \*name):** Checks if the string pointed to by name is contained in the system's environment list. If a match is found, it returns a pointer to the environment entry associated with the name, NULL otherwise.

**div\_t ldiv(long int a, long int b):** Similar to div(), except that the results are stored into the members of a ldiv\_t structure.

**void \*malloc(size\_t size):** Allocates a memory block of size bytes. It returns a pointer to the beginning of that memory or NULL if the memory can't be allocated.

**int mblen(const char \*str, int count):** It determines the validity of a multi-byte character pointed to by str and returns its length.

**int mbtowc(wchar\_t \*wchar, const char \*mbchar, size\_t count):** Converts count bytes of the multibyte character pointed to by mbchar into the code of a wide character and stores the result into the memory pointed to by wchar. It returns the length of the multibyte character.

**int mbstowcs(wchar\_t \*wchar, const char \*mbchar, size\_t count):** Converts count bytes of the multibyte character pointed to by mbchar into a string of wide character codes and stores the result into the array pointed to by wchar. If the conversion is successful, it returns the number of the converted multibyte characters, -1 otherwise.

**void \*qsort(void \*base, size\_t num, size\_t width, int (\*cmp)(const void \*elem1, const void \*elem2)):** Sorts the array pointed to by base. The parameters have the same meaning as in bsearch().

**int rand():** Generates and returns a random integer between 0 and RAND\_MAX. RAND\_MAX is defined in stdlib.h.

**void** \*realloc(**void** \*ptr, **size\_t** size): ptr points to a memory previously allocated with malloc(), realloc(), or calloc(). realloc() allocates a memory block of size bytes and copies the contents of the old memory if necessary. It returns a pointer to the beginning of the new memory or NULL if the memory can't be allocated.

**void** srand(**unsigned int** seed): Uses seed to initialize the sequence of the random numbers produced by rand(). Typically, the value of seed is the return value of time(), so that rand() generates different numbers each time the program runs.

**double** strtod(**const char** \*str, **char** \*\*endp): Converts the string pointed to by str to a floating-point number. It returns the number if conversion is successful, 0 otherwise. endp points to the first character that cannot be converted.

**long** strtol(**const char** \*str, **char** \*\*endp, **int** base): Converts the string pointed to by str to a long integer. base defines the radix of the number. It returns the number if conversion is successful, 0 otherwise. endp points to the first character that cannot be converted.

**unsigned long** strtoul(**const char** \*str, **char** \*\*endp, **int** base): Similar to strtol, except that it returns an unsigned long integer.

**int** system(**const char** \*str): Executes the operating system's command pointed to by str.

**int** wctomb(**char** \*mbchar, **wchar\_t** wchar): Converts the wchar wide character into the corresponding multibyte character and the result is stored into the memory pointed to by mbchar. It returns the number of the wchar bytes if conversion is successful, -1 otherwise.

**int** wcstombs(**char** \*mbchar, **const wchar\_t** \*wcstr, **size\_t** count): Similar to wcotmb(), except that a sequence of wide characters is converted into the corresponding multibyte characters.

## <string.h>

**void** \*memchr(**const void** \*str, **int** val, **size\_t** count): Searches for the val character into the first count characters of the string pointed to by str. It returns a pointer to its first occurrence if found, NULL otherwise.

**int** memcmp(**const void** \*ptr1, **void** \*ptr2, **size\_t** count): It returns a negative, zero, or positive value, depending on whether the first count bytes of the memory block pointed to by ptr1 are less than, equal to, or greater than the count bytes of the memory block pointed to by ptr2.

**void** \*memcpy(**void** \*dest, **const void** \*src, **size\_t** count): Copies count bytes from the memory pointed to by src to the memory pointed to by dest. If the memory blocks overlap the behavior is undefined. It returns the dest pointer.

**void** \*memmove(**void** \*dest, **const void** \*src, **size\_t** count): Similar to memcpy(), except that the copy operation will work properly even if the two memory blocks overlap. It returns the dest pointer.

**void** \*memset(**void** \*dest, **int** val, **size\_t** count): Stores val into the count bytes of the memory pointed to by dest. It returns the dest pointer.

`char *strcat(char *str1, const char *str2)`: Appends the string pointed to by str2 at the end of the string pointed to by str1 and adds the null character. It returns str1 which points to the concatenated string.

`char *strchr(const char *str, int val)`: Searches for the val character into the string pointed to by str. It returns a pointer to its first occurrence if found, NULL otherwise.

`int strcmp(const char *str1, const char *str2)`: Compares the strings pointed to by str1 and str2 and returns a negative, zero, or positive value depending on whether the first string is lexicographically less than, equal to, or greater than the second one.

`int strcoll(const char *str1, const char *ptr2)`: Compares the strings pointed to by str1 and str2 according to the rules of the current locale and returns the result of the comparison.

`char *strcpy(const char *dest, const char *src)`: Copies the string pointed to by src into the array pointed to by dest. It returns the dest pointer.

`size_t strcspn(const char *str, const char *set)`: It returns the length of the longest part of the string pointed to by str that does not contain any character of the string pointed to by set.

`char *strerror(int err)`: It returns a pointer to the string containing an error message corresponding to err.

`size_t strlen(const char *str)`: It returns the length of the string pointed to by str, not including the null character.

`char *strncat(char *str1, const char *str2, size_t count)`: Appends count characters from the string pointed to by str2 at the end of the string pointed to by str1 and adds the null character. It returns str1 which points to the concatenated string.

`int strncmp(const char *str1, const char *str2, size_t count)`: Compares count characters of the strings pointed to by str1 and str2 and returns the result of the comparison.

`char *strncpy(char* dest, const char *src, size_t count)`: Copies count characters of the string pointed to by src to the string pointed to by dest. It returns the dest pointer.

`char *struprbrk(const char *str, const char *set)`: It returns a pointer to the first character in the string pointed to by str that is contained in the string pointed to by set. It returns NULL if no match is found.

`char *strrchr(const char *str, int val)`: It returns a pointer to the last occurrence of val in the string pointed to by str. It returns NULL if it is not contained.

`size_t strspn(const char *str, const char *set)`: It returns the length of the longest part of the string pointed to by str that consists entirely of characters of the string pointed to by set.

`char *strstr(const char str1, const char *str2)`: Checks if the string pointed to by str2 is contained in the string pointed to by str1. If it is, it returns a pointer to its first occurrence in str1, NULL otherwise.

`char *strtok(const char *str, const char *set)`: Searches the string pointed to by str for a segment consisting of characters not in the string pointed to by set. If it is found, it returns a pointer to the first character of the segment, NULL otherwise.

`size_t strxfrm(char *dest, const char *src, size_t count)`: Transforms count characters of the string pointed to by src using the current locale's settings and stores the result into the array pointed to by dest. It returns the length of the transformed string.

`<time.h>`

`char *asctime(const struct tm *ptr)`: Converts the date and time information stored in a structure of type `tm` pointed to by `ptr` to a string of the form "Sun Nov 13 11:13:58 2016". It returns a pointer to the constructed string.

`clock_t clock(void)`: It returns the elapsed time since the beginning of program execution, measured in processor clock ticks. To convert it to seconds, divide that value by the constant `CLOCKS_PER_SEC`.

`char *ctime(const time_t *ptr)`: Converts the calendar time stored in a structure of type `time_t` pointed to by `ptr` to a string of the form "Sun Nov 13 11:13:58 2016", adjusted to the local time zone settings. It returns a pointer to the constructed string.

`double difftime(time_t t1, time_t t0)`: It returns the time difference `t1-t0`, measured in seconds.

`struct tm *gmtime(const time_t *ptr)`: Converts the calendar time pointed to by `ptr` into a universal time and returns a pointer to a structure of type `tm` which holds the results.

`struct tm *localtime(const time_t *ptr)`: Converts the calendar time pointed to by `ptr` into local time and returns a pointer to a structure of type `tm` which holds the results.

`time_t mktime(struct tm *ptr)`: Converts the local time pointed to by `ptr` into a calendar time and returns a pointer to a structure of type `time_t` which holds the results.

`size_t strftime(char *str, size_t size, const char *fmt, const struct tm *ptr)`: Formats the time information pointed to by `ptr` according to the format string pointed to by `fmt` and stores `size` characters in the array pointed to by `str`. It returns the number of the stored characters.

`time_t time(time_t *ptr)`: Returns the current calendar time.

## Appendix 4: Hexadecimal System

This appendix provides a brief description of the hexadecimal system. The base of the hexadecimal system is the number 16. The numbers 0-9 are the same of the decimal system. The numbers 10-15 are represented by the letters A to F. The correspondence of the hexadecimal system to the binary and decimal systems is depicted in Table A.4.1.

As shown, each hexadecimal digit (0 to F) is represented with four binary digits. For example, the hexadecimal number F4A is written in binary as 1111 0100 1010.

To find the decimal value of a hexadecimal number that consists of n digits (e.g.,  $d_{n-1} \dots d_2 d_1 d_0$ ), we apply the following formula:

$$\text{Decimal} = \sum_{i=0}^{n-1} (d_i \times 16^i).$$

For example, the decimal value of F4A is  $(A \times 16^0) + (4 \times 16^1) + (F \times 16^2) = 10 + 64 + 3840 = 3914$ .

**TABLE A.4.1**

Hex, Binary, and Decimal Systems

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

# *Index*

Page numbers followed by f and t indicate figures and tables, respectively.

- , (comma) operator, 56–57
    - in for statemen, 106
  - ?:(conditional) operator, 88–89
  - operator, 62
    - precedence, 64
- A**
- abstract interface, Java, 626, 627, 628
  - abstract modifier, Java, 617, 619–620
  - Accessibility
    - class data in C++, 581
    - one-dimensional arrays, elements, 152–154
  - accessiblity\_type, 590
  - Access modifiers (A\_M), Java, 612, 616–620,
    - 617
    - private, 617
    - protected, 617
    - public, 616–617
  - add () function, 606
  - add\_list\_end() function, 487
  - add\_node() function, 491
  - Address operator &, 35, 189
  - & address operator, 35, 189
  - add\_sort() function, 389–390
  - add\_stack() function, 477–478, 487
  - ALERTING message structure, unions, 442f
    - ISDN network, 443–445
  - Algorithms
    - binary search, 383–386
    - bubble sort, 395–398
    - insertion sort, 392–394, 393f
    - linear search, 381–383
    - quick sort, 398–400
    - selection sort, 386–392
  - American National Standard Institute (ANSI), 1
  - American Standard Code for Information Interchange (ASCII) code, 12, 15, 239, 253, 258, 284
    - character set, 242, 665, 666t
    - in files, 507
    - upper and lowercase letters and their values, 242
  - ANSI C (Standard C), 1
  - ANSI/ISO C standard, 1
  - a.out, 8, 577
  - argc (argument count) parameter, 364, 365, 612
  - args[] parameter, 612
  - Argument
    - array, 330
    - conversion, 330
    - defined, 307
    - in main (), 364
    - one-dimensional arrays as, 330–336
    - pointer, 308
    - structure as, 424–426
    - two-dimensional array as, 358–364
    - using const, 332
    - variable-length list, 367
    - vs. parameters, 307
  - argv (argument vector) parameter, 364, 365, 612
  - Arithmetic, pointer, 199–201
  - Arithmetic conversions, 15–18
    - explicit conversion, 17–18
    - implicit conversion, 16–17
  - Arithmetic operators, 46–47
  - Arithmetic overflow, 46
  - ARP packet structure, MAC, 355–357, 356f
  - ARP\_Reply, 355
  - ARP\_Request, 355
  - Array of pointers, 216–218
    - to functions, 231–235
  - Array(s), 151–184; *see also* Structures; Unions; Variable(s)
    - accessing, 152
    - address of, 204, 206, 303
    - as an argument, 330
    - common error, 153
    - const qualifier, 155
    - copying, 153
    - defined, 151
    - delete key from, 500–501
    - of five integers (example), 153–154
    - length of, 151, 154
    - memory size, 152
    - multidimensional, 171

one-dimensional  
  data\_type, 151–152  
  declaration, 151–152, 155–158  
  elements, accessing, 152–154  
  error, 164  
  example, 152  
  exercises, 155–170  
  frequency array (*freq*), 159–160  
  100 integers, program, 166–167  
  initialization, 154–155  
  shift operator, using, 164–165  
  10 integers, program, 162–163, 167–168  
  20 integers, program, 160–161  
overview, 151  
and pointers, relationship between, 153,  
  203–207, 213, 292  
of pointers, 216–218, 359  
  to functions, 231–235  
searching, 381–386  
  binary search algorithm, 383–386  
  linear search algorithm, 381–383  
sorting, 383, 386–400  
  bubble sort algorithm, 395–398  
  insertion sort algorithm, 392–394, 393f  
  quick sort algorithm, 398–400  
  selection sort algorithm, 386–392  
of strings, 292, 292f  
to structure, 420–422  
structure containing, 414  
two-dimensional, 171f  
  declaration, 170  
  elements, accessing, 171–172  
  exercises, 174–184  
  initialization, 172–173  
values, 154, 156  
  vs. linked list, 471  
ASCII set, *see* American Standard Code for  
Information Interchange (ASCII)  
code  
asm, 10  
assert() function, 553  
assert.h, 667  
Assignment operator, 13–14  
Assignment (=) operators, 45–46, 50, 74  
  wrong use of, 81  
Associativity, operators, 70  
atoi() function, 634  
AT&T Bell Laboratories, 579  
auto class, 321  
  local variables, 323  
auto int, 323  
Automatic storage class, of variables, 321

Automatic variables, 13  
  local variable as, 323  
  register class, 323  
  *vs.* global variable, 325  
avg() function, 304

## B

B, 1  
Bets, program for, 641–645  
Binary file(s), 507–508, 508f  
  content, 508f  
  data storing in, 507  
  disadvantages, 507–508  
  end of, 530  
  opening, 509  
  reading data from, 532  
    exercises, 534–544  
    fread() function, 533–534  
size, 507, 508f  
special character, 507  
test.bin binary file, 536–537, 540–542  
  *vs.* text file, 507  
writing data in, 532  
  exercises, 534–544  
  fwrite() function, 532–533  
Binary operators, 47  
  arithmetic conversions, 15  
Binary search algorithm, 383–386  
  exercises, 384–386  
Binary search tree, 488–495, 489f  
  add\_node() function, 491  
  advantage, 488  
  delete\_tree() function, 493  
  exercises, 495–496  
  implementation examples, 489–495  
  recursive function addition, 495  
  show\_inorder() function, 493  
  show\_preorder() function, 493  
Binary tree, 488  
Bit fields, structure, 417–419  
  advantage, 417–418  
  register layout, 418, 418f  
  unnamed, 418  
Bitwise operators, 60–69  
  , 62  
  &, 61  
  ^, 61  
  |, 61  
  in data coding in network communications,  
    66–67, 66f  
  exercises, 63–69

hardware communication, 6f, 68–69  
if statement, 64, 65  
shift, 62–63  
Blocks, memory, 449–450  
Block scope, 322, 323, 328  
Book library management application, program  
    for, 655–658  
Boolean AND operation, 61  
Boolean NOT operation, 62  
Boolean OR operation, 61  
Boolean XOR operation, 61  
braces, 4  
break statement, 111–113, 144  
    default case, 93, 112, 113  
    switch statement, 93, 94–95, 111–113  
bssearch() library function, 400–402  
    exercise, 401–402  
Bubble sort algorithm, 395–398  
    bubble\_sort() function, 395  
    exercises, 395–398  
bubble\_sort() function, 395  
Bug, 7  
*Build* command, 7, 577  
Building large projects, *see* Large programs  
Byte, 9  
Bytecode, 611

**C**

C; *see also* C++  
    advantages, 2  
    ANSI standard, 1  
    disadvantages, 2  
    flexibility, 2  
    history, 1  
    program, *see* Program  
    *vs.* C++, 579  
    *vs.* Java, 612–613

C++, 4, 300, 579–609  
    call by reference, 604  
    catch block, 606, 607  
    classes, 579–582  
    constructors, 582–584  
    default arguments, 605  
    delete operator, 602–603  
    destructors, 582–584  
    dynamic binding, 594  
    encapsulation, concept of, 579–580  
    exceptions, 606–609  
    function templates, 596–599  
    inheritance, 589–594  
    main() function, 579

new operator, 602–603  
objects, 579–582  
overloading of functions and operators,  
    584–589  
overview, 579  
polymorphism, 594–596  
private members, 581  
program overview, 579  
protected members, 581  
public members, 581  
reference variables, 604–606  
source file extension, 579  
standard template library, 600–601  
static binding, 584, 594  
templates, 596–600  
    class, 599–600  
    function, 596–599  
try, 606  
    *vs.* C, 579  
    *vs.* Java, 612613  
virtual, 595  
    virtual functions, 594  
Caesar, Julius, 646  
Caesar algorithm, 646–649  
Call, function, 305  
    with parameters, 307–311  
    printf() function, 307  
    test() function, 305–307, 308–311, 315  
    without parameters, 305–307  
Callback mechanism, 571  
Call by reference, 604  
Call by value, 604  
Called (end) user, 571  
calloc() function, 452  
Call Reference Value (CRV), 497  
call\_sim, 578  
Case-sensitive language, C as, 7, 9  
case statements  
    switch statement, 94–95  
Cast operator, 16  
    explicit arithmetic conversion, 17–18  
    implicit arithmetic conversion, 16–17  
catch block, 606, 607  
C data types, 10–13, 11t  
char\*, 258, 367  
Character constant, 15  
Character escapes, 20  
Characters, 239–251  
    ASCII character set, 242  
    char type, 239–242  
    conversion specifiers, 22  
    constants, 15

escapes, 20  
exercises, 242–245, 246–251  
getchar() function, 245–246  
if-else-if statements, 244–245  
for loop, 244–245  
null, 253, 254  
putchar() function, 245–246  
tigraph sequence, 241  
writing, 239

char array, 204  
char bit field, 417  
char data type, 11–12, 11t, 239, 299  
    function call, 308  
char type, characters, 239–242  
char variable, 246  
check digit, 249–251  
check() function, 584  
Cipher algorithms, 646–649  
Class(es), 578  
    C++, 579–582  
        accessibility, 581  
        constructor, 582–584  
        destructor, 582–584  
        form, 580  
        members of, 581  
    Java, 611–614  
        object or instance of, 580  
Class templates, C++, 599–600  
Closing, file, 510–511  
    fclose() function, 510, 511  
Code segment, memory, 449  
Collatz, Lothar, 375  
“Collatz conjecture,” 375–376  
Command line, 6, 8, 364, 365, 631, 634–635  
Comma (,) operator, 56–57  
    in for statement, 106  
Comments, 5–6  
Common errors, 7  
    arrays, 153  
    if statement, 74–75  
    strings, 254  
Communication  
    hardware, bitwise operators in, 68–69, 68f  
    network, bitwise operators in, 66–67, 66f  
Communication protocols, unions and, 442  
Compilation, program, 6–7  
    common errors, 7  
Compilers, 6  
Compile time (static binding), 584, 594  
Complex declaration, function, 303  
Compound assignment operators, 50–51  
Compound operators, 70

Compound statement, variables declarations, 323  
Conditional compilation, preprocessor  
    directives and, 557–562; *see also*  
    Directives, preprocessor  
Conditional expression  
    exercises, 90–92  
    if-else statement, 88–89, 90–92  
Conditional operator ?:, 88–89  
CONNECT\_ACK, 571  
CONNECT message, 571  
CONNECT message structure, unions, 442f  
    ISDN network, 443–445

Constants  
    character, 15  
    decimal, 14  
    defined, 18  
    enum type, 59–60  
    floating, 15  
    hexadecimal, 14  
    octal, 14  
    variables, assignment of, 13–15  
const qualifier, 155  
    use of, 198  
Constructor overloading, 623, 624  
Constructors  
    C++, 582–584  
        default, 583–584  
        nondefault, 583–584  
    Java, 614–616  
const variable, 18, 20, 329  
continue statement, 113–114  
    default case, 114  
    for loop, 113, 130  
    printf(), 113  
    switch statement, 113–114  
Conversion specifications, printf() function, 22–23, 23t, 25, 25f  
    field width, 27  
    flags, 28, 28t  
    optional fields, 25–28  
    precision, 25–26  
    prefix, 27  
Conversion specifiers, 22t, 23–23  
CRV (Call Reference Value), 497  
C11 standard, 1  
C99 standard, 1  
Ctrl+Z character, 507, 530  
ctype.h, 667

## D

Data coding  
    bitwise operators in, 66–67, 66f

Data encryption  
  ^ operator in, 63–64

Data input, 3

Data output, 3

Data segment, memory, 449

Data structures; *see also* Memory management  
  binary search tree, 488–495, 488–495, 489f, 489f  
  add\_node() function, 491  
  advantage, 488  
  delete\_tree() function, 493  
  exercises, 495–496  
  implementation examples, 489–495  
  recursive function addition, 495  
  show\_inorder() function, 493  
  show\_preorder() function, 493

dynamic, 471

implementation examples, 473–488  
  linked list, 481–488  
  queue, 478–480  
  stack, 473–478

linked list, 471–473, 471f–473f  
  static, 471  
  tree, 488

data\_type, 10  
  one-dimensional array, 151  
  pointers, 187–188  
  two-dimensional arrays, 170

\_DATE\_, 552

Debugger, 20

Decimal constant, 14

Declaration  
  array of pointers, 216  
  function, 299–300  
    complex, 303  
  global variable, 325–327  
  one-dimensional arrays, 151–152, 155–158  
  pointers, 187–188  
  pointer to pointer, 219  
  static variable, 328–330  
  structures, 405–408  
  two-dimensional arrays, 170  
  unions, 437–438  
  variables, 10–13

Decrement (--) operators, 47–49

default case, switch statement, 93–94, 112

Default constructor, 583–584

#define directive, 19–20, 59, 60, 334, 358, 549

defined operator, 561–562

delete[], 602, 603

delete operator, C++, 602–603

delete\_tree() function, 493

del\_node() function, 500–501

Destructors, C++, 582–584

Directives, preprocessor, 557–562  
  defined operator, 561–562  
  #elif, 558–560  
  #else, 558–560  
  #endif, 558–560  
  #if, 558–560  
  #ifndef, 559, 560–561  
  #endif, 559, 560–561  
  #define, 560–561

Dos/Windows applications, 530

double data type, 11, 11t, 12, 13, 239  
  function call, 308  
  two-dimensional arrays, 170

double poly(double a[], double x, int n), 349

double variable  
  scanf() function, 36

do-while loop  
  break statement, 111  
  continue statement, 113  
  nested, 124

do-while statement, 140–143; *see also* while statement

Dynamic binding, 594

Dynamic data structures, 471

Dynamic memory allocation, 451–458  
  exercises, 460–471  
  free() function, 455–458  
  linked list, 471  
  malloc() function, 452–455  
  queue, 478  
  size, 451–452  
  size of operator, 453  
  stack, 473

Dynamic memory deallocation, 455

Dynamic polymorphism, Java, 623

## E

#elif directive, 558–560, 561

Ellipsis, 366

#else directive, 558–560

else if, 78, 79

else statement, 76, 77–78; *see also* if-else statement; Nested if statements

E-mail application, 580

Encapsulation  
  in C++, 579–580  
  concept of, 579

Encryption key, 658–661

#endif directive, 558–560

End of File (EOF), 245, 263, 518

Enum type, 59–60

EOF (End of File), 245, 263, 518, 530–531  
EOF. `getc()` function, 526  
Equality (`==`) operator, 12, 49, 50, 74  
`#error` directive, 562  
Errors  
    common, 7  
        arrays, 153  
        if statement, 74–75  
        strings, 254  
    spelling, 7  
    syntactic, 7  
Escape sequences, `printf()` function, 20–22, 21t  
    character escapes, 20  
    numeric escapes, 20  
Euclid's algorithm, 346–347  
Exceptions, C++, 606–609  
.exe, 8  
EXIT\_FAILURE macros, 263  
`exit()` function  
    read strings, 263–264  
    *vs.* return statement, 263–264  
EXIT\_SUCCESS macros, 263  
Explicit arithmetic conversion, 17–18  
External linkage, of variables, 321  
External variables, 572  
extern storage class, 321, 573  
    global variable, 327

## F

`fabs()` function, 13  
`far`, 10  
`fclose()` function, 510, 511  
`feof()` function, 518, 544–545  
`fflush()` function, 38, 509, 510  
`fgetc()` function  
    exercises, 526–530  
    reading from text file, 526–530  
`fgets()` function; *see also* `gets()` function  
    exercise, 525–526  
    reading from text file, 523–526  
    read strings, 262, 263, 264  
    strings writing, 257  
Fibonacci sequence, Nth term of, 352–353  
Field width, conversion specifier, `printf()` function, 27  
FIFO (First In First Out) queue, 478, 494  
\_\_\_\_FILE\_\_\_\_, 552  
Files, 507–545  
    binary, 507–508, 508f  
    closing, 510–511  
    end of file, 530–531  
exercises, 513–514, 515–518, 521–522, 525–530, 534–544, 545  
`feof()` function, 544–545  
flushing, 510  
`fseek()` function, 531–532  
`ftell()` function, 531–532  
name, selection for large programs, 572  
opening, 508–510, 508t  
overview, 507  
processing, 511  
reading data from binary file, 532  
    exercises, 534–544  
`fread()` function, 533–534  
reading data from text file, 518  
    `fgetc()` function, 526–530  
    `fgets()` function, 523–526  
    `fscanf()` function, 518–523  
text, 507–508, 508f  
writing data in binary file, 532  
    exercises, 534–544  
`fwrite()` function, 532–533  
writing data in text file, 511  
    `fprintf()` function, 512–514  
    `fputc()` function, 514–518  
    `fputs()` function, 511–512  
File scope, 325  
FILE structure, 509  
final interface, Java, 626  
final modifier, Java, 617, 618  
`find_max()` function, 368–369  
Finite state machines (FSMs), 233  
    implementation, 233–234  
Firewall, 638  
First In First Out (FIFO) queue, 478, 494  
Fixed parameters, 367–368  
flag function, 329  
Flags, conversion specifier, `printf()` function, 28, 28t  
float data type, 11, 11t, 12–13, 239, 299  
    function call, 308  
Floating constant, 15  
Floating-point value, 13, 15  
Floating-point variables, 13, 25  
float variable, `scanf()` function, 36  
`fopen()` function, 508, 509, 510, 511  
for loop  
    break statement, 111  
    characters, 244–245  
    continue statement, 113, 130  
    insertion sort algorithm, 393–394  
    iterations, 645  
Format string, 20, 22, 24, 35–36

for statement, 105–110  
example, 107–108  
exercises, 109–110  
expressions, omitting, 108  
iterations, 107, 110, 114, 116  
nested loops, 108  
pointers and, 226  
printf(), 107, 109–110  
rand() function, 107, 108  
srand() function, 107, 108  
time() function, 107, 108  
vs. if statement, 105–106  
while statement and, 108, 129, 130

fprintf() function, 511; *see also* printf()  
function  
writing in text file, 512–514

fputc() function  
exercises, 515–518  
writing in text file, 514–518

fputs() function  
writing in text file, 511–512

fread() function  
exercises, 534–544  
reading from binary file, 533–534

free() function  
dynamic memory allocation, 455–458

free\_queue() function, 480

free\_stack() function, 480

Frequency array (freq), 159–160

fscanf() function; *see also* scanf() function  
exercises, 521–523  
reading from text file, 518–523  
return value, 519  
while statement, 519

fseek() function, 511, 531–532

fsm.c file, 572, 573, 574, 576–577

fsm.h file, 574

FSMs, *see* Finite state machines (FSMs)

ftell() function, 531–532

Function call, 305  
with parameters, 307–311  
printf() function, 307  
test() function, 305–307, 308–311, 315  
without parameters, 305–307

Function pointer, 228–231

Function prototype, 299

Function(s), 299–376; *see also* specific functions  
argument, structure as, 424–426  
arguments in main(), 364  
array as argument, 330  
array of pointers to, 231–235  
body, 301

call, 305–311  
callback example, 230  
call by reference, 309, 604  
call by value, 309  
with parameters, 307–311  
printf() function, 307  
test() function, 305–307, 308–311, 315  
without parameters, 305–307

complex declarations, 303

declaration, 299–300  
definition, 300–303  
example, 301  
exercises, 311–321, 336–358, 359–364,  
365–366, 368–369, 371–376

large program, 572

main() function, passing data in, 364–366

memory management, 458–460  
memcmp(), 458, 459–460  
memcpy(), 458  
memmove(), 458, 459

one-dimensional array as argument,  
330–336

overloading, in C++, 584–589

overview, 299

parameters, 300, 307; *see also* Parameters  
pointer to, 228–231

postfix operators, 303

prefix operator, 303

recursive, 369–376

return statement, 304–305

return value, 299–300

standard library, 3

storage classes and variables scope,  
321–330  
global variables, 325–327  
local variables, 322–325  
static variables, 327–330

strings, 270  
exercises, 270–278, 280–282, 285–291  
strcat() function, 282–284  
strcmp() function, 284–285  
strcpy() function, 278–280  
strlen() function, 270  
strncmp() function, 284–285  
strncpy() function, 278–280

structure as argument, 424

two-dimensional array as argument,  
358–364

with variable number of parameters,  
366–369

warning message, 302

Function templates, C++, 596–599

## fwrite() function

- exercises, 534–544
- sizeof operator and, 533
- writing in binary file, 532–533

## G

Gambling games' software, 641–645

Garbage collector mechanism, Java, 614

gcc (GNU Compiler Collection), 6, 7

- large programs, 577

- running program using, 8

general.h., 572–573

getc() function, 526

getchar() function, 245–246, 526

- exercise, 247–248

- read strings, 262–263, 264, 267–268

getHomeClub(), 628

gets() function; *see also* fgets() function

- read strings, 262

Getters, Java, 628

getVisitorClub(), 628

Global variable, 325–327, 449

- extern storage class, 327

- in functions, 325

- linkage of, 329

- naming, 325–327

- scope rule, 327

- vs.* automatic variables, 325

Gosling, James, 611

goto statement, 144–145

Green Project, 611

Greentalk, *see* Java

Guilty functions, 325

## H

.h, 3

Hangman, word-guessing game, 275–276

Hardware communication

- bitwise operators, 6f, 68–69

Hashing, 497–500

- algorithm development for, 497

- defined, 497

- exercise, 500–501

- implementation example, 497–500

Header file, 3, 301, 303

- large programs, 572

- general.h., 572–573

- information in, 572–573

Heap segment, memory, 152, 449

- size of, 452

hello.exe, 364

Hexadecimal system, 677, 677t

Hexadecimel constants, 14

## I

IDE, *see* Integrated development environment (IDE)

#ifdef directive, 559, 560–561

#if directive, 558–560, 561

if-else statement, 75–76; *see also* switch statement

- conditional expression, 88–89, 90–92

- firewall, 638

- sizeof operator, 58–59

- vs.* switch statement, 96

#ifndef directive, 559, 560–561

- large programs, 572–573

if statement(s), 12, 13, 73–75; *see also* Switch statement

- bitwise operators, 64, 65

- break statement, 111–112

- common errors, 74–75

- continue statement, 113

- nested, 76–79

- null statement, 74

- vs.* for statement, 105–106

- vs.* switch statement, 96

- while* statement and, 129

Image-editing programs, 435–437

IMEI (International Mobile Equipment Identifier), 249–251

"Implementing Communication Protocols," 354–357

implements keyword, Java, 626

Implicit arithmetic conversion, 16–17

import keyword, Java, 614

#include directive, 3–4, 12, 299, 549

- function definition, 301

- structures declaration, 406

- using, 550

Increment (++) operator, 47–49

\* indirection operator, 70, 190

indirection operator \*, 70, 190

Infinite loop, 18, 108, 130

Inheritance

- C++, 589–594

- advantage*, 591

- Java, 620–623

- multiple*, 622

- single*, 622

Initialization

- array, 154

- array of structures, 420–421

constant, 14  
one-dimensional arrays, 154–155  
pointers, 189  
static variable, 328–329  
structures, 410–412  
two-dimensional array, 172–173  
variable, 13

Inner loop, iterations, 125, 126, 127, 133  
Input stream, 508  
Insertion sort algorithm, 392–394, 393f  
`insert_sort()` function, 392, 393–394  
`int` array, 204, 224  
`int` bit field, 417  
`int` data type, 11, 11t, 12, 239, 299, 300  
    function call, 308  
    `getchar()` function, 245  
    one-dimensional array, 151  
    `*ptr`, 187–188

Integers  
    pointers and, 199–200

Integrated development environment (IDE), 3, 6

Integrated Services Digital Network (ISDN)  
    network, 571, 573

interface keyword, Java, 626

Interfaces, Java, 626–629

Internal linkage, of variables, 321

International Mobile Equipment Identifier (IMEI), 249–251

International Obfuscated C Code Contest, 2

The International Obfuscated C Code Contest, 242

International Standards Organization (ISO), 1

Internet Engineering Task Force (IETF)  
    organization, 571

Internet Protocol (IP) network, 571  
    local network, 354–357; *see also* Medium Access Control (MAC)  
    subnets, 649–652

Internet Protocol (IP) version 4 address (*IPv4*), 248–249  
    firewall, 638  
    header format, 635f  
    TCP segment, 635–638, 635f–636f

`int main()`, 4, 5  
`int` variable, 246  
`iostream` file, 579  
`isdigit()` function, 240  
`isdn_msg`, 443  
ISDN protocol, 442  
`islower()`, 354

ISO, *see* International Standards Organization (ISO)

ISO/IEC 9899:1990, 1

`isspace()`, 354  
`isupper()`, 354

Iteration loop, 105

Iterations  
    exercises, 114, 116, 118, 126, 127, 132–133  
    inner loop, 125, 126, 127, 133  
    for loop, 645  
    outer loop, 125, 126, 127, 133  
    pointers, 215–216  
    for statement, 107, 110, 114, 116  
    `strlen()` function, 273–274

ITU-T E.161 international standard, 246–247

*ITU-T Q.931* signaling protocol, 571

## J

Java, 611–629  
    abstract interface, 626  
    abstract modifier, 619  
    access modifiers, 612, 616–620, 617t  
    advantages, 611  
    background, 611  
    classes, 611–614  
    constructor overloading, 623  
    constructors, 614–616  
    doc comments, 613  
    final, 617  
    garbage collector mechanism, 614  
    inheritance, 620–623  
    instances, 613  
    interfaces, 626–629  
    javac, 612  
    `main()` function, 612, 627  
    nonaccess modifiers, 612, 616–620  
    objects, 611–614  
    overview, 611  
    packages, 614  
    polymorphism, 623–626  
        dynamic, 623  
        method overloading, 624–625  
        method overriding, 625–626  
        static, 623  
    source files, 611  
    subpackages, 614  
    *vs.* C/C++, 612–613

.java, 611  
javac compiler, 612  
javac `MainClass.java`, 612  
Java Virtual Machine (JVM), 611  
JDK (javadoc) tool, 613  
JVM (Java Virtual Machine), 611

## K

Kernighan, Brian, 1  
*Keywords*, *see specific keywords*  
K&R book, 1  
K&R style, 74

## L

Large programs; *see also Program Build command*, 577  
callback mechanism, 571  
call\_sim, 578  
commands, 577  
development, 571–578  
file name, selection of, 572  
fsm.c, 572, 573, 574, 576–577  
function definitions and declarations, 572  
gcc compiler, 577  
header file, 572  
general.h., 572–573  
information in, 572–573  
#ifndef directive, 572–573  
implementation, 572–578  
loopback testing, 576  
makefile, 577–578  
make tool, 578  
modules, interactions between, 571  
network protocol, 571  
overview, 571–572  
prtcl.c, 572, 573  
rcv.c, 576  
rcv\_msg() function, 573  
recv.c, 572  
send.c, 572, 575  
source file, 572  
    module, 572  
switch statement, 574–576

Last In First Out (LIFO) stack, 473

Library functions, 5, 667–675; *see also specific functions*

    bsearch(), 400–402  
    exercise, 401–402  
    qsort(), 400–402

Life cycle, of C program, 3  
LIFO (Last In First Out) stack, 473

\_\_LINE\_\_, 552

Linear search algorithm, 381–383  
    exercises, 381–383

linear\_search() function, 381

#line directive, 562

Linkage, of variables, 321

Linked list, 471–473, 471f

    implementing, 473, 481–483  
    nodes, 471  
        deletion, 472, 473f  
        insertion, 472, 472f  
    vs. arrays, 471

Linking, program, 7–8

locale.h, 667

Local variable, 322–325

    as automatic variable, 323  
    memory, 324  
    program, 323–324  
    register class, 323  
    returning address of, 324  
    storage class, 323

Logical operators, 51–55

    exercises, 53–55  
    ! operator, 51, 52  
    && operator, 52  
    || operator, 53

long bit field, 417

long data type, 11, 11t, 12

long double data type, 11t, 12, 13

long double variable

    scanf() function, 36

LOOPBACK\_MODE macro, 576

Loopback testing, 576

Loops, 105–145

    break statement, 111–113  
    continue statement, 113–114  
    do-while statement, 140–143  
    exercises, 114–124  
    goto statement, 144–145  
    iterations, 105, 125, 126, 127, 133  
    nested, 124–129  
    overview, 105  
    for statement, 105–110  
    while statement, 129–140

Loop statements, 74

Luhn digit, 249–251

lvalue, 45

## M

MAC, *see Medium Access Control (MAC)*

macro\_name, 549

Macros

    defined, 550  
    #define directive, 19–20  
    exercises, 563–567  
    extending, 551  
    LOOPBACK\_MODE, 576  
    with parameters, 554–555

predefined, 552, 552t  
simple, 549–553

Mail class, 580

MainClass.class, 612

MainClass.java file, 611, 612, 629

main() function, 4–5, 299, 301, 302  
    C++, 579  
    Java, 612, 627  
    large program, 572  
    local variable, 322–325  
    passing data in, 364–365  
        exercises, 365–366  
    recursive, 371

Makefile  
    advantage, 577  
    concept, 577  
    example, 577–578

Make tool, 578

malloc() function, 46, 359  
    dynamic memory allocation, 452–455  
    exercises, 460–461

Master-mind game, program for, 652–655

math.h, 668

max() function, 342

Maximum common divisor (MCD), 346–347

Medium Access Control (MAC), 354–357, 638  
    ARP packet structure, 355–357, 356f  
    frame structure, 355, 355f

memcmp() function, 458, 459–460  
    exercises, 462–463  
    vs. strcmp() function, 458–459

memcpy(), 153

memcpy() function, 458  
    vs. strcpy() function, 458–459

memmove() function, 458, 459

Memory  
    blocks, 449–450  
    code segment, 449  
    data segment, 449  
    distribution model, 449–450  
    fixed, 471  
    heap segment, 449  
    layout, 188f  
    local variable, 324  
    pointers and, 187, 188f  
    stack segment, 449

Memory address, 187, 188f

Memory allocation; *see also* Memory management  
    command line arguments and, 631, 634–635  
    dynamic, 451–458  
        exercises, 460–471  
        free() function, 455–458  
        malloc() function, 452–455

size, 451–452  
sizeof operator, 453

overview, 449

static, 450–451  
    actions, 450–451  
    example, 451  
    size, 450  
    variables, 450, 451

Memory management; *see also* Data structures;  
    Memory allocation  
functions, 458–460  
    memcmp(), 458, 459–460  
    memcpy(), 458  
    memmove(), 458

implementation examples, 473–488  
    linked list, 481–488  
    queue, 478–480  
    stack, 473–478  
    overview, 449

memset(), 633

Method overloading, 624–625

Method overriding, 625–626

MIN macro, 554

mk\_time() function, 426–427

mobile phone  
    IMEI, 249–251  
    wireless networks, 249–251

Modules, interactions between, 571

Multidimensional arrays, 171; *see also* One-dimensional arrays; Two-dimensional arrays  
    function, 358

Multiple inheritance, Java, 622

## N

Naming variables, 9–10

native modifier, Java, 617

Naughton, Patrick, 611

NDEBUG, 553

near, 10

Nested if statements, 76–79

Nested loops, 124–129; *see also* Loops  
    do-while, 124  
    iteration, 133  
    for statement, 108  
    while, 124

Nested structures, 415–417

Network communications, bitwise operators and, 66–67, 66f

Network protocol, 571; *see also* Internet Protocol (IP) network  
    described, 571

development of, 571  
implementation, 571  
*Q.931* protocol, 571  
New-line character, 5, 35, 241  
new operator, C++, 602–603  
Nodes, 471, 471f; *see also* Linked list  
  deletion, 472, 473f  
  insertion, 472, 472f  
No linkage, variables with, 321  
Nonaccess modifiers (N\_A\_M), Java, 612, 616–620  
  abstract, 617, 619–620  
  final, 617, 618  
  native, 617  
  static, 617–618  
  strictfp, 617  
  synchronized, 617  
  transient, 617  
  volatile, 617  
Nonsorted array, 381  
non-void function, 305  
Null character, 253, 254; *see also* Characters  
NULL macro, 189, 190  
Null pointers, 189–190  
Null statement, 74  
Numeric escapes, 20

## O

.o, 6  
.obj, 6  
Object-oriented programming, 578, 581, 611; *see also* C++; Java  
Object(s), 578  
  C++, 579–582  
  Java, 611–614  
Octal constant, 14  
offsetof, 408  
ondefault constructor, 583–584  
One-dimensional arrays; *see also*  
  Two-dimensional arrays  
    as argument of function, 330–336, 359  
    data\_type, 151–152  
    declaration, 151–152, 155–158  
    elements, accessing, 152–154  
    error, 164  
    example, 152  
    exercises, 155–170  
    frequency array (freq), 159–160  
    100 integers, program, 166–167  
    initialization, 154–155  
    shift operator, using, 164–165  
    10 integers, program, 162–163, 167–168  
    20 integers, program, 160–161  
open\_file(), 514  
Opening, file, 508–510  
  exercise, 513–514  
  fopen() function, 508, 509, 510  
  open modes, 508t  
Open modes, file, 508t  
Open Systems Interconnection (OSI) model, 572  
! operator, 51, 52  
# operator, 556–557  
## operator, 556–557  
& operator, 35, 36, 37, 61  
&& operator, 52  
^ operator, 61  
  in data encryption, 63–64  
| operator, 61  
|| operator, 53, 561  
== (equality) operator, 12, 49, 50, 74  
++ (increment) operator, 47–49  
Operators, 45–71; *see also* specific operators  
  , 62

- sizeof, 11, 57–59
  - ternary, 88
  - unary, 47
  - = (assignment) operators, 45–46, 50, 74
    - wrong use of, 81
  - (decrement) operators, 47–49
  - Ordinary characters
    - use of, scanf() function and, 37–38
  - Outer loop, iterations, 125, 126, 127, 133
  - Output stream, 508
  - Overloaded function, 584
  - Overloading; *see also* Polymorphism
    - constructor, 623, 624
    - defined, 584
    - of functions and operators in C++, 584–589
    - method, 624–625
  - Overriding, method, 625–626
- P**
- Packages, Java, 614
  - Parameters; *see also* specific entries
    - defined, 307
    - fixed, 367–368
    - function, 300
      - exercises, 311–315
      - switch statement, 313–314
      - with variable number of, 366–369
    - function call with, 307–311
    - function call without, 305–307
    - macros with, 554–555
    - vs. argument, 307
  - Permutation encryption algorithm, 658–661
  - ping command, 366
  - Pointer arithmetic, 199–201
    - exercises, 208–209, 210–211, 212–215, 227–228
  - Pointers, 187–235
    - arithmetic, 199–201, 208–209, 210–211
    - array of, 216–218, 359
    - array of pointers to functions, 231–235
    - arrays and, relationship between, 153, 203–207, 213, 292
    - comparison, 201
    - const qualifier, 198
    - data\_type, 187–188
    - declaration, 187–188
    - example, 190–191
    - exercises, 192–197, 201–203, 207–216, 218–219, 220–222, 225–228
    - to function, 228–231
    - initialization, 189
    - integers and, 199–200
    - iterations, 215–216
  - memory and, 187, 188f
  - null pointers, 189–190
  - overview, 187
  - pointer to, 219–220
  - sizeof operator, 188
  - for statement and, 226
  - string literals and, 258–261, 259f
  - to structure, 419–420
  - structure containing, 415
  - to structure member, 412–413
  - subtraction, 201
  - two-dimensional arrays and, 222–225
  - using, 190–191
  - void\*, 189, 197–198
  - while loop and, 195
- Pointer to a pointer variable, 219–220
  - Pointer to a structure member, 412–413
  - Polymorphism; *see also* Overloading
    - C++, 594–596
    - Java, 623–626
      - dynamic, 623
      - method overloading, 624–625
      - method overriding, 625–626
      - static, 623
    - pop() function, 480
    - pow() function, 317, 349
    - #pragma directive, 562
  - Precedence, operator, 70–71, 663, 663t
    - operator, 64
  - Precedence table, 663, 663t
  - Precision, conversion specifier, printf()
    - function, 25–26
  - Predefined macros, 552, 552t
  - Prefix, conversion specifier, printf() function, 27
  - Preprocessor, 549–567
    - directives and conditional compilation, 557–562
      - defined operator, 561–562
      - #elif, 558–560
      - #else, 558–560
      - #endif, 558–560
      - #if, 558–560
      - #ifdef, 559, 560–561
      - #ifndef, 559, 560–561
      - #undef, 560–561
    - exercises, 563–567
    - macros with parameters, 554–555
    - miscellaneous directives, 562
      - #error, 562
      - #line, 562
      - #pragma, 562
    - # operator, 556–557
    - ## operator, 556–557

overview, 549  
simple macros, 549–553

printf() function, 5, 12, 20–28, 299; *see also* fprintf() function

call, 307  
continue statement, 113  
conversion specifications, 22–23, 23t, 25, 25f  
field width, 27  
flags, 28, 28t  
optional fields, 25–28  
precision, 25–26  
prefix, 27

escape sequences, 20–22, 21t  
character escapes, 20  
numeric escapes, 20

exercises, 311  
printing variables, 24  
return value, 23–24

for statement, 107, 109–110  
strings writing, 255–257

unions, 439  
with variable number of parameters, 367  
vs. scanf() function, 36

Printing variables, 24

PrintStream, 613

private access modifier, Java, 617, 617t  
private interface, Java, 628–629

private members of class, C++, 581, 582, 593

Process, file, 511

Program; *see also* Large programs

- comments, 5–6
- compilation, 6–7
- execution, 8
- first, 3–6
- life cycle, 3
- linking, 7–8
- ways to run, 8
- writing, 3
- writing larger programs, 571; *see also* Large programs

Program control, 73–100

- conditional operator ?:, 88–89
- exercises, 79–88, 90–92, 96–100
- if-else statement, 75–76
- if statement, 73–75
- nested if statements, 76–79
- switch statement, 92–96
  - switch *vs.* if, 96

protected access modifier, Java, 617, 617t  
protected members of class, C++, 581, 582, 593

Protocol, network, *see* Network protocol

Prototype, function, 299

prtcl.c, 572, 573, 578  
\*ptr, 190, 191

public access modifier, Java, 616–617, 617t  
public interface, Java, 626–627, 628–629

public keyword, Java, 612

public members of class, C++, 581, 590

putc() function, 514

putchar() function, 245–246

- exercise, 246–247

puts() function

- strings writing, 255–257

Pythagoras Theorem, 219

## Q

Q.931 protocol, 571

qsort() library function, 400–402

- exercise, 401–402

Queue (data structure), implementing, 478–480

Quick sort algorithm, 398–400

quick\_sort function, 399–400

## R

RAM (random access memory), 9

rand() function

- for statement, 107, 108

RAND\_MAX, 108

Random access memory (RAM), 9

rcv.c file, 576

rcv\_msg() function, 573

Reading

- from binary file, 532
  - exercises, 534–544
- fread() function, 533–534
- from text file, 518
  - fgetc() function, 526–530
  - fgets() function, 523–526
  - fscanf() function, 518–523

Read strings, 261–265

- example, 262
- exercises, 264–270
- exit() function, 263–264
- fgets() function, 262, 263, 264
- getchar() function, 262–263, 264, 267–268
- gets() function, 262
- read\_text() function, 263
- realloc() function, 263
- scanf() function, 261–262, 264
- strlen() function, 263

read\_text() function, 339

- read strings, 263

realloc() function, 452  
  read strings, 263  
realloc\_mem() function, 461–462  
Recursive functions, 369–371  
  binary search tree, 495  
  exercises, 371–376  
  main() function, 371  
recv.c, 572  
Redirection, 512, 518  
Reference variables, C++, 604–606  
register class, 321  
  automatic variables, 323  
  local variable, 323

Relational operators, 49–50

RELEASE message, 571

reorder variable, 396

replacement\_characters, 549

return statement, 4

  in function execution, 304–305  
  switch statement, 94–95  
  *vs.* exit() function, 263–264

return\_type function, 228, 299–300

Return value

  function, 299–300  
  printf() function, 23–24  
  scanf() function, 38

RFC 791 standard, 571

RGB color model, 436

Richie, Dennis, 1

RLE (run length encoding) algorithm, 288–289

Run length encoding (RLE) algorithm, 288–289

Runtime binding, 623, 625

rvalue, 45

## S

scanf() function, 35–38, 241, 299, 518, 520; *see also*  
  fscanf() function  
described, 35, 38  
double variable, 36  
example, 35  
exercises, 311  
float variable, 36  
long double variable, 36  
& operator, 35, 36, 37  
ordinary characters, use of, 37–38  
overview, 35  
read strings, 261–262, 264  
return value, 38  
short variable, 36  
with variable number of parameters, 367  
  *vs.* printf() function, 36

Scope, of variables, 321, 323

Searching arrays, 381–386

  binary search algorithm, 383–386

  linear search algorithm, 381–383

SEEK\_CUR, 531

SEEK\_END, 531

SEEK\_SET, 531

Selection sort algorithm, 386–392

  add\_sort() function, 389–390

  exercises, 387–392

  sel\_sort() function, 386

  two-dimensional array, 390–392

sel\_sort() function, 386

Semicolon ;, 3

send.c, 572

send.c file, 575

Sequential search algorithm, *see* Linear search algorithm

setHomeClub(), 628

setjump.h, 668

Setters, Java, 628

SETUP message, 571, 573, 574

SETUP message structure, unions, 442f  
  ISDN network, 443–445

setVisitorClub(), 628

Sheridan, Mike, 611

Shift operator, 62–63

short int data type, 11, 11t  
  function call, 308

short variable

  scanf() function, 36

show() function, 591, 626

show\_inorder() function, 493

show\_preorder() function, 493

show\_queue() function, 480

show\_stack() function, 480

signal.h, 668–669

signed char data type, 11t, 12

signed int bit field, 417

Single inheritance, Java, 622

Single transformation algorithm, 290–291, 290f

sizeof(int), 508

sizeof operator, 11, 57–59, 145, 223

  array, 152

  array variable, 333–334

  dynamic memory allocation, 453

  fwrite() function and, 533

  if-else statement, 58–59

  pointer and, 206

  pointer variable, 188, 334

  read strings, 262

  union variable, 438

size\_stack() function, 477

size\_t type, 452

SOME\_TAG macro, 560–561  
Sorted subarray, 392, 393f  
Sorting arrays, 383, 386–400  
    bubble sort algorithm, 395–398  
    insertion sort algorithm, 392–394, 393f  
    quick sort algorithm, 398–400  
    selection sort algorithm, 386–392  
Source file  
    extension, C++ program, 579  
Java, 611  
large programs  
    fsm.c, 572  
    module, 572  
    prtcl.c, 572, 573  
    recv.c, 572  
    send.c, 572  
Spelling errors, 7  
SportMatchImpl class, 627, 628  
SportMatch interface, 626–627  
sqrt() function, 124, 318, 319  
rand() function  
    for statement, 107, 108  
Stack, memory, 152, 305, 449  
    implementing, 473–478  
    size, 452  
Stack overflow, 371, 451  
Standard template library, C++, 600–601  
Static binding (compile time), 584, 594  
Static data structures, 471  
static interface, Java, 626, 629  
static keyword, Java, 612  
Static memory allocation, 450–451  
    actions, 450–451  
    example, 451  
    size, 450  
    variables, 450, 451  
static modifier, Java, 617–618  
Static polymorphism, Java, 623  
Static storage class, of variables, 321, 327  
Static variable, 325, 327–330, 339, 449  
    declaration, 328–330  
    default properties, 328  
    initializing, 328–329  
stdarg.h, 669  
\_\_STDC\_\_, 552  
stderr (standard error stream), 512, 513  
stdin (standard input stream), 35, 512, 518  
stdio.h, 3, 5, 299, 509, 511, 512, 514, 523, 544,  
    669–671  
stdlib.h, 671–673  
stdlib.h file, 452  
stdout (standard output stream), 20, 512, m513  
Stealth Project, 611  
Storage class, of variables, 321  
    local variable, 323  
Storing  
    strings, 253–255  
strcat() function  
    in string, 282–284  
strchr() function, 375  
strcmp() function  
    exercises, 285–291, 344  
    in string, 284–285  
    vs. memcmp() function, 458–459  
strcpy() function  
    exercises, 280–282, 463–464, 649  
    in string, 278–280  
    vs. memcpy() function, 458–459  
Stream, defined, 508  
strictfp modifier, Java, 617  
string.h, 673–675  
String literals, 253  
    pointers and, 258–261, 259f  
Strings, 253–295, 254  
    array of, 292, 292f  
    common error, 254  
    exercises, 254–255, 257–258, 260–261, 264–270,  
        293–295  
    fgets() function, 257  
functions, 270  
    exercises, 270–278, 280–282, 285–291  
    strcat() function, 282–284  
    strcmp() function, 284–285  
    strcpy() function, 278–280  
    strlen() function, 270  
    strncmp() function, 284–285  
    strncpy() function, 278–280  
literals, 253  
literals, pointers and, 258–261, 259f  
overview, 253  
pointers and, 258  
printf() function, 255–257  
puts() function, 255–257  
read, 261–265  
    example, 262  
    exercises, 264–270  
    exit() function, 263–264  
    fgets() function, 262, 263, 264  
    getchar() function, 262–263, 264, 267–268  
    gets() function, 262  
    read\_text() function, 263  
    realloc() function, 263  
    scanf() function, 261–262, 264  
    strlen() function, 263  
storing, 253–255  
two-dimensional arrays and, 292–295

writing, 255–258  
exercises, 257–258  
fgets() function, 257  
printf() function, 255–257  
puts() function, 255–257

strlen() function, 553  
exercises, 270–278, 337–338  
iterations, 273–274  
read strings, 263  
in string, 270

strcmp() function  
exercises, 285–291  
in string, 284–285

strncpy() function  
exercises, 280–282, 288  
in string, 278–280

Stroustrup, Bjarne, 579

struct, 405

Structures, 405–437; *see also* Array(s); Unions  
accessing members, 410, 420  
array of, 420–422  
bit fields, 417–419, 418f  
comparing, 413  
containing arrays, 414  
containing pointers, 414–415  
containing structures, 415  
copy structure to structure, 413  
declaration, 405–408  
dynamic data structures, 471  
exercises, 422–424, 426–437  
as function argument, 424–426  
image-editing programs, 435–437  
incomplete declaration, 416  
initializing, 410–412  
members/fields, 405  
mk\_time() function, 426–427  
nested, 415–417  
operations, 413–414  
overview, 405  
pointer to, 419–420  
pointer to a structure member, 412–413  
sizeof, 408  
typedef specifier, 408–410  
*vs.* unions, 437

structure\_tag, 405

structure\_variable\_list, 405, 406

Subarray; *see also* Array(s)  
sorted, 392, 393f  
unsorted, 392, 393f

Subnets, IP network, 649–652

Subpackages, Java, 614

Subtraction  
pointers, 201

Sun Microsystems, 611

super keyword, 626

switch statement, 92–96, 144, 233; *see also* if-else statement; if statement(s)  
array of pointers to functions, 233–235  
break statement, 93, 94–95, 111–113  
under case label, 93  
case statements, 94–95  
continue statement, 113–114  
default case, 93–94, 112  
disadvantage, 96  
example, 93–94, 95–96  
exercises, 114–115  
function with parameter, 313–314  
large programs, 574–576  
return statement, 94–95  
*vs.* if-else-if statements, 96

/\* symbol, 5

/\* symbol, 5

Symmetric cryptography, 658

synchronized modifier, Java, 617

Syntactic errors, 7

## T

Templates, C++, 596–600  
class, 599–600  
function, 596–599

Ternary operator, 88

test.bin binary file, 536–537, 540–542, 655

test() function, 302, 329  
call with/without parameters, 305–307, 308–311, 315  
declaration, 334–335  
exercises, 339, 344–345, 346, 639–641, 649  
local variable, 322–325  
static memory allocation, 451  
structure as function argument, 425–426

test.h file, 550

test.txt file, 509, 518, 527

Text file(s), 507–508, 508f  
content, 508f  
end of, 530–531  
opening, 509  
reading data from, 518  
fgets() function, 526–530  
fgets() function, 523–526  
fscanf() function, 518–523

size, 507, 508f

special character, 507  
*vs.* binary file, 507

writing data in, 511  
fprintf() function, 512–514

`fputc()` function, 514–518  
`fputs()` function, 511–512

`this` keyword, Java, 616, 628  
Thompson, Ken, 1  
throw statement, 606–608

Tigraph sequence, 241  
`_TIME_`, 552  
time() function

    for statement, 107, 108

`time.h`, 675

*Toeplitz* matrix, 178

transient modifier, Java, 617

Transport Control Protocol (TCP) packet, 66f  
    bitwise operators, 66–67, 66f  
    header format, 636–638, 636f

Trapezoidal rule approximation, 319–321,  
    320f

Tree, 488

try block, 606, 607

Two-dimensional arrays, 171f; *see also* One-dimensional arrays

    as argument of function, 358–364  
        exercises, 359–364

    data\_type, 170

    declaration, 170

    elements, accessing, 171–172

    exercises, 174–184

    image-editing programs, 435–437

    initialization, 172–173

    pointers and, 222–225

    selection sort algorithm, 390–392

    strings and, 292, 292f

        exercises, 293–295

typedef, 406

typedef specifier, 408–410

Type qualifiers, 18

    const, 18

    volatile, 18

## U

Unary operators, 47

#undef directive, 560–561

Unions, 437–445; *see also* Array(s); Structures

    ALERTING message structure, 442f,  
        443–445

    in communication protocols development,  
        442

    CONNECT message structure, 442f,  
        443–445

    declaration, 437–438

    exercises, 443–445

    members access, 438–442

multiple views of same data, 441

overview, 405

SETUP message structure, 442f, 443–445

sizeof, 438

value, 439

    vs. structure, 437

Universal Product Code (UPC) barcode  
(example), 276–278

Unix/Linux systems

    compilers for, 6

Unix operating system, 1, 611

    characters, 507

    makefile, 577–578

Unnamed bit fields, 418

unsigned char data type, 11t, 12, 63, 119, 245

    getchar() function, 245

unsigned int bit field, 417

unsigned int type, 270, 408–409, 452

unsigned long int data type, 11, 11t

unsigned variables

    bitwise operators, 61

Unsorted subarray, 392, 393f

## V

`va_arg()` macro, 367

`va_end()` macro, 367

`va_list`, 367

Value

    arrays, 154, 156

    unions, 439

    variables, assignment of, 13–15

Variable number of parameters, functions with,  
    366–369

Variable(s); *see also* Array(s); *specific variables*

    address of, 189

    arithmetic conversions, 15–18

    assignment of values and constants, 13–15

    automatic, 13

    compound statement, 323

    constant, 18

    declaration, 10–13

    defined, 9

    extern, 321, 327

    floating-point, 13, 25

    global, 325–327

    linkage of, 321

    local, 322–325

    naming, 9–10

    with no linkage, 321

    pointer, 187–188; *see also* Pointers

    pointer to pointer, 219–220

    printing, 24

properties, 321  
reference, C++, 604–606  
register, 321  
scope of, 321, 323  
signed, 11  
sizeof, 11  
static, 327–330  
storage class of, 321, 323  
type qualifiers, 18  
uninitialized, 154  
va\_start() macro, 367  
virtual keyword, 625  
void function, 300, 304, 342–343, 358, 359  
    exercises, 318–319, 350–351  
void keyword, Java, 612  
void main(), 4–5  
Void\* pointers, 189, 197–198  
void test(int \*arr), 332  
void test(int arr[]), 332–333  
volatile modifier, Java, 617  
volatile qualifier, 18

## W

Warning message, function, 302  
while loop, 129  
    break statement, 111  
    continue statement, 113  
    execution, 645–646

insertion sort algorithm, 393–394  
nested, 124  
pointer and, 195  
while statement, 129–140; *see also* do-while  
    statement  
example, 129–131  
exercises, 132–140, 527  
fscanf() function, 519  
if statement and, 129  
for statement and, 108, 129, 130  
White-space character, 36, 37, 261  
Windows systems, 611  
    C compilers for, 6  
    characters, 507  
wireless networks  
    mobile phone, 249–251  
Writing  
    in binary file, 532  
        exercises, 534–544  
    fwrite() function, 532–533  
strings, 255–258  
    exercises, 257–258  
    fgets() function, 257  
    printf() function, 255–257  
    puts() function, 255–257  
in text file(s), 511  
    fprintf() function, 512–514  
    fputc() function, 514–518  
    fputs() function, 511–512