

Orhan Gazi

Modern C Programming

Including Standards C99, C11, C17, C23



Modern C Programming

Orhan Gazi

Modern C Programming

Including Standards C99, C11, C17, C23



Springer

Orhan Gazi
Electrical and Electronics Engineering
Ankara Medipol University
Altındağ, Ankara, Türkiye

ISBN 978-3-031-45360-1 ISBN 978-3-031-45361-8 (eBook)
<https://doi.org/10.1007/978-3-031-45361-8>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Preface

C programming language was introduced in 1972. After its introduction, it became the most popular programming language in the world in short time. In every branch of science and engineering, developments occur in time, and every technology needs to update itself in time. Otherwise, it becomes old fashion in short time. Today, the popularity cycle of technologies is becoming shorter.

C programming language was revised several times in the last several decades. The recent C standards are tabulated in Table 1.

In Table 1, all the formal names start with the same expression IEC 9899, and after symbol “:,” the year of the standard is written, i.e.,

ISO/IEC 9899:1999

ISO/IEC 9899:2011

ISO/IEC 9899:2018

ISO/IEC 9899:2023

The year of C17 standard is 2018. Some people may use the informal name as C18. But in literature, it is seen that C17 was adopted by many authorities. Let's briefly give information about these standards.

Table 1 C language standards since 1999

Informal name	Introduction year	Formal name	File name
C99	1999	ISO/IEC 9899:1999	N1256.pdf
C11	2011	ISO/IEC 9899:2011	N1570.pdf
C17	2018	ISO/IEC 9899:2018	N2310.pdf
C23	2023	ISO/IEC 9899:2023	N3096.pdf (last draft)

C99 Standard

Formal name of this standard is ISO/IEC 9899:1999. This standard is published in 1999. The file name of this standard is N1256.pdf. Using internet search engines, all standard document can be found easily.

C99 standard made major contribution to the C language; the standard introduced the header files

<pre><complex.h> <fenv.h> <inttypes.h> <stdbool.h> <stdint.h> <tgmath.h></pre>	<pre>complex number arithmetic floating-point environment formats for fixed-width integer types macros for Boolean data type fixed-width integer data types type-generic math</pre>
--	---

With the introduction of these header files, it became much easier to perform complex mathematical calculations, and perform trigonometric operations. And fixed-width integers were useful for efficient use of the resources.

New built-in data types introduced in C99 are

```
long long, unsigned long long
_Bool, _Complex, _Imaginary
inline, restrict
```

Variable length arrays were introduced in C99. Before C99, integer constants can be used for array sizes, and they must be defined before the array declaration. In C99 standard, we can declare an array whose size is known at run time. That is, you can get the array size from the user at run time.

In C99, the keyword static can be used inside the square brackets during array declaration, for example:

```
void myFunc(int a[static 20])
{
    // statements
}
```

Before C99, single line comments, //..., are not accepted. C99 introduced single line comments. In addition, before C99, variable declarations must be done at the beginning of the code; however, with C99, variable declarations can be made in any place of the code.

C11 Standard

Formal name of this standard is ISO/IEC 9899:2011. The file name where the standard is published is N1570.pdf. Using internet search engines, it is possible to find all the document standards easily.

After C99 release, a major contribution to the C language is made by the C11 standard. The header files introduced in C11 standard are

<code><stdalign.h></code>	alignas and alignof macros
<code><stdatomic.h></code>	atomic operations
<code><stdnoreturn.h></code>	noreturn macro
<code><threads.h></code>	thread functions
<code><uchar.h></code>	UTF-16 and UTF-32 encoding

With C11, standard parallel processing became possible using built in thread functions, and it was possible to avoid the race problem occurring in parallel processing using the **atomic** data types. Thread and atomic libraries were the powerful contribution of C11 standard.

The keywords added by C11 standard are

<code>_Alignas</code>	<code>_Alignof</code>	<code>_Atomic</code>	<code>_Generic</code>
<code>_Noreturn</code>	<code>_Static_assert</code>	<code>_Thread_local</code>	

C17 Standard

The standard C17 does not introduce new language features. In this standard, just some minor defects of the C11 standard are fixed.

C23 Standard

The major contribution of C23 standard is the introduction of two new header files

<code><stdckdint.h></code>	functions and macros for checked integer arithmetic
<code><stdbit.h></code>	functions and macros for bit manipulation

These files are useful to use the hardware as efficient as possible.

In this book, starting from the most basic topics of C programming language, we cover new language features introduced in the standards. Chapter 1 focuses on the representation of numbers in computer. This chapter is usually skipped in many C

books. However, for a good understanding of C programming, good knowledge of number representations in computer is needed. We advise the reader to study the first chapter before processing with C programming topics. Chapters [2](#), [3](#), [4](#), [5](#), [6](#), [8](#), and [9](#) cover the common topics of all the standards. Chapter [10](#) explains the pointers in C programming, and for good comprehension of this chapter, some knowledge about digital hardware fundamentals is needed.

Complex numbers and fixed bit length integers are covered in Chaps. [7](#) and [13](#). These concepts are introduced in C99 standard. In Chaps. [15](#) and [16](#), threads and atomic data types are explained. These concepts are introduced in standard C11 and they are also covered in C17 and C23. Macros, type qualifiers, storage classes, signals, enumerations included in all standards are covered in Chaps. [11](#), [12](#), and [14](#).

This book can be adapted as a text or supplementary book for C programming for a one-semester course, or it can be read for personal development.

I dedicate this book to my lovely brother Ilhan Gazi, and to those people who do something useful for humanity.

Ankara, Turkiye

Orhan Gazi

Wednesday, August 2, 2023

Contents

1	Representation of Numbers and Characters in Computer	1
1.1	Number Bases	1
1.1.1	Decimal Numbers	1
1.1.2	Binary Numbers	1
1.1.3	Octal Numbers	2
1.1.4	Hexadecimal Numbers	2
1.2	Conversion Between Bases	3
1.2.1	Binary to Decimal Conversion	3
1.2.2	Binary to Octal Conversion	3
1.2.3	Binary to Hexadecimal Conversion	4
1.2.4	Decimal to Binary Conversion	5
1.2.5	Octal to Binary Conversion	6
1.2.6	Hexadecimal to Binary Conversion	6
1.2.7	Hexadecimal to Decimal Conversion	7
1.3	Positive Integers	8
1.4	Two's Complement Form	8
1.5	Negative Integers	10
1.6	Registers	11
1.7	Memory Units	12
1.8	How Are the Integers Stored in Computer Memory, Big-Endian, and Little-Endian?	12
2	Data Types and Operators	15
2.1	How to Start Writing a C Program?	15
2.2	Comments in C Programming	17
2.3	The First C Program	17
2.4	Variables and Data Types	19
2.5	Binary Number Representation in Modern C	21
2.6	sizeof Operator in C	22
2.7	Unsigned Char Data Type	23

2.8	Left and Right Shift Operators in C	25
2.9	Integer Data Type	26
2.10	Hexadecimal and Octal Numbers	28
2.11	How Are Integers Stored in Computer Memory?	29
2.11.1	Short Integer Data Type	30
2.12	Why Do We Have Both Integer and Short Integer Data Types?	31
2.13	Long Integer and Long-Long Integer Data Types	32
2.14	Unsigned Integer Data Type	34
2.15	Floating-Point Number in C	39
2.15.1	IEEE 754 Floating-Point Standard (Single Precision)	39
2.16	Keyboard Input Using <code>scanf</code> in C	41
2.17	Operators in C Programming	43
2.17.1	Arithmetic Operators	43
2.17.2	Remainder Operator %	46
2.17.3	Augmented Assignment Operators	47
2.17.4	Logical Operators	47
2.17.5	Bitwise Operators in C	50
2.17.6	Increment and Decrement Operators	60
2.18	Operator Precedence	62
3	Type Conversion in C	67
3.1	Type Conversion Methods	67
3.1.1	Implicit Conversion	67
3.1.2	Explicit Conversion	70
3.2	Information Loss When a Higher-Order Data Is Converted to a Lower-Order Data	73
3.3	Information Loss When Conversion Is Performed Between Signed and Unsigned Data Types	74
4	Structures	77
4.1	Introduction	77
4.2	Initialization of Structure Elements	78
4.3	Initialization Using Designated Initializer List	79
4.4	Typedef for Structures	83
4.4.1	Alternative Use of <code>typedef</code> for Structures	85
4.5	Nested Structures	86
4.6	Structure Copying	88
4.7	Structures with Self-Referential	89
4.8	Bit Fields	91
4.9	Structures as Function Arguments	92
4.10	Structure Padding and Packing in C Programming	93
4.11	Unions	94

5 Conditional Statements	99
5.1 Conditional Structure	99
5.2 Conditional Ladder Structure (-If Ladder)	104
5.3 Multiconditional Structures	107
5.4 Syntax of Nested If-Else	110
5.5 Conditional Operator in C	112
5.6 switch Statement	116
6 Loop Statements	125
6.1 The For-Loop	125
6.1.1 Nested For-Loop	138
6.2 The While-Loop	140
6.2.1 Nested While-Loop	144
6.3 The Do-While Loop	147
6.4 Continue Statement	148
6.5 Break Statement	150
7 Complex Numbers in Modern C Programming	157
7.1 How to Define a Complex Number	157
7.2 Complex Operations	158
7.3 Calculation of Absolute Value (Norm, Modulus, or Magnitude) of a Complex Number	161
7.4 Complex Number Formation	161
7.5 Calculation of the Conjugate of a Complex Number in C	162
7.6 Calculation of the Argument, That Is, Phase Angle, of a Complex Number in C	164
7.7 Calculation of Complex Exponentials	164
7.8 Computation of the Complex Natural (Base-e) Logarithm of a Complex Number	165
7.9 Complex Power Calculation	166
7.10 Square Root of a Complex Number	167
7.11 Complex Trigonometric Functions	168
7.11.1 The csin Functions	168
7.11.2 The ccos Functions	168
7.11.3 The ctan Functions	169
7.11.4 The cacos Functions	170
7.11.5 The casin Functions	170
7.11.6 The catan Functions	171
7.11.7 Hyperbolic Functions	172
8 Arrays	175
8.1 Syntax for Array Declaration	175
8.2 Accessing Array Elements	175
8.3 Array Initialization Without Size	177
8.4 Array Initialization Using Loops	179
8.5 Strings as Array of Characters	184

8.6	Multidimensional Arrays	185
8.7	Passing an Array to a Function in C	189
9	Functions	193
9.1	Introduction	193
9.2	Types of Functions	197
9.3	Passing Parameters to Functions	198
9.4	Returning More Than One Value	200
9.5	Recursive Functions	202
9.6	Nested Functions	205
10	Pointers	207
10.1	Definition	207
10.2	Address of a Variable	207
10.3	NULL Pointer	215
10.4	Void Pointer	215
10.5	Types of Pointers	221
10.5.1	Pointer to a Constant Value	221
10.5.2	Pointer to a Constant Address (Constant Pointer)	223
10.5.3	Constant Pointer to a Constant Value	224
10.6	Function Pointers	224
10.6.1	Functions Returning Pointers	228
10.7	Pointers and Arrays	230
10.8	Multiple Pointers	237
10.9	Heap Stack and Code Memories	239
10.10	Dynamic Memory Allocation	240
10.10.1	malloc() Function	241
10.10.2	calloc() Function	245
10.10.3	free() Function	246
10.10.4	realloc() Function	247
10.11	Memory Functions	248
10.11.1	Memset Function	248
10.11.2	Memcpy Function	249
10.11.3	Memmove Function	250
10.11.4	Memcmp Function	251
11	Directives and Macros in C	259
11.1	Introduction	259
11.2	Preprocessor Directives as Macros	259
11.3	Macros as Functions	261
11.4	Multiline Macros	262
11.5	Directives Used for File Inclusion	265
11.6	Predefined Macros	267
11.7	Conditional Compilation	270
11.8	Concatenation Operator ##	274

12 Type Qualifiers, Enumerations, and Storage Classes in C	277
12.1 Type Qualifiers in C	277
12.1.1 Const	277
12.1.2 Restrict	279
12.1.3 Volatile	279
12.2 Storage Classes in C	280
12.2.1 Auto	280
12.2.2 Extern	280
12.2.3 Static	283
12.2.4 Register	284
13 Integer with Exactly N Bits	289
13.1 General Form of Fixed Width Integers	289
13.2 Macros for printf and scanf	289
13.3 uintN_t	293
13.4 int_leastN_t	295
13.5 uint_leastN_t	296
13.6 int_fastN_t	296
13.7 uint_fastN_t	297
13.8 Macros for printf	298
13.9 Macros for scanf	298
14 Signals in C	303
14.1 Introduction	303
14.2 Signal Handling	304
14.3 SIGINT	304
14.4 SIGQUIT	308
14.5 Artificial Signal Generation	310
14.6 Some of the Most Used Signals	311
15 Threads in C	313
15.1 Introduction	313
15.2 Thread Creation	313
15.3 Parallel Processing Using Threads	314
15.4 pthread_exit() Function	321
15.5 pthread_join() Function	323
15.6 pthread_self() Function	329
15.7 pthread_equal() Function	330
15.8 pthread_cancel() Function	331
15.9 pthread_detach() Function	331
15.10 Synchronizing Threads with Mutexes	332

16	Atomic Data Types	343
16.1	How to Define an Atomic Data Type?	343
16.2	Atomic Integer Types	344
16.3	Atomic Pointers	346
16.4	Race Prevention by Atomic Variables	347
16.5	Lock-Free Atomic Types	352
16.6	Atomic Assignments, Operators, and Functions	353
16.7	Atomic Functions	354
16.7.1	atomic_is_lock_free() Function	354
16.7.2	atomic_fetch_key() Function	354
16.7.3	atomic_store() Function	357
16.7.4	atomic_load() Function	358
16.7.5	atomic_exchange() Function	359
16.7.6	Comparison Functions	360
16.7.7	atomic_flag Macro	362
16.7.8	atomic_init() Function	363
16.8	Memory Order in C	364
16.8.1	Acquire, Release, and Consume	364
16.8.2	Memory Order	365
16.8.3	Atomic Functions with Memory Order	367
17	File Operations in C	369
17.1	File Types	369
17.2	File Operations	369
17.2.1	Opening a File	369
17.2.2	Closing a File	370
17.2.3	Reading and Writing of a Text File	371
17.2.4	Reading and Writing of a Binary File	379
	Bibliography	387
	Index	389

Chapter 1

Representation of Numbers and Characters in Computer



1.1 Number Bases

Number base is a positive integer, and a number with base N can contain digits less than N.

1.1.1 *Decimal Numbers*

If base equals to 10, then all the digits forming a number should be less than or equal to 9. The numbers under base 10 are called decimal numbers.

Example 1.1 We can write a few decimal numbers as

456 99988 67890 45433

In fact, in our daily life we use decimal numbers.

1.1.2 *Binary Numbers*

If the base equals 2, then the numbers are called binary numbers, and binary numbers can be formed using the digits 0 and 1.

Example 1.2 We can write a few binary numbers as

1011 10111101 101101010 11111111

1.1.3 Octal Numbers

If the base equals 8, then the numbers formed under this base are called octal numbers, and an octal number can be formed using the digits

0, 1, 2, 3, 4, 5, 6, 7

Example 1.3 We can write a few octal numbers as

76403 22334 54634

1.1.4 Hexadecimal Numbers

If the base equals 16, then the numbers formed under this base are called hexadecimal numbers, and a hexadecimal number can be formed using the digits

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

where the letters A, B, C, D, E, and F denote the numbers

10, 11, 12, 13, 14, 15.

Example 1.4 We can write a few hexadecimal numbers as

AB04 FF0A 456FEA09

We can use 0x or 0X prefix in front of the hexadecimal numbers. The hexadecimal number in the example can be written either as

0xAB04 0xFF0A 0x456FEA09

or as

0XAB04 0XFF0A 0X456FEA09

but usually small x, that is, 0x, is preferred.

1.2 Conversion Between Bases

Assume that we have a number under a base. The equivalent of this number in another base can be calculated. This procedure is called conversion between bases.

1.2.1 Binary to Decimal Conversion

The n-bit binary number

$$b_{n-1}b_{n-2}\dots b_1b_0$$

can be converted to decimal as

$$b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_12^1 + b_02^0$$

Example 1.5 The decimal equivalent of

$$10110$$

can be calculated as

$$2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

which is equal to

$$22$$

1.2.2 Binary to Octal Conversion

To convert a binary number to octal, starting from the rightmost position we first divide the binary string into groups having 3 bits, and then convert each 3 bits to an octal number.

Example 1.6 Convert the binary number

$$1010001101011$$

to a number in octal base.

Solution 1.6 Starting from the rightmost position, we first divide the binary string into groups having 3 bits as

$$\begin{array}{cccccc} 1 & \underline{010} & \underline{001} & \underline{101} & \underline{011} \\ \square & \square & \square & \square & \square \end{array}$$

where the leftmost bit can be left padded by zeros to make a group of 3 bits as

$$\begin{array}{cccccc} \underline{\textcolor{red}{001}} & \underline{010} & \underline{001} & \underline{101} & \underline{011} \\ \square & \square & \square & \square & \square \end{array}$$

and we convert each group to an octal number as in

$$\begin{array}{ccccc} \underline{001} & \underline{010} & \underline{001} & \underline{101} & \underline{011} \\ 1 & 2 & 1 & 5 & 3 \end{array}$$

Thus, the equivalent octal number is

$$12153$$

1.2.3 Binary to Hexadecimal Conversion

To convert a binary number to octal, starting from the rightmost position, we first divide the binary string into groups each having 4 bits, and then convert each 4 bits to a hexadecimal number.

Example 1.7 Convert the binary number

$$11010001101011$$

to a number in hexadecimal base.

Solution 1.7 Starting from the rightmost position, we first divide the binary string into groups having 3 bits as

$$\begin{array}{cccccc} 11 & \underline{0100} & \underline{0110} & \underline{1011} \\ \square & \square & \square & \square \end{array}$$

where the left most bit can be left padded by zeros to make a group of 3 bits as

$$\begin{array}{cccccc} \underline{\textcolor{red}{001}} & \underline{0100} & \underline{0110} & \underline{1011} \\ \square & \square & \square & \square \end{array}$$

and we convert each group to a hexadecimal number as in

$\underbrace{0011}_{3} \underbrace{0100}_{4} \underbrace{0110}_{6} \underbrace{1011}_{B}$

Thus, the equivalent hexadecimal number is

346B

1.2.4 Decimal to Binary Conversion

A decimal number can be converted to a binary number using successive division operation. In this method, the decimal number is divided by 2 and the remainder is recorded, the dividend is again divided by 2 and the remainder is recorded. This procedure is repeated with dividends until no more division operation can be achieved.

Example 1.8 Convert the decimal number 351 to binary.

Solution 1.8 We divide 351 by 2 and remainder equals 1 and dividend equals 350. We indicate this division operation as on the right-hand side of Fig. 1.1.

If we continue division operation in a successive manner, we obtain Fig. 1.2.

Finally, we collect the binary numbers from bottom to top as depicted in Fig. 1.3 and obtain

101011111

Exercise Find the decimal equivalent of the binary number

101011111

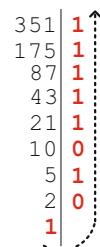
Fig. 1.1 Decimal to binary conversion for Example 1.8

$$\begin{array}{r}
 351 \Big| 2 \\
 \underline{-350} \quad \textcolor{red}{175} \\
 \hline
 \textcolor{blue}{1}
 \end{array}
 \longrightarrow
 \begin{array}{r}
 351 \Big| 1 \\
 \textcolor{red}{175} \quad \textcolor{red}{1} \\
 \hline
 \textcolor{blue}{175} \quad \textcolor{red}{1}
 \end{array}$$

Fig. 1.2 Successive division for Example 1.8

$$\begin{array}{r}
 351 \Big| 1 \\
 175 \quad \textcolor{red}{1} \\
 \underline{-175} \quad \textcolor{blue}{87} \\
 \hline
 \textcolor{blue}{87} \quad \textcolor{red}{43} \\
 \hline
 \textcolor{blue}{43} \quad \textcolor{red}{21} \\
 \hline
 \textcolor{blue}{21} \quad \textcolor{red}{10} \\
 \hline
 \textcolor{blue}{10} \quad \textcolor{red}{5} \\
 \hline
 \textcolor{blue}{5} \quad \textcolor{red}{2} \\
 \hline
 \textcolor{blue}{2} \quad \textcolor{red}{1} \\
 \hline
 \textcolor{blue}{1}
 \end{array}$$

Fig. 1.3 Bits are collected from bottom to top



1.2.5 Octal to Binary Conversion

To convert an octal number to binary, we first convert each octal digit to a binary string each having 3 bits, then concatenate all the bits and obtain the binary equivalent of the octal number.

Example 1.9 Convert the octal number 763015 to binary.

Solution 1.9 We first express each octal digit by 3 bits as shown in

$$\begin{array}{ccccccc} \overbrace{7} & \overbrace{6} & \overbrace{3} & \overbrace{0} & \overbrace{1} & \overbrace{5} \\ 111 & 110 & 011 & 000 & 001 & 101 \end{array}$$

and we concatenate the bits and obtain the binary equivalent number as

$$111 \ 110 \ 011 \ 000 \ 001 \ 101$$

where removing spaces we get

$$111110011000001101$$

1.2.6 Hexadecimal to Binary Conversion

To convert a hexadecimal number to binary, we first convert each hexadecimal digit to a binary string each having 4 bits, and concatenating all the bits we obtain the binary equivalent of the hexadecimal number.

Example 1.10 Convert the hexadecimal number 0x1AF395 to binary.

Solution 1.10 We first express each hexadecimal digit by 4 bits as shown in

$$\begin{array}{ccccccc} \overbrace{1} & \overbrace{A} & \overbrace{F} & \overbrace{3} & \overbrace{9} & \overbrace{5} \\ 0001 & 1010 & 1111 & 0011 & 1001 & 0101 \end{array}$$

then we concatenate the bits and obtain the binary equivalent number as

0001 1010 1111 0011 1001 0101

where removing spaces, we get

000110101111001110010101

1.2.7 Hexadecimal to Decimal Conversion

The n -digit hexadecimal number

$$h_{n-1}h_{n-2}\dots h_1h_0$$

is converted to decimal as

$$h_{n-1}16^{n-1} + h_{n-2}16^{n-2} + \dots + h_116^1 + h_016^0$$

Example 1.11 The decimal equivalent of

0xFF

can be calculated as

$$15 \times 16^1 + 15 \times 16^0$$

which is equal to

255

Example 1.12 The decimal equivalent of

0xFFFF

can be calculated as

$$15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$$

which is equal to

4095

1.3 Positive Integers

Positive integers are also called unsigned integers, and in computer positive integers are represented by their binary equivalents.

Example 1.13 The positive hexadecimal number 0xFF equals to the decimal number 255, which is represented in computer by 1111 1111.

1.4 Two's Complement Form

To find the 2's complement of a binary string, that is, binary number, we start from the rightmost position and proceed to the left until we meet the first 1, and after meeting the first 1, we go on proceeding to the left but we flip each bit to its complement form, that is, 1 is converted to 0, and 0 is converted to 1.

Example 1.14 Find the 2's complement of

10101110001000

Solution 1.14 We start from the rightmost bit and proceed to the left until we meet the first 1 as shown in Fig. 1.4.

Next, we go on proceeding to the left and take the complement of each bit as shown in Fig. 1.5.

and the 2's complement form is obtained as

0 1 0 1 0 0 0 1 1 1 1 0 0 0

where removing the spaces, we obtain

01010001111000

Fig. 1.4 Location of the first “1”

1 0 1 0 1 1 1 0 0 0 1 0 0 0

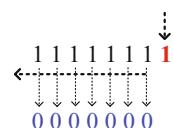
Fig. 1.5 Take the complement of each bit after first “1”

1 0 1 0 1 1 1 0 0 0 1 0 0 0
0 1 0 1 0 0 0 1 1 1

Fig. 1.6 Location of first “1”



Fig. 1.7 Complemented ones



Example 1.15 Find the 2's complement of

$$11111111$$

Solution 1.15 We start from the rightmost bit and proceed to the left until we meet the first 1; however, for this string, the first 1 is at the first position as shown in Fig. 1.6.

Next, we go on proceeding to the left and take the complement of each bit as shown in Fig. 1.7.

and the 2's complement form is obtained as

$$0\ 0\ 0\ 0\ 0\ 0\ 0\ 1$$

where removing the spaces, we obtain

$$00000001$$

Example 1.16 2's complement of

$$11111111$$

is

$$00000001$$

What is the 2's complement of

$$00000001$$

Solution 1.16 2's complement of

$$00000001$$

is

11111111

Hence, we can write that

If d is the 2's complement of a , i.e., $\text{d} = \text{2's comp}(a)$

then 2's complement of d is a , i.e., $a = \text{2's comp}(d)$

1.5 Negative Integers

Assume that a is a positive integer, in computer the positive integer a is represented by a binary string that is obtained by converting a into binary. That is,

a is represented by d , which is obtained by converting a into binary

Negative integers in digital devices are represented in 2's complement form.
That is,

If a is represented by d , then $-a$ is represented by 2's complement of d .

Example 1.17 How are the numbers 17 and -17 represented in computer? Use 8 bits for the representation.

Solution 1.17 Using 8 bits, we can write the binary equivalent of 17 as

00010001

Then, in computer the string

00010001

represents the number 17.

If the string 00010001 represents 17, then -17 is represented by

2's complement of 00010001

which is

11101111

Thus, in computer

the string 00010001 represents 17

and

the string 11101111 represents -17

In 2's complement form, the most significant bit is always 1.

Example 1.18 The string 11111111 represents a negative integer in computer. What is the decimal equivalent of the number?

Solution 1.18 Negative integers are represented by 2's complement of the binary representation of its opposite sign integer, that is, positive integer. We know that

If $d = 2$'s complement of a , then $a = 2$'s complement of d .

To find the value of negative integer, we first take the 2's complement of the binary string, which represents the negative integer, then convert the result to decimal and put a negative sign in front of it; accordingly, 2's complement of

11111111

is

00000001

and the decimal equivalent of this string is 1; then we can say that the string

11111111

represents -1 .

Example 1.19 The string 11111001 represents a negative integer in computer. What is the decimal equivalent of the number?

Solution 1.19 Decimal equivalent of the negative integer can be calculated as

$$-2\text{'s comp}(11111001) \rightarrow -00000111 \rightarrow -7$$

Thus, the string 11111001 represents -7 in computer.

1.6 Registers

Registers are memory-building blocks of memory units. A register is capable of holding a number of bits. A register is usually called by number of cells it contains. Each cell can hold only 1-bit information.

Fig. 1.8 A typical memory unit with 16-bit register addresses

Address	Content
0x000000000065FE18	0x78
0x000000000065FE19	0x56
0x000000000065FE1A	0x34
0x000000000065FE1B	0x12
0x000000000065FE1C	0x94
0x000000000065FE1D	0xEF
0x000000000065FE1E	0xCD
0x000000000065FE1F	0xAB

Example 1.20 An 8-bit register contains 8 cells and can hold 8 bits of information.

1.7 Memory Units

A memory is composed of a number of registers, and each register has an address. Address values of the registers are kept in another set of registers. Register addresses are usually expressed using hexadecimal numbers. In Fig. 1.8, a memory unit is depicted.

1.8 How Are the Integers Stored in Computer Memory, Big-Endian, and Little-Endian?

Numbers are stored in registers. Each register has an address. There are two types of data storage methods: little-endian and big-endian. In the little-endian method, the least significant byte is written to the memory first, whereas, in the big-endian method, the most significant byte is written to the memory first. In Fig. 1.9, little-endian and big-endian methods are explained.

In this book, we will use little-endian method in our examples.

Problems

- Convert the following decimal numbers to binary:

456 255 89 567 123

- Convert the following binary number to decimal, octal, and hexadecimal:

10101011110011110011

- Convert the hexadecimal number 0xA467B to decimal.

int a = 0x1245A78F			
Little-Endian	Big-Endian		
Address	Content	Address	Content
0x0000000000065FE14	0x8F	0x0000000000065FE14	0x12
0x0000000000065FE15	0xA7	0x0000000000065FE15	0x45
0x0000000000065FE16	0x45	0x0000000000065FE16	0xA7
0x0000000000065FE17	0x12	0x0000000000065FE17	0x8F

Fig. 1.9 Little-endian and big-endian memory storage

4. Find the representation of the following positive numbers in computer; use 16 bits for representation:

255 1024 4096 30000

5. Find the representation of the following negative numbers in computer; use 16 bits for representation:

– 255 – 1 – 32768 – 4095 – 154

6. The following strings represent some negative integers in computer; find the decimal equivalents of these integers:

1100111100000111 101111100111111 11001100

7. How is the hexadecimal number $0 \times \text{ABCDE789AA56}$ stored in memory according to little-endian and big-endian conventions?

Chapter 2

Data Types and Operators



2.1 How to Start Writing a C Program?

To write a C program, we should first form the main function of the program. For this purpose, we first write the word “int” as in Code 2.1.

```
int
```

Code
2.1

Then, next to “int” we write “main(void)” as in Code 2.2.

```
int main(void)
```

Code
2.2

We add curly brackets as in Code 2.3.

```
int main(void)  
{  
}
```

Code
2.3

The statement “return 0;“ is written before the closing curly bracket. Note that there is a semicolon after “return 0.”

<code>int main(void) { return 0; }</code>	Code 2.4
---	-------------

Semicolon can also be written after “},” but it is not required.

<code>int main(void) { return 0; };</code>	Code 2.5
--	-------------

The C program lines called statements are written in the main function of the program, and we get the template of the main function as in Code 2.6.

<code>int main(void) { // statements return 0; }</code>	Code 2.6
--	-------------

A return value of 0 indicates the successful completion of the program, while any other value indicates an error.

The template can be simplified. The word “void” in “main” can be omitted as in Code 2.7.

<code>int main() { // statements return 0; }</code>	Code 2.7
--	-------------

We can also omit return 0; statement as in Code 2.8.

<code>int main() { // statements }</code>	Code 2.8
---	-------------

Even we can omit “int” word before main, and we get Code 2.9.

```
main()
{
    // statements
}
```

Code
2.9

In this case, the compiler issues a warning, but it does not have any operational effect.

We can also use void before **main** word as in Code 2.10.

```
void main()
{
    // statements
}
```

Code
2.10

2.2 Comments in C Programming

Single-line comments can be written after // as in Code 2.11.

```
int main()
{
    // Single line comment
}
```

Code
2.11

Multiline comments can be written between the characters /* and */ as in Code 2.12.

```
int main()
{
    /* This is a multiple line
       comment in C */

    // This is a single line comment in C
}
```

Code
2.12

2.3 The First C Program

C programs usually use built-in library functions, and these functions are included using header directives as in Code 2.13.

```
#include <stdio.h>
int main()
{
    // statements
};
```

**Code
2.13**

In this program, stdio.h is the name of the header file, the brackets < > indicate the library directory where header file exists, and the sign # is used to write the directives.

Our first C program is shown in Code 2.14.

```
#include <stdio.h>
int main()
{
    printf("Hello World.");
};
```

**Code
2.14**

where printf() function is used to display the string on console. The output of the program is “Hello World.”

Example 2.1 The printf() function is used twice in Code 2.15.

```
#include <stdio.h>
int main()
{
    printf("Hello World.");
    printf("How are You.");
};
```

**Code
2.15**

Output(s) Hello World.How are You.

Example 2.2 To print the next sentence to a new line, we can use new line character \n as in Code 2.16.

```
#include <stdio.h>
int main()
{
    printf("Hello World.\n");
    printf("How are You.");
};
```

**Code
2.16**

Output(s)

Hello World.

How are You.

The new line character \n can be used at any place inside a sentence as in Code 2.17.

```
#include <stdio.h>
int main(void)
{
    printf("Hello \n World.\n");
    printf("How \n are You.");
}
```

**Code
2.17**

Output(s)

Hello

World.

How

are you

2.4 Variables and Data Types

A variable, as its name implies, can take a variety of values. But these values belong to a single data type. For instance, if *x* is an integer, then *x* can have many different integer values, but it cannot have a fractional value.

A variable is declared as

```
dataType variable_name;
```

Primary data types are

char—8-bit signed numbers

int—32-bit signed integers

float—32-bit signed single-precision fractional real numbers

double—32-bit signed double-precision fractional real numbers

The variables belonging to the same data type can be defined on the same line

```
dataType variable_name1, variable_name2;
```

Different data types should be separated by a semicolon as in Code 2.18.

```
#include <stdio.h>
int main(void)
{
    char ch1='A', ch2;
    int num1, num2=14;  double num3, num4=6.8;
}
```

**Code
2.18**

To print the values of a variable to the screen, we use the printf() function; the general use of the printf() function is

```
printf("%format", variable_name);
```

To print more than one variable value to the screen, we use the format

```
printf("%format1 %format2 ", variable_name1, variable_name2);
```

To print the character variables, that is, char, to the screen, we use

```
printf("%c", variable_name);
```

To print the integer variables, that is, int, to the screen, we use

```
printf("%d", variable_name);
```

where d indicates decimal.

Example 2.3 We define two variables, one of them is a character with value A, and the other is an integer with value 25, and we print both values using printf() function in Code 2.19.

```
#include <stdio.h>
int main(void)
{
    char ch='A';
    int num=25;

    printf("Letter is %c \n", ch);
    printf("Number is %d \n", num);
}
```

**Code
2.19**

The code can be written using a single printf() function as in Code 2.20.

```
#include <stdio.h>
int main(void)
{
    char ch='A';
    int num=25;

    printf("Letter is %c\nNumber is %d \n ", ch, num);
}
```

**Code
2.20**

Output(s)

Letter is A

Number is 25

The 8-bit data type char can take integer values between –128 and +127.

Example 2.4 The tilde symbol ~ has the ASCII value 126. Tilde symbol and its ASCII value can be printed as in Code 2.21.

```
#include <stdio.h>
int main(void)
{
    char ch=126;

    printf("Letter representation is %c ", ch);
    printf("\nNumber value is %d ", ch);
}
```

**Code
2.21**

Output(s)

Letter representation is ~

Number value is 126

2.5 Binary Number Representation in Modern C

Binary numbers can be expressed using 0b or 0B prefix.

Example 2.5 Code 2.21 can also be written using binary number representation as in Code 2.22.

```
#include <stdio.h>
int main(void) {
    char ch=0b01111110;
    printf("Letter representation is %c ", ch);
    printf("\nNumber value is %d ", ch);
}
```

**Code
2.22**

Output(s)

Letter representation is ~
Number value is 126

Example 2.6 The numbers outside the range –128...127 can be assigned to a char variable. In this case, the assigned number is converted to binary and only the least 8 bits are used for the variable. In Code 2.23, this concept is illustrated.

```
#include <stdio.h>
int main(void)
{
    char ch1=0b01111110;
    char ch2=0b11001101111110; // the least significant 8 bits are taken
    char ch3=126; // its binary representation is 01111110

    printf("Letter representations are %c %c %c", ch1, ch2, ch3);
    printf("\nNumber values are %d %d %d", ch1, ch2, ch3);
}
```

**Code
2.23**

Output(s)

Letter representations are ~ ~ ~
Number values are 126 126 126

In embedded hardware programming, binary assignment is preferred to the char variables to see the contents of the registers clearly.

2.6 sizeof Operator in C

The sizeof operator can be used to get the number of bytes occupied by a data type in C programming. The data type char holds 1-byte size in memory.

Example 2.7

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

    printf("Size of char is %d ", sizeof(char));
    printf("\nSize of ch is %d ", sizeof(ch));
}
```

Code
2.24

Output(s)

Size of char is 1

Size of ch is 1

Example 2.8

```
#include <stdio.h>
int main(void)
{
    // - 100 is represented by 10011100 in 2s complement form
    char ch1=0b10011100;
    char ch2=0b11001110011100; // the least significant 8 bits are taken
    char ch3=13212; // its binary representation is 11001110011100
    char ch4=-100;

    printf("%d %d %d %d", sizeof(ch1), sizeof(ch2), sizeof(ch3), sizeof(ch4));
}
```

Code
2.25

Output(s) 1 1 1 1**2.7 Unsigned Char Data Type**

Variables using **unsigned char** data type can have 8-bit non-negative integer values. The minimum value of unsigned char is

00000000

which equals decimal 0, and the maximum value of unsigned char is

11111111

and when this number is converted to integer, we get

$$2^0 + 2^1 + \dots + 2^7 = \frac{2^8 - 1}{2 - 1} \rightarrow 255$$

Hence, a variable of unsigned char data type can take values in the range

$$[0 \dots 255]$$

Example 2.9 In Code 2.26, the maximum value of unsigned char data type is printed in decimal, hexadecimal, and octal forms.

<pre>#include <stdio.h> #include <limits.h> int main() { unsigned char uc_max = UCHAR_MAX; // = 0xFF; printf("Maximum unsigned char: %u (decimal), ", uc_max); printf(" %#o (octal), %#x (hex)\n", uc_max, uc_max, uc_max); }</pre>	Code 2.26
---	---

Output(s) Maximum unsigned char: 255 (decimal), 0377 (octal), 0xff (hex)

Example 2.10 What happens if the negative number -100 is assigned to an **unsigned char** variable?

Solution 2.10 In computer, negative numbers are represented in 2's complement form. The number -100 is represented in 2's complement form using 8 bits as

10011011

If we type

```
unsigned char num = -100
```

then the 2's complement representation of -100 will be assigned to variable num, that is, we will have

```
num = 0B10011011
```

and the binary string is assumed to represent a non-negative number, and when this number is converted to integer, we get

$$2^0 + 2^1 + 2^3 + 2^4 + 2^7 \rightarrow 156$$

Hence, num has decimal value 156.

<pre>#include <stdio.h> #include <limits.h> int main() { unsigned char num=-100; printf("Number is: %u", num); }</pre>	Code 2.27
---	----------------------

Output(s) Number is: 156

2.8 Left and Right Shift Operators in C

The left shift and right shift operators are `<<` and `>>`. In left shift operations, the bits are shifted to the left and new locations are filled with zeros. On the other hand, for the right shift operation, the bits are shifted to the right and **new positions are filled with 0 if the number is a positive number**, or the **new positions are filled with 1 if the number is a negative number**.

That is, in the right shift operation, all the bits are shifted to the right and the new positions are filled with the sign bit.

Example 2.11 This example illustrates the left shift operation.

<pre>#include <stdio.h> int main() { // a = 5(00000101), b = 9(00001001), c = 255(11111111) unsigned char a = 5, b = 9, c=255; unsigned char d= a << 2; //d = 00010100 --> decimal 20 unsigned char e= b << 3; //e = 01001000 --> decimal 72 unsigned char f= c << 2; //f = 11111100 --> decimal 252 printf("d is %u", d); printf("\ne is %u", e); printf("\nf is %u", f); }</pre>	Code 2.28
---	----------------------

Output(s)

d is 20
e is 72
f is 252

Example 2.12 This example illustrates the right shift operation.

```
#include <stdio.h>
int main()
{
    signed char a = -5; // binary representation is 11111011
    signed char b= a >> 3; //b= 11111111 --> decimal -1

    printf("a is %d, ", a);
    printf("a >> 3 is %d", b);
}
```

Code
2.29

Output(s) a is -5, a >> 3 is -1

2.9 Integer Data Type

Integer variables can take values of whole number that have no fractional parts. An integer variable is defined as

```
int var_name;
```

The integer values can be displayed using the format specifier

%i or %d

The size of the integer data can be determined using the **sizeof** operator. The return type of the **sizeof** operator is **long unsigned**; for this reason, we use **%lu** in printf() function to display the return value of the **sizeof** operator.

Example 2.13

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

    printf("%lu", sizeof(num));
}
```

Code
2.30

Output(s) 4

In my computer, the integer size is 4 bytes. This means that an integer is represented by $4 \times 8 = 32$ bits.

Signed representation is used for integer data type. This means that for positive numbers the most significant bit is 0, and for negative numbers the most significant bit is 1, and for negative numbers 2's complement representation is used.

The largest whole number that can be represented by 32-bit integer data type is

01111111111111111111111111111111

and when this number is converted to decimal, we get

$$2^0 + 2^1 + \dots + 2^{31} = \frac{2^{32} - 1}{2 - 1} \rightarrow 2.147.483.647$$

which can be considered as 2 billion to keep in the mind.

The smallest negative number is

10000000000000000000000000000000

which is in 2's complement form. To find the decimal equivalent of this number, we take its 2's complement, convert it to decimal, and put a – sign in front of it. The 2's complement of this number equals itself. Then, the smallest negative number is

$$-2^{32} = -2.147.483.648$$

Thus, a variable of 32-bit integer data type can take values in the range

$$[-2.147.483.648 \dots 2.147.483.647]$$

If an integer value outside this range is assigned to an integer variable, the least significant 32 bits are taken into account from the binary representation of the value, and the rest is truncated.

If you forget how to calculate the maximum and minimum integer values, keep in mind that they can be displayed using INT_MIN and INT_MAX constant parameters, or you can consider hexadecimal values of the maximum and minimum values; the minimum value in hexadecimal is

$$0 \times 80000000$$

and the maximum value in hexadecimal is

$$0 \times 7FFFFFFF$$

Example 2.14

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("Minimum integer value is %d ",INT_MIN);
    printf("\nMaximum integer value is %d ",INT_MAX);
}
```

Code
2.31**Output(s)**

Minimum integer value is -2147483648
 Maximum integer value is 2147483647

2.10 Hexadecimal and Octal Numbers

Consider the decimal number 15. The binary representation of this number in C language is

0B1111 or 0b1111

Octal representation is

017

and hexadecimal representation can be one of these

0XF 0xF 0xf 0Xf 0X0F 0x0F 0x0f 0X0f

Example 2.15

```
#include <stdio.h>

int main(void)
{
    int num1=0xF; // this is a 4-bit number, F indicates 1111 in binary,
    int num2=15; // decimal assignment
    int num3=0B1111; // binary assignment
    int num4=017; // octal assignment

    printf("Numbers are %d %d %d %d ",num1, num2, num3, num4);
}
```

Code
2.32

Output(s) Numbers are 15 15 15 15

The integer numbers can be displayed in hexadecimal format using the format

```
printf(" %x ...%X ...", ...)
```

where capital X is used to display hexadecimal numbers using capital letters

Example 2.16

<pre>#include <stdio.h> int main(void) { int num1=0xF; // this is a 4-bit number, F indicates 1111 in binary, int num2=15; // decimal assignment int num3=0B1111; // binary assignment int num4=017; // octal assignment printf("Numbers are %X %x %X %x ",num1, num2, num3, num4); }</pre>	Code 2.33
--	----------------------

Output(s) Numbers are F f F f

2.11 How Are Integers Stored in Computer Memory?

Refer to Chap. 1 for little-endian and big-endian storage explanations. In this book, we will use little-endian method in our examples.

The address of an integer variable can be obtained using the & symbol in front of the variable name, that is, as

```
&variable_name
```

Example 2.17 Consider the 32-bit integer 0x12345678. When this value is assigned to an integer variable, the address of the stored register can be displayed using Code 2.34.

<pre>int main(void) { int num=0x12345678; printf("Address is %X ",&num); }</pre>	Code 2.34
---	----------------------

Table 2.1 Memory locations

Address	Content
28FEDC	78
28FEDD	56
28FEDE	34
28FEDF	12

In my computer, the output is 28FEDC, and this is the address of the least significant byte, that is, at this location we have the number 0x78. The address 28FEDC is the address of the top register of the memory block, and the address values increase downward as shown in Table 2.1

This means that the 32-bit integer is stored in the memory as shown in Table 2.1.

2.11.1 Short Integer Data Type

Short integers are defined either as

```
short int variable_name;
```

or as

```
short variable_name;
```

The short integer values can be displayed using the format

```
%hi or %hd
```

Short integers are 16-bit integers, that is, 2-byte integers. They are used to represent positive and negative whole numbers. For negative numbers, 2's complement representation is used.

The maximum number that can be represented by short integer data type is

```
0111111111111111
```

and when this number is converted to decimal, we get

$$2^0 + 2^2 + \dots + 2^{14} = \frac{2^{15} - 1}{2 - 1} \rightarrow 32767$$

and the smallest negative number that can be represented by short integer is

1000000000000000000

and this number corresponds to the negative number

$$-2^{15} = -32768$$

Thus, a variable of short integer data type can take values in the range

$$[-32768 \dots 32767]$$

If a number outside this range is assigned to a short integer variable, in this case, only the least significant 16 bits are taken into account and the rest is truncated. Minimum and maximum short integer values are defined as `SHRT_MIN` and `SHRT_MAX`.

Example 2.18

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    printf("Minimum short integer value is %hd ",SHRT_MIN);
    printf("\nMaximum short integer value is %hd ",SHRT_MAX);
}
```

Code
2.35

Output(s)

Minimum short integer value is -32768

Maximum short integer value is 32767

2.12 Why Do We Have Both Integer and Short Integer Data Types?

C language is very widely used in embedded programming. The code written by C language is converted to assembly code and hardware is programmed by the assembly code. If we are dealing with small integer numbers and use integer data type for the variables, this wastes memory use. Each integer value consumes 4-byte memory locations. If all the 4-byte memory locations are NOT used, it is a waste of memory.

2.13 Long Integer and Long-Long Integer Data Types

Long integers are defined either as

```
long int variable_name;
```

or as

```
long variable_name;
```

The long integer values can be displayed using the format

%li or %ld

Long integers are assumed to be 64-bit integers, that is, 8-byte integers. However, depending on the computer, long integers can be 32-bit integers. In this case, there is no difference between an integer and long integer data type.

Long-long integers are defined either as

```
long long int variable_name;
```

or as

```
long long variable_name;
```

The long-long integer values can be displayed using the format

%lli or %lld

Example 2.19

<pre>#include <stdio.h> #include <limits.h> int main(void) { printf("Size of short integer is %lu ",sizeof(short)); printf("\nSize of integer is %lu ",sizeof(int)); printf("\nSize of long integer is %lu ",sizeof(long)); printf("\nSize of long-long integer is %lu ",sizeof(long long)); }</pre>	Code 2.36
---	--

For my computer, the outputs are

Size of short integer is 2

Size of integer is 4

Size of long integer is 4

Size of long-long integer is 8

In my computer, long-long integers are 64-bit integers. The minimum and maximum numbers that can be represented by long-long integers are

$$-2^{63} \text{ and } 2^{63} - 1$$

Example 2.20

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    short num1=0x1234; // 2 bytes in my computer
    int num2=0x12345678; // 4 bytes in my computer
    long num3=0x12345678; // 4 bytes in my computer
    long long num4=0x1234567890ABCDEF; // 8 bytes in my computer

    printf("Short integer is %x ", num1);
    printf("\nInteger is %x ", num2);
    printf("\nLong integer is %x ", num3);
    printf("\nLong-long integer is %llx ", num4);
}
```

**Code
2.37**

Output(s)

Short integer is 1234

Integer is 12345678

Long integer is 12345678

Long-long integer is 1234567890ABCDEF

Note that the format **%x** prints in hexadecimal **without** prefix **0x**, whereas the format **%#x** prints in hexadecimal **with** prefix **0x**.

2.14 Unsigned Integer Data Type

Unsigned integers data types are used for non-negative numbers, that is, the numbers greater than or equal to zero. Variables for unsigned integer data types can be declared as

```
unsigned int variable_name; or unsigned variable_name  
unsigned short variable_name; unsigned long variable_name;  
unsigned long long variable_name;
```

The size of **unsigned int** and **unsigned long int** variables is 4 bytes, and the size of **unsigned short** variables is 2 bytes. However, these number may change on some computers.

The **unsigned short** data types are 16-bit numbers. The smallest number that can be represented by unsigned data type is

0000000000000000

which equals decimal number 0. And the maximum **unsigned short integer** number is

1111111111111111

and when this number is converted to decimal, we obtain

$$2^0 + 2^1 + \dots + 2^{15} = \frac{2^{16} - 1}{2 - 1} \rightarrow 65535$$

Thus, a variable of **unsigned short int** data type can take values in the range

$[0 \dots 65535]$

In a similar manner, we can calculate the range of number that can be represented by **unsigned int** data type as

$$[0 \dots 2^{32} - 1] \rightarrow [0 \dots 4.294.967.295]$$

which is also the range for **unsigned long int** data type for my computer. For **unsigned long long int** data type, the range is

$$[0 \dots 2^{64} - 1] \rightarrow [0 \dots 18.446.744.073.709.551.615]$$

To display the **unsigned integer** data types, we use the format

%hu %u %lu %llu

for **unsigned short**, **unsigned int**, **unsigned long int**, and **unsigned long long int** data types.

For hexadecimal displays, we use the format

%hx %x %lx %llx

and for **%hx**, we can also use **%x** format.

Example 2.21

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    unsigned short int num1=0x1234; // 2 bytes
    unsigned int num2=0x12345678; // 4 bytes
    unsigned long int num3=0x12345678; // 4 bytes
    unsigned long long int num4=0x1234567890ABCDEF; // 8 bytes

    printf("Unsigned short number is %x ", num1);
    printf("\nUnsigned integer number is %x ", num2);
    printf("\nUnsigned long integer number is %x ", num3);
    printf("\nUnsigned long-long integer number is %llx ", num4);
}
```

**Code
2.38**

Output(s)

Unsigned short number is 1234

Unsigned integer number is 12345678

Unsigned long integer number is 12345678

Unsigned long-long integer number is 1234567890ABCDEF

Example 2.22

```
#include <stdio.h>
#include <limits.h>

int main(void)
{
    unsigned short int num1=0x1234; // 2 bytes
    unsigned int num2=0x12345678; // 4 bytes
    unsigned long int num3=0x12345678; // 4 bytes
    unsigned long long int num4=0x1234567890ABCDEF; // 8 bytes

    printf("Unsigned short number is %hu ", num1);
    printf("\nUnsigned integer number is %u ", num2);
    printf("\nUnsigned long integer number is %lu ", num3);
    printf("\nUnsigned long-long integer number is %llu ", num4);
}
```

**Code
2.39**

Output(s)

Unsigned short number is 4660
 Unsigned integer number is 305419896
 Unsigned long integer number is 305419896
 Unsigned long-long integer number is 1311768467294899695

We can use # symbol to print the base symbol before values

Example 2.23

<pre>#include <stdio.h> #include <limits.h> int main(void) { unsigned short int num1=0x1234; // 2 bytes unsigned int num2=0x12345678; // 4 bytes unsigned long int num3=0x12345678; // 4 bytes unsigned long long int num4=0x1234567890ABCDEF; // 8 bytes printf("Unsigned short number is %#hx ", num1); printf("\nUnsigned integer number is %#x ", num2); printf("\nUnsigned long integer number is %#lx ", num3); printf("\nUnsigned long-long integer number is %#llx ", num4); }</pre>	Code 2.40
--	---

Output(s)

Unsigned short number is 0x1234
 Unsigned integer number is 0x 12345678
 Unsigned long integer number is 0x12345678
 Unsigned long-long integer number is 0X1234567890ABCDEF

Example The integer –1 is represented in 2's complement form in 32-bit registers as

11111111111111111111111111111111

That is, 32 ones in computer represent –1. Its hexadecimal representation is

0xFFFFFFFF

That is, 8 F letters represent –1 in hexadecimal base in 2's complement form.

Example 2.24 Find the outputs of Code 2.41.

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

    printf("\na is %X", a);
    printf("\na is %u", a);
    printf("\na is %d", a);
}
```

**Code
2.41**

The integer -1 , which is represented by 32 ones in 2's complement form, is assigned to an unsigned integer variable. The assigned 32 ones are accepted as representing an unsigned number.

The first statement

```
printf("\na is %X", a);
```

prints the hexadecimal number

0xFFFFFFFF

The second statement

```
printf("\na is %u", a);
```

prints the decimal equivalent of

0xFFFFFFFF

which is

4294967295

In the last statement

```
printf("\na is %d", a);
```

the operator `%d` just operates on the value of `a`, and it interprets it as a negative number since the most significant bit is 1, and `%d` is used to print integer values. The output of the last statement is -1 . Thus, the outputs are

```
a is 0xFFFFFFFF
a is 4294967295
a is -1
```

Example 2.25 What are the outputs of Code 2.42?

<pre>#include <stdio.h> int main() { unsigned char a=-1; printf("\na is %x", a); printf("\na is %u", a); printf("\na is %d", a); }</pre>	Code 2.42
--	----------------------

Unsigned char is 8-bit data type, and -1 is represented by 8 ones in 2's complement form

1111111

The first statement

```
printf("\na    is %X", a);
```

interprets the value as signed 32-bit integer, and the binary representation is expanded as

000000000000000000000000001111111

and the printf() function prints the hexadecimal number

0xFF

The second statement

```
printf("\na    is %u", a);
```

interprets the value as 32-bit unsigned integer, and the binary representation is expanded as

```
00000000000000000000000000001111111
```

and the `printf()` function prints the decimal number

```
255
```

In the last statement,

```
printf("\na is %d", a);
```

the operator `%d` just operates on the value of `a`, and it interprets it as a positive number since the most significant bit is 0, and `%d` is used to print integer values. The output of the last statement is 255.

Thus, outputs are

```
a is 0xFF
a is 255
a is 255
```

2.15 Floating-Point Number in C

For real numbers, two data types, **float** and **double**, are used in C programming. For my computer, the size of **float** is 4 bytes and the size of **double** is 8 bytes.

2.15.1 IEEE 754 Floating-Point Standard (Single Precision)

The format of the 32-bit floating-point number is

$$\underbrace{S}_{\text{Sign}} \underbrace{E_7 E_6 E_5 E_4 E_3 E_2 E_1 E_0}_{\text{Exponent}} \underbrace{F_{-1} F_{-2} F_{-3} \cdots F_{-23}}_{\text{Fraction}}$$

whose value is calculated as

$$\text{Value} = (-2S + 1) (2^{127 - \text{Exponent in decimal}}) (1 + \text{Fraction value in decimal})$$

A 32-bit floating-point data type is called single-precision data type, and a 64-bit floating-point data type is called double-precision data type.

For 64-bit floating number, sign has 1 bit, exponent has 11 bits, and fractional part has 52 bits, and the decimal value of the number is calculated using

$$\text{Value} = (-2S + 1)(2^{1023 - \text{Exponent in decimal}})(1 + \text{Fraction value in decimal})$$

The floating-point binary strings that represent negative numbers have 1 as their most significant bit.

Example Calculate the value of the 32-bit floating-point number

1 01111110 1111111111111111111111001

Solution Here, the sign bit is 1, which indicates that the string represents a negative number, and the string is in 2's complement form. To find the real value of the number, we first take the 2's complement of the string

0 10000001 0000000000000000000000000111

and using the formula

$$\text{Value} = (-2S + 1)(2^{127 - \text{Exponent in decimal}})(1 + \text{Fraction value in decimal})$$

we obtain

$$\text{Value} = -(2^{127 - 129})(1 + 2^{-1} + 2^{-2} + 2^{-3}) \rightarrow$$

$$\text{Value} = -0.25(1 + 0.5 + 0.25 + 0.125) \rightarrow$$

$$\text{Value} = -0.46875$$

Example 2.26

```
#include <stdio.h>
int main()
{
    printf("Size of float is %lu", sizeof(float));
    printf("\nSize of double is %lu", sizeof(double));
    printf("\nSize of long double is %lu", sizeof(long double));
}
```

Code
2.43

Output(s)

Size of float is 4

Size of double is 8

Size of long double is 12

To display the float, double, and long double variables, we use the format

%f %lf %Lf

with printf() function as

```
printf( " %f    %lf    %Lf ", variable_f, variable_d, variable_ld );
```

Example 2.27

<pre>#include <stdio.h> int main() { float num1=12.34; double num2=34.67; long double num3=34755.6798; printf("Numbers are %f %lf %Lf", num1, num2, num3); }</pre>	Code 2.44
--	----------------------

Output(s) Numbers are 12.340000 34.670000 34755.679800

2.16 Keyboard Input Using scanf in C

The C function **scanf()** is used to get input from the user via a keyboard. Its use is

```
scanf( " %data_type1    %data_type2 ... ", &var_name1, &var_name1, ... )
```

where the most frequently used formats are

%d	for int
%ld	for long int
%lld	for long long int
%u	for unsigned int
%hi	for short int
%hu	for unsigned short int
%i	for decimal, octal, hexadecimal integers
%o	for unsigned octal integers
%x	for unsigned hexadecimal integers
%f	for float
%lf	for double
%Lf	for long double
%c	for char
%s	for strings

Example 2.28 This program inputs two integers from the user and prints them to the screen.

```
#include <stdio.h>

int main()
{
    int num1, num2;

    printf("Please enter first integer number: ");
    scanf("%d", &num1);

    printf("Please enter second integer number: ");
    scanf("%d", &num2);

    printf("You entered %d and %d", num1, num2);
}
```

Code
2.45

Output(s)

Please enter first integer number: 45
Please enter second integer number: 78
You entered 45 and 78

This code can also be written as

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

    printf("Please enter first and second integer numbers: ");
    scanf("%d %d", &num1, &num2);

    printf("You entered %d and %d", num1, num2);
}
```

Code
2.46

Output(s)

Please enter first and second integer numbers: 56 78
You entered 56 and 78

2.17 Operators in C Programming

Operators can be classified as binary and unary operators. A binary operator operates on two parameters, for instance, + is a binary operator, which is used as

$$x + y$$

where x is called left operand, and y is called right operand.

The operators used in C programming can be classified as

Arithmetic operators

Logical operators

Relational operators

Assignment operators

Cast operators

2.17.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations, and these operators and their functions can be outlined as

- + addition
- subtraction
- * multiplication
- / division
- % remainder

Example 2.29 Write a C program that gets two real numbers from the user and displays the sum of these two numbers.

```
#include <stdio.h>
Code
2.47

int main()
{
    float num1, num2, result;

    printf("Please enter the first real number: ");
    scanf("%f", &num1);

    printf("Please enter the second real number: ");
    scanf("%f", &num2);

    result=num1+num2;
    printf("Sum of the numbers is: %f", result);
    // or we can write printf("Sum of the numbers is: %f", num1+num2);
}
```

Output(s)

Please enter the first real number: 12.7

Please enter the second real number: 87.16

Sum of the numbers is: 99.86

Example 2.30 Write a C program that gets two real numbers from the user and displays the subtraction, multiplication, and division results of these two numbers.

```
#include <stdio.h>
Code
2.48

int main()
{
    float num1, num2, result;

    printf("Please enter the first real number: ");
    scanf("%f", &num1);

    printf("Please enter the second real number: ");
    scanf("%f", &num2);

    printf("Results are: %f    %f    %f", num1-num2, num1/num2, num1*num2);
}
```

Output(s)

Please enter the first real number:

Please enter the second real number:

Sum of the numbers is:

2.17.1.1 Division of Integer Numbers

If two integers are involved in a division operation, then the result is also an integer; otherwise, if one of the operands is a real number, then the result is also a real number. For example,

$$\frac{7}{2} \rightarrow 3 \quad \frac{7}{2.0} \rightarrow 3.5 \quad \frac{7.0}{2} \rightarrow 3.5$$

Example 2.31

```
#include <stdio.h>
Code
2.49

int main()
{
    int a=7, b=2;
    float c=7.0, d=2.0;

    printf("Results are: %d %f %f", a/b, a/d, c/b);
}
```

Output(s) Results are: 3 3.5 3.5

2.17.1.2 Multiplication of Integer Numbers

If two integers are involved in a multiplication operation, then the result is also an integer; otherwise, if one of the operands is a real number, then the result is a real number.

Example 2.32

```
#include <stdio.h>
Code
2.50

int main()
{
    int a=7, b=2;
    float c=7.0, d=2.0;

    printf("Results are: %d %f %f", a*b, a*d, c*b);
}
```

Output(s) Results are: 14 14.0 14.0

2.17.2 Remainder Operator %

The remainder or modulus operator is used to find the remaining number after the division of two integers.

Example 2.33

```
#include <stdio.h>
int main()
{
    int a=14, b=3;
    printf("Remainder is: %d", a%b);
}
```

Code
2.51

Output(s)

Remainder is: 2

The sign of the remainder for

a%b

is the same as the sign of a.

Example 2.34

```
#include <stdio.h>
int main()
{
    int a=14, b=3;
    printf("Remainders are: %d %d %d %d", a%b, a%-b, -a%b, -a%-b);
}
```

Code
2.52

Output(s) Remainders are 2 2 -2 -2

2.17.3 Augmented Assignment Operators

The augmented assignment operators used in C programming are

`+ = - = * = / = % =`

and

<code>a+=b equals a=a+b</code>
<code>a-=b equals a=a-b</code>
<code>a*=b equals a=a*b</code>
<code>a/=b equals a=a/b</code>
<code>a%b equals a=a%b</code>

Example 2.35

```
#include <stdio.h>
int main()
{
    int a=14, b=3;
    printf("a, b are %d %d", a, b);

    a+=b;
    printf("\n after a+=b, a is %d ", a);
}
```

**Code
2.53**

Output(s)

a, b are 14 3
after a+=b, a is 17

2.17.4 Logical Operators

Boolean Data Type

The header file `<stdbool.h>` contains the definition of **bool** data type. It must be included in the C program to use the Boolean data type.

Example 2.36

```
#include <stdio.h>
#include <stdbool.h>

int main()
{
    bool a = true;
    bool b = false;

    printf("True is : %d", a);
    printf("\nFalse is : %d", b);
}
```

**Code
2.54**

Output(s)

True is : 1

False is : 0

There is no format specifier for bool data type for printf() function. We can use

```
printf("%s", x ? "true": "false");
```

to display true and false words for Boolean value.

Example 2.37

```
#include <stdio.h>
#include <stdbool.h>

int main()
{
    bool a = true;

    if (a == true)
    {
        printf("True");
    }
    else
    {
        printf("False");
    }
}
```

**Code
2.55**

Output(s) True

The Boolean equivalent of nonzero numbers is **true** or 1, and the only **false** value is the zero number.

Example 2.38

```
#include <stdbool.h>
int main(void)
{
    int a = 36;
    int b = 48;
    int c = 0;

    printf("a, b, and c are %d %d %d", (bool)(a), (bool)(b), (bool)(c));
}
```

Code
2.56**Output(s) a, b, and c are 1 1 0**

The logical operators can be listed as

&&	logical AND operator
 	logical OR operator
!	unary complement operator

When logical operators are operated on operands, each operand is converted to its Boolean equivalent value and the results are calculated according to

For AND operation

true && true → true
true && false → false
false && false → false

For OR operation

true || true → true
true || false → true
false || false → false

For complement operation

!true → false
!false → true

Example 2.39

<pre>#include <stdio.h> #include <stdbool.h> int main(void) { int a = 36; int b = 48; int c = 0; bool r1 = a && b; bool r2 = a && c; printf("AND results are %d, %d", r1, r2); }</pre>	Code 2.57
---	----------------------------

Output(s) AND results are 1, 0

In Code 2.57, the variables a and b have nonzero values and each of them are evaluated as **true**, and the result of the AND operation for a and b is **true**. The variable c has zero value, and it is evaluated as **false**. The result of the AND operation for a and c is zero.

2.17.5 Bitwise Operators in C

Bitwise operators are completely different operators than the logical operators we covered in the previous section.

Bitwise operators are very frequently used by the embedded software engineers. They are used in hardware programming, such as in microprocessor, and chip programming.

The bitwise operators are

Bitwise AND: &

It operates on 2 bits, and the result is 1 if both bits are 1.

Bitwise OR: |

It operates on 2 bits, and the result is 1 if one of the bits is 1.

Bitwise XOR: ^

It operates on 2 bits, and the result is 1 if bits are different from each other.

Bitwise left shift: <<

It has two operands, the bits of the first operand are left shifted, and the second operand decides the number of places to shift.

Bitwise right shift: >>

It has two operands, the bits of the first operand are right shifted, and the second operand decides the number of places to shift.

Bitwise complement: ~

It takes one number and inverts all bits of it. **Example 2.40** Let us define the variables a and b as

```
unsigned char a = 5, b = 9, c;
```

Since unsigned char data type uses 8 bits, the binary representations of these variables are

```
a = 00000101      b = 00001001
```

Now consider

```
c = a & b
```

The corresponding bits of a and b at the same positions are ANDed; the result is assigned to c. The binary value of c happens to be

```
c = 00000001
```

whose decimal value is 1.

<pre>#include <stdio.h> int main() { unsigned char a = 5, b = 9, c; printf("a = %d, b = %d\n", a, b); c=a&b; printf("c = %d\n", c); }</pre>	Code 2.58
--	----------------------

Output(s)

```
a = 5, b = 9
c = 1
```

Example 2.41 Again let us define the variables a and b as

```
unsigned char a = 5, b = 9, c;
```

Since unsigned char data type uses 8 bits, the binary representations of these variables are

```
a = 00000101      b = 00001001
```

Now consider

```
c = a | b
```

The corresponding bits of a and b at the same positions are ORed; the result is assigned to c.

Using

```
a = 00000101
b = 00001001
```

the binary value of c is calculated as

```
c = 00001101
```

whose decimal value is 13.

Example 2.42

<pre>#include <stdio.h> int main() { unsigned char a = 5, b = 9, c; printf("a = %d, b = %d\n", a, b); c=a b; printf("c = %d\n", c); }</pre>	Code 2.59
--	----------------------

Output(s)

a = 5, b = 9
 c = 13

Example 2.43 Let us define the variables a and b as

```
char a = -6, b = 9, c;
```

The 8-bit representation of -6 in 2's complement form is 11111010 . Char data type uses 8 bits signed representation. The binary representations of the variables are

```
a = 11111010
b = 00001001
```

Now consider

```
c = a | b
```

The corresponding bits of a and b at the same positions are ORed; the result is assigned to c. The binary value of c happens to be

```
c = 11111011
```

which represents a negative number in 2's complement form. The decimal value of this number is

-2 's complement (11111011) = -00000101 which is -5

Using Code 2.60, we get the same results.

<pre>#include <stdio.h> int main() { char a = -6, b = 9, c; printf("a = %d, b = %d\n", a, b); c=a b; printf("c = %d\n", c); }</pre>	Code 2.60
--	--

Output(s)

a = -6 , b = 9
 c = -5

Example 2.44 Let us define the variable a as

```
char a = -6;
```

The 8-bit representation of -6 in 2's complement form is 11111010.

What is displayed with

```
printf("a = %u", a);
```

?

Answer

The negative number -6 is represented by 8-bit string 11111010.

In the statement,

```
printf("a = %u", a);
```

The format `%u` interprets the value as integer, and numbers are represented by 32 bits in integer data type. Then, -6 is represented by a 32-bit string

11111111111111111111111111111111 11111010

and the format `%u` considers this string as an unsigned number, that is, positive number, and its decimal equivalent is printed by

```
printf("a = %u", a);
```

The decimal equivalent of

11111111111111111111111111111111 11111010

can be calculated as

$$2^{31} + 2^{30} + \dots + 2^3 + 2^1$$

which is

4294967290

Code 2.61 gives the same result.

```
#include <stdio.h>
int main()
{
    char a = -6;
    printf("a = %u\n", a);
}
```

**Code
2.61**

Output(s) a = 4294967290

Example 2.45

```
#include <stdio.h>
int main()
{
    char a = 0b01101010;
    char b = 0b00011001;
    // a^b = 0b00110001

    printf("a XOR b is %c\n", a^b);
    printf("a XOR b is %d", a^b);
}
```

**Code
2.62**

Output(s)

a XOR b is : s

a XOR b is : 115

Bitwise left shift: <<

The bitwise left shift operator is used as

a << b;

where a is the first operand whose bits are shifted to the left by b times, and b is the second operand.

Example 2.46

```
#include <stdio.h>
int main()
{
    char a = 0b10000001;
    char b= a << 2;
    printf("a is : %d\n", a);
    printf("b is : %d", b);
}
```

**Code
2.63**

Output(s)

a is : -127

b is : 4

The variable a has binary value 10000001 whose most significant bit is 1, and it indicates a negative number in 2's complement form. The decimal value of this value is

$$-2^7 \text{ complement } (10000001) \rightarrow -0111111 \rightarrow -127$$

When `char b= a << 2` is performed, the bits of a are shifted to the left by two places and b happens to be

00000100

whose decimal value is 4. We get the same results using Code 2.63. Now we modify Code 2.63 and obtain Code 2.64.

```
#include <stdio.h>
int main()
{
    char a = 0b10000001;
    printf("a is: %d\n", a );
    printf("a << 2 is: %d", a << 2);
}
```

**Code
2.64**

Output(s)

a is : -127

a << 2 is : -508

In the statement

```
printf("a << 2 is: %d", a << 2);
```

the value `a << 2` is accepted as a 32-bit integer. It is not truncated to 8 bits as in the previous example, and shifting operation is performed on 32 bits. The value of `a` is written using 32 bits as

and when this string is shifted to the left by two places, we get

which represents a negative integer in 2's complement form, and the decimal equivalent of this integer is -508.

Example 2.47

```
int main()
{
    int a=0b10000000000000000000000000000011; // 32-bit integer

    printf("a is: %d\n", a);

    printf("a << 2 is: %d\n", a << 2);
}
```

Code
2.65

Output(s)

a is : -2147483645

a << is : 12

The 32-bit integer a has binary value

which represents a negative number in 2's complement form since the most significant bit is 1. The decimal equivalent of this negative number is calculated as

When the operation $a \ll 2$ is performed, we get

and this string represents a positive number and decimal equivalent of this number is 12.

Bitwise right shift: >>

The bitwise right shift operator is used as

```
a >> b;
```

where a is the first operand whose bits are shifted to the right by b times, and b is the second operand.

Example 2.48

<pre>#include <stdio.h> int main() { char a = 0b10000000; char b= a >> 2; printf("a is : %d\n", a); printf("b is : %d", b); }</pre>	Code 2.66
---	----------------------

Output(s)

a is : -128

b is : -32

The variable a has binary value 10000000 whose most significant bit is 1, and it indicates a negative number in 2's complement form. The decimal value of this value is

$$-2^7 \text{ complement}(10000000) \rightarrow -10000000 \rightarrow -128$$

When char b= a >> 2 is performed, the bits of a are shifted to the right by two places, and in this shifting operation **the leftmost bit is used for the new positions**, and the shifted bit string happens to be

11100000

which represents a negative number in 2's complement form, and the decimal equivalent of this number is

$$-2^7 \text{ complement}(11100000) \rightarrow -00100000 \rightarrow -32$$

Example 2.49

```
#include <stdio.h>
int main()
{
    char a = 0b10000000;
    printf("a is: %d\n", a );
    printf("a >> 2 is: %d", a >> 2);
}
```

**Code
2.67**

Output(s)

a is : -128
a >>2 is : -32

In the statement

```
printf("a >> 2 is: %d", a >> 2);
```

the value `a >> 2` is accepted as a 32-bit integer. It is not truncated to 8 bits as in the previous example, and shifting operation is performed on 32 bits. The value of a can be written using 32 bits as

11111111111111111111111111 **10000000**

and when this string is shifted to the right by two places, we get

11111111111111111111111111 **100000**

which represents a negative integer in 2's complement form, and the decimal equivalent of this integer is -32.

Bitwise complement: ~

The bitwise complement operator is a unary operator. When bitwise operator is applied on a bit of string, then all the 1's become 0's and vice versa.

For instance,

$\sim 0000\ 0111 \rightarrow 1111\ 1000$

Example 2.50

```
#include <stdio.h>
int main()
{
    int num = 4;
    printf("Bitwise complement of %d is %d", num, ~num);
}
```

**Code
2.68**

Output(s) Bitwise complement of 4 is -5 .

The 32-bit representation of the integer 4 is

0000000000000000000000000000100

and the complement of this string is

11111111111111111111111111011

which represents a number in 2's complement form, and the number represented by this string is

$-2^{\text{'s comp}}(111111111111111111111111011)$

which is

$-0000000000000000000000000000101 \rightarrow -5$

2.17.6 Increment and Decrement Operators

The prefix increment and decrement operators are used as

`++variable_name` `--variable_name`

The postfix increment and decrement operators are used as

`variable_name++` `variable_name--`

The expression with prefix increment

```
result = ++var_name;
```

equals to

```
var_name = var_name + 1;
result = var_name;
```

The expression with postfix increment

```
result = var_name++;
```

equals to

```
result = var_name;
var_name = var_name + 1;
```

Example 2.51 In this example, we use postfix increment.

<pre>#include <stdio.h> int main() { int a = 12, b; b = a++; printf("a = %d, b = % d", a, b); }</pre>	Code 2.69
--	----------------------

Output(s) a = 13, b = 12

Example 2.52 In this example, we use prefix increment.

<pre>#include <stdio.h> int main() { int a = 12, b; b = ++a; printf("a = %d, b = % d", a, b); }</pre>	Code 2.70
--	----------------------

Output(s) a = 13, b = 13

Example 2.53 In this example, we use postfix decrement.

```
#include <stdio.h>
int main()
{
    int a = 12, b;
    b = a--;
    printf("a = %d, b = % d", a, b);
}
```

Code
2.71

Output(s) a = 11, b = 12

Example 2.54 In this example, we use prefix decrement.

```
#include <stdio.h>
int main()
{
    int a = 12, b;
    b = --a;
    printf("a = %d, b = % d", a, b);
}
```

Code
2.72

Output(s) a = 11, b = 11

2.18 Operator Precedence

The precedence of the operators from highest to lowest is shown in Table 2.2.

Example 2.55

```
#include <stdio.h>
int main()
{
    int a = 7, b = 5, c = -1, d = 15;
    if (a < b > c < d) // evaluated as (((a < b)> c) < d)
        printf("Result is TRUE.");
    else
        printf("Result is FALSE.");
}
```

Code
2.73

Table 2.2 Operator precedence

Precedence	Operator	Description	Associativity
1	[]	Array subscripting	Left-to-right
	()	Function call or parentheses	
	++ , --	Postfix increment and decrement	
	- >	Member access through pointer	
	.	Structure and union member access	
2	++ / --	Prefix increment, decrement	Right-to-left
	+ / -	Unary plus, minus	
	(type)	Cast operator	
	! , ~	Logical NOT and bitwise NOT	
	*	Dereference operator	
	&	Address of operator	
	sizeof	Determine size in bytes	
	_Alignof	Alignment requirement	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	Left-to-right
5	<< >>	Bitwise shift left and bitwise shift right	Left-to-right
6	< <=	Relational operators < and <=	Left-to-right
	> >=	Relational operators > and >=	
7	== !=	Relational operators == and !=	Left-to-right
8	&	Bitwise AND	Left-to-right
9	^	Bitwise XOR, i.e., exclusive OR	Left-to-right
10		Bitwise OR or inclusive OR	Left-to-right
11	&&	Logical AND	Left-to-right
12		Logical OR	Left-to-right
13	? :	Ternary conditional	Right-to-left
14	=	Assignment	Right-to-left
	+= -=	Augmented addition and subtraction	
	*= /=	Augmented multiplication and division	
	%= &=	Augmented remainder and bitwise AND	
	^= =	Augmented bitwise exclusive and inclusive OR	
	<<=	Augmented bitwise shift left and augmented bitwise shift right	
	>>=		
15	,	Comma (expression separator)	Left-to-right

Output(s) Result is TRUE.

Problems

1. Write a program that declares four variables of type char, int, float, and double, respectively.
2. Define two variables for float and int data types, initialize them, and print their values using printf() function.

3. What is the output of Code 2.74?

```
#include <stdio.h>
int main(void)
{
    char ch = 0b11011101;
    printf("ch = %d ", ch);
}
```

**Code
2.74**

4. What is the output of Code 2.75?

```
#include <stdio.h>
int main(void)
{
    char ch = 0b1100111111100;
    printf("ch = %d", ch);
}
```

**Code
2.75**

5. Write a program that displays the decimal number -100 in octal and hexadecimal formats.

6. By drawing explain how are the values in

```
int num = 0x12345678
```

stored in memory?

7. The binary representation of a **short int** type number is 1111011100000001.

Write the binary representation of an **int** type number having the same decimal value.

8. Which format is used in **printf()** function to display a **long long int** number?

9. What is the size of a **long long int** data types?

10. What are the outputs of Code 2.76?

```
#include <stdio.h>
int main()
{
    unsigned char a = -8;

    printf("\na is %x", a);
    printf("\na is %u", a);
    printf("\na is %d", a);
}
```

**Code
2.76**

11. What are the outputs of Code 2.77?

```
#include <stdio.h>
int main()
{
    int a = 9, b = 2;
    float c = 11.0, d = 2.0;
    printf("Results are: %d %f %f", a/b, a/d, c/b);
```

Code
2.77

12. What are the outputs of Code 2.78?

```
#include <stdio.h>
int main()
{
    unsigned char a = 8, b = 17, c;
    printf("a = %d, b = %d\n", a, b);
    c=a|b;
    printf("c = %d\n", c);
```

Code
2.78

13. What are the outputs of Code 2.79?

```
#include <stdio.h>
int main()
{
    char a = -13, b = 18, c;
    printf("a = %d, b = %d\n", a, b);
    c=a|b;
    printf("c = %d\n", c);
```

Code
2.79

14. What are the outputs of Code 2.80?

```
#include <stdio.h>
int main()
{
    char a = 0b11000011;
    printf("a is: %d\n", a );
    printf("a >> 2 is: %d", a >> 2);
}
```

Code
2.80

15. What are the outputs of Code 2.81?

```
#include <stdio.h>
int main()
{
    int num = -1;
    printf("Bitwise complement of %d is %d", num, ~num);
}
```

Code
2.81

Chapter 3

Type Conversion in C



3.1 Type Conversion Methods

There are two types of conversion in C:

Implicit conversion (automatic)

Explicit conversion (manual)

3.1.1 *Implicit Conversion*

Implicit conversion is performed automatically by the compiler whenever a value of one type is assigned to another type. For example, if you assign an integer value to a character type or vice versa .

Example 3.1 Code 3.1 illustrates integer to float implicit conversion.

```
#include <stdio.h>
int main()
{
    // Automatic conversion: int to float
    float myFloat = 18; // 18 is converted to 18.000000
    printf("%f", myFloat);
}
```

Code
3.1

Output(s) 18.000000

Example 3.2 Automatic conversion of **float** to **int** is illustrated in this example.

```
#include <stdio.h>
int main()
{
    // Automatic conversion: float to int
    int a = 23.45;

    printf("%d \n", a);

    printf("%d", 23.45);
}
```

**Code
3.2**

Output(s)

23

858993459

In the statement

```
printf("%d", 23.45);
```

the 32-bit binary representation of floating-point number 23.45 is accepted representing an integer, and the binary string is converted to decimal, which equals 858993459.

Example 3.3 The division result of two integers is also an integer.

```
#include <stdio.h>
int main()
{
    double a = 7 / 2;

    printf("%lf \n", a);

    printf("%lf", 7/2);
}
```

**Code
3.3**

Output(s)

3.000000

3.000000

In the statement

```
printf("%lf", 7/2);
```

the integer division 7/2 results in 3, and this is interpreted as a double result, that is, implicit conversion is performed inside printf() function.

Example 3.4 Characters have ASCII values.

Code
3.4

```
#include <stdio.h>

int main()
{
    int x = 12; // integer x

    char y = 'a'; // y is a character, ASCII value of 'a' is 97

    x = x + y; // y is implicitly converted to integer

    printf("x = %d", x);
}
```

Output(s) x = 109

Example 3.5 This example illustrates double to integer implicit conversion.

Code
3.5

```
#include<stdio.h>

int main()
{
    // a is double variable
    double a = 6754.38;

    printf("Double number : %.2lf\n", a);

    // implicit conversion from double to integer
    int b = a;

    printf("Integer number : %d", b);
}
```

Output(s)

Double number : 6754.38

Integer number: 6754

Example 3.6 This example illustrates character to integer implicit conversion.

```
#include<stdio.h>
int main()
{
    // character variable
    char x = 'a';
    printf("Character value : %c\n", x);

    // assign character value to integer variable
    int y = x;

    printf("Integer (ASCII) value : %d", y);
}
```

Code
3.6

Output(s)

Character value : a
Integer (ASCII) value : 97

Example 3.7

```
#include <stdio.h>
int main()
{
    int a = 15;
    char c = 'k'; /* ASCII value is 107 */
    float s;

    s = a + c;

    printf("s = %.1f \n", s);
}
```

Code
3.7

Output(s) s = 122.0

3.1.2 Explicit Conversion

Explicit conversion is written manually as

(type) expression

Example 3.8 This example illustrates explicit conversion of integer to float.

```
#include <stdio.h>
int main()
{
    // Explicit conversion of int to float
    float a = (float) 7 / 2;

    printf("a = %.1f", a);
}
```

Code
3.8

Output(s) a = 3.5

In the statement

```
float a = (float) 7 / 2;
```

the integer 7 is converted to float data type, that is, it becomes 7.0, and when this number is divided by 2, we get 3.5.

Example 3.9

```
#include <stdio.h>
int main()
{
    // Explicit conversion of int to float
    float a = (float) 7 / 2;

    printf("7/2 = %f\n", 7/2);

    printf("a = %f\n", a);

    printf("a = %.1f\n", a);
}
```

Code
3.9

Output(s)

7/2 = 0.000000

a = 3.500000

a = 3.5

In the statement

```
printf("7/2 = %f\n", 7/2);
```

the integer division 7/2 results in 3 and the 32-bit binary string representing the integer 3 is interpreted as representing a floating-point number, and when this string is converted to a floating-point value, a very small number close to 0 is obtained.

Example 3.10 This example illustrates explicit conversion of integer to double.

```
#include <stdio.h>
Code
3.10
int main()
{
    int a= 5;
    int b= 2;
    double c= (double) a / b;

    printf("c = %.1lf", c);
}
```

Output(s) c = 2.5

Example 3.11 This example illustrates explicit conversion of float to integer.

```
#include <stdio.h>
Code
3.11
int main()
{
    float a = 4.7;
    int b = (int)a;

    printf("a = %f\n", a);
    printf("b = %d\n", b);
}
```

Output(s)

a = 4.700000

b = 4

Example 3.12 This example illustrates explicit conversion of double to integer.

```
#include<stdio.h>
Code
3.12
int main()
{
    double a = 3.7;

    // Explicit conversion from double to int
    int b = (int)a + 1;

    printf("b = %d", b);
}
```

Output(s) b = 4

3.2 Information Loss When a Higher-Order Data Is Converted to a Lower-Order Data

Data loss may occur when conversion is performed between different data types. In general, when a higher-order data is converted to a lower-order data as shown in Fig. 3.1, information loss can occur.

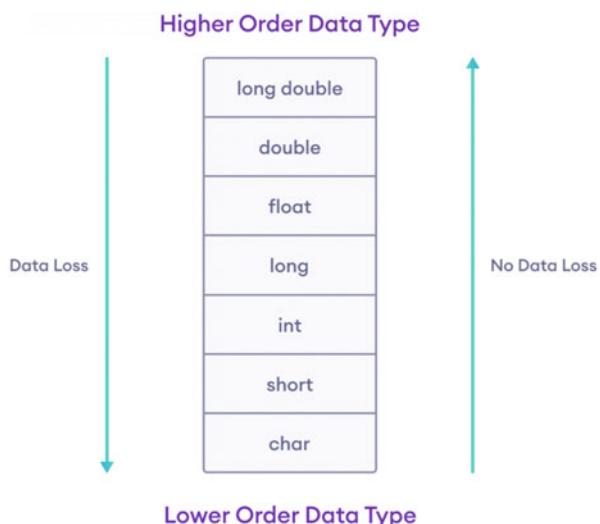
Example 3.13

```
#include <stdio.h>
Code
3.13
int main()
{
    int a = 0B01111111111111000000000000000000;
    short b = a; // b = 0000000000000000
    printf("a = %d \n", a);
    printf("b = %d \n", b);
}
```

Output(s)

a = 2147418112
b = 0

Fig. 3.1 Information loss order



In this code, the integer variable “a” has a 32-bit value, and the decimal equivalent of this value is 2147418112.

Short integer data type is a 16-bit data type. When the assignment

```
short b = a;
```

is performed, only the least significant 16 bits of a is assigned to b, the leftmost 16 bits of b are all 0's. Then, the decimal equivalent of b is 0.

3.3 Information Loss When Conversion Is Performed Between Signed and Unsigned Data Types

When unsigned and signed integers are mixed in arithmetic operations, unexpected results may be obtained.

Example 3.14

<pre>#include <stdio.h> int main() { short int a = -8; unsigned short int b = 4; unsigned short int c = a+b; printf("a = %hd \n", a); printf("b = %hu \n", b); printf("c = %hu \n", c); }</pre>	Code 3.14
---	----------------------

Output(s)

a = -8
b = 4
c = 65532

In the statement

```
short int a = -8;
```

the short integer a is represented by 16 bits in 2's complement as

11111111111111000

When the statement

```
unsigned short int c = a+b;
```

is performed, the value of the variable a is converted to unsigned short int data type, that is, the bit string is accepted representing an unsigned short integer, and the decimal equivalent of the bit string is

$$11111111111111000 \rightarrow 2^{15} + 2^{14} + \dots + 2^3 \rightarrow 65528$$

and when this number is summed by 4, we obtain 65532.

Problems

- What are the outputs of Code 3.15?

<pre>#include <stdio.h> int main() { char a = 0B11100101; unsigned short b = a; printf("a = %d \n", a); printf("b = %hu \n", b); }</pre>	Code 3.15
--	----------------------

- What are the outputs of Code 3.16?

<pre>#include <stdio.h> int main() { char a = -64; unsigned char b = a; printf("a = %d \n", a); printf("b = %d \n", b); }</pre>	Code 3.16
---	----------------------

3. What are the outputs of Code 3.17?

<pre>#include <stdio.h> int main() { printf("%d \n", 9/2); printf("%f \n", 9/2); printf("%f \n", (float)9/2); printf("%f \n", (float)(9/2)); printf("%.1f \n", (float)9/2); printf("%.1f \n", (float)(9/2)); }</pre>	Code 3.17
--	----------------------

4. What are the outputs of Code 3.18?

<pre>#include <stdio.h> int main() { int cnt = 0; for(short int indx = 65535; indx > 0; indx--) { cnt++; } printf("cnt = %d \n", cnt); }</pre>	Code 3.18
---	----------------------

5. What is the output of Code 3.19?

<pre>#include <stdio.h> int main() { short int x = 65530; // integer x char y = 'a'; // character y, ASCII value of 'a' is 97 x = x + y; // y is implicitly converted to short integer printf("x = %d", x); }</pre>	Code 3.19
--	----------------------

Chapter 4

Structures



4.1 Introduction

The C structures are used to group a number of variables employing different data types, and the group is seen as a single data type.

Syntax

The syntax of the structure declaration is

```
struct structureName
{
    dataType1 var1;
    dataType2 var2;

    .
};


```

Note that there is a **semicolon** at the end of structure declaration.

Example 4.1 This example shows how to create a structure data type having two members.

```
#include <stdio.h>           Code
                                4.1
int main()
{
    struct myStruct
    {
        int x;
        double y;
    };
}
```

Fig. 4.1 Defining a structure object, that is, variable

```
#include <stdio.h>
int main()
{
    struct myStruct
    {
        int x;
        double y;
    };
    struct myStruct s;
}
```

Two of them represents a new data type

Variable name

Example 4.2 We can create a variable, that is, object, using the structure we defined in the previous example as in Code 4.2.

```
#include <stdio.h>
```

Code
4.2

```
int main()
{
    struct myStruct
    {
        int x;
        double y;
    };

    struct myStruct s;
}
```

In Fig. 4.1, we explain how to define a variable, that is, object, for a structure data type.

4.2 Initialization of Structure Elements

The initialization of structure elements can be performed either as

```
struct structureName str;

str.variableName1 = value1;
str.variableName2 = value2;
str.variableName3 = value3;
.
.
.
```

or as

```
struct structureName str = {value1, value2, value3,...};
```

Example 4.3 This example illustrates how to initialize the structure elements.

```
#include <stdio.h>
int main()
{
    struct myStruct
    {
        int x;
        double y;
    };

    struct myStruct s = {4, 3.4};

    printf("s.x = %d    ", s.x);
    printf("s.y = %.lf", s.y);
}
```

Code
4.3

Output(s) s.x = 4 s.y = 3.4

4.3 Initialization Using Designated Initializer List

We can initialize the members of the structure using explicit initialization method called designated initializer list method.

Syntax

```
struct structureName str = \
{ .varName1 = value1, .varName2 = value2, .varNameN = valueN};
```

Example 4.4 This example illustrates designated initializer list approach.

```
#include <stdio.h> Code  
4.4  
  
int main()  
{  
    struct myStruct  
    {  
        int x;  
        double y;  
    };  
  
    struct myStruct s = {.x=4, .y=3.4};  
  
    printf("s.x = %d ", s.x);  
    printf("s.y = %.lf", s.y);  
}
```

Output(s) s.x = 4 s.y = 3.4

Example 4.5 This is another example of structure initialization.

```
#include <stdio.h> Code  
4.5  
  
int main()  
{  
    struct myStruct  
    {  
        int x;  
        double y;  
    };  
  
    struct myStruct s;  
  
    s.x = 4;  
    s.y = 3.4;  
}
```

Fig. 4.2 Alternative way for variable definition for structures

```
struct myStruct
{
    int x;
    double y;
} ----- ;
↑
Variables can be
written here
```

```
struct myStruct
{
    int x;
    double y;
}
----- ← Variables can be
----- written here
;
```

Example 4.6 A structure can contain arrays as elements.

```
#include <stdio.h>

int main()
{
    struct myStruct
    {
        int x;
        double y;
        char z[20];
    };

    struct myStruct s = {4, 3.4, "Hello"};
    printf("s.z = %s", s.z);
}
```

Code
4.6

It is possible to define the variables as shown in Fig. 4.2.

Example 4.7 This example illustrates defining structure variables using the method in Fig. 4.2.

```
#include <stdio.h>

int main()
{
    struct myStruct
    {
        int x;
        double y;
    } a, b, c;
}
```

Code
4.7

Example 4.8 We can initialize the variables of the previous example. First, we open a vertical space as in Code 4.8.

```
#include <stdio.h>
Code
4.8
int main()
{
    struct myStruct
    {
        int x;
        double y;
    }

    // open a vertical space here
    ;
}

}
```

Next, we write the variable names as in Code 4.9.

```
#include <stdio.h>
Code
4.9
int main()
{
    struct myStruct
    {
        int x;
        double y;
    }
    a, // write the variable names
    b,
    c;
}
```

In the third step, the variables are initialized as in Code 4.10.

```
#include <stdio.h>
Code
4.10
int main()
{
    struct myStruct
    {
        int x;
        double y;
    }
    a={4, 3.4 },
    b={6, 7.8 },
    c={9, 2.5 };
}
```

In Code 4.11, the values of structure variables are printed.

<pre>#include <stdio.h> int main() { struct myStruct { int x; double y; } a={4, 3.4 }, b={6, 7.8 }, c={9, 2.5 }; printf("a.x = %d a.y = %.1f \n", a.x, a.y); printf("b.x = %d b.y = %.1f \n", b.x, b.y); printf("c.x = %d c.y = %.1f", c.x, c.y); }</pre>	Code 4.11
---	----------------------

Output(s)

a.x = 4 a.y = 3.4
 b.x = 6 b.y = 7.8
 c.x = 9 c.y = 2.5

4.4 Typedef for Structures

Structure variables can be defined using less words with the help of **typedef** reserved word.

Assume that **a** represents a data type. A variable is defined for this data type as

a var_name;

Typedef is used as

typedef a b;

where a is a data type and its new name is b. Then,

a var_name; and **b var_name;**

are the same thing.

We define a variable for a structure as in

struct myStruct s;

where `-struct myStruct-` is the data type. We can use `typedef` for struct as

```
typedef struct myStruct a mySt;
```

then we can define a variable for structure as

```
mySt s;
```

Example 4.9 In this example, we use `typedef` to define a structure variable.

<pre>#include <stdio.h> int main() { typedef struct myStruct mySt; struct myStruct { int x; double y; }; mySt s; }</pre>	Code 4.12
---	----------------------

Example 4.10 Structure objects, that is, variables, can be initialized when they are defined with `typedef` utility.

<pre>#include <stdio.h> int main() { typedef struct myStruct mySt; struct myStruct { int x; double y; }; mySt s = {4, 6.7}; printf("s.x = %d s.y = %.1f \n", s.x, s.y); }</pre>	Code 4.13
---	----------------------

Output(s) s.x = 4 s.y = 6.7

Example 4.11 Structures can be used to define global variables as well.

```
#include <stdio.h>
typedef struct myStruct mySt;

struct myStruct
{
    int x;
    double y;
};

mySt s = {4, 6.7}; // this is a global variable

int main()
{
    printf("s.x = %d  s.y = %.1f \n", s.x, s.y);
}
```

Code
4.14

Output(s) s.x = 4 s.y = 6.7

4.4.1 Alternative Use of `typedef` for Structures

Typedef can also be used as

```
typedef struct{ .... } nameForStructure;
```

Example 4.12 Structure name is shown in red in Code 4.15.

```
#include <stdio.h>
int main()
{
    typedef struct
    {
        int x;
        double y;
    } mySt;

    mySt s = {4, 6.7};

    printf("s.x = %d  s.y = %.1f \n", s.x, s.y);
}
```

Code
4.15

Output(s) s.x = 4 s.y = 6.7

4.5 Nested Structures

It is possible to define a structure inside another one.

Example 4.13 Let us form a nested structure; for this purpose, first, let us define a structure as in Code 4.16.

```
#include <stdio.h>

int main()
{
    struct myStruct1
    {
        int x;

    };
}
```

Code
4.16

Now, we can define another structure and its variable as the member of myStruct1 as in Code 4.17.

```
#include <stdio.h>

int main()
{
    struct myStruct1
    {
        int x;

        struct myStruct2
        {
            int y;
            double z;
        } s2;
    };
}
```

Code
4.17

Finally, we can initialize the structure elements as in Code 4.18.

```
#include <stdio.h>
int main()
{
    struct myStruct1
    {
        int x;

        struct myStruct2
        {
            int y;
            double z;

        } s2;
    };

    struct myStruct1 s1;

    s1.x = 5;
    s1.s2.y = 6;
    s1.s2.z = 3.4;

    printf("x = %d    y = %d    z = %.lf", s1.x, s1.s2.y, s1.s2.z);
}
```

Code
4.18

Output(s) x = 5 y = 6 z = 3.4

Example 4.14 In this example, we use one structure as data type in another structure.

```
#include <stdio.h>
int main()
{
    struct myStruct2
    {
        int y; double z;
    };
    struct myStruct1
    {
        int x;
        struct myStruct2 s2;
    };

    struct myStruct1 s1;

    s1.x = 5;
    s1.s2.y = 6;
    s1.s2.z = 3.4;

    printf("x = %d    y = %d    z = %.lf", s1.x, s1.s2.y, s1.s2.z);
}
```

Code
4.19

Output(s) x = 5 y = 6 z = 3.4

4.6 Structure Copying

Objects or variables belonging to the same structure type can be copied to each other.

Example 4.15 Simple assignment can be used to copy one structure variable to another one.

```
#include <stdio.h>

int main()
{
    struct myStruct
    {
        int x;
        double y;
    };

    struct myStruct s1 = {4, 8.7};

    struct myStruct s2;

    s2=s1;

    printf("s2.x = %d    s2.y = %.1lf", s2.x, s2.y);
}
```

Code
4.20

In this example, s1 and s2 are both variables of the same structure.

Output(s) s2.x = 4 s2.y = 8.7

Structure Pointers

We can define pointers to structures as

```
struct structureName* sptr;
```

Example 4.16 In this example, we explain pointers to structures. First, let us create a structure and initialize a structure as in Code 4.21.

```
#include <stdio.h>

int main()
{
    struct myStruct
    {
        int x;
        double y;
    };

    struct myStruct s = {4,
8.7};
```

Code
4.21

We can define a pointer to the structure variable and access to the elements of structure using → as in Code 4.22.

```
#include <stdio.h>
int main()
{
    struct myStruct
    {
        int x;
        double y;
    };

    struct myStruct s = {4, 8.7};

    struct myStruct* sptr = &s;

    printf("s.x = %d    s.y = %.1lf", sptr->x, sptr->y);
}
```

Code
4.22

Output(s) s.x = 4 s.y = 8.7

4.7 Structures with Self-Referential

The syntax of a self-referral structure is

```
struct structureName
{
    dataType1 var1;
    dataType1 var2;
    .
    .
    struct structureName* sptr;
}
```

Example 4.17 Let us define a self-referential structure as in Code 4.23.

```
#include <stdio.h>
int main()
{
    struct node
    {
        int x;
        double y;
        struct node* next;
    };
}
```

Code
4.23

We can introduce two structure variables and initialize them as in Code 4.24.

```
#include <stdio.h>
int main()
{
    struct node
    {
        int x;
        double y;
        struct node* next;
    };

    struct node n1 = {2, 3.4};
    struct node n2 = {5, 6.7, NULL};
}
```

Code
4.24

The address of the second structure variable is assigned to the pointer of the first structure variable as in Code 4.25.

```
#include <stdio.h>
int main()
{
    struct node
    {
        int x;
        double y;
        struct node* next;
    };

    struct node n1 = {2, 3.4};
    struct node n2 = {5, 6.7, NULL};

    n1.next = &n2;

    printf("n2.x = %d    n2.y = %.1lf", n1.next->x, n1.next->y);
}
```

Code
4.25

Output(s) n2.x = 5 n2.y = 6.7

4.8 Bit Fields

Bit fields are used to prevent the waste usage of the memory. We use sufficient number of bits for the variables in a structure.

The syntax for the use of bit fields is

```
struct structureName
{
    dataType1 varName1: bit-width1;
    dataType2 varName2: bit-width3;
    .
    .
};


```

Example 4.18 First, let us define two structures:

```
#include <stdio.h>

struct myStruct1
{
    int a;
    char c;
};

struct myStruct2
{
    int a : 16;
    char c;
};
```

Code
4.26

The structure variables can be initialized as in Code 4.27.

```
#include <stdio.h>                                         Code
                                                               4.27

struct myStruct1
{
    int a;
    char c;
};

struct myStruct2
{
    int a : 16;
    char c;
};

int main()
{
    struct myStruct1 s1 = {12, 'a'};

    struct myStruct2 s2 = {38, 'b'};

    printf("Size of s1 is : %lu \n", sizeof(s1));

    printf("Size of s2 is : %lu", sizeof(s2));
}
```

Output(s)

Size of s1 is : 8

Size of s2 is : 4

It is obvious from the outputs that the second structure objects use less memory, thus better memory use is achieved.

4.9 Structures as Function Arguments

Structures are data types; the structure variables can be used in function arguments.

Example 4.19 In Code 4.28, we define a structure and introduce the prototype of a function.

```
#include <stdio.h>
struct myStruct
{
    int x;
    double y;
};

void disp(struct myStruct s);
```

**Code
4.28**

The implementation of the function and passing of structure variable as function argument is shown in Code 4.29.

```
#include <stdio.h>

struct myStruct
{
    int x;
    double y;
};

void disp(struct myStruct s);

int main(void)
{
    struct myStruct s = {12, 5.9};

    disp(s);
}

void disp(struct myStruct s)
{
    printf("s.x = %d  s.y = %.1f \n", s.x, s.y);
}
```

**Code
4.29**

Output(s) s.x = 12 s.y = 5.9

Example 4.20 In the previous example, the structure is a global data type; it is seen by every unit of the program. In this example, we define structure inside the main function. Function argument cannot identify the defined data type.

<pre>#include <stdio.h> void disp(struct myStruct s); int main(void) { struct myStruct { int x; double y; }; struct myStruct s = {12, 5.9}; disp(s); } void disp(struct myStruct s) { printf("s.x = %d s.y = %.1f \n", s.x, s.y); }</pre>	Code 4.30
---	----------------------

Output(s) error: type of formal parameter 1 is incomplete, disp(s)

4.10 Structure Padding and Packing in C Programming

The structure shown in Code 4.31 contains two variables belonging to char and int data types.

<pre>struct myStruct { char x; int y; };</pre>	Code 4.31
--	----------------------

In Code 4.32, we calculate the size of a structure variable:

```
#include <stdio.h>

int main(void)
{
    struct myStruct
    {
        char x;
        int y;
    };
    printf("Size of myStruct is : %lu", sizeof(struct myStruct));
}
```

Code
4.32

The output of the Code-4.32 is “Size of myStruct is : 8”

If we look at the inside of the structure in Code 4.32, we see that a **char** and an **int** variable are used. We expect the size of the structure to be

$$\text{sizeof}(\text{char}) + \text{sizeof}(\text{int}) = 1 + 4 \rightarrow 5$$

however, we got the output 8. The reason for this output is that different data types cannot be stored in memory in a consecutive manner. Padding is used between storage regions of different data types in memory.

To use memory for the storage of the structures efficiently, we use pragma directive pack as shown in Code 4.33. Although the use of pack pragma results in efficient memory use, it decreases the performance of the system due to longer memory access time.

```
#include <stdio.h>

#pragma pack(1)

int main(void)
{
    struct myStruct
    {
        char x;
        int y;
    };

    printf("The size of myStruct is : %lu", sizeof(struct myStruct));
}
```

Code
4.33

When Code 4.33 is executed, we get:

The size of myStruct is : 5

4.11 Unions

Unions are very similar to structures, but the size of the union equals to the size of one of its members, and this member has the largest size considering all the members of the union. Definitions and member access rules of the unions are similar to the structures with some differences. Only the first member of the union can be initialized. One storage location is shared by all the members. The value of the last modified element of the union is shared by all the other members.

Example 4.21

```
#include <stdio.h>
int main(void)
{
    struct myStruct
    {
        char a;
        int b;
    };
    union myUnion
    {
        char a;
        int b;
    };

    printf("The size of myStruct is : %lu \n", sizeof(struct myStruct));
    printf("The size of myUnion is : %lu", sizeof(union myUnion));
}
```

Code
4.34**Output(s)**

The size of myStruct is : 8

The size of myUnion is : 4

Example 4.22 If initialization is performed for more than one member, only the first one is initialized.

```
#include <stdio.h>
int main(void)
{
    union myUnion
    {
        char a;
        int b;
    };

    union myUnion u = {'x', 67};

    printf("%c %c %d", u.a, u.b, u.b);
}
```

Code
4.35**Output(s)** x x 120

Example 4.23 One memory location is shared by all the members of the union.

```
#include <stdio.h>
Code
4.36
int main(void)
{
    union myUnion
    {
        char a;
        int b;
    };

    union myUnion u;

    u.a = 'x';

    printf("%c %c %d", u.a, u.b, u.b);
}
```

Output(s) x x 120

Example 4.24 ASCII values can be used for characters.

```
#include <stdio.h>
Code
4.37
int main(void)
{
    union myUnion
    {
        char a;
        int b;
    };

    union myUnion u;

    u.b = 120;

    printf("%c %d %d", u.a, u.a, u.b);
}
```

Output(s) x 120 120

Problems

1. Write a structure that has two integers, one float, and one char variable as structure elements. Declare two variables of this structure and initialize them.
2. Repeat the previous problem but use the keyword `typedef` while declaring the structure.
3. Write a structure called `student_card`. The structure has variables called `st_Number`, `st_Name`, which are used for student number and student name. In

the main part of the program, declare an array of student_card with size 3. Ask the user to enter the values of the each structure element and get them using scanf() function and initialize the structure elements.

4. Declare a structure called complex_number; this structure has two elements called real and imaginary. Write functions that get two complex_number structure variables and find the summation and multiplication of the variables considering the summation and multiplication of classical complex numbers and return the result. Write a test program to initialize the structure elements and test the functions.

5. The person structure is declared in Code 4.38.

```
typedef struct personData
{
    int age;
    char* name;
    char* address;
}person;
```

Code
4.38

Write a test program where variables for this structure are defied and initialized. The members of structure elements are displayed using element access rules.

6. What is missing in Code 4.39?

```
#include <stdio.h>

struct student
{
    int ID;
    char name[40];
};

int main()
{
    struct student s;
}
```

Code
4.39

Chapter 5

Conditional Statements



5.1 Conditional Structure

The conditional statements are the statements that are executed when the condition is true. The syntax of the conditional statement containing the word **if** is

```
if(condition)
{
    // Statements to be executed
    // when condition is true
}
```

The syntax of the conditional statement containing the words **if-else** is

```
if(condition)
{
    // Statements to be executed
    // when condition is true
}
else
{
    // Statements to be executed
    // when condition is false
}
```

A condition is accepted true if it has a nonzero value or if it is a Boolean expression with true value.

Example 5.1 Any integer value different than zero is accepted as true.

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

    if(a)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

Code
5.1

Output(s) Inside if part

Example 5.2 Any float value different than zero is accepted as true.

```
#include <stdio.h>
int main()
{
    float a = -2.5;

    if(a)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

Code
5.2

Output(s) Inside if part

Example 5.3 Any double value different than zero is accepted as true.

```
#include <stdio.h>
int main()
{
    double a = 0.0001;

    if(a)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

Code
5.3

Output(s) Inside if part

Example 5.4 A zero value is accepted as false.

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

    if(b)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

Code
5.4

Output(s) Inside else part

Example 5.5 The header file <stdbool.h> contains Boolean data type.

```
#include <stdbool.h>
#include <stdio.h>

int main()
{
    bool a = true;

    if(a)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

Code
5.5

Output(s) Inside if part

Example 5.6

```
#include <stdbool.h>
#include <stdio.h>

int main()
{
    bool b = false;

    if(b)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

Code
5.6

Output(s) Inside else part

Example 5.7 Logical comparisons produce Boolean results.

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

    if (a < 10)
    {
        printf("a < 10 --> %d \n", a < 10);

        printf("a is less than 10 \n");
    }
    else
    {
        printf("a is greater than 10");
    }
}
```

Code
5.7

Output(s)

a < 10 --> 1
a is less than 10

Example 5.8 Equality comparison symbol, ==, can be used in conditional parts.

```
#include <stdio.h>
int main()
{
    int a = 14;

    if (a % 2 == 0)
    {
        printf("%d is even number", a);

        return 0;
    }

    if (a % 2 == 1)
    {
        printf("%d is odd number", a);

        return 0;
    }
}
```

Code
5.8

Output(s) 14 is even number

In Example 5.8, after `printf()` function, `return 0` is written. When `return 0` is met, the program terminates.

Example 5.9

```
#include <stdio.h>
int main()
{
    int num = 23;
    if (num > 35)
        printf("Inside if part");
    printf("Outside if part");
}
```

**Code
5.9**

Output(s) Outside if part

5.2 Conditional Ladder Structure (-If Ladder)

The template for the conditional ladder structure is shown in Code 5.10.

```
if(condition-1)
{
    statements-1;
}
else if(condition-2)
{
    statements-2;
}
else if(condition-3)
{
    statements-3;
}
.
.
else
{
    statements;
}
```

**Code
5.10**

Example 5.10 This example illustrates the use of the conditional ladder structure.

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

    printf("Please enter an integer : ");
    scanf("%d", &num);

    if(num > 0)
    {
        printf("You entered a positive integer.");
    }
    else if(num < 0)
    {
        printf("You entered a negative integer.");
    }
    else
    {
        printf("You entered zero.");
    }
}
```

Code
5.11

Output(s)

Please enter an integer : -13
You entered a negative integer.

Example 5.11 This example illustrates the use of the conditional ladder structure with combined conditional expressions.

```
#include <stdio.h>
int main()
{
    int avg_mark = 83;

    if (avg_mark <= 100 && avg_mark >= 90)

        printf("A+ Grade");

    else if (avg_mark < 90 && avg_mark >= 80)

        printf("A Grade");

    else if (avg_mark < 80 && avg_mark >= 70)

        printf("B Grade");

    else if (avg_mark < 70 && avg_mark >= 60)

        printf("C Grade");

    else if (avg_mark < 60 && avg_mark >= 50)

        printf("D Grade");

    else

        printf("F Failed");
}
```

Code
5.12

Output(s) A Grade

Example 5.12 This example illustrates the use of the combined conditional expressions in the ladder structure.

```
#include <stdio.h>
Code
5.13

int main()
{
    int num = 66;

    printf("Please enter an integer between 0 and 100 : ");

    scanf("%d", &num);

    // Check if num is between 0 and 25
    if (num >= 0 && num <= 25)
        printf("You entered a number between 0 and 25");

    // Since entered num is not between 0 and 25
    // Check if num is between 26 and 50
    else if (num >= 26 && num <= 50)
        printf("You entered a number between 26 and 50");

    // Since entered num is not between 26 and 50
    // Check if num is between 51 and 75
    else if (num >= 51 && num <= 75)
        printf("You entered a number between 51 and 75");

    // Since entered num is not between 51 and 75
    // It means num is greater than 75
    else
        printf("You entered a number greater than 75");
}
```

Output(s)

Please enter an integer between 0 and 100 : 64

You entered a number between 51 and 75

5.3 Multiconditional Structures

Multiple conditions can be checked using the logical AND and OR operators

&& ||

The AND operator gives true result if all the conditions are true; on the other hand, the OR operator gives true if one of the conditions is true.

Example 5.13

```
#include <stdio.h>
Code
5.14

int main()
{
    int a = 4, b = -1, c = 13;

    int and_result = (a == 4) && (b < 0) && (c > 10);

    printf("and_result = %d", and_result);
}
```

Output(s)

and_result = 1

Example 5.14

```
#include <stdbool.h>
#include <stdio.h>
Code
5.15

int main()
{
    bool a = true;
    bool b = false;

    int c = 12;

    bool and_result = a && (!b) && (c > 10);

    printf("and_result = %d", and_result);
}
```

Output(s) and_result = 1

Example 5.15

```
#include <stdbool.h>
#include <stdio.h>

int main()
{
    bool a = true;
    bool b = false;

    int c = 12;

    bool result1 = a && b || (c > 10);

    bool result2 = a || b && (c > 10);

    printf("result1 = %d\n", result1);
    printf("result2 = %d", result2);
}
```

**Code
5.16**

Output(s)

result1 = 1

result2 = 1

Note that the AND operator has higher precedence than the OR operator.

Example 5.16

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

    printf("Please enter two integers : \n");
    scanf("%d %d", &num1, &num2);

    if (num1 > 0 && num2 > 0)
    {
        printf("You entered two positive integers.");
    }
    else if (num1 < 0 && num2 < 0)
    {
        printf("You entered one positive and one negative integer.");
    }
    else if ((num1 > 0 && num2 < 0) || (num1 < 0 && num2 > 0))
    {
        printf("You entered one positive and one negative integer.");
    }
    else if (num1 == 0 && num2 == 0)
    {
        printf("You entered two zeros.");
    }
    else
    {
        printf("One of the numbers is zero.");
    }
}
```

Code
5.17

Output(s)

Please enter two integers : 5 -12

You entered one positive and one negative integer.

5.4 Syntax of Nested If-Else

The syntax of the nested if-else is shown in Code 5.18 where nesting is done in only if part.

```
if (condition-1)
{
    // Executed when condition-1 is true

    if (condition-2)
    {
        // Executed when condition-2 is true
    }
    else
    {
        // Executed when condition-2 is false
    }

}
else
{
    // Executed when condition-1 is false
}
```

Code
5.18

The syntax of the nested if-else where nesting is done both in if part and else part is shown in Code 5.19.

```
if (condition-1)
{
    // Executed when condition-1 is true
    if (condition-2)
    {
        // Executed when condition-2 is true
    }
    else
    {
        // Executes when condition-2 is false
    }
}
else
{
    // Executed when condition-1 is false
    if (condition-3)
    {
        // Executed when condition-3 is true
    }
    else
    {
        // Executes when condition-3 is false
    }
}
```

Code
5.19

Example 5.17

This example illustrates the use of nested if-else structure.

```
#include <stdio.h>
Code
5.20

int main()
{
    int num;

    printf("Please enter an integer between 0 and 100 : ");

    scanf("%d", &num);

    if (num < 50)
    {
        printf("You entered a number smaller than 50. \n");
        if (num < 25)
            printf("You entered a number smaller than 25. \n");
        else
            printf("You entered a number greater than 25. \n");
    }
    else
    {
        printf("You entered a number greater than 49. \n");
        if (num > 75)
            printf("You entered a number greater than 75. \n");
        else
            printf("You entered a number smaller than 75. \n");
    }
}
```

Output(s)

Please enter an integer between 0 and 100 : 67
 You entered a number greater than 49.
 You entered a number smaller than 75.

5.5 Conditional Operator in C

The syntax of the conditional operator is

(a) ? [executed if a is true] : [executed if a is false];

Note that every value other than zero is accepted as true.
 The conditional operator can also be used as

`variable = condition ? valueT : valueF;`

which is equivalent to

```
(condition) ? (variable = valueT) : (variable = valueF);
```

which means

```
if(condition)
{
    variable = valueT;
}
else
{
    variable = valueF;
}
```

Example 5.18

In Code 5.21, the variable “a” has a value of 7, which is accepted as true.

```
#include <stdio.h>

int main()
{
    int a = 7;

    int b;

    b = (a) ? 6 : 8;

    printf("b is : %d", b);
}
```

**Code
5.21**

Output(s)

b is : 6

Example 5.19 In Code 5.22, the variable “a” has a value of 0, which is accepted as false.

```
#include <stdio.h>

int main()
{
    int a = 0;

    int b;

    b = (a) ? 6 : 8;

    printf("b is : %d", b);
}
```

**Code
5.22**

Output(s) b is : 8

Example 5.20 An alternative form of the conditional operator is illustrated in Code 5.23.

```
#include <stdio.h>
int main()
{
    int a = 5;
    int b;
    (a) ? (b = 6) : (b = 8);
    printf("b is : %d", b);
}
```

Code
5.23

Output(s) b is : 6

Example 5.21

```
#include <stdio.h>
int main()
{
    int a = 0;
    int b;
    (a) ? (b = 6) : (b = 8);
    printf("b is : %d", b);
}
```

Code
5.24

Output(s) b is : 8

Example 5.22 Note that parentheses are necessary when conditional operator is used in the format

```
(condition) ? (variable = valueT) : (variable = valueF);
```

If parentheses are not used, error arises.

```
#include <stdio.h>
int main()
{
    int a = 0;
    int b;
    (a) ? b = 6 : b = 8;
    printf("b is : %d", b);
}
```

**Code
5.25**

Output(s)

main.cpp: In function ‘main’:

main.cpp:9:20: error: lvalue required as left operand of assignment

Example 5.23

```
#include <stdio.h>
int main()
{
    int a = 8, b = 5;
    (a > b) ? printf("a > b") : printf("a < b");
}
```

**Code
5.26**

Output(s) a > b

Example 5.24 The code in the previous example can be written as in Code 5.27.

```
#include <stdio.h>
int main()
{
    int a = 8, b = 5;
    (a > b) ?
        printf("a > b")
        : printf("a < b");
}
```

**Code
5.27**

Output(s) a > b

Example 5.25 A second alternative form of the conditional operator is illustrated in Code 5.28.

```
#include <stdio.h>
int main()
{
    int a = 8, b = 5;

    int larger;

    larger = (a > b) ? a : b;

    printf("Larger number is : %d", larger);
}
```

Code
5.28

Output(s) Larger number is : 8

5.6 switch Statement

The syntax of the switch statement is shown in Code 5.29.

```
switch(variable-name)
{
    case value-1: statements-1;
    break;

    case value-2: statements-2;
    break;
    .
    .
    .
    case value-N: statements-N;
    break;

    default: statements;
}
```

Code
5.29

In Code 5.30, when the value of the variable matches one of the listed values, the corresponding statements are executed.

Example 5.26 In this example, an integer is entered by the user and a message is printed according to the entered value.

```
#include <stdio.h>
Code
5.30

int main()
{
    int num;

    printf("Enter an integer between 1 and 4, \n"
           "1 and 4 are included : \n");

    scanf("%d", &num);

    switch (num)
    {
        case 1:
            printf("You entered 1.");
            break;

        case 2:
            printf("You entered 2.");
            break;

        case 3:
            printf("You entered 3.");
            break;

        default:
            printf("You entered 4.");
            break;
    }
}
```

Output(s)

Enter an integer between 1 and 4,

1 and 4 are included : 3

You entered 3

Example 5.27 Between **case** and **break** keywords, more than one line can be written.

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

    printf("Enter an integer between 1 and 4, \n"
           "1 and 4 are included : ");

    scanf("%d", &num);

    switch (num)
    {
        case 1:
            printf("You entered 1.\n");
            printf("Thank you.");
            break;

        case 2:
            printf("You entered 2.\n");
            printf("Thank you.");
            break;

        case 3:
            printf("You entered 3.\n");
            printf("Thank you.");
            break;

        default:
            printf("You entered 4.\n");
            printf("Thank you.");
            break;
    }
}
```

Code
5.31

Output(s)

Enter an integer between 1 and 4,
1 and 4 are included : 3
You entered 3.
Thank you.

Example 5.28 We can use curly parentheses between **case** and **break** keywords.

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

    printf("Enter an integer between 1 and 4, \n"
           "1 and 4 are included : ");

    scanf("%d", &num);

    switch (num)
    {
        case 1:
        {
            printf("You entered 1.\n");
            printf("Thank you.");
            break;
        }

        case 2:
        {
            printf("You entered 2.\n");
            printf("Thank you.");
            break;
        }

        case 3:
        {
            printf("You entered 3.\n");
            printf("Thank you.");
            break;
        }

        default:
        {
            printf("You entered 4.\n");
            printf("Thank you.");
            break;
        }
    }
}
```

Code
5.32

Output(s)

Enter an integer between 1 and 4,

1 and 4 are included : 3

You entered 3.

Thank you.

Example 5.29 If **break** of a **case** is not written, the next line is also executed.

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

    printf("Enter an integer between 1 and 4, \n"
           "1 and 4 are included : ");

    scanf("%d", &num);

    switch (num)
    {
        case 1:
            printf("You entered 1.\n");
            break;

        case 2:
            printf("You entered 2.\n");
            /* */
        case 3:
            printf("You entered 3.\n");
            break;

        default:
            printf("You entered 4.\n");
            break;
    }
}
```

Code
5.33

Output(s)

Enter an integer between 1 and 4,
1 and 4 are included : 2
You entered 2.
You entered 3.

Example 5.30 If **break** of a **case** is not written, the rest of the code is executed until **break** keyword is met.

```
#include <stdio.h>

int main()
{
    int num;

    printf("Enter an integer between 1 and 4, \n"
           "1 and 4 are included : ");

    scanf("%d", &num);

    switch (num)
    {
        case 1:
            printf("You entered 1.\n");
            break;

        case 2:
            printf("You entered 2.\n");

        case 3:
            printf("You entered 3.\n");

        default:
            printf("You entered 4.\n");
            break;
    }
}
```

Code
5.34

Output(s)

Enter an integer between 1 and 4,

1 and 4 are included : 2

You entered 2.

You entered 3.

You entered 4.

Problems

1. What is the output of Code 5.35?

```
#include <stdio.h>
int main()
{
    char a = '0';

    if(a)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

**Code
5.35**

2. What is the output of Code 5.36?

```
#include <stdio.h>
int main()
{
    char a = '0';

    if(6 < 7 > -5)
    {
        printf("Inside if part \n");
    }
    else
    {
        printf("Inside else part \n");
    }
}
```

**Code
5.36**

3. Which header file do we need to include in our source file to be able to write a statement like

```
bool a = true;  
?
```

4. Write a C program that inputs an integer from the user and determines whether the number is even or odd.
5. Write a C program that inputs an integer from the user and displays a “1-digit number” if it is a 1-digit integer; displays a “2-digit number” if it is a 2-digit integer; and displays a “3-digit number” if it is a 3 digit-integer; otherwise, it displays “a larger number.”

6. Write a program that inputs an integer from a user and determines whether the number is a prime number or not.
7. Write a C program that inputs three numbers from the user and determines and displays the largest of three numbers.
8. Write a C program that inputs five numbers from the user and determines and displays the largest of five numbers.
9. Write a C program that determines whether a triangle is equilateral, isosceles, or scalene. Input the side lengths from the user.
10. What is the output of Code 5.37?

```
#include <stdio.h>
int main()
{
    int a = -3;
    int b;
    b = (a) ? 9 : 17;
    printf("b is : %d", b);
}
```

Code
5.37

11. What is the output of Code 5.38?

```
#include <stdio.h>
int main()
{
    int a = 0;
    int b;
    (a) ? (b = 45) : (b = 92);
    printf("b is : %d", b);
}
```

Code
5.38

12. Write a program using the switch statement that inputs digits from 1 to 5 and displays the digit entered by the user.

Chapter 6

Loop Statements



6.1 The For-Loop

The structure of the for-loop is

```
for(st0; st1; st2)
{
    st3
}
```

where st0, st1, st2, and st3 are the statements, and they are executed in rowwise as

```
st0, st1, st2, st3
st1, st2, st3
st1, st2, st3
st1, st2, st3
....
```

until the loop terminates.

The formal syntax of the for-loop is

```
for(initialization statements; loop execution condition; update statements)
{
    // Loop statements
}
```

where at initialization statements, we have variable initialization expressions, such as

```
int a = 0;
```

loop execution condition is a Boolean expression, for example,

a < 3

update statements are usually increment or decrement statements, for example,

a++ or a--

however, other updates like $a=a+2$, $a*=2$ are also possible.

Example 6.1 Let us write a code involving the for-loop. For this purpose, we first write the for-loop structure as in Code 6.1.

```
#include <stdio.h>
int main()
{
    for ( ; ; )
    {
    }
}
```

**Code
6.1**

Initialization is written as in Code 6.2.

```
#include <stdio.h>
int main()
{
    for (int a = 0; ; )
    {
    }
}
```

**Code
6.2**

Conditional expression is added in Code 6.3.

```
#include <stdio.h>
int main()
{
    for (int a = 0; a < 3; )
    {
        }

}
```

Code
6.3

Finally, update is written in Code 6.4.

```
#include <stdio.h>
int main()
{
    for (int a = 0; a < 3; a++)
    {
        }

}
```

Code
6.4

Two printf() functions are written in Code 6.5.

```
#include <stdio.h>
int main()
{
    for (int a = 0; a < 3; a++)
    {
        printf("Inside for-loop\n\n");
    }

    printf("\nOutside for-loop");
}
```

Code
6.5

To understand the operation of the for-loops, let us use printf() functions inside the for header. For this purpose, let us use back slash and obtain three lines as in Code 6.6.

```
#include <stdio.h>
int main()
{
    for (int a = 0 ;\n
         a < 3;\n
         a++)
    {
        printf("Inside for-loop\n\n");
    }

    printf("\nOutside for-loop");
}
```

**Code
6.6**

The first printf() function is shown in Code 6.7.

```
#include <stdio.h>
int main()
{
    for (int a = 0 && printf("Initializatin is performed, a = 0\n");\n
         a < 3;\n
         a++)
    {
        printf("Inside for-loop\n\n");
    }

    printf("\nOutside for-loop");
}
```

**Code
6.7**

The second printf() function is used in Code 6.8.

```
#include <stdio.h>
int main()
{
    for (int a = 0 && printf("Initializatin is performed, a = 0 \n");\n
         printf("a = %d, a < 3 is %s \n",a, a < 3 ? "true" : "false") && a < 3;\n
         a++)
    {
        printf("Inside for-loop\n\n");
    }

    printf("\nOutside for-loop");
}
```

**Code
6.8**

The third printf() function is used in Code 6.9.

```
#include <stdio.h>
int main()
{
    for (int a = 0 && printf("Initializatin is performed, a = 0 \n"); \
        printf("a = %d, a < 3 is %s \n", a, a < 3 ? "true" : "false") && a < 3; \
        printf("a++ --> a is incremented\n") && a++)
    {
        printf("Inside for-loop\n\n");
    }

    printf("\nOutside for-loop");
}
```

Code
6.9

Output(s) When the program is executed, we get the following output, which explains the operation of the for-loop:

a = 0, a < 3 is true

Inside for-loop

a++ --> a is incremented

a = 1, a < 3 is true

Inside for-loop

a++ --> a is incremented

a = 2, a < 3 is true

Inside for-loop

a++ --> a is incremented

a = 3, a < 3 is false

Outside for-loop

Example 6.2 We can write the update part of the for-loop in the body of the for-loop.

```
#include <stdio.h>
int main()
{
    for (int a = 0; a < 3; )
    {
        printf("Inside for-loop, a = %d \n",a);
        a++;
    }

    printf("\nOutside for-loop");
}
```

Code
6.10

Output(s)

Inside for-loop, a = 0

Inside for-loop, a = 1

Inside for-loop, a = 2

Outside for-loop

Example 6.3 Update can be a decrement operation. In Code 6.11, decrement is used for update operation.

```
#include <stdio.h>
int main()
{
    for (int a = 2; a >= 0; a--)
    {
        printf("Inside for-loop, a = %d \n",a);
    }

    printf("\nOutside for-loop");
}
```

Code
6.11

Output(s)

Inside for-loop, a = 2

Inside for-loop, a = 1

Inside for-loop, a = 0

Outside for-loop

Example 6.4 The variable defined at the initialization part of the for-loop is not accessible outside the body of the for-loop.

```
#include <stdio.h>
int main()
{
    for (int a = 0; a < 3; a++)
    {
        printf("a = %d \n",a);
    }

    printf("a = %d \n",a); // a is not accessible, error
}
```

Code
6.12

Output(s)

[Error] ‘a’ undeclared (first use in this function)

Example 6.5 A local variable outside the for-loop structure can be used for counting index as in Code 6.13.

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

    for (a = 0; a < 3; a++)
    {
        printf("Inside for-loop, a = %d \n",a);
    }

    printf("\nOutside for-loop, a = %d \n",a);
}
```

Code
6.13

Output(s)

Inside for-loop, a = 0

Inside for-loop, a = 1

Inside for-loop, a = 2

Outside for-loop, a = 3

Example 6.6 The initialization part of the for-loop can be written outside the for-loop structure as in Code 6.14.

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

    for ( ; a < 3; a++)
    {
        printf("Inside for-loop, a = %d \n",a);
    }
    printf("\nOutside for-loop");
}
```

Code
6.14

Output(s)

Inside for-loop, a = 0
 Inside for-loop, a = 1
 Inside for-loop, a = 2

Outside for-loop, a = 3

Example 6.7 In Code 6.15, both initialization and update parts of the for-loop are written outside the for-loop header.

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

    for ( ; a < 3; )
    {
        printf("Inside for-loop, a = %d \n",a);
        a++;
    }
    printf("\nOutside for-loop");
}
```

Code
6.15

Output(s)

Inside for-loop, a = 0
 Inside for-loop, a = 1
 Inside for-loop, a = 2

Outside for-loop, a = 3

Example 6.8 More than one variable can be used at the header of the for-loop.

```
#include <stdio.h>
int main()
{
    for (int a = 0, b = 0; a < 3, b < 3; a++, b++)
    {
        printf("Inside for-loop, a = %d ", a);
        printf("b = %d \n", b);
    }
    printf("\nOutside for-loop");
}
```

Code
6.16

Output(s)

Inside for-loop, a = 0 b = 0

Inside for-loop, a = 1 b = 1

Inside for-loop, a = 2 b = 2

Outside for-loop

Example 6.9 Conditional part of the for-loop can contain multiconditional expressions.

```
#include <stdio.h>
int main()
{
    for (int a = 0, b = 0; a < 3 && b < 3; a++, b++)
    {
        printf("Inside for-loop, a = %d ", a);
        printf("b = %d \n", b);
    }
    printf("\nOutside for-loop");
}
```

Code
6.17

Output(s)

Inside for-loop, a = 0 b = 0

Inside for-loop, a = 1 b = 1

Inside for-loop, a = 2 b = 2

Outside for-loop

Example 6.10 The logical operator `&&` produces false, if one of the operands evaluates to be false.

```
#include <stdio.h>
Code
6.18

int main()
{
    for (int a = 0, b = 0; a < 3 && b < 785; a++, b = b+2)
    {
        printf("Inside for-loop, a = %d ",a);
        printf("b = %d \n",b);
    }
    printf("\nOutside for-loop");
}
```

Output(s)

Inside for-loop, a = 0 b = 0

Inside for-loop, a = 1 b = 2

Inside for-loop, a = 2 b = 4

Outside for-loop

Example 6.11 In this example, logical OR `||` is used at the conditional part of the for-loop.

```
#include <stdio.h>
Code
6.19

int main()
{
    for (int a = 0, b = 0; a < 3 || b < 4; a++, b++)
    {
        printf("Inside for-loop, a = %d ",a);

        printf("b = %d \n",b);
    }
    printf("\nOutside for-loop");
}
```

Output(s)

Inside for-loop, a = 0 b = 0

Inside for-loop, a = 1 b = 1

Inside for-loop, a = 2 b = 2

Inside for-loop, a = 3 b = 3

Outside for-loop

Example 6.12 Different updates can be used at the update part of the for-loop.

```
#include <stdio.h>
int main()
{
    for (int a = 0, b = 2; a < 3 || b < 4; a++, b = b+5)
    {
        printf("Inside for-loop, a = %d ",a);
        printf("b = %d \n",b);
    }
    printf("\nOutside for-loop");
}
```

Code
6.20

Output(s)

Inside for-loop, a = 0 b = 2

Inside for-loop, a = 1 b = 7

Inside for-loop, a = 2 b = 12

Outside for-loop

Example 6.13 Floating point variables can be used at the header of the for-loop.

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

    for (a = 0, b = 2.7; a*b < 25.6; a++, b++)
    {
        printf("Inside for-loop, a = %d ",a);
        printf("b = %.1f \n",b);
    }

    printf("\nQuitted for-loop\n\n");

    printf("Outside for-loop, a = %d ",a);
    printf("b = %.1f \n",b);
}
```

Code
6.21

Output(s)

Inside for-loop, a = 0 b = 2.7

Inside for-loop, a = 1 b = 3.7

Inside for-loop, a = 2 b = 4.7

Inside for-loop, a = 3 b = 5.7

Quitted for-loop

Outside for-loop, a = 4 b = 6.7

Example 6.14 In this example, it is shown that two for-loops can employ the same parameter at their headers.

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

    for (a = 0; a < 3; a++)
    {
        printf("Inside for-loop-1, a = %d \n",a);
    }

    printf("\nOutside for-loop-1, a = %d \n\n",a);

    for (a = 8; a < 11; a++)
    {
        printf("Inside for-loop-2, a = %d \n",a);
    }

    printf("\nOutside for-loop-2, a = %d \n",a);
}
```

Code
6.22

Output(s)

Inside for-loop-1, a = 0

Inside for-loop-1, a = 1

Inside for-loop-1, a = 2

Outside for-loop-1, a = 3

Inside for-loop-2, a = 8

Inside for-loop-2, a = 9

Inside for-loop-2, a = 10

Outside for-loop-2, a = 11

Example 6.15 Infinite loop can be created using the structure in Code 6.23.

```
#include <stdio.h>
int main()
{
    for ( ; ; )
    {
        printf("Infinite loop\n");
    }
}
```

Code
6.23

Output(s)

Infinite loop

Infinite loop

Infinite loop

.

.

.

Example 6.16 Another infinite loop is written in Code 6.24.

```
#include <stdio.h>
int main()
{
    for ( ; 7.34 ; )
    {
        printf("Another infinite loop\n");
    }
}
```

Code
6.24

Output(s)

Another infinite loop

Another infinite loop

Another infinite loop

.

.

.

Example 6.17 In Code 6.25, the for-loop header has only conditional part, and the condition becomes false when the value of “b” equals 3.

```
#include <stdio.h>
#include <stdbool.h>

int main()
{
    bool a = true;
    int b = 0;

    for (; a ; )
    {
        printf("Inside for-loop, b = %d \n",b);

        b++;

        if(b > = 3)
            a = false;
    }

    printf("\nOutside for-loop, b = %d \n",b);
}
```

Code
6.25

Output(s)

Inside for-loop, b = 0
 Inside for-loop, b = 1
 Inside for-loop, b = 2

Outside for-loop, b = 3

6.1.1 Nested For-Loop

For-loops can be used in a nested manner. The structure of nested for-loops is shown in Code 6.26.

```
for(init1; condition1; update1)
{
    // outer loop statements1

    for(init2; condition2; update2)
    {
        // inner loop statements
    }

    // outer loop statements2
}
```

Code
6.26

Example 6.18 Let us write a nested for-loop. For this purpose, first let us place the first for-loop as in Code 6.27.

```
#include <stdio.h>
int main()
{
    for(int idxN = 0; idxN < 3; idxN++)
    {
        }
}
```

Code
6.27

We write a printf() function inside the first loop as in Code 6.28.

```
#include <stdio.h>
int main()
{
    for(int idxN = 0; idxN < 3; idxN++)
    {
        printf("\n\nInside outer loop, idxN = %d", idxN);
    }
}
```

Code
6.28

Inside the first for-loop, we place the structure of the second for-loop as in Code 6.29.

```
#include <stdio.h>
int main()
{
    for(int idxN = 0; idxN < 3; idxN++)
    {
        printf("\n\nInside outer loop, idxN = %d", idxN);
        for(int idxM = 0; idxM < 3; idxM++)
        {
            }
    }
}
```

Code
6.29

Two printf() functions, one inside the second for-loop and the other one outside both for-loops, are written as in Code 6.30.

```
#include <stdio.h>
int main()
{
    for(int idxN = 0; idxN < 3; idxN++)
    {
        printf("\n\nInside outer loop, idxN = %d", idxN);
        for(int idxM = 0; idxM < 3; idxM++)
        {
            printf("\nInside inner loop, idxM = %d", idxM);
        }
    }
    printf("\n\nOutside outer loop");
}
```

Code
6.30

Output(s)

Inside outer loop, idxN = 0

Inside inner loop, idxM = 0

Inside inner loop, idxM = 1

Inside inner loop, idxM = 2

Inside outer loop, idxN = 1

Inside inner loop, idxM = 0

Inside inner loop, idxM = 1

Inside inner loop, idxM = 2

Inside outer loop, idxN = 2

Inside inner loop, idxM = 0

Inside inner loop, idxM = 1

Inside inner loop, idxM = 2

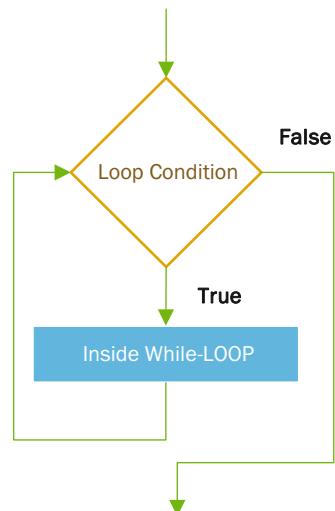
Outside outer loop

6.2 The While-Loop

In the while-loop, first initialization is performed, then loop condition is checked, and loop body is executed; updating can be performed inside the loop body.

The syntax of the while-loop is

Fig. 6.1 The while-loop execution logic



```
// Initialization statements

while (condition)
{
    // Loop statements
}
```

The statements inside the while-loop parentheses are executed as long as the condition is true. The operation of the while-loop is illustrated in Fig. 6.1.

Example 6.19 This example illustrates the use of the while-loop.

```
#include <stdio.h>

int main()
{
    int a = 2;

    while(a < 9)
    {
        printf("Inside while-loop, a = %d\n", a);

        a = a+2;
    }

    printf("\nOutside while-loop, a = %d ", a);
}
```

Code
6.31

Output(s)

Inside while-loop, a = 2

Inside while-loop, a = 4

Inside while-loop, a = 6

Inside while-loop, a = 8

Outside while-loop, a = 10

Example 6.20 Inside the while-loop, we can use conditional statements as in Code 6.32.

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

    while(a)
    {
        printf("Inside while-loop, a = %d \n", a);

        if(a == 8)
        {
            a = 0;
        }
        else
            a = a+2;
    }
    printf("\nOutside while-loop, a = %d ", a);
}
```

**Code
6.32**

Output(s)

Inside while-loop, a = 2

Inside while-loop, a = 4

Inside while-loop, a = 6

Inside while-loop, a = 8

Outside while-loop, a = 0

Example 6.21 The while-loop condition can contain logical expressions.

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

    while(a < 10)
    {
        printf("Inside while-loop, a = %d\n", a);

        a = a+2;
    }
    printf("\nOutside while-loop, a = %d", a);
}
```

**Code
6.33**

Output(s)

Inside while-loop, a = 2

Inside while-loop, a = 4

Inside while-loop, a = 6

Inside while-loop, a = 8

Outside while-loop, a = 10

Example 6.22 It is possible to define an infinite while-loop as in Code 6.34.

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

    while(1)
    {
        printf("a = %d\n", a);

        a = a+2;
    }
}
```

**Code
6.34**

Output(s)

a = 2

a = 4

a = 6

.

.

.

Example 6.23 Multiconditional expressions may appear inside the while-loop.

```
#include <stdio.h>
Code
6.35

int main()
{
    int a = 1;

    while(a < 20)
    {

        if(a % 7 == 0 || a % 9 == 0)
            printf("Inside while-loop, a = %d\n", a);
        a++;
    }

    printf("\nOutside while-loop, a = %d ", a);
}
```

Output(s) -6.xx

Inside while-loop, a = 7
 Inside while-loop, a = 9
 Inside while-loop, a = 14
 Inside while-loop, a = 18

Outside while-loop, a = 20

6.2.1 Nested While-Loop

The structure of the nested while-loop is shown in Code 6.36.

```
while(condition1)
{
    // outer loop statements1

    while(condition2)
    {
        // inner loop statements
    }

    // outer loop statements2
}
```

Code
6.36

Example 6.24 Let us form a nested while-loop. For this purpose, we write the first while-loop as in Code 6.37.

```
#include <stdio.h>

int main()
{
    int idxN = 0;

    while(idxN < 3)
    {
        }

    }
}
```

Code
6.37

We add printf and update statements as in Code 6.38.

```
#include <stdio.h>

int main()
{
    int idxN = 0, idxM = 0;

    while(idxN < 3)
    {
        printf("\n\nInside outer loop, idxN = %d", idxN);

        idxN++;
    }
}
```

Code
6.38

The structure of the second while-loop is placed into the first while-loop as in Code 6.39.

```
#include <stdio.h>

int main()
{
    int idxN = 0, idxM = 0;

    while(idxN < 3)
    {
        printf("\n\nInside outer loop, idxN = %d", idxN);

        while(idxM < 3)
        {

            }

        idxN++;
    }
}
```

Code
6.39

We add printf and parameter update expressions inside the second while-loop, and add one printf() function outside both loops as in Code 6.40.

```
#include <stdio.h>
int main()
{
    int indxN = 0, indxM = 0;

    while(indxN < 3)
    {
        printf("\n\nInside outer loop, indxN = %d", indxN);

        while(indxM < 3)
        {
            printf("\nInside inner loop, indxM = %d", indxM);
            indxM++;
        }

        indxN++;
        indxM = 0;
    }

    printf("\n\nOutside outer loop");
}
```

**Code
6.40**

Output(s)

Inside outer loop, indxN = 0

Inside inner loop, indxM = 0

Inside inner loop, indxM = 1

Inside inner loop, indxM = 2

Inside outer loop, indxN = 1

Inside inner loop, indxM = 0

Inside inner loop, indxM = 1

Inside inner loop, indxM = 2

Inside outer loop, indxN = 2

Inside inner loop, indxM = 0

Inside inner loop, indxM = 1

Inside inner loop, indxM = 2

Outside outer loop

6.3 The Do-While Loop

The syntax of the do-while loop is

```
// Initialization statements

do
{
    // Loop statements

} while (condition);
```

Loop statements are executed as long as the condition is true. Note that the do-while loop is executed at least once.

Example 6.25 Do-while loop is executed at least once.

```
#include <stdio.h>                                         Code
6.41

int main()
{
    int a = 0;

    do
    {
        printf("do-while Loop is executed at least once.\n");
    } while(a);

    printf("Outside do-while loop.");
}
```

Output(s)

do-while Loop is executed at least once.

Outside do-while loop.

Example 6.26 Loop condition can be a logical expression.

```
#include <stdio.h>                                         Code
6.42

int main()
{
    int a = 2;

    do
    {
        printf("Inside do-while loop, a = %d\n", a);

        a = a+2;

    } while(a < 5);

    printf("\nOutside do-while loop, a = %d\n", a);
}
```

Output(s)

Inside do-while loop, a = 2
 Inside do-while loop, a = 4
 Inside do-while loop, a = 6
 Inside do-while loop, a = 8

Outside do-while loop, a = 10

Example 6.27 In this example, we form a product table using the do-while loop.

```
#include <stdio.h>
Code
6.43

int main()
{
    int a = 7, b = 1;

    do
    {
        printf("%d      x %5d      = %5d \n", a, b, a * b);

        b++;
    }
    while (b < 5);

}
```

Output(s)

7 x 1 = 7
 7 x 2 = 14
 7 x 3 = 21
 7 x 4 = 28

6.4 Continue Statement

Continue statement is used to skip the rest of the statements when a condition is met, and program execution returns to the beginning of the loop.

The use of the continue statement in the while-loop is illustrated in Fig. 6.2.

Fig. 6.2 Behavior of the continue statement in the while-loop

```
→ while (condition1)
{
    // statements
    if (condition2)
    {
        continue;
    }
    // statements
}
```

```

→ for(initialization; condition1; update)
{
    // statements
    if(condition2)
    {
        continue;
    }
    // statements
}

```

Fig. 6.3 Behavior of the continue statement in the for-loop**Fig. 6.4** Behavior of the continue statement in the do-while loop

```

→ do
{
    // statements
    if(condition1)
    {
        continue;
    }
    // statements
} while(condition2);

```

The use of the continue statement in the for-loop is illustrated in Fig. 6.3.

The use of the continue statement in the do-while loop is illustrated in Fig. 6.4.

Example 6.28 In this example, we use the continue statement in a for-loop.

<pre> #include <stdio.h> int main() { for(int a = 0; a < 7; a++) { if(a == 3) continue; printf("a=%d ", a); } } </pre>	Code 6.44 x
--	----------------------------------

When $a==3$, the `printf()` function is not executed, program execution goes to the beginning of the loop.

Output(s)

$a = 0 \quad a = 1 \quad a = 2 \quad a = 4 \quad a = 5 \quad a = 6$

Example 6.29 In this example, we use the continue statement in a while-loop.

```
#include <stdio.h>
Code
6.45

int main()
{
    int a = 0;

    while(a < 5)
    {
        a++;

        printf("while-loop upper part is executed, a = %d\n",a);

        if(a==3)
        {
            printf("\n\"continue\" statement is executed, a = %d\n",a);
            printf("while-loop lower part is skipped\n\n");
            continue;
        }

        printf("while-loop lower part is executed, since a = %d\n",a);
    }

    printf("\nOutside while-loop");
}
```

When $a == 3$, the rest of the code is not executed, program execution goes to the beginning of the loop.

Output(s)

while-loop upper part is executed, a = 1
 while-loop lower part is executed, a = 1
 while-loop upper part is executed, a = 2
 while-loop lower part is executed, a = 2
 while-loop upper part is executed, a = 3

“continue” statement is executed, since $a = 3$
 while-loop lower part is skipped

while-loop upper part is executed, a = 4
 while-loop lower part is executed, a = 4
 while-loop upper part is executed, a = 5
 while-loop lower part is executed, a = 5

Outside while-loop

6.5 Break Statement

Break statement is used to quit the loop when a condition is met.

The use of the break statement in the while-loop is illustrated in Fig. 6.5.

Fig. 6.5 Behavior of the break statement in the while-loop

```
while (condition1)
{
    // statements
    if (condition2)
    {
        break;
    }
    // statements
}
```



Fig. 6.6 Behavior of the break statement in the for-loop

```
for (initialization; condition1; update)
{
    // statements
    if (condition2)
    {
        break;
    }
    // statements
}
```



Fig. 6.7 Behavior of the break statement in the do-while loop

```
do
{
    // statements
    if (condition1)
    {
        break;
    }
    // statements
} while (condition2)
```



The use of the break statement in the for-loop is illustrated in Fig. 6.6.

The use of the break statement inside the do-while loop is illustrated in Fig. 6.7.

Example 6.30 Let us use the break statement for a for-loop. First, let us form the structure of the for-loop as in Code 6.46.

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

    for(a = 0; a < 7; a++)
    {
        }
}
```

**Code
6.46**

We add printf() function in Code 6.47.

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

    for(a = 0; a < 7; a++)
    {
        printf("Inside for-loop, a = %d \n", a);
    }
}
```

**Code
6.47**

We add a condition part as in Code 6.48.

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

    for(a = 0; a < 7; a++)
    {
        if(a == 3)
        {
            }

        printf("Inside for-loop, a = %d \n", a);
    }
}
```

**Code
6.48**

Inside the condition part, we add printf and break statements, and add one more printf() function outside the for-loop as in Code 6.49.

```
#include <stdio.h>
Code
6.49

int main()
{
    int a;

    for(a = 0; a < 7; a++)
    {
        if(a == 3)
        {
            printf("Quitting for-loop, a = %d \n", a);
            break;
        }

        printf("Inside for-loop, a = %d \n", a);
    }

    printf("\nOutside for-loop, a = %d ", a);
}
```

Output(s)

Inside for-loop, a = 0
 Inside for-loop, a = 1
 Inside for-loop, a = 2
 Quitting for-loop, a = 3

 Outside for-loop, a = 3

Example 6.31 In this example, we use the break statement inside a while-loop.

```
#include <stdio.h>
Code
6.50

int main()
{
    int a = 2;

    while(1)
    {
        printf("Inside while-loop, a = %d \n", a);

        a = a+2;

        if (a > 8)
            break;
    }
    printf("\nOutside while-loop, a = %d ", a);
}
```

Output(s)

Inside while-loop, a = 2

Inside while-loop, a = 4

Inside while-loop, a = 6

Inside while-loop, a = 8

Outside while-loop, a = 10

Example 6.32 In this example, we use **bool** data type for the while-loop condition.

<pre>#include <stdio.h> #include <stdbool.h> int main() { bool a = true; double b = 0; while(a) { printf("Inside while-loop, b = %.1f \n", b); b = b+2.6; if(b > 10) a = false; } printf("\nOutside while-loop, b = %.1f ", b); }</pre>	Code 6.51
--	----------------------

Output(s)

Inside while-loop, b = 0.0

Inside while-loop, b = 2.6

Inside while-loop, b = 5.2

Inside while-loop, b = 7.8

Outside while-loop, b = 10.4

Example 6.33 In this example, we use the break statement in an infinite do-while loop after an if statement. No curly parentheses are used for the if statement.

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

    do
    {
        printf("Inside do-while loop, a = %d \n", a);

        a = a+2;

        if (a > 8)
            break;

    } while(1);

    printf("\nOutside do-while loop, a = %d ", a);
}
```

Code
6.52**Output(s)**

Inside do-while loop, a = 2

Inside do-while loop, a = 4

Inside do-while loop, a = 6

Inside do-while loop, a = 8

Outside do-while loop, a = 10

Example 6.34 In this example, we use the break statement inside the parentheses of a conditional expression.

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

    do
    {
        printf("Please enter a negative number : ");
        scanf("%d", &a);

        printf("Inside the do-while loop, you entered %d \n", a);

        if (a > 0)
        {
            printf("\nYou entered a positive number. Quitting the loop.");
            break;
        }

    } while(1);

    printf("\n\nOutside the do-while loop, you last entered %d \n", a);
}
```

Code
6.53

Output(s)

Please enter a negative number : -2
 Inside the do-while loop, you entered -2
 Please enter a negative number : -3
 Inside the do-while loop, you entered -3
 Please enter a negative number : 1
 Inside the do-while loop, you entered 1
 You entered a positive number. Quitting the loop.
 Outside the do-while loop, you last entered 1

Problems

- What are the outputs of Code 6.54?

<code>#include <stdio.h></code>	Code 6.54
<pre>int main() { for (int a = 5; a > 0; a = a-2) { printf("Inside for-loop\n"); } printf("\nOutside for-loop"); }</pre>	

- Write a program that displays the first 10 even integers. Use the for-loop in your code.
- Write a C program that inputs an integer from the user and calculates the factorial of the entered number.
- Write a program that calculates and displays the sum of the series

$$1 + \frac{1}{3} + \left(\frac{1}{3}\right)^2 + \left(\frac{1}{3}\right)^3 + \dots$$

- Write three separate C programs that display the patterns in Fig. 6.8.
- Write a C program that inputs an integer from the user to determine the number of digits in the entered number.
- Write a C program that converts a binary number to decimal.

Fig. 6.8 Patterns to be displayed

*	1	*
**	22	**
***	333	** *
****	4444	** * *
*****	55555	** * * *
*****	666666	** * * * *

Chapter 7

Complex Numbers in Modern C Programming



7.1 How to Define a Complex Number

To define complex numbers and perform operations on complex numbers, we need to include the header file <complex.h> in our program.

The variables employing complex data type can be defined either using

```
double __Complex var_name;  
float __Complex var_name;  
long double __Complex var_name;
```

or using

```
double complex var_name;  
float complex var_name;  
long double complex var_name;
```

The real part of a complex number can be extracted using the functions

```
crealf, creal, creall
```

which have prototypes

```
float crealf( float complex z );  
double creal( double complex z );  
long double creall( long double complex z );
```

The imaginary part of a complex number can be extracted using the functions

`cimagf, cimag, cimags`

which have prototypes

```
float cimagf( float complex z );
double cimag( double complex z );
long double cimags( long double complex z );
```

If the range is not a concern, we can use the functions

```
double creal( double complex z );
double cimag( double complex z );
```

for float and long double data types.

7.2 Complex Operations

Standard arithmetic operators `+`, `-`, `*`, `/` can be used with real, complex types in any combination.

Example 7.1 We can define complex numbers using both `_Complex` and `complex`.

```
#include <complex.h>
#include <stdio.h>

int main()
{
    float _Complex a = 1 + 2*I;
    float complex b = -2 + 3*I;

    printf("a = %.1f + %.1fi \n", creal(a), cimag(a));
    printf("b = %.1f + %.1fi \n", creal(b), cimag(b));
}
```

Code
7.1

Output(s)

a = 1.0 + 2.0i

b = -2.0 + 3.0i

Example 7.2 We can multiply two complex numbers as in Code 7.2.

```
#include <complex.h>
#include <stdio.h>

int main()
{
    float complex a = 1 + 2*I;
    float complex b = -2 + 3*I;

    float complex c = a*b;

    printf("c = %.1f + %.1fi \n", creal(c), cimag(c));
}
```

Code
7.2

Output(s) $c = -8.0 + -1.0i$

Example 7.3 Division and addition operations can be performed on complex numbers.

```
#include <complex.h>
#include <stdio.h>

int main()
{
    float complex a = 1 + 2*I;
    float complex b = -2 + 3*I;

    float complex r1 = a*b;

    float complex r2, r3;
    r2 = a/b;
    r3 = a+b;

    printf("r1 = %.1f + %.1fi \n", creal(r1), cimag(r1));
    printf("r2 = %f + %fi \n", creal(r2), cimag(r2));
    printf("r3 = %.1f + %.1fi \n", creal(r3), cimag(r3));
}
```

Code
7.3

Output(s)

$r1 = -8.0 + -1.0i$
 $r2 = 0.307692 + -0.538462i$
 $r3 = -1.0 + 5.0i$

Example 7.4 In this example, we define double and long double complex numbers.

```
#include <complex.h>
#include <stdio.h>

int main()
{
    float complex a = 1 + 2*I;
    double complex b = -2 + 3*I;
    long double complex c = 4-5*I;

    long double complex d = a*b;

    printf("d = %.1f + %.1fi \n", creal(d), cimag(d));
}
```

Code
7.4

Output(s) $d = -8.0 + -1.0i$

Example 7.5 Arithmetic operations can be performed between complex numbers and real numbers.

```
#include <complex.h>
#include <stdio.h>

int main()
{
    float complex a = 1 + 2*I;
    float b = -2;

    float complex c = a+b;
    float complex d = a*b;

    float e = a*b;

    printf("c = %.1f + %.1fi \n", creal(c), cimag(c));
    printf("d = %.1f + %.1fi \n", creal(d), cimag(d));
    printf("e = %.1f ", e);
}
```

Code
7.5

Output(s)

$c = -1.0 + 2.0i$

$d = -2.0 + -4.0i$

$e = -2.0$

7.3 Calculation of Absolute Value (Norm, Modulus, or Magnitude) of a Complex Number

The absolute value of a complex number in C can be calculated using the functions

`cabsf, cabs, cabsl`

The prototypes of these functions are

```
float cabsf( float complex z );
double cabs( double complex z );
long double cabsl( long double complex z );
```

Example 7.6 In this example, we calculate the norms of float, double, and long double complex numbers.

```
#include <complex.h>
#include <stdio.h>

int main()
{
    float complex a = -3 - 4*I;
    double complex b = 3 + 4*I;
    long double complex c = -3 + 4*I;

    float norm_a = cabsf(a);
    double norm_b = cabs(b);
    long double norm_c = cabsl(c);

    printf("Norms are : %.1f, %.1lf, %.1Lf", norm_a, norm_b, norm_c);
}
```

**Code
7.6**

Output(s)

Norms are : 5.0, 5.0, 5.0

7.4 Complex Number Formation

A complex number can be obtained by providing its real and imaginary parts to the macros

`CMPLXF, CMPLXF, CMPLXL`

and these macros are used as

```
double complex CMPLX ( double x, double y );
float complex CMPLXF ( float x, float y );
long double complex CMPLXL (long double x, long double y );
```

Example 7.7 In this example, we form a float complex number using CMPLXF function.

```
#include <complex.h>
#include <stdio.h>

int main(void)
{
    float real = 1.5, imag = 1.7;
    float complex a = CMPLXF(real, imag);

    printf("a = %.1f + %.1fi", creal(a), cimag(a));
}
```

**Code
7.7**

Output(s) a = 1.5 + 1.7i

Example 7.8 In this example, we form a double complex number using the CMPLXL function.

```
#include <complex.h>
#include <stdio.h>

int main(void)
{
    double real = 1.5, imag = 1.7;
    double complex a = CMPLX(real, imag);

    printf("Absolute value = %.1f", cabs(a));
}
```

**Code
7.8**

Output(s) Absolute value = 2.3

7.5 Calculation of the Conjugate of a Complex Number in C

The conjugate of a complex number can be calculated using the functions

`conjf, conj, conjl`

whose declarations are given as

```
float complex conjf ( float complex z );
double complex conj ( double complex z );
long double complex conjl ( long double complex z );
```

Example 7.9 In this example, we print a complex number and its conjugate.

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex z = 1.0 + 2.0*I;
    double complex conj_z = conj(z);

    printf("z = %.1f + %.1fi\n", creal(z), cimag(z));
    printf("z* = %.1f + %.1fi", creal(conj_z), cimag(conj_z));
}
```

Code
7.9

Output(s)

$z = 1.0 + 2.0i$

$z^* = 1.0 + -2.0i$

Example 7.10 In this example, we multiply a complex number by its conjugate and print the result.

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex z = 1.0 + 2.0*I;
    double complex conj_z = conj(z);

    double complex z2 = z * conj_z;

    printf("z x z* = %.1f + %.1fi\n", creal(z2), cimag(z2));
}
```

Code
7.10

Output(s) $z \times z^* = 5.0 + 0.0i$

7.6 Calculation of the Argument, That Is, Phase Angle, of a Complex Number in C

The angle of a complex number can be calculated using the functions

`cargf, carg, cargl`

whose prototypes are given as

```
float cargf ( float complex z );
double carg ( double complex z );
long double cargl ( long double complex z );
```

Example 7.11 In this example, the phase of a complex number is displayed in both radians and degrees.

```
#include <stdio.h>
#include <complex.h>
#include <math.h>

int main(void)
{
    double complex z = 1.0 + 1.0*I;

    printf("z = %.1fi + %.1fi\n", creal(z), cimag(z));
    printf("Phase angle of z in radians : %f\n", carg(z));
    printf("Phase angle of z in degrees : %.1f\n", (carg(z)/M_PI)*180);
    printf("The value of pi here is : %lf\n", M_PI);
}
```

Code
7.11

Output(s)

$z = 1.0 + 1.0i$

Phase angle of z in radians : 0.785398

Phase angle of z in degrees : 45.0

The value of pi here is : 3.141593

7.7 Calculation of Complex Exponentials

The exponential expression

$$e^z$$

where z is a complex number can be calculated in C language using the functions

`cexpf, cexp, cexpl`

and the prototypes of these functions are

```
float complex cexpf ( float complex z );
double complex cexp ( double complex z );
long double complex cexpl ( long double complex z );
```

For complex number

$$z = a + ib$$

the exponential term

$$e^z$$

is calculated as

$$e^z = e^{a+ib} \rightarrow e^z = e^a(\cos b + i \sin b)$$

Example 7.12 In this example, calculation of complex exponential is illustrated.

<pre>#include <stdio.h> #include <math.h> #include <complex.h> int main(void) { double PI = acos(-1); double complex z = 0.5+I * PI/4; double complex exp_z = cexp(z); printf("z = %.1f + %.1fi\n", creal(z), cimag(z)); printf("e^z = %.1f + %.1fi", creal(exp_z), cimag(exp_z)); }</pre>	Code 7.12
--	---

Output(s)

$z = 0.5 + 0.8i$

$e^z = 1.2 + 1.2i$

7.8 Computation of the Complex Natural (Base-e) Logarithm of a Complex Number

The complex natural logarithm of complex float, complex double, and complex long-long double numbers can be calculated using the functions

`clogf, clog, clogl`

whose prototypes are

```
float complex clogf ( float complex z );
double complex clog ( double complex z );
long double complex clogl ( long double complex z );
```

Example 7.13 In this example, the natural logarithm of a complex number is calculated and displayed.

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex z = 1+I;
    double complex loge_z = clog(z);

    printf("z = %.1f + %.1fi\n", creal(z), cimag(z));
    printf("log_e(z) = %.1f + %.1fi", creal(loge_z), cimag(loge_z));
}
```

Code
7.13
x

Output(s)

$z = 1.0 + 1.0i$

$\log_e(z) = 0.3 + 0.8i$

7.9 Complex Power Calculation

The complex power x^y can be calculated using the functions

`cpowf, cpow, cpowl`

whose prototypes are

```
float complex cpowf ( float complex x, float complex y );
double complex cpow ( double complex x, double complex y );
long double complex cpowl ( long double complex x, long double complex y );
```

Example 7.14 In this example, the complex power calculation is illustrated.

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex z = cpow(1.0-3.0*I, 3);
    printf("(1-3i)^3 = %.1f + %.1fi \n", creal(z), cimag(z));
}
```

Code
7.14

Output(s) $(1 - 3i)^3 = -26.0 + 18.0i$

7.10 Square Root of a Complex Number

The square root of a complex number can be calculated using the functions

`csqrtf`, `csqrt`, `csqrtd`

whose prototypes are

```
float complex csqrtf ( float complex z );
double complex csqrt ( double complex z );
long double complex csqrtd ( long double complex z );
```

Example 7.15 In this example, the square root of a complex number is calculated and the result is printed.

```
#include <stdio.h>
#include <complex.h>

int main(void)
{
    double complex z = csqrt(-4 + 3*I);

    printf("Square root of -4 + 3i is : %.1f + %.1fi \n", creal(z), cimag(z));
}
```

Code
7.15

Output(s) Square root of $-4 + 3i$ is : $0.7 + 2.1i$

7.11 Complex Trigonometric Functions

7.11.1 The *csin* Functions

The *csin* functions

`csinf, csin, csinl`

are used to compute the complex sine of z , and the prototypes of these functions are

```
float complex csinf ( float complex z );
double complex csin ( double complex z );
long double complex csinl ( long double complex z );
```

Example 7.16 In this example, the complex sine of a complex number is calculated and the result is printed.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(void)
{
    double complex z = csin(-2+3*I);
    printf("csin(-3+2i) = %.2f + %.2fi \n", creal(z), cimag(z));
}
```

Code
7.16

Output(s) $\text{csin}(-3+2i) = -9.15 + -4.17i$

7.11.2 The *ccos* Functions

The *ccos* functions

`ccosf, ccos, ccosl`

are used to compute the complex cosine of z , and the prototypes of these functions are

```
float complex ccosf ( float complex z );
double complex ccos ( double complex z );
long double complex ccosl ( long double complex z );
```

Example 7.17 In this example, the complex cosine of a complex number is calculated and the result is printed.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(void)
{
    double complex z = ccos(-2+3*I);
    printf("ccos(-3+2i) = %.2f + %.2fi \n", creal(z), cimag(z));
}
```

Code
7.17

Output(s) $\text{ccos}(-3+2i) = -4.19 + 9.11i$

7.11.3 The *ctan* Functions

The *ctan* functions

ctanf, *ctan*, *ctanl*

are used to compute the complex tangent of *z*, and the prototypes of these functions are

```
float complex ctanf ( float complex z );
double complex ctan ( double complex z );
long double complex ctanl ( long double complex z );
```

Example 7.18 In this example, the complex tangent of a complex number is calculated and the result is printed.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(void)
{
    double complex z = ctan(-2+3*I);
    printf("ctan(-3+2i) = %.4f + %.4fi \n", creal(z), cimag(z));
}
```

Code
7.18

Output(s) $\text{ctan}(-3+2i) = 0.0038 + 1.0032i$

7.11.4 The *cacos* Functions

The *cacos* functions

`cacosf, cacos, cacosl`

are used to compute the complex arc cosine of complex number z , and the prototypes of these functions are

```
float complex cacosf (float complex z);
double complex cacos (double complex z);
long double complex cacosl (long double complex z);
```

Example 7.19 In this example, the complex arc cosine of a complex number is calculated and the result is printed.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(void)
{
    double complex z = cacos(-2+3*I);

    printf("cacos(-3+2i) = %.2f + %.2fi \n", creal(z), cimag(z));
}
```

Code
7.19

Output(s) $\text{cacos}(-3+2i) = 2.14 + -1.98i$

7.11.5 The *casin* Functions

The *casin* functions

`casinf, casin, casinl`

are used to compute the complex arc sine of complex number z , and the prototypes of these functions are

```
float complex casinf (float complex z);
double complex casin (double complex z);
long double complex casinl (long double complex z);
```

Example 7.20 In this example, the complex arc sine of a complex number is calculated and the result is printed.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(void)
{
    double complex z = casin(-2+3*I);

    printf("casin(-3+2i) = %.2f + %.2fi \n", creal(z), cimag(z));
}
```

Code
7.20

Output(s) $\text{casin}(-3+2i) = -0.57 + 1.98i$

7.11.6 The catan Functions

The catan functions

`catanf, catan, catanl`

are used to compute the complex arc tangent of complex number z , and the prototypes of these functions are

```
float complex catanf (float complex z);
double complex catan (double complex z);
long double complex catanl ( long double complex z);
```

Example 7.21 In this example, the complex arc tangent of a complex number is calculated and the result is printed.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main(void)
{
    double complex z = catan(-2+3*I);

    printf("catan(-3+2i) = %.2f + %.2fi \n", creal(z), cimag(z));
}
```

Code
7.21

Output(s) $\text{catan}(-3+2i) = -1.41 + 0.23i$

7.11.7 Hyperbolic Functions

The hyperbolic functions of complex numbers can be calculated using built-in hyperbolic complex function. In this section, we explain these functions.

The `csinh` Functions

The `csinh` functions compute the complex hyperbolic sine of z , and the prototypes of these functions are

```
float complex csinhf (float complex z);
double complex csinh (double complex z);
long double complex csinhl (long double complex z);
```

The `ccosh` Functions

The `ccosh` functions compute the hyperbolic cosine of z , and the prototypes of these functions are

```
float complex ccoshf (float complex z);
double complex ccosh (double complex z);
long double complex ccoshl (long double complex z);
```

The `ctanh` Functions

The `ctanh` functions compute the hyperbolic tangent of z , and the prototypes of these functions are

```
float complex ctanhf (float complex z);
double complex ctanh (double complex z);
long double complex ctanhl (long double complex z);
```

The `casinh` Functions

The `casinh` functions compute the complex arc hyperbolic sine of z , and the prototypes of these functions are

```
float complex casinhf (float complex z);
double complex casinh (double complex z);
long double complex casinhl (long double complex z);
```

The `cacosh` Functions

The `cacosh` functions compute the complex arc hyperbolic cosine of z , and the prototypes of these functions are

```
float complex cacoshf (float complex z);
double complex cacosh (double complex z);
long double complex cacoshl (long double complex z);
```

The catanh Functions

The catanh functions compute the complex arc hyperbolic tangent of z , and the prototypes of these functions are

```
float complex catanhf (float complex z);
double complex catanh (double complex z);
long double complex catanhl (long double complex z);
```

Problems

1. Which header file should we include in the source code to be able to define the complex numbers?
2. Define a complex number, initialize it to $3 + 4j$, and print it.
3. Fill the inside of printf() function to print the real and imaginary parts of the complex number as in Code 7.22.

```
#include <complex.h>
#include <stdio.h>

int main()
{
    float complex a= 5 + 2*I;
    printf("....");
}
```

Code
7.22

4. Write a program that inputs the real and imaginary parts of a complex number from the user and calculates its magnitude and displays it.
5. Write a C program that calculates and prints the magnitude and phase of $3 - 4j$. Phase is printed both in radian and degree units.
6. Write a C program that calculates and prints the sine, cosine, Ln of the complex number $2 + \sqrt{3}j$.

Chapter 8

Arrays



8.1 Syntax for Array Declaration

The syntax for array declaration is

```
dataType arrayName[arraySize] = {val1, val2, ..., valN};
```

8.2 Accessing Array Elements

We can access array elements using subscript operator [] and the index value of the element as in

```
arrayName[index]
```

The indexing in the array always starts with 0, and the index of the last element is $N - 1$, so new values to the array elements are assigned using

```
array_name[i] = new_value;
```

Example 8.1 We can define **int**, **char**, **float**, and **double** arrays as in Code 8.1.

```
#include <stdio.h>
int main()
{
    int a[6];
    char b[5];
    float c[4];
    double d[7];
}
```

Code
8.1

Example 8.2 Array elements can be initialized.

```
int main()
{
    int a[1] = {1};
    int b[2] = {4, 6};
}
```

Code
8.2

Example 8.3 Code 8.3 and Code 8.4 achieve the same goal.

```
int main()
{
    int a[3] = {7, 13, 54};
```

Code
8.3

=

```
int main()
{
    int a[3];
    a[0] = 7;
    a[1] = 13;
    a[2] = 54;
```

Code
8.4

Example 8.4 Initialization starts from the first element, noninitialized elements equal to zero by default.

```
#include <stdio.h>
int main()
{
    int a[4] = {2, 7};

    printf("a[0]=%d  ", a[0]);
    printf("a[1]=%d  \n\n", a[1]);
    printf("a[2]=%d  ", a[2]);
    printf("a[3]=%d  ", a[3]);
```

Code
8.5

Output(s)

a[0] = 2 a[1] = 7
a[2] = 0 a[3] = 0

Example 8.5 Array name shows an address in the memory. If we use an index value larger than the array size, we refer to a memory location that is not reserved for the array variable, and these locations may be inaccessible, or if they are accessible, they can contain values.

```
#include <stdio.h>
int main()
{
    int a[2] = {3,6};

    printf("a[0]=%d    ", a[0]);
    printf("a[1]=%d    ", a[1]);
    printf("a[7]=%d    ", a[7]);

    printf("a[10]=%d   ", a[10]);
    printf("a[100]=%d  ", a[100]);
}
```

**Code
8.6**

Output(s)

a[0] = 3 a[1] = 6 a[7] = 32678 a[10] = 1829073257 a[100] = -
1864110261

8.3 Array Initialization Without Size

We can define an array and initialize as in

```
dataType arrayName [] = {val1, val2, ..., valN};
```

In this case, array size is determined by the compiler considering the number of initialization values.

Example 8.6 In Code 8.7, we define an array without size and initialize it when it is defined.

```
#include <stdio.h>
Code
8.7
int main()
{
    int a[] = {3, 6, 8, 9};

    printf("a[0]=%d    ", a[0]);
    printf("a[1]=%d    \n\n", a[1]);
    printf("a[2]=%d    ", a[2]);
    printf("a[3]=%d    ", a[3]);
}
```

Output(s)

a[0] = 3 a[1] = 6
 a[2] = 8 a[3] = 9

Example 8.7 In Code 8.8, for the initialization of the array, two values are used, and array size is automatically decided as 2, and if indexes larger than array size used to access the array elements, arbitrary values are displayed.

```
#include <stdio.h>
Code
8.8
int main()
{
    int a[] = {3, 6};

    printf("a[0]=%d    ", a[0]);
    printf("a[1]=%d    ", a[1]);
    printf("a[55]=%d    ", a[55]);

    printf("a[10]=%d    ", a[10]);
    printf("a[867]=%d    ", a[867]);
}
```

Output(s)

a[0] = 3 a[1] = 6 a[55] = 0 a[10] = 4225392 a[867] = 826162750

Example 8.8 The size of an integer array equals 4× number of elements.

```
#include <stdio.h>
int main()
{
    int a[] = {3,6,8,9};
    printf("Size of a is : %lu", sizeof(a));
}
```

Code
8.9

Output(s) Size of a is : 16

8.4 Array Initialization Using Loops

Arrays can be initialized using loops; usually for-loops are used for the initialization of the arrays.

Example 8.9 In this example, we first initialize the array and then print the array elements.

```
#include <stdio.h>
int main()
{
    float a[5];
    for (int i = 0; i < 5; i++)
    {
        a[i] = i*3.12;
    }
    for (int i = 0; i < 5; i++)
    {
        printf("a[%d] = %.1f      ", i, a[i]);
    }
}
```

Code
8.10

Output(s)

a[0] = 0.0 a[1] = 3.1 a[2] = 6.2 a[3] = 9.4 a[4] = 12.5

Example 8.10 Array elements can be initialized by the user-entered values.

```
#include <stdio.h>
Code
8.11

int main()
{
    int a[3];

    for (int i = 0; i < 3; i++)
    {
        printf("Please enter a[%d]: ", i);
        scanf("%d", &a[i]);
    }

    printf("\nYou entered: \n\n");

    for (int i = 0; i < 3; i++)
    {
        printf("a[%d] = %d    ", i, a[i]);
    }
}
```

Output(s)

Please enter a[0]: 2

Please enter a[1]: 6

Please enter a[2]: 7

You entered:

a[0] = 2 a[1] = 6 a[2] = 7

Example 8.11 In this example, the maximum value of all the array elements is found.

```
#include <stdio.h>
Code
8.12

int main()
{
    int numbers[5] = { 123, 45, 345, 67, 78 };

    int maxNum = numbers[0];

    for (int i = 1; i < 5; i++)
    {
        if (maxNum < numbers[i])
        {
            maxNum = numbers[i];
        }
    }

    printf("The maximum of the numbers in the array is : %d", maxNum);
}
```

Fig. 8.1 Typical memory map for Example 8.12

	Address	Content
a=100 →	100	78
	101	56
	102	34
	103	12
	104	94
	105	EF
	106	CD
	107	AB

Output(s)

The maximum of the numbers in the array is : 345

Example 8.12 In Code 8.13, we use hexadecimal numbers to initialize array elements.

```
#include <stdio.h>
int main()
{
    int a[2];
    a[0] = 0x12345678;
    a[1] = 0xABCDDEF94;
}
```

**Code
8.13**

Each of the integers $a[0]$ and $a[1]$ are stored on 4-byte consecutive memory locations. The letter “a” indicates the address of the first byte in the memory used to store the first integer $a[0]$. For simplicity of explanation, let us assume that $a=100$, which is the address of the first byte of $a[0]$ stored into memory. In Fig. 8.1, the storage of the integer bytes in consecutive register locations is illustrated. We assumed that little-endian is used by the compiler.

Example 8.13 Array name indicates a register address, which is the starting address of a block where array values are stored.

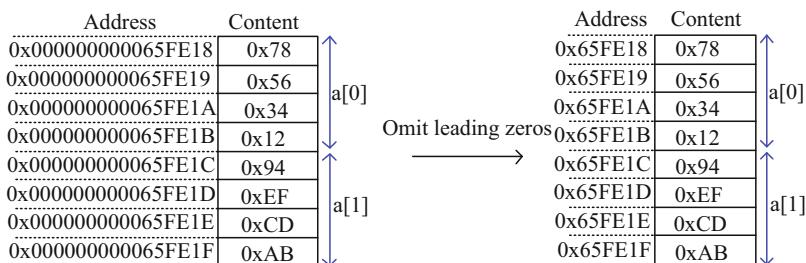


Fig. 8.2 Memory map for Example 8.13

```
#include <stdio.h>
Code
8.14

int main()
{
    int a[2];

    a[0] = 0x12345678;
    a[1] = 0xABCD E94;

    printf("a[0] = %x\n", a[0]);
    printf("a[1] = %x\n", a[1]);
    printf("Address (a) is : %p \n", a); // 000000000065FE18 in my computer
    printf("Address size is : %lu", sizeof(a));
}
```

Output(s)

a[0] = 12345678

a[1] = ABCDEF94

Address (a) is : 000000000065FE18

Address size is : 8

In Fig. 8.2, memory map is explained for the array of Example 8.13.

Example 8.14 In this example, we define a character array and display its elements. Array name is the address of storage block.

```
#include <stdio.h>
Code
8.15

int main()
{
    char a[3] = {'A', 'b', 'c'};

    printf("a[0] = %c\n", a[0]);
    printf("a[1] = %c\n", a[1]);
    printf("a[2] = %c\n", a[2]);

    printf("Address (a) is : %p \n", a); // 000000000065FE1D in my computer
}
```

Output(s)

a[0] = A
 a[1] = b
 a[2] = c
 Adress (a) is : 000000000065FE1D

Example 8.15 Characters are also 8-bit numbers. In this example, we show hexadecimal values of array characters and illustrate how they are stored in memory (Fig. 8.3).

```
#include <stdio.h>
Code
8.16

int main()
{
    char a[3] = {'A', 'b', 'c'};

    printf("a[0] = %c\n", a[0]);
    printf("a[1] = %c\n", a[1]);
    printf("a[2] = %c\n", a[2]);

    printf("Adress (a) is : %p \n", a); // 000000000065FE1D in my computer

    printf("a[0] = %x\n", a[0]);
    printf("a[1] = %x\n", a[1]);
    printf("a[2] = %x\n", a[2]);
}
```

Output(s)

a[0] = A
 a[1] = b
 a[2] = c
 Adress (a) is : 000000000065FE1D
 a[0] = 41
 a[1] = 62
 a[2] = 63

Fig. 8.3 Memory map for Example 8.15

Address	Content	Address	Content
0x65FE1D	'A'	0x65FE1D	0x41
0x65FE1E	'b'	0x65FE1E	0x62
0x65FE1F	'c'	0x65FE1F	0x63

8.5 Strings as Array of Characters

The array of characters terminated by a NULL character is called a string. The array of characters terminated by a NULL can be printed using a for-loop or using the format %s in printf() function.

Example 8.16 A string can be printed using a for-loop or using a printf() function.

<pre>#include <stdio.h> int main() { char a[6] = { 'H', 'e', 'l', 'l', 'o', '\0' }; for(int i = 0; i < 6; i++) { printf("%c", a[i]); } printf("\n%s", a); }</pre>	Code 8.17
--	----------------------

Output(s)

Hello Hello

If the array of characters is NOT terminated by a NULL, then it may not be printed properly using the format %s in printf() function.

Example 8.17

<pre>#include <stdio.h> int main() { char a[5] = { 'H', 'e', 'l', 'l', 'o' }; for(int i = 0; i < 5; i++) { printf("%c", a[i]); } printf("\n%s", a); }</pre>	Code 8.18
--	----------------------

Output(s)

Hello Hello♣

```
#include <stdio.h>
int main()
{
    char a[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
    char b[6] = "Hello"; // a equals to b
    for(int i=0; i < 6; i++)
    {
        printf("%c", b[i]);
    }
    printf(" %s", a);
    printf(" %s", b);
}
```

Code
8.19

Example 8.18 In this example, we define a string in two different ways.

The array of characters and the array of NULL terminated characters are two different concepts. The latter one is the string, and for string manipulations, a number of built-in library functions are available, and these functions do not work properly for the array of characters that are not NULL terminated.

Output(s)

Hello Hello Hello

8.6 Multidimensional Arrays

A two-dimensional array is defined as

```
dataType arrayName[size1][size2];
```

and this array can be considered a single-dimensional array having size1 elements each of which is an array having size2 elements of dataType.

```
arrayName[index1][index2];
```

Example 8.19 A multidimensional array can be initialized in two different methods.

```
#include <stdio.h>
int main()
{
    int a[2][3] = { 10, 20, 30, 40, 50, 60 };

    int b[2][3] = {
        {10, 20, 30},
        {40, 50, 60}
    };
}
```

Code
8.20

In this example, both initialization methods assign the same values to the same index locations.

Example 8.20 Nested for-loops can be used to initialize a multidimensional array.

```
#include<stdio.h>
int main()
{
    int M = 4, N = 4;

    int a[M][N];

    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            a[i][j] = 0;
        }
    }
}
```

Code
8.21

Example 8.21 The multidimensional array elements can be displayed using nested for-loops.

```
#include <stdio.h>
Code
8.22
int main()
{
    int a[2][3] = { 67, 20, 87, 34, 50, 70 };

    int b[2][3] = {
        {67, 20, 87},
        {34, 50, 70}
    };

    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            printf("%d ",a[i][j]);
        }
        printf("\n");
    }

    printf("\n");

    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            printf("%d ",b[i][j]);
        }
        printf("\n");
    }
}
```

Output(s)

67 20 87

34 50 70

67 20 87

34 50 70

Example 8.22 The first dimension of a multidimensional array can be omitted. In this case, the value of the first dimension is determined by the compiler regarding the initialization data.

```
#include<stdio.h>
int main()
{
    int a[][] = {
        {1,2},
        {3,4},
        {5,6}
    };

    // int b[][] = { {1,2}, {3,4}, {5,6} };
}
```

**Code
8.23**

Example 8.23 Only the first dimension of a multidimensional array can be omitted, all the other dimensions must be written.

```
#include<stdio.h>
int main()
{
    int a[][] = { {1,2}, {3,4}, {5,6} };

    int b[][][] = { { {1,2}, {3,4}, {5,6} },
                    { {7,8}, {9,10}, {11,12} } };

    // error: declaration of 'b' as multidimensional array must have bounds
    // for all dimensions except the first
}
```

**Code
8.24**

Example 8.24 Only the first dimension of a multidimensional array can be omitted, all the other dimensions must be written.

```
#include<stdio.h>
int main()
{
    int a[][] = { {1,2}, {3,4}, {5,6} };

    int b[][][3][2] = { { {1,2}, {3,4}, {5,6} },
                        { {7,8}, {9,10}, {11,12} } }; // ok
}
```

**Code
8.25**

8.7 Passing an Array to a Function in C

Arrays can be used in function arguments. Function arguments contain array definition without the first dimension. Array names are solely used in function calls.

Example 8.25

```
#include <stdio.h>
void dispArray(double a[]);
int main()
{
    double a[4] = {2.3, 5.6, 3.1, 12.35};
    printf("Number of elements is : %d\n", sizeof(a)/sizeof(double));
    dispArray(a); // only array name is passed to the function call
}
void dispArray(double a[])
{
    for (int i = 0; i < 4; i++)
    {
        printf("a[%d] = %.2lf    ", i, a[i]);
    }
}
```

Code
8.26

Output(s)

Number of elements is : 4

a[0] = 2.30 a[1] = 5.60 a[2] = 3.10 a[3] = 12.35

If we use N-dimensional arrays at the inputs of the functions, except for the first array dimension, all the other dimensions must be specified at the declaration of functions.

Example 8.26 In this example, the first dimension of the array is not written in function arguments.

```
#include <stdio.h>
Code
8.27

void dispArray(int a[][3], int n) // first dimension is not used
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            printf("%d ", a[i][j]);
        }
    }
}

int main()
{
    int a[][] = {{7, 8, 3}, {9, 2, 1}, {4, 0, 5}};
    dispArray(a, 2);
}
```

Output(s) 7 8 3 9 2 1

Example 8.27 Dimensions of an array can be written separately in function arguments.

```
#include <stdio.h>
Code
8.28

int M = 3, N = 3;

void dispArray(int a[][N], int M, int N)
{
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%d ", a[i][j]);
        }
    }
}

int main()
{
    int a[][] = {{7, 8, 3}, {9, 2, 1}, {4, 0, 5}};
    dispArray(a, M, N);
}
```

Output(s)

7 8 3 9 2 1 4 0 5

Problems

1. Input five integers from the user, store them in an array, and print them.
2. Write a program in C that inputs six integers from the user, stores them in an array, and prints them.
3. Write a C program that calculates the sum of all elements of an integer array.
4. Write a C program that finds the maximum number of a float array.
5. Write a C program that calculates the frequency of each element of an array.
6. Write a C program to sort the elements of the array in ascending order.
7. Write a C program to find equal elements of an integer array.
8. Initialize the array in Code 8.29 and print its elements.

```
#include <stdio.h>
int main()
{
    float a[10];
    // initialize the array using for-loop
    // and print the array elements
}
```

**Code
8.29**

9. Write a function that takes an array as its argument, reverses the elements of the array, and returns it.

Chapter 9

Functions



9.1 Introduction

A C function is a set of statements written for a specific task enclosed by curly braces. A function may return a value.

The syntax of a function is

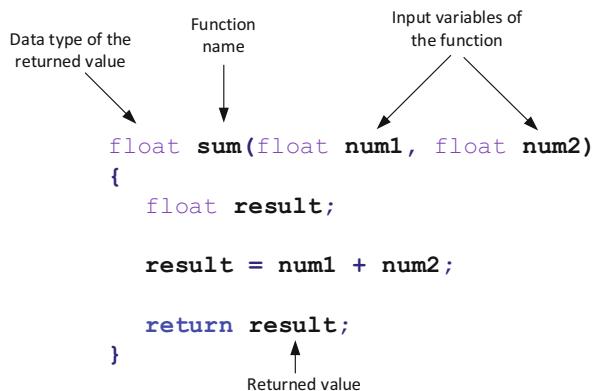
```
returned_data_type function_name(dataType1 var1, dataType1 var1, ...)  
{  
    // statements  
    ...  
    ...  
    return value;  
}
```

In Code 9.1, we show how to place the prototype and body of a function in a program.

```
#include <stdio.h>  
  
// function prototype  
returned_data_type function_name (parameter_list);  
  
int main()  
{  
    // statements  
}  
  
// function implementation  
returned_data_type function_name (parameter_list)  
{  
    // function statements  
}
```

Code
9.1

Fig. 9.1 Explanation of the parts of a function



The body of the function can be placed before the main function as shown in Code 9.2.

```

#include <stdio.h>
Code
9.2
// function implementation
returned_data_type function_name(parameter_list)
{
    // function statements
}

int main()
{
    // statements
}
  
```

If a function is written after the main function, its declaration must be written before the main function. The declaration of the function only contains the function header. Its variable names can be omitted; however, variable data types have to be written.

In Fig. 9.1, the structure of a function is explained.

The prototype, that is, declaration of the function, can be written either as

```
float sum(float num1, float num2);
```

or as

```
float sum(float, float);
```

Note that semicolon “;” is used at the end of the function declarations.

Example 9.1 Function definition can be written before the main part.

```
#include <stdio.h>
double mult(double a, double b)
{
    double result;

    result = a * b; // double result = a * b;

    return result;
}

int main()
{
    double r = mult(3.14, 8.97);

    printf("Result is: %.2lf", r);
}
```

Code
9.3

Output(s) Result is: 28.17

Example 9.2 If the function definition is written after the main part, then the prototype of the function should be written before the main part.

```
#include <stdio.h>
double mult(double a, double b); // function prototype

int main()
{
    double r = mult(3.14, 8.97);

    printf("Result is: %.2lf", r);
}

double mult(double a, double b)
{
    double result;

    result = a * b; // double result = a * b;

    return result;
}
```

Code
9.4

Example 9.3 In function prototypes, variable names can be omitted.

```
#include <stdio.h>
double mult(double, double); // function prototype

int main()
{
    double r = mult(3.14, 8.97);

    printf("Result is: %.2lf", r);
}

double mult(double a, double b)
{
    return a * b;
}
```

Code
9.5

Example 9.4 In this example, we write a function that finds the greater of two integers. Integers are entered by the user.

```
#include <stdio.h>
int myMax(int x, int y);

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

    printf("Please enter two different integers : ");
    scanf("%d %d", &a, &b);

    c = myMax(a, b);

    if(c != -1)
        printf("The maximum of %d and %d is %d", a, b, c);
    else
        printf("The numbers are equal to each other. ");
}

int myMax(int x, int y)
{
    if(x > y)
        return x;
    else if(x < y)
        return y;

    return -1;
}
```

Code
9.6

Output(s)

Please enter two different integers: 8 11

The maximum of 8 and 11 is 11

9.2 Types of Functions

We do not need to write every function. Some of the functions are already written, and they are called library or built-in functions. These functions are part of the compiler and can be used directly. Some of the library functions are

`exp()`, `pow()`, `sqrt()`

To be able to use these functions, we need not know their prototypes. The prototypes of the function can be found in the header file `<math.h>`, and the prototypes are defined as

```
double exp(double x);
double pow(double x, double y);
double sqrt(double x);
```

and these function are used to calculate the mathematical expressions

$$e^x \quad x^y \quad \sqrt{x}$$

Example 9.5 In this example, we use built-in functions `exp`, `pow`, and `sqrt`.

<pre>#include <math.h> #include <stdio.h> int main() { printf("%.3lf ", exp(2.5)); printf("%.3lf ", pow(2.5, 1.3)); printf("%.3lf ", sqrt(2.5)); }</pre>	Code 9.7
---	---------------------------

Output(s)

12.182 3.291 1.581

The functions written by a developer are called user-defined functions.

9.3 Passing Parameters to Functions

Function parameter values can be supplied with two methods:

Pass-by value method

Pass-by reference method

Example 9.6 Function call by pass-by value is illustrated in this example.

```
#include <stdio.h>

void myFunc(int a, int b); // function prototype

int main()
{
    int a = 10, b = 20;

    printf("Before function call \n");
    printf("a = %d ", a);
    printf("b = %d ", b);

    myFunc(a, b);

    printf("\n\nAfter function call \n");
    printf("a = %d ", a);
    printf("b = %d ", b);
}

void myFunc(int a, int b)
{
    a = a + 15;
    b = b + 15;
}
```

Code
9.8

Output(s)

Before function call

a = 10 b = 20

After function call

a = 10 b = 20

Example 9.7 Function call by pass-by reference is illustrated in this example.

```
#include <stdio.h>

void myFunc(int* x, int* y); // function prototype

int main()
{
    int a = 10, b = 20;

    printf("Before function call \n");
    printf("a = %d ", a);
    printf("b = %d ", b);

    myFunc(&a, &b);

    printf("\n\nAfter function call \n");
    printf("a = %d ", a);
    printf("b = %d ", b);

}

void myFunc(int* x, int* y)
{
    *x = *x + 15;
    *y = *y + 15;
}
```

Code
9.9

Output(s)

Before function call

a = 10 b = 20

After function call

a = 25 b = 35

Example 9.8 Swapping can be achieved by reference call.

```
#include <stdio.h>
void swap(int* x, int* y);

int main()
{
    int a = 34, b = 45;

    printf("Before swap operation : a = %d b = %d \n", a, b);

    swap(&a, &b); // swap a and b

    printf("After swap operation : a = %d b = %d ", a, b);
}

void swap(int* x, int* y)
{
    int t = *x;

    *x = *y;

    *y = t;
}
```

Code
9.10

Output(s)

Before swap operation: a = 34 b = 45

After swap operation: a = 45 b = 34

9.4 Returning More Than One Value

A function returns only a single value. However, two methods can be employed to return more than one value. In the first method, global variables can be used. In the second approach, we can use pointers to return more than one value.

Example 9.9 Global variables can be used inside a function to return more than one value.

```
#include <stdio.h>
int a, b; // global variables
void myFunc(int x, int y);
int main()
{
    int x = 6, y = 7;
    myFunc(x, y);
    printf("Square of %d and %d are : %d and %d ", x, y, a, b);
}
void myFunc(int x, int y)
{
    a = x*x;
    b = y*y;
}
```

Code
9.11**Output(s)** Square of 6 and 7 are: 36 and 49**Example 9.10** Pointers can be used in function arguments to return more than one value.

```
#include <stdio.h>
void myFunc(int* x, int* y);
int main()
{
    int x = 6, y = 7;
    printf("Square of %d and %d are : ", x, y);
    myFunc(&x, &y);
    printf("%d and %d", x, y);
}
void myFunc(int* x, int* y)
{
    *x = (*x) * (*x);
    *y = (*y) * (*y);
}
```

Code
9.12**Output(s)** Square of 6 and 7 are: 36 and 49

9.5 Recursive Functions

Recursive functions are self-calling function. In Code 9.13, the structure of a recursive function is shown.

```
void myFunc()
{
    // ...
    myFunc();
    // ...
}
```

Code
9.13

Recursive functions can contain conditional expressions before self-calling as in Code 9.14.

```
void myFunc()
{
    // ...
    if (condition)
        myFunc();
    // ...
}
```

Code
9.14

Example 9.11 Let us write a recursive function that displays the integers

$$a \quad a - 1 \quad a - 2 \dots 0$$

We will use $a = 6$ to test the recursive function. First, we write Code 9.15.

```
void countDown(int a)
{
    printf("%d ", a);
}
```

Code
9.15

In the second step, we add the self-calling and decrement statements as in Code 9.16.

```
void countDown(int a)
{
    printf("%d ", a);

    a--;
    countDown(a);
}
```

Code
9.16

The conditional part for self-calling is added as in Code 9.17.

```
void countDown(int a)
{
    printf("%d ", a);
    a--;

    if (a >= 0)
        countDown(a);
}
```

Code
9.17

The recursive function can be tested inside a main function as in Code 9.18.

```
#include <stdio.h>

void countDown(int a);

int main()
{
    countDown(6);
}

void countDown(int a)
{
    printf("%d ", a);
    a--;

    if (a >= 0)
        countDown(a);
}
```

Code
9.18

Output(s) 6 5 4 3 2 1 0

Example 9.12 Factorial of integers can be calculated using a recursive function.

<pre>#include <stdio.h> int myFactorial(int a); int main() { int a; printf("Enter a : "); scanf("%d", &a); int result = myFactorial(a); printf("%d", result); } int myFactorial(int a) { int r; if (a > 0) { r = a * myFactorial(a - 1); return r; } else if(a == 0) { return 1; } }</pre>	Code 9.19
---	----------------------

Output(s)

Enter a: 4

24

The function myFactorial can be written in a compact form as in Code 9.20.

<pre>int myFactorial(int a) { return ((a == 1 a == 0) ? 1 : a * myFactorial(a - 1)); }</pre>	Code 9.20
---	----------------------

9.6 Nested Functions

A function may be defined inside another function. Some compilers may support nested functions.

```
#include <stdio.h>
void myFunc1();
int main()
{
    myFunc1();
}

void myFunc1()
{
    printf("Inside MyFunc1 \n");
    void myFunc2()
    {
        printf("Inside MyFunc2 \n");
    }
    myFunc2();
}
```

Code
9.21

Output(s) Dev-C++ compiler for a C project gives the outputs

Inside MyFunc1
Inside MyFunc2

Problems

1. Write a C function that takes two integers and swaps their values.
2. Write a C function that finds the largest element of an array and returns it.

3. Write a C function that determines whether an integer is a prime number or not.
4. What is the output of Code 9.22?

```
#include <stdio.h>
void myFunc(int* x, int* y);

int main()
{
    int x = 13, y = 67;

    myFunc(&x, &y);

    printf("%d and %d", x, y);
}

void myFunc(int* x, int* y)
{
    *x = (*x) + (*x);
    *y = (*x) + (*y) - 5;
}
```

**Code
9.22**

5. Using pass-by reference method, write a C function that takes three integers and replaces each number by its square.

Chapter 10

Pointers



10.1 Definition

Pointers are data types that are used to keep the address of memory locations where values of other variables are stored. A pointer variable is defined as

```
dataType* ptr;
```

where dataType can be any built-in data types available in C, such as int, float, double, struct, etc.

10.2 Address of a Variable

The address of a variable can be obtained using the & operator. For instance, the address of variable a in

```
int a = 10;
```

is obtained using &a, and the content of the address is obtained using the dereferencing operator *, that is,

```
*address = content of the address;
```

For instance, if

```
int a = 10;
```

then

`*(&a)` equals 10

Example 10.1 In this example, address and dereferencing operators are used together.

```
#include <stdio.h>
int main()
{
    int a = 10;
    printf("a = %d", *(&a));
}
```

Code
10.1

Output(s) a = 10

Example 10.2 Integers are 4-byte numbers. In this example, the storage address of an integer is displayed. Little-endian storage type is assumed.

```
#include <stdio.h>
int main()
{
    int a = 0x1245A78F; // 4-byte integer value
    int* ptr = &a;

    printf("Value of a is = %X \n", a);
    printf("Address of a is = %X \n", &a);
    printf("Pointer value is = %p \n", ptr);
}
```

Code
10.2

Output(s)

Value of a is = 1245A78F

Address of a is = 65FE14

Pointer value is = 0000000065FE14

It is seen from the above memory map that address of a or pointer value is the value of memory address where the least significant byte (little-endian) of the integer a is stored (Fig. 10.1).

Fig. 10.1 Address and content relations of Example 10.2

	Address	Content
ptr=&a=	0x000000000065FE14	0x8F
	0x000000000065FE15	0xA7
	0x000000000065FE16	0x45
	0x000000000065FE17	0x12

Fig. 10.2 Address and content relations of Example 10.4

	Address	Content
ptr=	0x000000000065FE14	0x8F
	0x000000000065FE15	0xA7
	0x000000000065FE16	0x45
	0x000000000065FE17	0x12

Example 10.3 Pointer is a data type, and its size can be calculated using the sizeof operator.

```
#include <stdio.h>
Code
10.3
int main()
{
    int a = 0x1245A78F;
    int* ptr = &a;

    printf("Pointer value is ptr = %p \n", ptr);
    printf("Sizeof pointer variable is : %lu \n", sizeof(ptr));
}
```

Output(s)

Pointer value is ptr = 000000000065FE14

Sizeof pointer variable is: 8

Example 10.4 Pointed address has content, and the content can be read by dereferencing operator (Fig. 10.2).

```
#include <stdio.h>
Code
10.4
int main()
{
    int a = 0x1245A78F;
    int* ptr = &a;

    printf("Pointer value is ptr = %p \n", ptr);
    printf("Value at address ptr = %p is *ptr = %X \n",ptr, *ptr);
}
```

Output(s)

Pointer value is ptr = 000000000065FE14

Value at address ptr = 000000000065FE14 is *ptr = 1245A78F

Example 10.5 Pointer data type is used for addresses, that is, pointers hold addresses, and these address values are stored in registers that have addresses as well (Fig. 10.3).

```
#include <stdio.h>
int main()
{
    int a = 0x1245A78F;
    int* ptr1 = &a;
    int* ptr2 = &ptr1;
    printf("Pointer value is ptr1 = %p \n", ptr1);
    printf("Pointer value is ptr2 = %p \n", ptr2);
}
```

Code
10.5

Output(s)

Pointer value ptr1 = 000000000065FE14

Pointer value ptr2 = 000000000065FE08

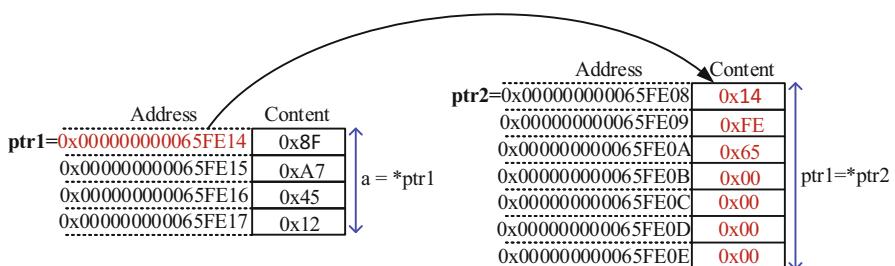


Fig. 10.3 Visual illustration of Example 10.5

Example 10.6 A pointer can point to the address of another pointer.

```
#include <stdio.h>
Code
10.6
int main()
{
    int a = 0x1245A78F;

    int* ptr1 = &a;

    int* ptr2 = &ptr1;

    int* ptr3 = &ptr2;

    printf("Pointer value ptr1 = %p \n", ptr1);
    printf("Pointer value ptr2 = %p \n", ptr2);
    printf("Pointer value ptr3 = %p \n", ptr3);
}
```

Output(s)

Pointer value ptr1 = 000000000065FE14

Pointer value ptr2 = 000000000065FE08

Pointer value ptr3 = 000000000065FE00

Pointed addresses and stored values are shown in (Fig. 10.4)

	Address	Content	Address	Content
ptr1=	0x000000000065FE14	0x8F	ptr2=	0x000000000065FE08
	0x000000000065FE15	0xA7		0x000000000065FE09
	0x000000000065FE16	0x45		0x000000000065FE0A
	0x000000000065FE17	0x12		0x000000000065FE0B
				0x000000000065FE0C
				0x000000000065FE0D
				0x000000000065FE0E

	Address	Content
ptr3=	0x000000000065FE00	0x08
	0x000000000065FE01	0xFE
	0x000000000065FE02	0x65
	0x000000000065FE03	0x00
	0x000000000065FE04	0x00
	0x000000000065FE05	0x00
	0x000000000065FE06	0x00

Fig. 10.4 Visual illustration of Example 10.6

Example 10.7 In this example, the second pointer points to the first pointer.

```
#include <stdio.h>
int main()
{
    int a = 0x1245A78F;

    int* ptr1 = &a;

    int* ptr2 = &ptr1;

    printf("Pointer value ptr1 = %p \n", ptr1);
    printf("Pointer value ptr2 = %p \n", ptr2);
    printf("*ptr2 = %p \n", *ptr2);
}
```

**Code
10.7**

Output(s)

Pointer value ptr1 = 000000000065FE14
 Pointer value ptr2 = 000000000065FE08
 $*\text{ptr2} = 000000000065FE14$

Property

Let

```
dataType a;
dataType* p=&a
```

If the value of p is v, then the value of $p+1$ is

```
v+sizeof(dataType)
```

Example 10.8 If ptr is a pointer pointing to an integer variable, the value of ptr+1 is 4 more than the value of ptr.

```
#include <stdio.h>
Code
10.8
int main()
{
    int a = 0x1245A78F;

    int* ptr = &a;

    printf("Value of ptr = %p \n", ptr);

    printf("Value of ptr+1 = %p \n", ptr+1);

    // value of ptr+1 is 4 more than value of ptr
}
```

Output(s)

Value of ptr = 000000000065FE14

Value of ptr+1 = 000000000065FE18

Example 10.9 If ptr is a pointer pointing to a long-long integer variable, the value of ptr+1 is 8 more than the value of ptr.

```
#include <stdio.h>
Code
10.9
int main()
{
    long long int a = 0x1245A78F1245A78F;

    long long int* ptr = &a;

    printf("Value of ptr = %p \n", ptr);

    printf("Value of ptr+1 = %p \n", ptr+1); // ptr+1 is 8 more than ptr
}
```

Output(s)

Value of ptr = 000000000065FE10

Value of ptr+1 = 000000000065FE18

Example 10.10 If ptr is a pointer pointing to a char variable, the value of ptr+1 is 1 more than the value of ptr.

```
#include <stdio.h>
Code
10.10
int main()
{
    char a = 'A';

    char* ptr = &a;

    printf("Value of ptr = %p \n", ptr);

    printf("Value of ptr+1 = %p \n", ptr+1);

    printf("Value of *ptr is %c \n", *ptr);
}
```

Output(s)

Value of ptr = 000000000065FE17

Value of ptr+1 = 000000000065FE18

Value of *ptr is A

Example 10.11 We can use pointers to change the value of a variable.

```
#include<stdio.h>
Code
10.11
int main()
{
    float a = 4.8;

    float* ptr = &a;

    *ptr = (*ptr) * 3.8 + 4.7;

    printf("a = %f", a);
}
```

Output(s)

a = 22.940001

10.3 NULL Pointer

A pointer having no address value is called a NUL pointer. NULL pointers can be obtained assigning NULL value to the pointer. If a pointer is not initialized to NULL value, it may have an arbitrary value, and it may be dangerous to assign a value to pointer arbitrarily.

Example 10.12 It is better convention to initialize the pointers.

```
#include <stdio.h>
int main()
{
    int* ptr1;
    int* ptr2 = NULL;
    printf("Pointer value ptr1 = %p \n", ptr1); //this is an arbitrary value
    printf("Pointer value ptr2 = %p \n", ptr2);
}
```

Code
10.12

Output(s)

Pointer value ptr1 = 00000000007213A0

Pointer value ptr2 = 0000000000000000

10.4 Void Pointer

Void pointer is also called generic pointer. Its pointer data type can be changed.

Example 10.13 The size of void pointer can be obtained using the sizeof operator.

```
#include <stdio.h>
int main()
{
    void* vp = 0; // or void* ip = NULL;
    printf("sizeof (void*) is %lu \n", sizeof(vp));
    printf("vp = %p", vp);
}
```

Code
10.13

Output(s)

sizeof (void*) is 8
vp = 0000000000000000

Example 10.14 Void pointers can point to any variable.

```
#include <stdio.h>
Code
10.14

int main()
{
    int a = 0x12345678;

    void* vp = NULL;

    printf("Before assignment\n");
    printf("sizeof (void*) is %lu \n", sizeof(vp));
    printf("vp = %p\n\n", vp);

    vp = &a;
    printf("After assignment\n");
    printf("vp = %p\n", vp);
    printf("sizeof (void*) is %lu", sizeof(vp));

    //printf("a = %x", *vp); // gives error
}
```

Output(s)

Before assignment
sizeof (void*) is 8
vp = 0000000000000000

After assignment
vp = 0000000000065FE14
sizeof (void*) is 8

Example 10.15 Void pointers can point to any data type, but casting must be used to print the value of the pointed data.

```
#include <stdio.h>

int main()
{
    int a = 0x12345678;

    void* vp = NULL;

    printf("Before assignment\n");
    printf("sizeof (void*) is %lu \n", sizeof(vp));
    printf("vp = %p\n\n", vp);

    vp = &a;
    printf("After assignment\n");
    printf("vp = %p\n", vp);
    printf("sizeof (void*) is %lu\n\n", sizeof(vp));
    //printf("a = %x", *vp); // gives error

    printf("If casting is used\n");
    printf("sizeof (int*) is %lu \n", sizeof((int*)vp));
    printf("vp = %p\n", (int*)vp);
    printf("a = *vp = 0x%x", *(int*)vp);
}
```

Code
10.15

Output(s)

Before assignment
sizeof (void*) is 8
vp = 0000000000000000

After assignment
vp = 000000000065FE14
sizeof (void*) is 8

If casting is used
sizeof (int*) is 8
vp = 000000000065FE14
a = *vp = 0x12345678

Example 10.16 The address pointed out by void pointer can be changed. Explicit type conversion should be done to print the value of the pointed variable.

```
#include <stdio.h>
Code
10.16
int main()
{
    int a = 0x12345678;
    char b = 'k';
    void* vp = NULL;
    vp = &a;
    printf("a = *vp = 0x%x\n\n", *(int*)vp);
    vp = &b;
    printf("b = *vp = %c", *(char*)vp);
}
```

Output(s)

a = *vp = 0x12345678

b = *vp = k

Example 10.17 The default increment amount for void pointer is 1; however, when void pointer is converted to any other pointer type, increment amount changes depending on the pointed data type.

```
#include <stdio.h>
Code
10.17
int main()
{
    int a = 0x12345678;
    void* vp = NULL;
    vp = &a;
    printf("No casting:\n");
    printf("vp = %p\n", vp);
    printf("vp+1 = %p\n\n", vp+1);
    printf("If (int*) casting is used:\n");
    printf("vp = %p\n", (int*)vp);
    printf("vp+1 = %p\n\n", (int*)vp+1);
}
```

Output(s)

No casting:

```
vp = 000000000065FE14
vp+1 = 000000000065FE15
```

If (int*) casting is used:

```
vp = 000000000065FE14
vp+1 = 000000000065FE18
```

Example 10.18 Void pointer can be converted to any other pointer type by explicit conversion.

<pre>#include <stdio.h> int main() { int a = 0x12345678; long long int b = 34.5; double d = 45.67; void* vp = NULL; vp = &a; printf("Integer is pointed, No casting:\n"); printf("vp = %p\n", vp); printf("vp+1 = %p\n\n", vp+1); printf("If (int*) casting is used:\n"); printf("vp = %p\n", (int*)vp); printf("vp+1 = %p\n\n", (int*)vp+1); vp = &b; printf("Long long integer is pointed, No casting:\n"); printf("vp = %p\n", vp); printf("vp+1 = %p\n\n", vp+1); printf("If (long long int*) casting is used:\n"); printf("vp = %p\n", (long long int*)vp); printf("vp+1 = %p\n\n", (long long int*)vp+1); vp = &d; printf("Double is pointed, No casting:\n"); printf("vp = %p\n", vp); printf("vp+1 = %p\n\n", vp+1); printf("If (double*) casting is used:\n"); printf("vp = %p\n", (double*)vp); printf("vp+1 = %p\n\n", (double*)vp+1); printf("Sizeof double is %lu.\n", sizeof(double)); }</pre>	Code 10.18
---	--

Output(s)

Integer is pointed, No casting:

`vp = 000000000065FE14`

`vp+1 = 000000000065FE15`

If (`int*`) casting is used:

`vp = 000000000065FE14`

`vp+1 = 000000000065FE18`

Long long integer is pointed, No casting:

`vp = 000000000065FE08`

`vp+1 = 000000000065FE09`

If (`long long int*`) casting is used:

`vp = 000000000065FE08`

`vp+1 = 000000000065FE10`

Double is pointed, No casting:

`vp = 000000000065FE00`

`vp+1 = 000000000065FE01`

If (`double*`) casting is used:

`vp = 000000000065FE00`

`vp+1 = 000000000065FE08`

Sizeof double is 8.

Example 10.19 Using void pointer, it is possible to get read value at any memory location (Fig. 10.5).

Fig. 10.5 Visual illustration of Example 10.19

	Address	Content	
<code>vp=&a=0x000000000065FE0C</code>	0x78		a
0x000000000065FE0D	0x56		
<code>cp=vp+2=0x000000000065FE0E</code>	0x34		*cp
0x000000000065FE0F	0x12		

```
#include <stdio.h>
int main()
{
    int a = 0x12345678;
    void* vp = NULL;
    vp = &a;

    printf("Integer is pointed, No casting:\n");
    printf("vp = %p\n", vp);

    char* cp = (char*) (vp+2);
    printf("cp = vp+2 = %p\n", cp);

    printf("The value at position vp+2 is ", vp+2);
    printf("%x", *cp);
}
```

Code
10.19

Output(s)

Integer is pointed, No casting:
vp = 00000000065FE0C

cp = vp+2 = 000000000065FE0E
The value at position vp+2 is 34

10.5 Types of Pointers

10.5.1 Pointer to a Constant Value

A pointer to a constant value is defined as

```
const dataType* ptr;
```

The data pointed by the pointer is a constant and its value cannot be changed. The pointer can change the pointed variable, that is, the pointed address may be changed but the content of the address cannot be changed.

Example 10.20 If address content is constant, it cannot be changed.

```
#include <stdio.h>
int main()
{
    int a = 45;
    int b = 67;

    const int* ip = &a; // ok, point to the address of a
    *ip = 40; // error, address content cannot be changed, it is constant
    ip = &b; // ok, pointed address can be changed
    *ip = 67; // error, address content cannot be changed
}
```

Code
10.20

Pointers to constant values are usually employed in function arguments to prevent the accidental change of the values used in function calls.

Example 10.21 Constant variable value can be changed by a pointer. Compare Code 10.20 with Code 20.21.

```
#include <stdio.h>
int main()
{
    const int a = 45;

    printf("a = %d \n", a); // a = 45
    int* ip = &a; // ok, point
    *ip = 89; // ok
    printf("a = %d \n", a); // a = 89
}
```

Code
10.21

Output(s)

a = 45
a = 89

Example 10.22 Function arguments can contain pointers to constant values.

```
#include <stdio.h>
int myFunc(const int* ip)
{
    int b = *ip+2; // ok

    *ip = *ip+2; // error

    return b;
}

int main()
{
    int a = 10, d;

    d = myFunc(&a);
}
```

Code
10.22

10.5.2 Pointer to a Constant Address (Constant Pointer)

The pointer points to a constant memory address, and the value at that address can be changed because it is a variable, but the pointer can only point to the same address and the pointed address cannot be changed. The syntax of the constant pointer is

```
dataType* const ptr = &varName
```

Note that initialization has to be performed when constant pointer is declared.

Example 10.23 For constant address pointers, the pointed variable cannot be changed.

```
#include <stdio.h>
int main()
{
    int a = 45;

    int b = 67;

    int* const ip = &a; // ok, point to the address of a
    *ip = 40; // ok, content can be changed

    ip = &b; // error, address is constant, it cannot be changed
}
```

Code
10.23

Example 10.24 For constant address pointers, initialization must be done when pointer is defined.

```
#include <stdio.h>
int main()
{
    int a = 45;
    int* const ip; // initialization must be done here
    ip = &a; // error, initialization should be done in the previous line
}
```

Code
10.24

10.5.3 Constant Pointer to a Constant Value

A constant pointer to a constant value is defined as

```
const dataType* const ptr = &varName
```

Example 10.25 For constant address, constant value pointers, neither pointed address nor content of the address can be changed.

```
#include <stdio.h>
int main()
{
    int a = 45, b = 67;
    const int* const ip = &a; // initialization must be done here
    *ip = 98; // error, content can not be changed
    ip = &b; // error, address can not be changed
}
```

Code
10.25

10.6 Function Pointers

Variables are stored in memory locations and each variable has a memory address; similarly instructions of a function are stored in memory and each function has an address.

The name of an array is a pointer to the head of the memory address where array values are stored. The name of a function is a pointer to the memory address where function instructions are stored.

Example 10.26 Function and array names are also pointers.

```
#include <stdio.h>
void myFnc()
{
    printf("Inside function \n");
}

int main()
{
    int a = 34;

    int b[3] = {3,5,6};

    printf("Address of a = %p \n", &a);

    printf("Address of array b = %p \n", b);

    printf("Address of the myFnc() = %p \n", myFnc);

    printf("Address of the myFnc() = %p", &myFnc); // second method
}
```

Code
10.26

Output(s)

Address of a = 00000000065FE1C

Address of array b = 00000000065FE10

Address of the myFnc() = 000000000401560

Address of the myFnc() = 000000000401560

Syntax of Function Pointer in C

The syntax of the function pointer is

```
returnedDataType (*PointerName) (argument1, argument2, ...);
```

For example, for the function declaration

```
int myFunc(int, int);
```

a function pointer in C can be defined as

```
float (*fncPointer) (int, int);
```

and function address is assigned to the pointer using either

```
fncPointer= myFunc
```

or

```
fncPointer=&myFunc
```

and the function is called using the function pointer using either

```
fncPointer(intVal1, intVal2);
```

or

```
(*fncPointer) (intVal1, intVal2);
```

Example 10.27 This example illustrates the use of the function pointers.

<pre>#include <stdio.h> int mySum(int a, int b) { return a+b; } int main() { int a = 34, b = 65, sm; int (*fp)(int, int); // define a function pointer fp = mySum; // assign function address to the function pointer, // alternative assignment is fp=&mySum sm = fp(a, b); // or use sm = (*fp)(a, b); printf("Sum of the integers %d and %d is %d", a, b, sm); }</pre>	Code 10.27
--	---

Output(s)

Sum of the integers 34 and 65 is 99

Example 10.28 Explain the prototype

```
void* (*func_ptr) (int *, int *);
```

We should inspect such statements inside out. Here,

```
(*func_ptr) (int *, int *)
```

is a function pointer taking two inputs, which are integer pointers, and the return expression

```
void*
```

is a void pointer

Example 10.29 It is possible to define an array of function pointers.

```
#include <stdio.h>
int mySum(int a, int b){ return a+b;}
int mySubt(int a, int b){return a-b;}
int myMult(int a, int b){return a*b;}

int main()
{
    int a = 34, b = 65, sm, df, ml;

    int (*fp[3])(int, int); // define an array of function pointers

    fp[0] = mySum;//assign mySum function address to the function pointer
    fp[1] = mySubt;//assign mySubt function address to the function pointer
    fp[2] = myMult;//assign myMult function address to the function pointer

    sm = fp[0](a, b);
    df = fp[1](a, b);
    ml = fp[2](a, b);

    printf("a = %d b = %d, a+b = %d\n", a, b, sm);
    printf("a = %d b = %d, a-b = %d\n", a, b, df);
    printf("a = %d b = %d, a*b = %d\n", a, b, ml);
}
```

Code
10.28

Output(s)

a = 34 b = 65, a+b = 99
 a = 34 b = 65, a-b = -31
 a = 34 b = 65, a*b = 2210

Example 10.30 Functions can have function pointers in their arguments, and function names can be passed for function pointers.

```
#include <stdio.h>

void myFunc1 ()
{
    printf("Inside function-1\n");
}
void myFunc2 ()
{
    printf("Inside function-2\n");
}
void callFunc(void (*fp) ())
{
    fp();
}
```

```
int main()
{
    callFunc(myFunc1);
    callFunc(myFunc2);
}
```

Code
10.29

Output(s)

Inside function-1
 Inside function-2

Example 10.31 In this example, we define a function pointer in main function and pass it to a function whose arguments involve a function pointer.

```
#include <stdio.h>
int squaredSum(int a, int b, int (*fp)())
{
    int sqSum;

    sqSum = fp(a*a, b*b);

    return sqSum;
}

int mySum(int a, int b)
{
    return a+b;
}

int main()
{
    int a = 5, b = 4, sm;

    int (*fp)(int, int); // define a function pointer

    fp = mySum; // assign function address to the function pointer,

    int res = squaredSum(a, b, fp); // call the function squaredSum

    printf("Squared sum of the integers %d and %d is %d.", a, b, res);
}
```

Code
10.30

Output(s)

Squared sum of the integers 5 and 4 is 41.

10.6.1 Functions Returning Pointers

Pointer is a data type, its size is 8. Function can return pointers as returned values.

Example 10.32 Pointer values are addresses, and a function can return an address of a global variable.

```
#include <stdio.h>
int a = 18; // global variable
int* myFunc()
{
    return (&a);
}
int main()
{
    int* ptr;
    ptr = myFunc();
    printf("Pointed address is %p\n", ptr); // ok
    printf("Value at address is %d\n", *ptr); // ok
}
```

Code
10.31

Output(s)

Pointed address is 0000000000403010

Value at address is 18

Example 10.33 Local variables inside a function are destroyed when function is terminated. Returning local variable addresses creates a problem.

```
#include <stdio.h>
int* myFunc()
{
    int a = 18;// local variable
    return (&a); // local variables are destroyed, when function is quitted
}
int main()
{
    int* ptr;
    ptr = myFunc();
    printf("Pointed address is %p\n", ptr); // wrong output
    printf("Value at address is %d\n", *ptr); // no output
}
```

Code
10.32

Output(s)

Pointed address is 0000000000000000

warning: function returns address of local variable [-Wreturn-local-addr]

Example 10.34 Functions can be called by pass-by value method. In this case, the values in the function arguments can be considered local variables inside function. Returning the address of argument values creates a problem.

```
#include<stdio.h>
float* findLarger(float*, float*);

int main()
{
    float a = 24.6, b = 56.8;

    float* larger_num;

    larger_num = findLarger(&a, &b);

    printf("Larger of %.1f and %.1f is %.1f", a, b, *larger_num);
}

float* findLarger(float* c, float* d)// pass by value is used here
{
    // c and d can be considered as local
    // variables
    if (*c > *d)
    {
        return c;
    }
    else
        return d;
}
```

Code
10.33

Output(s)

warning: function returns address of local variable [-Wreturn-local-addr]

10.7 Pointers and Arrays

Pointers and arrays are closely related to each other. Array names can be considered constant pointers pointing to the first element of the array.

Example 10.35 A string itself indicates a constant address in memory.

```
#include<stdio.h>
int main()
{
    char* a = "Hello";
    printf("a is %p\n",a);
    printf("&a is %p\n\n",&a);
    printf("Constant address is %p\n","Hello");
    printf("&Constant address is %p\n",&"Hello");
}
```

Code
10.34

Output(s)

a is 0000000000404000

&a is 000000000065FE18

Constant address is 0000000000404000

&Constant address is 0000000000404000

In the expression `char* a`, the letter a indicates an address, and the assigned value should be an address, thus the string "Hello" on the right-hand side of `char* a="Hello";` is also an address.

The constant on the right-hand side of `char* a="Hello";` is stored on the code memory.

On the other hand, the pointer variable "a" is stored on the stack memory.

Example 10.36

```
#include<stdio.h>
int main()
{
    char* a = "Hello";
    char b[6] = "Hello";
    printf("Size of a is %lu \n",sizeof(a));
    printf("Size of b is %lu",sizeof(b));
}
```

Code
10.35

Output(s)

Size of a is 8

Size of b is 6

In this example, “a” is a pointer, its size is 8, and “b” is an array variable, its size is 6, that is, it is an array of six characters.

Example 10.37

```
#include<stdio.h>
int main()
{
    char* a = "Hello";
    char b[6] = "Hello";
    printf("Size of a is %lu \n", sizeof(a));
    printf("Size of b is %lu \n", sizeof(b));
    printf("b is %p \n", b);
    printf("&b is %p \n", &b);
    printf("Size of &b is %lu \n", sizeof(&b));
}
```

**Code
10.36**

Output(s)

Size of a is 8

Size of b is 6

b is 0x7ffeb161d182

&b is 0x7ffeb161d182

Size of &b is 8

In this example, we see the strange sides of C compilers. Although the size of b is 6, it can be displayed as pointer. b and &b have the same values. The size of &b is 8.

Example 10.38 Array name is a constant pointer, its value is constant, and it cannot be modified.

```
#include<stdio.h>
Code
10.37
int main()
{
    char* a = "Hello";
    char b[6] = "Hello";
    printf("a is %p \n",a); // ok
    printf("++a is %p \n",++a); // ok
    printf("++b is %p",++b); // error
}
```

Example 10.39 A string can be considered an address. It cannot be assigned to an array name. Since array name is a constant address, it cannot be changed.

```
#include<stdio.h>
Code
10.38
int main()
{
    char* a = "Hello";
    char b[6] = "Hello";
    a = "World"; // ok
    b = "World"; // error
}
```

Example 10.40 In this example, we explain the differences between a character array and a character pointer initialized with a string.

```
#include<stdio.h>
int main()
{
    char* a = "Hello";
    const char* a1 = "Hello";
    char b[6] = "Hello";
    char c[6] = {'H','e','l','l','o','\0'};
    printf("%s %s %s \n",a, b, c);
    // a[1] = 'A'; // Does not work
    b[1] = 'A'; // ok
    c[1] = 'A'; // ok
    printf("%s %s %s",a, b, c);
}
```

**Code
10.39**

Output(s)

Hello Hello Hello
Hello HAllo HAllo

Differences between **char* a="Hello"** and **char b[]="Hello"**

char* a="Hello";
a is a pointer
pointer size is 8
a and &a are different
a is at the stack memory but "Hello" is stored at the code section of memory
a="Hallo" is valid also an address
a++ is valid
a[1]='A' is invalid, code section is read-only memory

char b[]="Hello";
b is a variable name for an array
sizeof(b) is 6
b and &b are the same
"Hello" is stored in the stack
b="Hallo" is not valid
b is an address and string constant is also an address
b++ is invalid
b[1]="A" is valid

Example 10.41 Array names are constant pointers. They can be assigned to nonconstant pointers.

```
#include<stdio.h>
Code
10.40
int main()
{
    int a[] = {41, 25, 38, 47, 58} ;
    int* ptr = a;
    printf("%d, %d", ptr[2], *(ptr + 2));
}
```

Output(s) 38, 38

Example 10.42 We can use pointers to access array elements.

```
#include<stdio.h>
Code
10.41
int main()
{
    int a[3] = {1,2,3};
    int* b;
    b = a;
    printf("b[0] = %d    b[1] = %d    b[2] = %d \n",b[0],\n
           b[1], b[2]);
}
```

Output(s) b[0] = 1 b[1] = 2 b[2] = 3

Example 10.43 Although array name behaves as a pointer, its size depends on the number and types of its elements.

```
#include<stdio.h>
Code
10.42
int main()
{
    int a[] = {3, 4, 5};
    int* ip = a;
    printf("Size of a[] is %lu \n", sizeof(a) );
    printf("Size of ip is %lu \n", sizeof(ip) );
}
```

Output(s)

Size of a[] is 12

Size of ip is 8

Example 10.44 Array names behave as constant pointers.

```
#include<stdio.h>
int main()
{
    int a[] = {3, 4, 5}, b = 23;
    int* ip = &b; // ok
    a = &b; // error
    a = ip; // error
}
```

Code
10.43

If **a** is an array name, ptr is a pointer, and ptr=a, array elements can be accessed using one of

$$a[i]^*(a + i) \quad \text{ptr}[i]^*(\text{ptr} + i)$$

where i is the index value.

Example 10.45 Array elements can be accessed in different ways.

```
#include<stdio.h>
int main()
{
    int a[] = {3, 4, 5};
    int* ip = a; // ok

    printf("Array elements are \n\n");
    printf("a[0] = %d a[1] = %d a[2] = %d \n", a[0], a[1], a[2]);
    printf("a[0] = %d a[1] = %d a[2] = %d \n", *a, *(a+1), *(a+2));
    printf("a[0] = %d a[1] = %d a[2] = %d \n", ip[0], ip[1], ip[2]);
    printf("a[0] = %d a[1] = %d a[2] = %d \n", *ip, *(ip+1), *(ip+2));
}
```

Code
10.44**Output(s)**

Array elements are

a[0] = 3 a[1] = 4 a[2] = 5

a[0] = 3 a[1] = 4 a[2] = 5

```
a[0] = 3  a[1] = 4  a[2] = 5  
a[0] = 3  a[1] = 4  a[2] = 5
```

10.8 Multiple Pointers

It is possible to define a pointer that points to another pointer.

Example 10.46 In this example, we explain the use of multiple pointers.

```
#include <stdio.h>  
  
int main()  
{  
    int a = 0xABCD;  
  
    int* ip1, ** ip2, *** ip3;  
  
    /* int* ip1;  
     *int** ip2;  
     *int*** ip3; */  
  
    ip1 = &a;  
    ip2 = &ip1;  
    ip3 = &ip2;  
  
    printf("Pointer value ip1 = %p\n", ip1);  
    printf("Pointer value ip2 = %p\n", ip2);  
    printf("Pointer value ip3 = %p\n\n", ip3);  
  
    printf("*ip3 = %p\n", *ip3);  
    printf("**ip3 = %p\n", **ip3);  
    printf("****ip3 = %X\n", ***ip3);  
}
```

**Code
10.45**

Output(s)

Pointer value ip1 = 00000000065FE14
Pointer value ip2 = 0000000000065FE08
Pointer value ip3 = 0000000000065FE00

*ip3 = 0000000000065FE08
**ip3 = 0000000000065FE14
***ip3 = ABCD

The relationships between addresses are explained in Fig. 10.6.

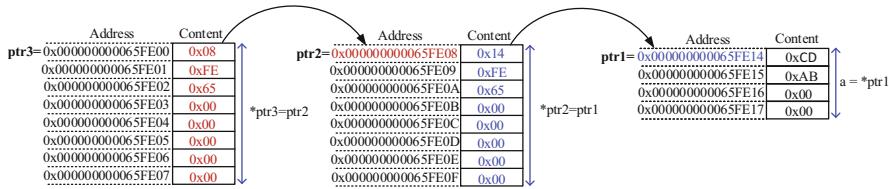


Fig. 10.6 Visual illustration of Example 10.46

Example 10.47 A pointer pointing to another pointer holds the address of another pointer.

Code 10.46

```
#include <stdio.h>

int main()
{
    int a = 36;

    int* ip1, ** ip2, *** ip3;

    ip1 = &a;
    ip2 = &ip1;
    ip3 = &ip2;

    printf("Pointer value ip1 = %p\n", ip1);
    printf("Pointer value ip2 = %p\n", ip2);
    printf("Pointer value ip3 = %p\n\n", ip3);

    unsigned char* ucp=(unsigned char*) ip3;

    printf("ucp = %p\n", ucp);
    printf("*ucp = %.2X \n\n", *ucp);

    ucp++;
    printf("ucp++ = %p\n", ucp);
    printf("*ucp = %X \n\n", *ucp);

    ucp++;
    printf("ucp++ = %p\n", ucp);
    printf("*ucp++ = %X \n\n", *ucp);
}
```

Output(s)

```

Pointer value ip1 = 000000000065FE0C
Pointer value ip2 = 000000000065FE00
Pointer value ip3 = 000000000065FDF8

ucp = 000000000065FDF8
*ucp = 00

ucp++ = 000000000065FDF9
*ucp = FE

ucp++ = 000000000065FDFA
*ucp++ = 65

```

Exercise Draw the memory maps for this example and show the relationships between pointers.

10.9 Heap Stack and Code Memories

The variables, functions, instruction codes, and dynamically created variables are stored in different parts of the memory. In Fig. 10.7, different parts of the memory and their relations to code parameters are explained.

In the sample code in Fig. 10.8, it is illustrated that dynamic allocation is performed in the heap section of the program memory.

Example 10.48 In Code 10.47, all these variables are stored on stack.

<pre> int main() { int a; int b[6]; int c = 34; double d = 4.5; float e; int* p; } </pre>	Code 10.47
---	-----------------------

Example 10.49 In Code 10.48, memory allocation is performed for five double numbers on the heap.

<pre> int main() { double *dp = malloc(5*sizeof(double)); } </pre>	Code 10.48
--	-----------------------

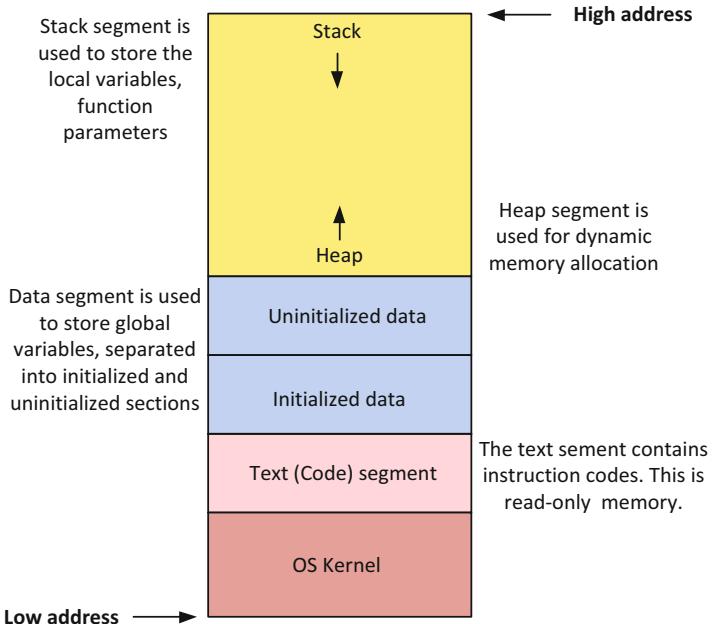


Fig. 10.7 Heap stack and code memories

```
#include <stdio.h>

int main()
{
    int* ip=malloc(sizeof(int));
    double* dp=malloc(2*sizeof(double));
    *ip=45;
    dp[0]=34.5;
    dp[1]=67.8;
}
```

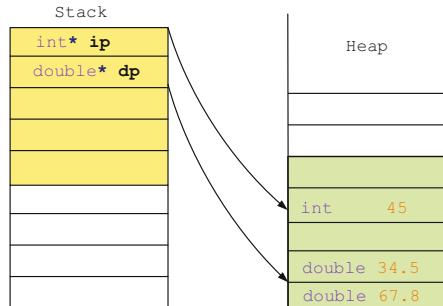


Fig. 10.8 Dynamic memory allocation

10.10 Dynamic Memory Allocation

There are four functions provided by C language to perform dynamic memory allocation operations; these functions are defined in the header file `<stdlib.h>` as follows:

`malloc()` `calloc()` `free()` `realloc()`

10.10.1 *malloc()* Function

The malloc() function is used as

```
dataType* ptr = (dataType*) malloc(total-size_to_be_allocated)
```

The allocation is performed in the heap memory. When the expression

```
int* ip = (int*) malloc(20 * sizeof(int));
```

is performed, a total of 20×4 bytes = 80 bytes are allocated in the heap memory. Note that each integer requires 4 bytes of memory location. For 20 integers, a total size of 80 bytes is allocated.

In C11, aligned_alloc function is defined as

```
void* aligned_alloc(size_t alignment, size_t size);
```

where `size_t` is `unsigned int` data type defined in the header file `<stddef.h>`. When the expression

```
char* ptr = aligned_alloc(256, 1024);
```

is performed, 1024 bytes are allocated with 256 bytes alignment. Memory alignment decreases the memory access time. That is, performance of the program increases. However, alignment involves padding and some of the allocated memory regions are never used; they are used for zero padding. For this reason, efficient memory use decreases. For this example, although 1024 bytes of memory are allocated, in fact more than this amount is used for this allocation. The tradeoff between execution speed and amount of memory used should be considered when C functions with memory alignments are to be used.

Example 10.50 In this example, we allocate memory location for two integers and explain the allocated regions (Fig. 10.9).

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* ip;

    ip = (int*)malloc(2 * sizeof(int));

    if (ip == NULL)
    {
        printf("Memory allocation failed.\n");
        exit(0);
    }
    else
    {
        printf("Memory allocation is successful.\n");

        printf("ip = %p", ip);
    }
}
```

**Code
10.49**

Output(s)

Memory allocation is successful.

ip = 00000000001E1400

When memory allocation is performed using malloc() function, a number of consecutive memory locations are reserved for storage.

This can be considered as reserving a number of tables in a restaurant. However, this does not mean that nonreserved tables can be used. If they are free, they can be used. However, there is no guarantee that free tables are available.

Fig. 10.9 Visual illustration for Example 10.50.

ip = 0x00000000001E1400	?
0x00000000001E1401	?
0x00000000001E1402	?
0x00000000001E1403	?
0x00000000001E1404	?
0x00000000001E1405	?
0x00000000001E1406	?
0x00000000001E1407	?

Example 10.51 In this example, memory allocation is performed for two integers. The malloc function returns a pointer to the head of the allocated block.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* ip;

    ip = (int*)malloc(2 * sizeof(int));

    if (ip == NULL)
    {
        printf("Memory allocation failed.\n");
        exit(0);
    }
    else
    {
        printf("Memory allocation is successful.\n");

        printf("ip = %p\n", ip);

        printf("ip[0] = %d\n", ip[0]); // reserved, have arbitrary value
        printf("ip[1] = %d\n", ip[1]); // reserved, have arbitrary value

        printf("ip[2] = %d\n", ip[2]);
        printf("ip[3] = %d\n", ip[3]);
        printf("ip[4] = %d\n", ip[4]);
        printf("ip[5] = %d\n", ip[5]);
    }
}
```

Code
10.50

In this example, “ip” holds the address of the head of the allocated block, and, ip +3, ip+4 are the addresses of the consecutive locations, but they are not reserved; however, these locations may also be accessible. They hold arbitrary values.

Output(s)

Memory allocation is successful.

ip = 000000000A21400

ip[0] = A261F0

ip[1] = 0

ip[2] = A20150

ip[3] = 0

ip[4] = 454B5F49

ip[5] = 6F6E3D59

Example 10.52 In this example, we dynamically allocate memory and write values to allocated addresses. Besides, we also try to write values for nonallocated addresses.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* ip;

    ip = (int*)malloc(2 * sizeof(int));

    if (ip == NULL)
    {
        printf("Memory allocation failed.\n");
        exit(0);
    }
    else
    {
        printf("Memory allocation is successful.\n");

        printf("ip = %p \n\n", ip);

        ip[0] = 0x34; // ok, reserved location
        ip[1] = 0x23456789; // ok, reserved location

        ip[2] = 0xABCD3456; // may give error, non-reserved location
        ip[3] = 0x6783DEFA; // may give error, non-reserved location

        printf("ip[0] = %X \n", ip[0]);
        printf("ip[1] = %X \n\n", ip[1]);

        printf("ip[2] = %X \n", ip[2]);
        printf("ip[3] = %X \n", ip[3]);
    }
}
```

Code
10.51

Output(s)

Memory allocation is successful.

ip = 000000000B01400

ip[0] = 34
 ip[1] = 23456789

ip[2] = ABCD3456
 ip[3] = 6783DEFA

10.10.2 *calloc()* Function

“calloc” means “contiguous allocation.”

It is similar to malloc() but the reserved locations are initialized with 0. The prototype of calloc function is

```
dataType* ptr = (dataType*) calloc(number of elements, dataType size);
```

Example 10.53 When calloc is used for memory allocation, the allocated registers are initialized with zeros.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* ip;

    ip = (int*)calloc(3, sizeof(int));

    if (ip == NULL)
    {
        printf("Memory allocation failed.\n");
        exit(0);
    }
    else
    {
        printf("Memory allocation is successful.\n");

        printf("ip = %p \n\n", ip);

        printf("ip[0] = %d \n", ip[0]);
        printf("ip[1] = %d \n", ip[1]);
    }
}
```

**Code
10.52**

Output(s)

Memory allocation is successful.

ip = 000000000B31400

ip[0] = 0

ip[1] = 0

10.10.3 *free()* Function

The function free() is used to deallocate the memory. It is used as

```
free(ptr);
```

where ptr is the pointer pointing to a memory location allocated by malloc or calloc functions.

Example 10.54 Free function can be used for both malloc and calloc allocated regions.

<pre>#include <stdio.h> #include <stdlib.h> int main() { float* fp1, * fp2; fp1 = (float*)malloc(2 * sizeof(float)); fp2 = (float*)calloc(3, sizeof(float)); if (fp1 == NULL fp2 == NULL) { printf("Memory not allocated.\n"); exit(0); } else { printf("Malloc allocation is successful.\n"); free(fp1); printf("Malloc allocated memory is freed.\n\n"); printf("Calloc allocation is successful.\n"); free(fp2); printf("Calloc allocated memory is freed.\n"); } }</pre>	Code 10.53
---	---

Output(s)

Malloc allocation is successful.

Malloc allocated memory is freed.

Calloc allocation is successful.

Calloc allocated memory is freed.

10.10.4 *realloc()* Function

The size of a previously allocated memory can be changed using the realloc() function; its prototype is

```
ptr = realloc(ptr, new size);
```

where ptr points to a previously allocated memory location. New locations are initialized with arbitrary values.

Example 10.55 In this example, realloc function is used to increase the amount of reserved memory from two integer size to four integer size.

Code
10.54

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int* ip;

    ip = (int*)malloc(2*sizeof(int));

    if (ip == NULL)
    {
        printf("Memory could not be allocated. \n");
        exit(0);
    }
    else
    {
        printf("Memory is allocated. \n");
        ip[0] = 43;
        ip[1] = 56;

        printf("ip[0] = %d  ", ip[0]);
        printf("ip[1] = %d  \n\n", ip[1]);

        ip = realloc(ip, 4 * sizeof(int));

        printf("Memory is re-allocated. \n");

        ip[2] = 78;
        ip[3] = 95;

        for (int i = 0; i < 4; i++)
        {
            printf("ip[%d] = %d  ", i, ip[i]);
        }

        free(ip);
    }
}
```

Output(s)

Memory is allocated.

ip[0] = 43 ip[1] = 56

Memory is re-allocated.

ip[0] = 43 ip[1] = 56 ip[2] = 78 ip[3] = 95

10.11 Memory Functions

10.11.1 Memset Function

The prototype of the memset() function is

```
void* memset(void* ptr, int ch, size_t N)
```

The memset() function sets the first N bytes starting from address ptr to the value ch. Although ch is of type int, it is converted to unsigned character. This function is defined in the header file <string.h>.

Example 10.56 In this example, the allocated memory is initialized with four “A” letters.

<pre>#include <stdio.h> #include <stdlib.h> #include <string.h> // void *memset(void* ptr, int ch, size_t N); int main() { unsigned char* ptr; ptr = (unsigned char*) malloc(4 * sizeof(unsigned char)); memset(ptr, 'A', 4); printf("%c, %c, %c, %c", ptr[0], ptr[1], ptr[2], ptr[3]); }</pre>	Code 10.55
--	--

Output(s) A, A, A, A

Example 10.57 Integer numbers can also be used with memset function to initialize allocated memory locations.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    unsigned int* ptr;

    ptr = (unsigned int*) malloc(3 * sizeof(unsigned int));

    memset(ptr, 0x56, 12);

    printf("%X \n", ptr[0]);
    printf("%X \n", ptr[1]);
    printf("%X \n", ptr[2]);
}
```

Code
10.56

Output(s)

56565656

56565656

56565656

10.11.2 Memcpy Function

The prototype of the memcpy() function is

```
void* memcpy(void* destination, const void *source, size_t N)
```

The memcpy() function copies the first N bytes from a memory block, whose starting address is source, to the memory block whose starting address is destination. The function returns a pointer to the destination address. If source and destination addresses overlap, undefined behavior may be observed when memcpy() function is used.

Example 10.58 In this example, we illustrate the use of memcpy function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// void *memset(void* ptr, int ch, size_t N);
// void* memcpy(void* destination, const void *source, size_t N)

int main()
{
    unsigned char* ptr_src;
    unsigned char* ptr_dest;

    ptr_src = (unsigned char*) malloc(4 * sizeof(unsigned char));
    ptr_dest = (unsigned char*) malloc(4 * sizeof(unsigned char));

    memset(ptr_src, 'A', 4);

    memcpy(ptr_dest, ptr_src, 4);

    printf("%c, %c, %c, %c", ptr_dest[0], ptr_dest[1], ptr_dest[2], \
           ptr_dest[3]);
}
```

Code
10.57

Output(s) A, A, A, A

10.11.3 Memmove Function

The prototype of the memmove() function is

```
void* memmove(void* destination, const void *source, size_t N)
```

The memmove() function moves the first N bytes from a memory block, whose starting address is source, to the memory block whose starting address is destination. The function returns a pointer to the destination address. The move operation is different from the copy operation. After move operation, source addresses do not contain old data, that is, consider moving your house, old house is taken by others. Even if source and destination addresses overlap, memmove() function works fine.

Example 10.59 In this example, we illustrate the use of memmove function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// void *memset(void* ptr, int ch, size_t N);
// void* memcpy(void* destination, const void *source, size_t N)

int main()
{
    unsigned char* ptr_src;
    unsigned char* ptr_dest;

    ptr_src = (unsigned char*) malloc(4 * sizeof(unsigned char));
    ptr_dest = (unsigned char*) malloc(4 * sizeof(unsigned char));

    memset(ptr_src, 'A', 4);

    memmove(ptr_dest, ptr_src, 4);

    printf("Destination values are: %c, %c, %c, %c \n",
           ptr_dest[0], ptr_dest[1], ptr_dest[2], ptr_dest[3]);

    printf("After move, source values are: %c, %c, %c, %c \n",
           ptr_dest[0], ptr_dest[1], ptr_dest[2], ptr_dest[3]);
}
```

Code
10.58

Output(s)

Destination values are: A, A, A, A

After move, source values are: A, A, A, A

10.11.4 Memcmp Function

The prototype of the memcmp() function is

```
int memcmp(const void* ptr1, const void* ptr2, size_t N);
```

The memcmp() function compares the content of the memory block pointed by ptr1 containing N bytes to the memory content of the memory block pointed by ptr2 containing N bytes, and return value is

- Negative if the content of block-1 is less than the content of block-2
- Zero if the contents are equal to each other
- Positive if the content of block-1 is greater than the content of block-2

Example 10.60 In this example, we illustrate the use of memcmp function.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    unsigned int* ptr1;
    unsigned int* ptr2;

    ptr1 = (unsigned int*) malloc(2 * sizeof(unsigned int));
    ptr2 = (unsigned int*) malloc(2 * sizeof(unsigned int));

    ptr1[0] = 0x11111111; ptr1[1] = 0x11111111;
    ptr2[0] = 0x11111111; ptr2[1] = 0x21111111;

    int result = memcmp(ptr1, ptr2, 8);

    printf("Result is : %d", result);
}
```

Code
10.59

Output(s) Result is: -1

Problems

1. An integer variable is defined as

```
int a = 56;
```

Print the address value of this variable.

2. What is the output of Code 10.60?

```
#include <stdio.h>
int main()
{
    int a = 56;
    printf("a = %d", *(&a));
}
```

Code
10.60

3. What is the output of Code 10.61?

```
void myFunc(float*);  
  
int main()  
{  
    float a = 4.8;  
  
    myFunc(&a);  
  
    printf("%.1f", a);  
}  
void myFunc(float* fp)  
{  
    *fp = 7.9;  
}
```

Code
10.61

4. What are the outputs of Code 10.62?

```
#include <stdio.h>  
  
int main()  
{  
    float a = 3.4;  
  
    float* fp = &a;  
  
    *fp += 2.3;  
  
    printf("a = %.1f \n", a);  
  
    printf("*fp = %.1f \n", *fp);  
}
```

Code
10.62

5. Run the code in Code 10.63, and by drawing on a paper show how integer value is stored in memory.

```
#include <stdio.h>
Code
10.63

int main()
{
    int a = 0xACDEF9D4; // 4-byte integer value

    int* ptr = &a;

    printf("Value of a is = %X \n", a);
    printf("Address of a is = %X \n", &a);
    printf("Pointer value is = %p \n", ptr);
}
```

6. Assume that you have a pointer ptr pointing to an integer variable. If the value of pointer ptr is 0, then what is the value of ptr+1?
 7. What is the size of a pointer variable?
 8. What is the output of Code 10.64?

```
#include<stdio.h>
Code
10.64

int main()
{
    int myArray[] = {1, 2, 3, 4, 5, 6, 7, 8};

    int *ptr1 = myArray;
    int *ptr2 = myArray + 3;

    printf("Number of integers between two addresses are: %d\n",
           ptr2 - ptr1);

    printf("Number of bytes between two addresses are: %d",
           (char*)ptr2 - (char*) ptr1);
}
```

9. What is the output of Code 10.65?

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

    char* cp;

    cp = (char*) &a;

    a = 0xAAAABBBB;

    cp[0] = 0xDD;

    cp[1] = 0xEE;

    printf("%X\n", a);
}
```

Code
10.65

10. Find the error in Code 10.66.

```
#include <stdio.h>
int main()
{
    int a = 67;

    int b = 89;

    const int* ip=&a;

    *ip = 40;

    ip = &b;

    *ip = 67;
}
```

Code
10.66

11. Find the error in Code 10.67.

Code
10.67

```
#include <stdio.h>

void myFunc(const int* ip)
{
    *ip = *ip+2;
}

int main()
{
    int a = 10;

    myFunc(&a);
}
```

12. Find the error in Code 10.68.

Code
10.68

```
#include <stdio.h>

int main()
{
    int a = 45;

    int* const ip;
    ip=&a;
}
```

13. Find the error in Code 10.69.

Code
10.69

```
#include <stdio.h>

int main()
{
    float myArray[5] = {16.5, 70.6, 56.5, 36.4, 56.5};

    float* fp = myArray;

    fp = fp +2;

    printf("%.1f", *fp);
}
```

14. What is the difference between heap and stack memory?
15. What is the output of Code 10.70?

```
#include<stdio.h>
void myFunc(int** ptr)
{
    **ptr = 35;
}

int main()
{
    int a = 10, *ptr1, **ptr2;

    ptr1 = &a;

    ptr2 = &ptr1;

    myFunc(ptr2);

    printf("a = %d ",a);
}
```

Code
10.70

Chapter 11

Directives and Macros in C



11.1 Introduction

The program lines that start have a symbol, `#`, called directives, and directives are processed with a built-in program called preprocessor, where the prefix `-pre-` implies that preprocessors process codes before compilation. A directive has syntax

```
# ...
```

When the symbol `#` is met inside the code, the preprocessor replaces the corresponding directive with a code, and we get a code without the `#` symbol, then the compiler compiles the code. The operation of directives is illustrated in Fig. 11.1.

We use preprocessor directives for

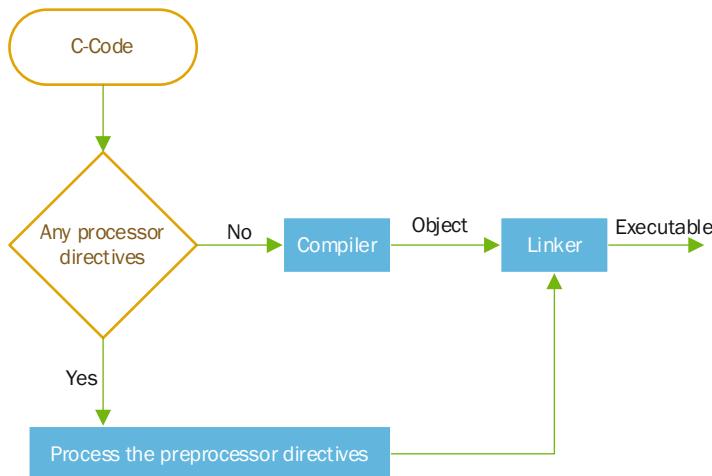
- Writing macros
- File inclusion
- Conditional compilation

11.2 Preprocessor Directives as Macros

A macro is defined using the syntax

```
#define macroName macroValue
```

and after preprocessing, the `macroName` will be replaced by `macroValue` in the program whenever it is met.

**Fig. 11.1** Macros are preprocessed

Example 11.1 We define two macros called myInteger and myFloat.

```
#include <stdio.h>
#define myInteger 12
#define myFloat 45.8

int main()
{
    printf("myInteger = %d \n", myInteger);

    double r = myInteger + myFloat;

    printf("r = %.1lf \n", r);
}
```

**Code
11.1**

Output(s)

myInteger = 12
r = 57.8

Example 11.2 The mathematical number π can be defined as macro and can be used to calculate the area of a circle.

```
#include<stdio.h>
#define PI 3.14

int main()
{
    int r;

    printf("Please enter radius of the circle :");

    scanf("%d", &r);

    float a = PI * (r*r);      // area of a circile

    printf("Circle are is %.2f", a);
}
```

**Code
11.2**

Output(s)

Please enter radius of the circle :4
 Circle are is 50.24

11.3 Macros as Functions

Although a macro can perform a similar task to a function, they are completely different things. Macros are preprocessed before the entire program is compiled. On the other hand, functions are compiled, they are NOT preprocessed.

Example 11.3 In Code 11.3, we define a macro that acts as a function.

```
#include <stdio.h>
#define macFnc(a, b) (a > b) ? a+b : a-b

int main()
{
    int a = 6, b = 8;

    printf("result = %d \n", macFnc(a, b));
}
```

**Code
11.3**

Output(s) result = -2

Example 11.4 In this example, we calculate the discriminant of a second-order mathematical equation using macro.

```
#include <stdio.h>
#define macFnc(a, b, c) ((b)*(b)-4*(a)*(c)) / (2*(a))

int main()
{
    int a = 1, b = -5, c = 4;
    printf("result = %d \n", macFnc(a, b, c));
}
```

Code
11.4**Output(s)** result = 4**Example 11.5** In this example, the area of a circle is calculated using a macro.

```
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)

int main()
{
    float r, a;

    printf("Please enter radius of the circle :");
    scanf("%f", &r);

    a = circleArea(r);

    printf("Circle are is %.2f", a);
}
```

Code
11.5**Output(s)**

Please enter radius of the circle :4

Circle are is 50.26

11.4 Multiline Macros

We can write multiline macros by placing “\” at the end of each line.

Example 11.6 In multiline macros, at the end of each line there is “\”.

```
#include <stdio.h>
#define DISP(start, N) \
    for(int indx = start; indx < N; indx++) \
    { \
        printf("Hello World!\n"); \
    }
int main()
{
    DISP(0, 3);
}
```

Code
11.6

Output(s)

Hello World!
Hello World!
Hello World!

Example 11.7 In the previous example, macro is called by some inputs. In this example, macro is called by just its name.

```
#include <stdio.h>
#define disp \
    printf("Hello World!\n"); \
    printf("Hello World!"); \
    printf("\n");
int main()
{
    disp;
}
```

Code
11.7

Output(s)

Hello World!
Hello World!

Example 11.8 Macro is expanded into code before compilation.

```
#include <stdio.h>
#define disp { \
    printf("Hello World!\n"); \
    printf("Hello World!"); \
    printf("\n"); \
}
int main()
{
    if (1)
        disp;
    else
        printf("Macro is not called \n");
}
```

Code
11.8

When this program is run, we get error`else`' without a previous 'if'

Note that preprocessor expands the macros into the code, and then Code 11.8 happens to be as in Code 11.9, which is an erroneous code.

```
#include <stdio.h>
int main()
{
    if (1)
        printf("Hello World!\n");
        printf("Hello World!");
        printf("\n");
    else
        printf("Macro is not called \n");
}
```

Code
11.9

To overcome this bottleneck, we can use curly parenthesis in if-else statement as in Code 11.10.

```
#include <stdio.h>
#define disp { \
    printf("Hello World!\n"); \
    printf("Hello World!"); \
    printf("\n"); \
}
int main()
{
    if (1)
    { // <---
        disp;

    } // <---
    else
        printf("Macro is not called \n");
}
```

**Code
11.10**

When the macro is expanded, we get the error-free Code 11.11.

```
#include <stdio.h>
int main()
{
    if (1)
    {
        printf("Hello World!\n");
        printf("Hello World!");
        printf("\n");
    }
    else
        printf("Macro is not called \n");
}
```

**Code
11.11**

Expansion of the macro is illustrated in Fig. 11.2.

Identifiers are simple macros that do not contain programming statements.

11.5 Directives Used for File Inclusion

The preprocessor directive **#include** is used to include the header files. To include the header files in compiler's library, we use the directive

```
#include <fileName.h>
```

and to include the user-developed header files we use the directive

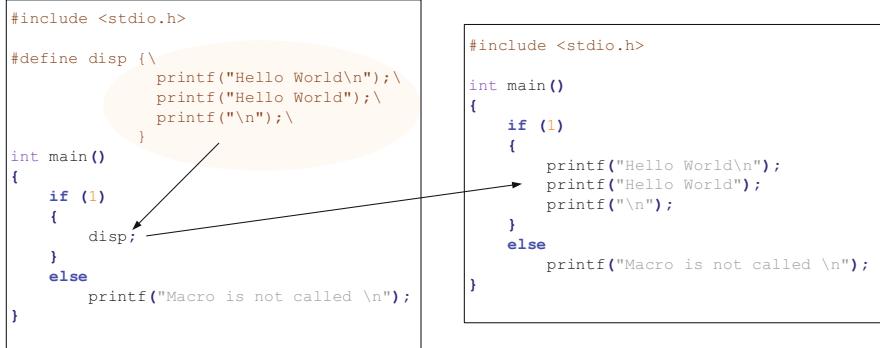


Fig. 11.2 Macro is expanded into the code before compilation

```
#include "fileName.h"
```

where we assume that the header file stays in the same folder as the source file; otherwise, we should write the path of the header file, that is, we use the directive as

```
#include "C:/.../ .../fileName.h"
```

Example 11.9 We have a header file with name myFile.h.

```
// "myFile.h"
// content of myFile.h

#define PI 3.1415
#define circleArea(r) (PI*r*r)
```

and we include this header file in our course code.

<pre>#include <stdio.h> #include "myFile.h" int main() { float r, a; printf("Please enter radius of the circle :"); scanf("%f", &r); a = circleArea(r); printf("Circle area is : %.2f", a); }</pre>	Code 11.12
--	-------------------

Output(s)

Please enter radius of the circle :4
Circle area is 50.26

11.6 Predefined Macros

Compilers have some predefined macros available in their libraries. These macros can be used directly in a C program. Let us see some of these macros.

__LINE__

This is a predefined macro that expands to the current line number in the C program as an integer. `__LINE__` is used for log statements, for clarifying the error location of in a code, and for debugging code.

Example 11.10 In this example, the use of the predefined macro `__LINE__` is illustrated.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Line number is: %d\n", __LINE__);
6 }
```

**Code
11.13**

Output(s) Line number is: 5

```
1 #include <stdio.h>
2
3 int main()
4 {
5     //
6     //
7     printf("Line number is: %d\n", __LINE__);
8 }
```

**Code
11.14**

Output(s) Line number is: 7**__FILE__**

It expands to the file name of the current program running.

Example 11.11 In this example, the use of the predefined macro `__FILE__` is illustrated.

```
#include <stdio.h>
int main()
{
    printf("File name is %s\n", __FILE__);
}
```

**Code
11.15**

Output(s) File name is main.c

__STDC_VERSION__

This macro expands to the C Standard's version number, which is in the form yyyyymm where yyyy stands for year and mm stands for month.

Example 11.12 In this example, the use of the predefined macro __STDC_VERSION__ is illustrated.

```
#include <stdio.h>
int main()
{
    printf("Version Number is %ld\n", __STDC_VERSION__);
}
```

**Code
11.16**

Output(s) Version Number is 199901

Here 1999 is the year and 01 is the month.

__DATE__

It is expanded to the compilation date of the program, and the date is in the format “month dd yyyy.”

Example 11.13 In this example, the use of the predefined macro __DATE__ is illustrated.

```
#include <stdio.h>
int main()
{
    printf("Compilation date is %s\n", __DATE__);
}
```

**Code
11.17**

Output(s) Compilation date is Jul 17 2023

__TIME__

It is expanded to the compilation time of the program, and the time is in the format hour:minute:second.

Example 11.14 In this example, the use of the predefined macro `_TIME_` is illustrated.

```
#include <stdio.h>
int main()
{
    printf("Compilation time is %s\n", __TIME__);
}
```

**Code
11.18**

Output(s) Compilation time is 22:16:09

Identifiers are simple macros which do not contain programming statements.

`__func__`

If this predefined identifier is inside a function, it is replaced by the name of the function; otherwise, if it is inside the main function, it is replaced by the word “main.”

Example 11.15 In this example, the use of the predefined macro `__func__` is illustrated.

```
#include <stdio.h>
void myFunc()
{
    printf("%s \n", __func__);
}

int main()
{
    printf("%s      ", __func__);
    myFunc();
}
```

**Code
11.19**

Output(s) main myFunc

`__cplusplus__`

This directive can be used to identify whether the current compiler is a cpp compiler or not.

Example 11.16 In this example, the use of the predefined macro `__cplusplus__` is illustrated.

```
#include <stdio.h>
int main()
{
    #ifdef __cplusplus
        printf("CPP is supported.");
    #else
        printf("CPP is NOT supported.");
    #endif
}
```

**Code
11.20**

Output(s) CPP is NOT supported

11.7 Conditional Compilation

Conditional compilation directives are used to compile a specific part of the program and skip the rest. Conditional compilation is achieved using the directives

```
#if conditionalExpression
#define identifier
#ifndef identifier
#elif conditionalExpression
#define identifier      (defined in C23)
#ifndef identifier      (defined in C23)
#else
#endif
```

The syntax of `#ifdef...#endif` pair for conditional compilation is given in Code 11.21.

```
#ifdef identifier
    // statements
    // to be compiled
#endif
```

**Code
11.21**

If identifier is defined, then the code inside is compiled.

In a similar manner, we can use `#ifndef...#endif` pair for conditional compilation; its syntax is shown in Code 11.22.

```
#ifndef identifier
    // statements
    // to be compiled
#endif
```

Code
11.22

In this case, if identifier is **NOT** defined, then the code inside is compiled.

Example 11.17 In this example, we define the macro P1 directly.

```
#include <stdio.h>
#define P1

int main(void)
{
    #ifdef P1
        printf("P1 is defined \n");
        printf("P1 has no value");
    #else
        printf("P1 is NOT defined \n");
    #endif
}
```

Code
11.23

Output(s)

P1 is defined

P1 has no value

Example 11.18 In this example, we define the macro P1 inside a conditional expression.

```
#include <stdio.h>
#ifndef P1
    #define P1
#endif

int main()
{
    #ifdef P1
        printf("P1 is defined \n");
        printf("P1 has no value");
    #else
        printf("P1 is NOT defined \n");
    #endif
}
```

Code
11.24

Output(s)

P1 is defined

P1 has no value

Example 11.19 Code 11.23 can also be written as Code 11.25.

```
#include <stdio.h>
#define P1

int main(void)
{
    #if defined P1
        printf("P1 is defined \n");
        printf("P1 has no value");
    #else
        printf("P1 is NOT defined \n");
    #endif
}
```

**Code
11.25**

Output(s)

P1 is defined

P1 has no value

Example 11.20 We define the macro P1 with a value.

```
#include <stdio.h>
#define P1 45.8

int main(void)
{
    #ifdef P1
        printf("P1 is defined \n");
        printf("P1 = %.1f", P1);
    #else
        printf("P1 is NOT defined \n");
    #endif
}
```

**Code
11.26**

Output(s)

P1 is defined

P1 = 45.8

Example 11.21 A macro previously defined can be undefined.

```
#include <stdio.h>
#define P1 45.8
#undef P1

int main(void)
{
    #ifdef P1
        printf("P1 is defined \n");
        printf("P1 = %.1f", P1);
    #else
        printf("P1 is NOT defined \n");
    #endif
}
```

Code
11.27

Output(s) P1 is NOT defined

Example 11.22 In this example, the use of macros in a conditional ladder structure is illustrated.

```
#include <stdio.h>
#define P1
#define P2

int main(void)
{
    #ifdef P1
        #define VAL_A 16
        printf("P1 is defined. \n");
        printf("VAL_A is now defined.");
    #elif defined P2
        #define VAL_B 34
        printf("P2 is defined \n");
        printf("VAL_B is now defined.");
    #else
        #define VAL_C 34
        printf("P1, P2 are NOT defined. \n");
        printf("VAL_C is defined.");
    #endif
}
```

Code
11.28

Output(s)

P1 is defined.

VAL_A is now defined.

Example 11.23 We use #undef macro in this example.

```
#include <stdio.h>
#define P1
#define P2
#undef P1

int main(void)
{
    #ifdef P1
        #define VAL_A 16
        printf("P1 is defined. \n");
        printf("VAL_A is now defined.");
    #elif defined P2
        #define VAL_B 34
        printf("P2 is defined \n");
        printf("VAL_B is now defined.");
    #else
        #define VAL_C 34
        printf("P1, P2 are NOT defined. \n");
        printf("VAL_C is defined.");
    #endif
}
```

Code
11.29

Output(s)

P2 is defined

VAL_B is now defined.

11.8 Concatenation Operator

The operator ## is used to concatenate macro arguments. It is also called either token pasting operator or merging operator.

Syntax

```
#define macroName(p1, p2) p1##p2
```

Example 11.24 In this example, we concatenated two integers 12 and 96 using a macro.

```
#include <stdio.h>
#define myConcat(a, b) a##b

int main()
{
    int a = myConcat(12, 96);

    printf("a = %d", a);
}
```

**Code
11.30**

Output(s)

a = 1296

Problems

1. In C programming, if a macro, which is not previously defined, appears inside an expression, its value is accepted as zero. However, it is still undefined. Considering this fact, find the output of Code 11.31.

```
#include <stdio.h>

#if M1 == 0
    #define M2 6
#else
    #define M2 8
#endif

int main()
{
    printf("M1 = %d", M2);
    //printf("M1 = %d", M1); // gives error, it is still undefined
}
```

**Code
11.31**

2. While Code 11.32 is ok, Code 11.33 gives an error. Explain the reasoning behind this error.

```
#include <stdio.h>

#define X \
    printf("Hello World");

int main()
{
    X;
}
```

**Code
11.32**

```
#include <stdio.h>

#define X 5

#if X \
    printf("Hello World");
#endif

int main()
{
    X;
}
```

**Code
11.33**

3. What is the output of Code 11.34?

```
#include <stdio.h>
#define cube(a) (a)*(a)*(a)

int main()
{
    float b = 64.0/(cube(8));
    printf("%.1f", b);
}
```

**Code
11.34**

4. There is a big difference between

```
#define cube(a) (a)*(a)*(a)
```

and

```
#define cube(a) a*a*a
```

Give an example that shows this difference.

5. What is the output of Code 11.35?

```
# include <stdio.h>
# define myPrintf "%s Hello World"

int main()
{
    printf(myPrintf,myPrintf);
}
```

**Code
11.35**

6. Convert the function in Code 11.36 to a macro.

```
void disp()
{
    printf("Hello World!\n");
}
```

**Code
11.36**

Chapter 12

Type Qualifiers, Enumerations, and Storage Classes in C



12.1 Type Qualifiers in C

The type qualifiers used in C programming are

```
const      restrict      volatile
```

12.1.1 *Const*

A type qualifier is used to add additional attributes to a variable. For instance, in the definition

```
double num;
```

num is the variable name, the data type used by the variable is a double data. If we use **const** qualifier for the variable num, we get

```
const double num;
```

and in this case the data type used by the variable is still a double data but its value cannot be changed, that is, the variable has an additional property.

The type qualifier **const** can be used with pointers. In this case, the one to the right of **const** has constant value and cannot be changed.

Example 12.1

```
float num = 3.4;
const float* fp = &num;
```

in the pointer expression, to the right of **const**, there is data type **float**; this shows that data value is constant.

Example 12.2

```
float num = 3.4;
float* const fp = &num;
```

in the pointer expression, to the right of **const**, there is pointer variable **fp**; this shows that pointer value is constant. It means the pointer cannot point to another variable, that is, the pointer address cannot be changed.

Example 12.3 Constant data pointed by a pointer cannot be changed.

<pre>#include <stdio.h> int main() { int a = 13; const int* ptr = &a; *ptr = 54; // error, value cannot be changed }</pre>	Code 12.1
--	----------------------

Output(s) error: assignment of read-only location ‘*ptr’

Example 12.4 The address of a constant pointer cannot be changed.

<pre>#include <stdio.h> int main(void) { int a = 23, b = 49; int *const ptr = &a; ptr = &b; //error, pointer address cannot be changed }</pre>	Code 12.2
--	----------------------

Output(s) error: assignment of read-only variable ‘ptr’

12.1.2 Restrict

The **const** and **volatile** qualifiers are introduced in C89; later, the **restrict** qualifier is defined in the C99 standard. This keyword is useful for engineers working on embedded systems. This qualifier is used to optimize the performance of the compiler for memory access. The value pointed by restrict pointer is held in a register, that is, cached, and during intermediate operations cached value is used.

A pointer having restrict qualifier indicates that a particular region of memory should be accessed by one pointer and never another. If a pointer is defined to be restrict and it points to a variable, and then the variable is accessed by another pointer, then undefined behavior may happen.

Example 12.5

```
void myFnc(int* ptra, int* ptrb, int* ptrc)
{
    *ptra += *ptrc;
    // *ptrc is read from memory, and *ptra += *ptrc is performed

    *ptrb += *ptrc;
    // *ptrc is read from memory, and *ptrb += *ptrc is performed
}
```

Code
12.3

In Code 12.3, four instructions are performed. Let us use the restrict keyword for the pointers as in Code 12.4.

```
void myFnc(int* ptra, int* ptrb, int* restrict ptrc)
{
    *ptra += *ptrc;
    // *ptrc is read from memory, and *ptra += *ptrc is performed

    *ptrb += *ptrc;
    // *ptrb += *ptrc is performed
}
```

Code
12.4

Due to the use of restrict keyword in Code 12.4, three instructions are performed. One less instruction is needed compared to Code 12.3.

12.1.3 Volatile

This qualifier is useful especially for embedded software engineers or hardware programming engineers. The volatile keyword is used to take attention of the compiler to the variables whose values can change instantly, and the compiler avoids optimization techniques that can cause the miss detection of instant changes of variables.

For instance, a global variable representing a data port can receive a value whenever an interrupt occurs, and in this case the global variable that is used for the data port should be declared as volatile in order to catch the latest data available at the port. Electronic devices utilizing touch sensors should process every input immediately.

If the variable is not qualified as volatile, the compiler applies optimization methods to speed up the program execution, and when the port is read, it is placed into a catch, that is, register, and the cached value is used for some time as port value, the port is not read every time, and when the catch time expires the port value is updated with the new input if there is. Besides, in multithread applications shared global variables should be qualified as volatile due to a similar reasoning as in the case of hardware interrupts.

In summary, if a variable is qualified as volatile, then compiler does not do optimization for that variable. For example,

```
volatile int a;
```

12.2 Storage Classes in C

The keywords used to classify storage method in C are

```
auto      extern      static      register
```

12.2.1 Auto

Auto is used for default storage, and most time, it is not explicitly used.

Example 12.6

```
auto int a;
```

is the same as

```
int a;
```

12.2.2 Extern

To understand the role of keyword **extern**, let us explain the meanings of declaration and definition.

Declaration

Declaration of a variable or function is nothing but making compiler aware of the existence of a variable or a function. That means giving pre-information to the

compiler about a variable or a function. Function declaration involves function name, input arguments, and returned data type. Memory is not allocated for variable or function. It is similar to calling a restaurant and making them aware of you that you are a customer candidate, but a reservation is not made for you. To declare a variable, we use the keyword **extern**.

Definition

When a variable or function is defined, memory is automatically allocated for variable or function. It is similar to calling a restaurant and reserving a table for your dinner.

Example 12.7 A declaration is made in

```
extern float a;
```

Example 12.8 A function declaration is made in

```
float myFunc(float, double);  
or in  
float myFunc(float a, double d);
```

Example 12.9 A definition is made in

```
float a;
```

Example 12.10 A function definition is made in

```
float myFunc(float a, float b)  
{  
    return a*b;  
}
```

Example 12.11 A declaration and a definition are made in

```
extern float a = 20.45;
```

Here, we inform the compiler that the variable a can be used in another file.

Example 12.12 Declaration is made for variable “a” in Code 12.5, and “a” is defined in Code 12.6.

```
// file name is : main.c
#include <stdio.h>

extern float a;

int main()
{
    printf("a = %.2f",a);
}
```

**Code
12.5**

```
// file name is variables.c
float a = 23.56;
```

**Code
12.6**

When these two files are compiled in a project, we get the output a = 23.56.

Example 12.13 If we write a header file, variables.h, and include it in our code, then it is not necessary to declare the variable using the keyword extern.

```
// file name is : main.c
#include <stdio.h>
#include "variables.h"

// no need for extern declaration

// extern float a;

int main()
{
    printf("a = %.2f",a);
}
```

**Code
12.7**

```
// file name is variables.h
float a = 23.56;
```

**Code
12.8**

Example 12.14 In this example, we have three separate files. The file main.c contains two extern declarations.

```
// main.c
#include <stdio.h>

extern float a;
extern void myFunc();

int main()
{
    printf("Inside main, a = %.2f \n",a);

    myFunc();
}
```

**Code
12.9**

The file variables.c contains definition for the variable a.

```
// variables.c
float a = 23.56;
```

**Code
12.10**

The file functions.c contains the declaration n for the variable a.

```
// functions.c
#include <stdio.h>

extern float a;

void myFunc()
{
    printf("Inside function, a = %.2f \n", a);
```

**Code
12.11**

When these three files are compiled in a project, we get the output

Inside main, a = 23.56

Inside function, a = 23.56

Let us remove the extern declaration from functions.c as in Code 12.12.

```
// functions.c
#include <stdio.h>

void myFunc()
{
    printf("Inside function, a = %.2f \n", a);
```

**Code
12.12**

When we now compile three files, we get the output
functions.c: In function ‘myFunc’:

functions.c:9:48: error: ‘a’ undeclared (first use in this function)

That is, we get an error.

12.2.3 Static

Static variables preserve their last values until the main program terminates. A static value needs to be initialized at its declaration. A static variable is defined as

```
static dataType variable_name = value;
```

Example 12.15 The last value of a static variable is kept, and this property is very useful in function calls.

```
#include <stdio.h>
Code
12.13
void myFunc()
{
    static int a = 10;

    a++;

    printf("Inside function, a = %d \n", a);
}

int main()
{
    myFunc();
    myFunc();
    myFunc();
}
```

Output(s)

Inside function, a = 11

Inside function, a = 12

Inside function, a = 13

12.2.4 Register

The most frequently used variables can be attributed with register keyword. When a variable is defined as register variable, it is stored in special registers and these registers are accessed in a fast manner compared to stack memory. We can define a register variable as

```
register float a = 4.7;
```

Fast access registers do not have explicit addresses as stack memory registers, and for this reason we cannot use address operator, &, for the register variables.

Example 12.16 Address operator & cannot be used with register variables.

```
#include <stdio.h>
int main()
{
    register int a = 45;
    int* ptr = &a; // error
    printf("%p", ptr);
}
```

Code
12.14

Output(s) error: address of register variable ‘a’ requested

Pointers can be used with register keyword. Do not forget that pointers are variables and any variable can be used with register keyword.

Example 12.17 Pointers can be used with **register** variables.

```
#include <stdio.h>
int main()
{
    int a = 10;
    register int* ptr = &a;
    printf("address is : 0x%p", ptr);
}
```

Code
12.15

Output(s) address is: 0x0x7ffc7620f544

Register keyword cannot be used for static and global variables.

Example 12.18 Static register variables cannot be defined.

```
#include <stdio.h>
int main()
{
    register static int a = 10; // error
    printf(" a = %d", a);
}
```

Code
12.16

Output(s) error: multiple storage classes in declaration specifiers.

Example 12.19 Global register variables cannot be defined.

<pre>#include <stdio.h> register static int a = 10; // error int main() { printf(" a = %d", a); }</pre>	Code 12.17
---	-----------------------

Output(s) error: multiple storage classes in declaration specifiers.

Problems

1. Is there any difference between the definitions

```
int volatile a;
```

and

```
volatile int a;
```

?

2. Explain the differences in the pointer definitions in the expressions

```
float volatile* fp;
volatile float* fp
float* volatile fp;
float (*volatile fp);
volatile float* volatile fp;
float volatile* volatile fp;
```

3. Why do we use **restrict** qualifier in C programming? Should every code use it?
Which engineering field needs this qualifier the most?
4. What are the outputs of Code 12.18?

```
#include <stdio.h>
Code
12.18

void myFunc()
{
    static float a = 2.2;

    a = a + 3.2;

    printf("Inside function, a = %d \n", a);
}

int main()
{
    myFunc();
    myFunc();
    myFunc();
}
```

5. Are **extern** and **register** keywords used for type qualifying or are they storage classes?

Chapter 13

Integer with Exactly N Bits



13.1 General Form of Fixed Width Integers

`intN_t`

The data type

`intN_t`

is used for signed integer with exactly N bits, and every compiler according to C99 standard must support the values for 8, 16, 32, and 64. However, a compiler is free to support other values for N such as 24, 48, etc.

Thus, a standard compiler has the data types

`int8_t int16_t int32_t int64_t`

printf macros used for `intN_t` are

`PRIdN PRIiN PRIoN PRIxN PRIxN`

13.2 Macros for printf and scanf

The macros for printf begin with

`PRI`

and the macros for scanf begin with

SCN.

To print the signed integers with printf() function, we use the macros

PRI_dN PRI_iN PRI_oN PRI_xN PRI_XN

where N can be 8, 16, 32, and 64. For the scanf() function, we use the macros

SCN_dN SCN_iN SCN_oN SCN_xN SCN_XN

where N can be 8, 16, 32, and 64.

To be able to use these macros, we need to include the header file <inttypes.h> in our program.

Example 13.1 The printf() function is used in a different manner for fixed length integer types.

```
#include <inttypes.h>
#include <stdio.h>

int main(void)
{
    int8_t num = 23;

    printf("num = % " PRId8 "\n", num);
    printf("num = % " PRIi8 "\n", num);
}
```

**Code
13.1**

Output(s)

num = 23

num = 23

Example 13.2 Different sizes are available for different fixed length integer types.

```
#include <stdio.h>
#include <stdint.h>

int main()
{
    int8_t a = 0xFF; // 8-bit signed integer
    int16_t b = 0xFFFF; // 16-bit signed integer
    int32_t c = 0xFFFFFFFF; // 32-bit signed integer
    int64_t d = 0xFFFFFFFFFFFFFFFF; // 64-bit signed integer

    printf("sizeof int8_t is %lu \n", sizeof(int8_t));
    printf("sizeof int16_t is %lu \n", sizeof(int16_t));
    printf("sizeof int32_t is %lu \n", sizeof(int32_t));
    printf("sizeof int64_t is %lu \n", sizeof(int64_t));
}
```

Code
13.2

Output(s)

sizeof int8_t is 1
 sizeof int16_t is 2
 sizeof int32_t is 4
 sizeof int64_t is 8

Example 13.3 For negative integers, 2's complement representation is used. -1 is represented by all 1's.

```
#include <stdio.h>
#include <stdint.h>

int main()
{
    int8_t a = 0xFF; // 8-bit signed integer
    int16_t b = 0xFFFF; // 16-bit signed integer
    int32_t c = 0xFFFFFFFF; // 32-bit signed integer
    int64_t d = 0xFFFFFFFFFFFFFFFF; // 64-bit signed integer

    printf("a = % " PRId8 "\n", a);
    printf("b = % " PRId16 "\n", b);
    printf("c = % " PRId32 "\n", c);
    printf("d = % " PRId64 "\n", d);
}
```

Code
13.3

Output(s)

a = -1
 b = -1
 c = -1
 d = -1

The minimum values of the signed integers

```
int8_t int16_t int32_t int64_t
```

are defined by the macro constants

```
INT8_MIN INT16_MIN INT32_MIN INT64_MIN
```

and similarly their maximum values are defined by the macro constants

```
INT8_MAX INT16_MAX INT32_MAX INT64_MAX
```

Example 13.4

```
#include <stdio.h>
#include <inttypes.h>

int main()
{
    printf("INT8_MIN = % " PRId8 "\n", INT8_MIN);
    printf("INT16_MIN = % " PRId16 "\n", INT16_MIN);
    printf("INT32_MIN = % " PRId32 "\n", INT32_MIN);
    printf("INT64_MIN = % " PRId64 "\n", INT64_MIN);
}
```

**Code
13.4**

Output(s)

```
INT8_MIN = -128
INT16_MIN = -32768
INT32_MIN = -2147483648
INT64_MIN = -9223372036854775808
```

Example 13.5

```
#include <stdio.h>
#include <inttypes.h>

int main()
{
    printf("INT8_MAX = % " PRId8 "\n", INT8_MAX);
    printf("INT16_MAX = % " PRId16 "\n", INT16_MAX);
    printf("INT32_MAX = % " PRId32 "\n", INT32_MAX);
    printf("INT64_MAX = % " PRId64 "\n", INT64_MAX);
}
```

**Code
13.5**

Output(s)

```
INT8_MAX = 127
INT16_MAX = 32767
INT32_MAX = 2147483647
INT64_MAX = 9223372036854775807
```

13.3 uintN_t

The data type

```
uintN_t
```

is used for unsigned signed integers with exactly N bits, and every compiler according to C99 standard must support the values for 8, 16, 32, and 64. However, a compiler is free to support other values for N such as 24, 48, etc.

Thus, a standard compiler has the data types

```
uint8_t uint16_t uint32_t uint64_t
```

Macros for printf are

```
PRIuN PRIoN PRIxN PRIxN
```

and macros for scanf are

```
SCNuN SCNoN SCNxN SCNxN
```

where N values can be 8, 16, 32, and 64.

Example 13.6

<pre>#include <inttypes.h> #include <stdio.h> int main() { uint8_t a = 0xFF; // 8-bit unsigned integer uint16_t b = 0xFFFF; // 16-bit unsigned integer uint32_t c = 0xFFFFFFFF; // 32-bit unsigned integer uint64_t d = 0xFFFFFFFFFFFFFFFF; // 64-bit unsigned printf("a = %"PRIu8"\n", a); printf("b = %"PRIu16"\n", b); printf("c = %"PRIu32"\n", c); printf("d = %"PRIu64"\n", d); }</pre>	Code 13.6
---	----------------------------

Output(s)

a = 255
 b = 65535
 c = 4294967295
 d = 18446744073709551615

Example 13.7

```
#include <inttypes.h>
#include <stdio.h>

int main()
{
    uint8_t a = 0xFF; // 8-bit unsigned integer
    uint16_t b = 0xFFFF; // 16-bit unsigned integer
    uint32_t c = 0xFFFFFFFF; // 32-bit unsigned integer
    uint64_t d = 0xFFFFFFFFFFFFFFFF; // 64-bit unsigned

    printf("a = %"PRIx8"\n", a);
    printf("b = %"PRIx16"\n", b);
    printf("c = %"PRIx32"\n", c);
    printf("d = %"PRIx64"\n", d);
}
```

**Code
13.7**

Output(s)

a = ff
 b = ffff
 c = FFFFFFFF
 d = FFFFFFFFFFFFFF

The minimum values of the signed integers

uint8_t uint16_t uint32_t uint64_t

are 0 and their maximum values are defined by the macro constants

UINT8_MAX UINT16_MAX UINT32_MAX UINT64_MAX

Example 13.8

```
#include <inttypes.h>
#include <stdio.h>

int main()
{
    printf("UINT8_MAX = %"PRIx8"\n", UINT8_MAX);
    printf("UINT16_MAX = %"PRIx16"\n", UINT16_MAX);
    printf("UINT32_MAX = %"PRIx32"\n", UINT32_MAX);
    printf("UINT64_MAX = %"PRIx64"\n", UINT64_MAX);
}
```

**Code
13.8**

Output(s)

UINT8_MAX = ff
 UINT16_MAX = FFFF
 UINT32_MAX = ffffffff
 UINT64_MAX = FFFFFFFFFFFFFFFF

13.4 int_leastN_t

The data type

int_leastN_t

is used for signed integers with at least N bits. A standard compiler has the data types

int_least8_t int_least16_t int_least32_t int_least64_t

Macros for printf for **int_leastN_t** are

PRI_dLEASTN PRI_iLEASTN PRI_xLEASTN PRI_XLEASTN PRI_oLEASTN

where N can be 8, 16, 32, and 64.

Macros for scanf for **int_leastN_t** are

SCNdLEASTN SCN_iLEASTN SCN_xLEASTN SCNxLEASTN SCN_oLEASTN

where N can be 8, 16, 32, and 64.

Minimum values for **int_leastN_t** are

INT_LEAST8_MIN INT_LEAST16_MIN INT_LEAST32_MIN INT_LEAST64_MIN

Maximum values for `int_leastN_t` are

```
INT_LEAST8_MAX INT_LEAST16_MAX INT_LEAST32_MAX INT_LEAST64_MAX
```

13.5 `uint_leastN_t`

The data type

```
uint_leastN_t
```

is used for unsigned integers with at least N bits. A standard compiler has the data types

```
uint_least8_t uint_least16_t uint_least32_t uint_least64_t
```

Macros for printf and scanf for `uint_leastN_t` are

```
PRIuLEASTN PRIxLEASTN PRIxLEASTN PRIoLEASTN
```

where N can be 8, 16, 32, and 64.

Macros for scanf for `uint_leastN_t` are

```
SCNuLEASTN SCNxLEASTN SCNxLEASTN SCNoLEASTN
```

where N can be 8, 16, 32, and 64.

Minimum values for `uint_leastN_t` are 0.

Maximum values for `int_leastN_t` are

```
UINT_LEAST8_MAX UINT_LEAST16_MAX UINT_LEAST32_MAX UINT_LEAST64_MAX
```

13.6 `int_fastN_t`

The data type

```
int_fastN_t
```

is used for signed integer with at least N bits and fastest processing. A standard compiler has the data types

```
int_fast8_t int_fast16_t int_fast32_t int_fast64_t
```

Macros for printf for `int_fastN_t` are

```
PRIdFASTN PRIiFASTN PRIxFASTN PRIoFASTN
```

where N can be 8, 16, 32, and 64.

Macros for scanf for `int_fastN_t` are

```
SCndFASTN SCniFASTN SCnxFASTN SCnoFASTN
```

where N can be 8, 16, 32, and 64.

Minimum values for `int_fastN_t` are

```
INT_FAST8_MIN INT_FAST16_MIN INT_FAST32_MIN INT_FAST64_MIN
```

Maximum values for `int_fastN_t` are

```
INT_FAST8_MAX INT_FAST16_MAX INT_FAST32_MAX INT_FAST64_MAX
```

13.7 uint_fastN_t

The data type

```
uint_fastN_t
```

is used for unsigned integer with at least N bits and fastest processing. A standard compiler has the data types

```
uint_fast8_t uint_fast16_t uint_fast32_t uint_fast64_t
```

Macros for printf for `uint_fastN_t` are

```
PRIuFASTN PRIxFASTN PRIoFASTN
```

where N can be 8, 16, 32, and 64.

Macros for scanf for `uint_fastN_t` are

```
SCnuFASTN SCnxFASTN SCnoFASTN
```

where N can be 8, 16, 32, and 64.

Maximum values for `uint_fastN_t` are

```
UINT_FAST8_MAX UINT_FAST16_MAX UINT_FAST32_MAX UINT_FAST64_MAX
```

13.8 Macros for printf

All the macros for printf() function are shown in Table 13.1.

In this table, we can have N = 8, 16, 32, and 64.

13.9 Macros for scanf

All the macros for scanf() function are shown in Table 13.2.

In this table, we can have N = 8, 16, 32, and 64.

C23 introduces macros for the bit width of the data types, and some of these macros are

```
INT8_WIDTH INT_FAST8_WIDTH INT_LEAST8_WIDTH
INT16_WIDTH INT_FAST16_WIDTH INT_LEAST16_WIDTH
INT32_WIDTH INT_FAST32_WIDTH INT_LEAST32_WIDTH
INT64_WIDTH INT_FAST64_WIDTH INT_LEAST64_WIDTH
```

Problems

1. How do the macros begin with for printf and scanf() function for fixed width integers?
2. Which header file needs to be included in the source file to be able to use the macros to print fixed width integers?
3. Can a compiler define 19-bit fixed width integer data type?
4. Fill the inside part of printf() function in Code 13.9 to print the fixed width integer value.

<pre>#include <inttypes.h> #include <stdio.h> int main(void) { int8_t num = 45; printf("...."); }</pre>	Code 13.9
--	----------------------

Table 13.1 The macros used for printf() function

		Macros for data types					
		[u] intN_t	[u] int leastN_t	[u] int fastN_t	[u] intmax_t	[u] dintptr_t	
d		PRIdN	PRIFASTN	PRIFASTN	PRIIMAX	PRIDPTR	
i		PRIiN	PRIiFASTN	PRIiFASTN	PRIIMAX	PRIiPTR	
u		PRIuN	PRIuFASTN	PRIuFASTN	PRIuMAX	PRIuPTR	
o		PRIoN	PRIoFASTN	PRIoFASTN	PRIoMAX	PRIoPTR	
x		PRIxN	PRIxFASTN	PRIxFASTN	PRIxMAX	PRIxPTR	
X		PRIxN	PRIxFASTN	PRIxFASTN	PRIxMAX	PRIxPTR	

Table 13.2 The macros used for scanf() function

Equivalent for <code>int</code> or <code>unsigned int</code>		Description		Macros for data types			
		[u] <code>intx_t</code>	[u] <code>int_leastx_t</code>	[u] <code>int_fastx_t</code>	[u] <code>intmax_t</code>	[u] <code>intptr_t</code>	
d	Input of a signed decimal integer value	<code>SCNd</code> x	<code>SCNdLEAST</code> x	<code>SCNdFAST</code> x	<code>SCNdMAX</code>	<code>SCNdPTR</code>	
i	Input of a signed integer value (base is determined by the first characters parsed)	<code>SCNi</code> x	<code>SCNiLEAST</code> x	<code>SCNiFAST</code> x	<code>SCNiMAX</code>	<code>SCNiPTR</code>	
u	Input of an unsigned decimal integer value	<code>SCNu</code> x	<code>SCNuLEAST</code> x	<code>SCNuFAST</code> x	<code>SCNuMAX</code>	<code>SCNuPTR</code>	
o	Input of an unsigned octal integer value	<code>SCNo</code> x	<code>SCNoLEAST</code> x	<code>SCNoFAST</code> x	<code>SCNoMAX</code>	<code>SCNoPTR</code>	
x	Input of an unsigned hexadecimal integer value	<code>SCNx</code> x	<code>SCNxLEAST</code> x	<code>SCNxFAST</code> x	<code>SCNxMAX</code>	<code>SCNxPTR</code>	

5. Find the output of Code 13.10.

```
#include <stdio.h>
#include <inttypes.h>

int main()
{
    printf("INT8_MAX = %" PRIx8 "\n", INT8_MAX);
    printf("UINT8_MAX = %" PRIx8 "\n", UINT8_MAX);
}
```

Code
13.10

6. What is the difference between `int_least64_t` and `int64_t`. Assume that there is a 9-bit machine, can we use both data types in this machine?

Chapter 14

Signals in C



14.1 Introduction

Signals are generated when an interrupt is received. Signals can also be called as interrupt signals. Interrupts are abnormal cases, and upon their occurrences, some signals are generated, and when these signals are detected, some actions are performed.

The interrupts are can be divided into two main categories:

- Hardware interrupts
- Software interrupts

When a signal is received, we understand that either some default actions are performed or a user-defined function is executed. For instance, when Ctrl + C is pressed, a software interrupt is generated. The default action is to terminate the current process. However, a function, which is executed when Ctrl + C interrupt is received, can be written. Signals can also be generated from OS kernel directly when a hardware fault such as a bus error occurs or an illegal instruction is performed.

The default actions of signals are

The signal is discarded after it is received

The current process is terminated when the signal is received

A core file is written, and the process is stopped

The current process is stopped when the signal is received

The signals can be artificially generated. For this purpose, C provides the `raise()` function with prototype

```
int raise(int sig)
```

where `sig` is an integer that defines the signal type, and they are defined as macros in the header file `<signal.h>`

Some of these macros are

```
#define SIGHUP 1 /* hangup */
#define SIGINT 2 /* interactive interrupt */
#define SIGQUIT 3 /* quit (ASCII FS) */
#define SIGILL 4 /* illegal instruction (not reset when caught) */
#define SIGABRT 6 /* used by abort, replace SIGIOT in the future */
#define SIGKILL 9 /* kill (cannot be caught or ignored) */
#define SIGBUS 10 /* bus error */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
```

14.2 Signal Handling

When an interrupt occurs, a signal is generated, and the generated signal can be caught by the built-in function called signal handler. The signal handler

- May let the default action to happen
- Can block the signal; however, this may not be possible for every signal
- A function can be called by the signal handler

The built-in signal handler is defined as

```
int (*signal(int sig, void (*func)()))();
```

which can be used as

```
signal(signalMacroName, pointerName)
```

if default action is to be performed, or it can be used as

```
signal(signalMacroName, functionName)
```

if a function with name **functionName** is to be called.

14.3 SIGINT

This signal macro is used to capture interactive attention signal. It can be considered as interrupt signal generated by the user. For instance, this signal is generated when the user presses Ctrl + C from the keyboard.

If default action is to be considered when

```
signal(signalMacroName, pointerName)
```

is executed, the pointer name can be **SIG_DFL**, which is a pointer to a system default function **SIG_DFL()**, and the process is terminated upon the reception of interrupt signal.

If the pointer name is **SIG_IGN**, the signal action is ignored.

Example 14.1 Let us write a program involving signals. For this purpose, let us first write an infinite loop as in Code 14.1.

<pre>#include <signal.h> #include <stdio.h> int main() { while (1) { printf("Infinite Loop \n"); sleep(1); } }</pre>	Code 14.1
---	----------------------

Signal handler for SIGINT is added as in Code 14.2.

<pre>#include <signal.h> #include <stdio.h> int main() { signal(SIGINT, myFunc); while (1) { printf("Infinite Loop \n"); sleep(1); } }</pre>	Code 14.2
--	----------------------

We add the definition of signal handler function myFunc as in Code 14.3.

```
#include <signal.h>
#include <stdio.h>

void myFunc(int sig)
{
    printf("Interrupt signal is received: MACRO: %d\n", sig);

}

int main()
{
    signal(SIGINT, myFunc);

    while (1)
    {
        printf("Infinite Loop \n");
        sleep(1);
    }
}
```

**Code
14.3**

Output(s) When the user pressed Ctrl + C, an interrupt signal is generated and it is handled by the signal handler, that is, myFunc.

```
Infinite Loop
Infinite Loop
^CInterrupt signal is received: MACRO: 2
Infinite Loop
Infinite Loop
Infinite Loop
^CInterrupt signal is received: MACRO: 2
Infinite Loop
Infinite Loop
....
```

Note that Sleep() is defined in <windows.h> for Windows compiler, and sleep() function is defined in <unistd.h> for Linux. However, without these headers, the program works with warning.

Example 14.2 Signal handler can perform default action as well. The default action SIG_DFL terminates the program.

```
#include <signal.h>
#include <stdio.h>

int main()
{
    signal(SIGINT, SIG_DFL);

    while (1)
    {
        printf("Infinite Loop \n");
        sleep(1);
    }
}
```

**Code
14.4**

Output(s) When Ctrl + C is pressed, the signal handler performs the default action, which is the termination of the program.

Infinite Loop

Infinite Loop

Infinite Loop

^C

...Program finished with exit code 0

Example 14.3 One of the default actions for the signal SIGINT is the SIG_IGN, which ignores the interrupt request.

```
#include <signal.h>
#include <stdio.h>

int main()
{
    signal(SIGINT, SIG_IGN);

    while (1)
    {
        printf("Infinite Loop \n");
        sleep(1);
    }
}
```

**Code
14.5**

Output(s) When Ctrl + C is pressed, the signal handler performs the default action SIG_IGN, which disregards, that is, ignores the signal action. The program goes on running.

Infinite Loop

Infinite Loop

Infinite Loop

^C

Infinite Loop

Infinite Loop

....

14.4 SIGQUIT

This signal macro is similar to **SIGINT**; however, the interrupt signal is generated when Ctrl + \ is pressed. Note that Ctrl + \ means we press Ctrl and \ at the same time.

Example 14.4 Interrupt signal can be generated by Ctrl + C or Ctrl + \. Some compilers support both of them, some of them support only Ctrl + C. In this example, we write a code that identifies whether Ctrl + C or Ctrl + \ is pressed.

We first write the program for SIGINT as in Code 14.6 where interrupt is generated when Ctrl + C is pressed. Note that Ctrl + C means we press Ctrl and C at the same time.

```
#include <stdio.h>
#include <signal.h>

void func1();

int main()
{
    signal(SIGINT, func1);

    for(;;)
    {
        printf("Infinite Loop \n");
        sleep(1);
    }
}

void func1()
{
    printf("\nCtrl+C is pressed \n");
}
```

Code
14.6

Next, we add the signal handler for SIGQUIT, which is generated when Ctrl + \ is pressed.

```
#include <stdio.h>
#include <signal.h>

void func1();
void func2();

int main()
{
    signal(SIGINT, func1);
    signal(SIGQUIT, func2);

    for(;;)
    {
        printf("Infinite Loop \n");
        sleep(1);
    }
}

void func1()
{
    printf("\nCtrl+C is pressed  \n");
}

void func2()
{
    printf("\nCtrl+ \\ is pressed \n");
}
```

Code
14.7

Output(s)

Infinite Loop

Infinite Loop

Infinite Loop

Infinite Loop

^C

Ctrl + C is pressed

Infinite Loop

Infinite Loop

Infinite Loop

^\\

Ctrl + \ is pressed

Infinite Loop

Infinite Loop

Infinite Loop

...

14.5 Artificial Signal Generation

Interrupt signals can be generated by hardware, such as by pressing Ctrl + C, or they can be generated by software. The `raise()` function can be used to generate interrupt signals. The prototype of the `raise()` function is

```
int raise(int sig);
```

Example 14.5 In the previous examples, the interrupt signal SIGINT is generated by pressing Ctrl + C. In this example, we generate the signal SIGINT using the `raise()` function.

```
#include <signal.h>
#include <stdio.h>

void myFunc(int sig)
{
    printf("Received SIGINT Signal. Signal MACRO: %d\n", sig);
}

int main(void)
{
    signal(SIGINT, myFunc);

    printf("Generating SIGINT Signal. Signal MACRO: %d\n", SIGINT);

    raise(SIGINT);

    printf("Quits Program. \n");
}
```

Code
14.8

Output(s)

Generating SIGINT Signal. Signal MACRO: 2
 Received SIGINT Signal. Signal MACRO: 2
 Quits Program.

SIG_ERR

It indicates an error in signal handling.

SIG_ACK

It indicates successful signal handling.

Example 14.6 In this example, we improve the previous example using the SIG_ERR and return value of raise() function.

```
#include <signal.h>
#include <stdio.h>

void myFunc(int sig)
{
    printf("Has received the signal SIGTERM");
}

int main(void)
{
    if (signal(SIGTERM, myFunc) == SIG_ERR)
    {
        printf("Error while handling the signal.\n");

        exit(0);
    }

    printf("Generating the signal SIGTERM.\n");

    if (raise(SIGTERM) != 0)
    {
        printf("Error while generating the signal SIGTERM.\n");

        exit(0);
    }
}
```

Code
14.9

Output(s)

Generating the signal SIGTERM.
Has received the signal SIGTERM

14.6 Some of the Most Used Signals

SIGHUP HUP is an abbreviation for “hang up.” This signal is generated when the process terminates abruptly. The SIGHUP signal is generated when a remote connection is lost.

SIGABRT The SIGABRT is generated when an error is detected by the program and abort() function is called.

SIGFPE This signal is generated when overflow, division by zero, etc., occur in arithmetic operations.

SIGSEGV This name is an abbreviation for signal segmentation violation. When a program tries to read or write to a forbidden location, this signal is generated.

SIGALRM This is a timeout signal that is sent by alarm(). SIGALRM is sent when the timer expires after a certain amount of time.

SIGILL It is an abbreviation for signal for illegal instruction. This signal is generated when code is corrupted or an attempt is made to execute data, or the program loads a corrupted dynamic library.

SIGUSR1 and SIGUSR2 The signals SIGUSR1 and SIGUSR2 may be used as you wish. Handlers for these two signals can be written for simple inter-process communication.

SIGTERM This signal is used to quit the program in a clean manner while **SIGKILL** is an abnormal termination signal.

Problems

1. Explain the use of **raise** function in C signals.
2. How many ways are available to call the signal handler?
3. Write a C program that increments the value of a global variable every time Ctrl + C is pressed. The program stays inside an infinite loop, and whenever the global value reaches 10 the program terminates.
4. Write a C program that raises another interrupt signal whenever Ctrl + C is pressed.

Chapter 15

Threads in C



15.1 Introduction

The threads are introduced in C11 standard, and it is included in C17 and C23 standards as well. Threads enable parallel processing operations. Parallel processing decreases the computation time of the scientific algorithms. Figure 15.1 illustrates the concept of threads.

15.2 Thread Creation

To be able to use the thread facility in our programs, we need to include the header file `<threads.h>` where macros, types, enumeration constants, and functions that support multiple threads of execution are defined.

The data type `pthread_t` is used to uniquely identify a thread, and in a standard compiler it is defined as

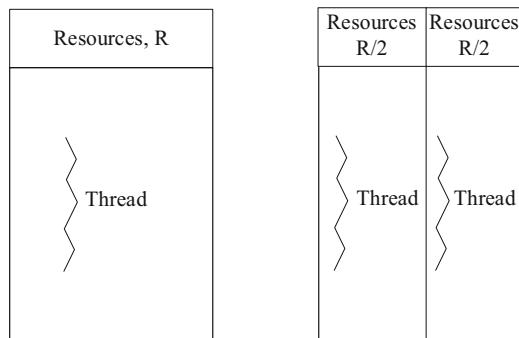
```
typedef unsigned long int pthread_t;
```

Other implementations may differ.

Threads are created using `pthread_create()` function

```
int pthread_create(
    pthread_t * thread,
    const pthread_attr_t * attr,
    void* (*start_routine)(void *), void *arg
);
```

Fig. 15.1 Single versus multiple threads



which is used in a program as

```
pthread_create(thread, attr, start_routine, arg)
```

In the function prototype,

thread: can be NULL, or a pointer to a **pthread_t** object,
attr: is a pointer to a **pthread_attr_t** structure that specifies the attributes of the new thread, and if attr is NULL, the default attributes are used
start_routine : is the thread function
arg: is the argument to pass to the thread function, you can pass what is necessary for the function using this parameter
int (return type) : if thread is created successfully, the return value will be 0, otherwise **pthread_create** will return an error number of type integer

15.3 Parallel Processing Using Threads

Threads are used for parallel processing. Parallel processing decreases the computation latency of the complex algorithms. Threads are not useful if heavy computation is not performed. Since the use of threads brings some extra overheads to the compiler, and if heavy computation is not performed, using threads may be a disadvantage.

Example 15.1 We can calculate the size of a thread object using **sizeof()** operator.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int main ()
{
    pthread_t thrd; //thrd will hold the thread ID
    printf("Size of pthread_t is : %lu \n", sizeof(pthread_t)) ;
}
```

**Code
15.1**

Output(s)

Size of pthread_t is : 8

Example 15.2

In this example, we demonstrate parallel processing operation using a thread. First, let us write an infinite loop in the main function as in Code 15.2.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int main ()
{
    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}
```

**Code
15.2**

Next, we write a function that contains an infinite loop as in Code 15.3. This function will be used as a thread function.

```
void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc \n");
    }
}
```

**Code
15.3**

We form the structure of our program as in Code 15.4.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* myFunc(void* ptr);

int main ()
{
    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}

void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc \n");
    }
}
```

**Code
15.4**

We define a thread ID as in Code 15.5. We also included <pthread.h> header file.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* myFunc(void* ptr);

int main ()
{
    pthread_t thrd; //thrd will hold the thread ID

    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}

void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc \n");
    }
}
```

**Code
15.5**

Thread creation is done in Code 15.6.

<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void* myFunc(void* ptr); int main () { pthread_t thrd; //thrd will hold the thread ID int rc; rc = pthread_create(&thrd, NULL, myFunc, NULL); if(rc) { printf("Error : cannot create thread."); exit(-1); } while(1) { printf("Inside main \n"); sleep(1); } } void* myFunc(void* ptr) { while(1) { sleep(1); printf("Inside myFunc \n"); } }</pre>	Code 15.6
--	----------------------

Output(s) When we run the program, we get the following output. It is seen that main function and thread function run in parallel.

Inside main
Inside main
Inside myFunc
Inside main
Inside myFunc
Inside main
Inside myFunc
...
...

Example 15.3 In this example, we show how to pass a message to thread function. For this purpose, we use the third argument of the `pthread_create()` function as in Code 15.7.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* myFunc(void* ptr);

int main ()
{
    pthread_t thrd; //thrd will hold the thread ID
    char* msg = "myFunc is called";
    int rc;

    rc = pthread_create(&thrd, NULL, myFunc, (void*) msg);
    if(rc)
    {
        printf("Error : cannot create thread");
        exit(-1);
    }
    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}
void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);

        char* msg;
        msg = (char*) ptr;
        printf("%s \n", msg);

        printf("Inside myFunc \n");
    }
}
```

Code
15.7

Output(s)

Inside main

Inside main

myFunc is called

Inside myFunc

Inside main

myFunc is called

Inside myFunc

Inside main

...

Example 15.4

We can use more than one thread function. In this example, two threads (thread functions) are used.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* myFunc1(void* ptr);
void* myFunc2(void* ptr);

int main ()
{
    pthread_t thr1, thr2; // will hold the thread IDs

    int rc1, rc2;

    pthread_create(&thr1, NULL, myFunc1, NULL);
    pthread_create(&thr2, NULL, myFunc2, NULL);

    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}

void* myFunc1(void* ptr)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc1 \n");
    }
}

void* myFunc2(void *threadid)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc2 \n");
    }
}
```

**Code
15.8**

Output(s)

Inside main
Inside myFunc1
Inside myFunc2
Inside main
Inside myFunc1
Inside main
Inside myFunc2
...

Example 15.5 Threads can be created for the same function. In this example, we create two threads that use the same function.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* myFunc(void* ptr);

int main ()
{
    pthread_t thr1, thr2;

    pthread_create(&thr1, NULL, myFunc, NULL);
    pthread_create(&thr2, NULL, myFunc, NULL);

    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}

void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc \n");
    }
}
```

Code
15.9

Output(s) Two threads use the same function; for this reason, the message “Inside myFunc” is printed twice after every “Inside main” message.

```
Inside main
Inside myFunc
Inside myFunc
Inside main
Inside myFunc
Inside myFunc
Inside main
Inside myFunc
Inside main
Inside myFunc
...
...
```

15.4 pthread_exit() Function

This function is used to terminate a thread. Its prototype is

```
void pthread_exit(void* retval);
```

where **retval** is the pointer that contains the return status of the thread terminated.

Example 15.6 In this example, thread functions contain infinite loops. However, when pthread_exit() function is met, the threads are terminated.

<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void* myFunc1(void* ptr); void* myFunc2(void* ptr); int main () { pthread_t thr1, thr2; // will hold the thread IDs int rc1, rc2; pthread_create(&thr1, NULL, myFunc1, NULL); pthread_create(&thr2, NULL, myFunc2, NULL); printf("Thread ID1 is : %lu \n", thr1); printf("Thread ID2 is : %lu \n", thr2); while(1) { printf("Inside main \n"); sleep(1); } } void* myFunc1(void* ptr) { while(1) { sleep(1); printf("Inside myFunc1 \n"); pthread_exit(NULL); } } void* myFunc2(void *threadid) { while(1) { sleep(1); printf("Inside myFunc2 \n"); pthread_exit(NULL); } }</pre>	Code 15.10
---	-----------------------------

Output(s)

Thread ID1 is : 1

Thread ID2 is : 2

Inside main

Inside myFunc1

Inside myFunc2

Inside main

Inside main

Inside main

...

Example 15.7 In this example, we show that when even the main function, that is, main thread terminates, any thread can go on running until it is terminated.

<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void* myFunc(void* ptr); int main () { pthread_t thrd; //thrd will hold the thread ID pthread_create(&thrd, NULL, myFunc, NULL); while(1) { printf("Inside main \n"); sleep(1); pthread_exit(NULL); } } void* myFunc(void* ptr) { while(1) { sleep(1); printf("Inside myFunc \n"); } }</pre>	Code 15.11
--	---

Output(s)

Inside main

Inside myFunc

Inside myFunc

Inside myFunc

....

15.5 pthread_join() Function

The function

```
pthread_join()
```

is used to wait for the termination of a target thread; if there are multiple threads running, all the other threads wait for the termination of the target thread. Besides, it is also possible that the termination of the main program before threads complete their tasks. Using pthread_join() function, it is guaranteed that the threads complete their run before the main program terminates.

The prototype of **pthread_join()** function is

```
int pthread_join(pthread_t th, void** ptr);
```

where

th: is the thread id of the target thread for which all the other threads wait
ptr: is the pointer to the location of the target thread where exit status is stored

If the call for pthread_join() is successful, zero is returned; otherwise, an error number is returned.

Example 15.8 In this example, the function **myFunc** has an infinite loop, and it never finishes. For this reason, the main part has no chance to be executed since it waits for the termination of the thread.

Code
15.12

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* myFunc(void* ptr);

int main ()
{
    pthread_t thrd; //thrd will hold the thread ID

    pthread_create(&thrd, NULL, myFunc, NULL);

    pthread_join(thrd , NULL);

    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}

void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc \n");
    }
}
```

Output(s)

Inside myFunc
 Inside myFunc
 Inside myFunc
 Inside myFunc
 Inside myFunc
 Inside myFunc

Example 15.9 In this example, we use pthread_join() function for two threads.

<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void* myFunc1(void* ptr); void* myFunc2(void* ptr); int main() { pthread_t thrd1, thrd2; pthread_create(&thrd1, NULL, myFunc1, NULL); pthread_create(&thrd2, NULL, myFunc2, NULL); pthread_join(thrd1, NULL); pthread_join(thrd2, NULL); printf("Inside main \n"); } void* myFunc1(void* ptr) { while(1) { printf("Inside myFunc1 \n"); sleep(1); } } void* myFunc2(void* ptr) { while(1) { printf("Inside myFunc2 \n"); sleep(1); } }</pre>	Code 15.13
--	-----------------------------

Output(s)

Inside myFunc1
 Inside myFunc2
 Inside myFunc1

Inside myFunc2

Inside myFunc1

....

Example 15.10 In this example, threads use the same function; messages are passed to the threads running in parallel.

**Code
15.14**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* myFunc(void* ptr);

int main()
{
    pthread_t thrd1, thrd2;

    char *msg1 = "Thread 1 executes";
    char *msg2 = "Thread 2 executes";

    pthread_create(&thrd1, NULL, myFunc, (void*) msg1);
    pthread_create(&thrd2, NULL, myFunc, (void*) msg2);

    pthread_join(thrd1, NULL);
    pthread_join(thrd2, NULL);

    printf("Inside main \n");
}

void* myFunc(void* ptr)
{
    while(1)
    {
        char *msg = (char *) ptr;
        printf("%s \n", msg);
        sleep(1);
    }
}
```

Output(s)

Thread 1 executes

Thread 2 executes

Thread 1 executes

Thread 2 executes

Thread 1 executes

....

Example 15.11 In this example, we show how to get a message from a terminated thread. Thread is terminated using `pthread_exit()` function. In Code 15.15, we first write the message in yellow to be returned by terminated thread.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

char* ret="Function is terminated, and returned to main.;"

void* myFunc(void* ptr);

int main ()
{
    pthread_t thrd;
    void* ret_ptr;

    pthread_create(&thrd, NULL, myFunc, NULL);

    printf("Inside main \n");

    sleep(1);

}

void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);

        printf("Inside myFunc \n");

        pthread_exit(ret);
    }
}
```

Code
15.15

Returned message from the terminated thread is obtained in the second argument of the `pthread_join()` function as shown in Code 15.16.

<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void* myFunc(void* ptr); char* ret="Function is terminated, and returned to main."; int main () { pthread_t thrd; void* ret_ptr; pthread_create(&thrd, NULL, myFunc, NULL); pthread_join(thrd, &ret_ptr); printf("Inside main \n"); sleep(1); printf("%s \n", (char*) ret_ptr); } void* myFunc(void* ptr) { while(1) { sleep(1); printf("Inside myFunc \n"); pthread_exit(ret); } }</pre>	Code 15.16
---	-----------------------

Output(s)

Inside myFunc
Inside main
Function is terminated, and returned to main.

Example 15.12 In Code 15.17, sky blue parts show the message passing from the main function to thread, and yellow parts show message passing from terminated thread to the main function.

```
#include <stdio.h>
#include <pthread.h>

int globalNum = 65;

void* myFunc(void* ptr)
{
    printf("Inside myFunc : Received localNum from main is: ");
    printf("%d \n", *(int*)ptr);

    pthread_exit(&globalNum);
}
int main(void)
{
    int localNum = 18;
    int* ip;

    pthread_t thrd;
    pthread_create(&thrd, NULL, myFunc, &localNum);

    pthread_join(thrd, (void**) &ip);

    printf("Inside main: Recevied globalNum from myFunc is : ");
    printf("%d \n", *ip);
}
```

**Code
15.17**

Output(s)

Inside myFunc : Received localNum from main is: 18

Inside main: Recevied globalNum from myFunc is : 65

Example 15.13 In this example, the terminated thread returns the integer 51 to the main function.

```
#include <stdio.h>
#include <pthread.h>

void* myFunc(void* ptr)
{
    printf("Terminating thread.\n");
    pthread_exit((void*) 51);
}
int main()
{
    int num;

    pthread_t thrd;

    pthread_create(&thrd, NULL, myFunc, NULL);

    pthread_join(thrd, (void**) &num);

    printf("Num = %d\n", num);
}
```

**Code
15.18**

Output(s)

Terminating thread.

Num = 51

15.6 pthread_self() Function

This function is used to get the thread id of the current thread, and its prototype is

```
pthread_t pthread_self(void);
```

Example 15.14 In this example, thread IDs of two threads are printed both in the main part and inside the threads.

<pre>#include <stdio.h> #include <pthread.h> void* myFunc1(void* ptr); void* myFunc2(void* ptr); int main() { pthread_t thrd1, thrd2; pthread_create(&thrd1, NULL, myFunc1, NULL); pthread_create(&thrd2, NULL, myFunc2, NULL); pthread_join(thrd1, NULL); pthread_join(thrd2, NULL); printf("Inside main \n"); printf("Thread ID1 is : %lu \n",thrd1); printf("Thread ID2 is : %lu \n\n",thrd2); } void* myFunc1(void* ptr) { pthread_t tid1 = pthread_self(); printf("Inside myFunc1 \n"); printf("Thread ID is : %lu \n\n",tid1); } void* myFunc2(void* ptr) { printf("Inside myFunc2 \n"); pthread_t tid2 = pthread_self(); printf("Thread ID is : %lu \n\n",tid2); }</pre>	Code 15.19
--	-----------------------

Output(s)

Inside myFunc1

Thread ID is : 140552266073664

Inside myFunc2

Thread ID is : 140552257680960

Inside main

Thread ID1 is : 140552266073664

Thread ID2 is : 140552257680960

15.7 pthread_equal() Function

It checks whether two threads are the same or not. If the two threads are equal, then the function returns a nonzero value, otherwise zero is returned. The prototype of the function is

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Example 15.15 This example illustrates the use of the pthread_equal() function.

```
#include <stdio.h>
#include <pthread.h>

pthread_t glb_tid;
void* myFunc(void* ptr);

int main()
{
    pthread_t thrd;

    pthread_create(&thrd, NULL, myFunc, NULL);

    glb_tid = thrd;

    pthread_join(thrd, NULL);
}
void* myFunc(void* ptr)
{
    pthread_t tid = pthread_self();

    if (pthread_equal(glb_tid, tid))
    {
        printf("Thread IDs are equal\n");
    }
    else
    {
        printf("Thread IDs are NOT equal\n");
    }
}
```

Code
15.20

Output(s)

Thread IDs are equal

15.8 pthread_cancel() Function

The pthread_cancel() function is used to cancel a target thread. The function prototype is

```
int pthread_cancel(pthread_t thread);
```

15.9 pthread_detach() Function

A thread is a joinable property by default, which means that when the thread terminates some information about the thread, it stays in the memory, that is, some artifacts about the terminated thread still stay. For instance, another thread can obtain the return status of the terminated thread using pthread_join() function. Sometimes, we do not care about the remaining information about the terminated thread, for instance, its return status, etc., we just want the system to automatically clean up and remove the thread artifacts when it terminates, and this is achieved using the pthread_detach() function. The prototype of the function is

```
int pthread_detach(pthread_t thrd);
```

If you use NULL for second argument of the pthread_create() function, the created thread will have joinable property.

The resources of a joinable threads cannot be freed by the thread itself; it can be freed by the other threads using the pthread_join() function.

The resources of a detached thread are automatically freed upon its termination. Once a thread is detached, it cannot be used with the pthread_join() function. If synchronization is not an issue, and efficient use of the resources is required, it is better to use pthread_detach().

Example 15.16 This example illustrates the use of the pthread_detach() function.

<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void* myFunc(void* ptr); int main() { pthread_t thrd; pthread_create(&thrd, NULL, myFunc, NULL); pthread_detach(thrd); printf("Inside main \n"); sleep(1); } void* myFunc(void* ptr) { printf("Inside myFunc \n"); }</pre>	Code 15.21
--	--

Output(s)

Inside main
Inside myFunc

15.10 Synchronizing Threads with Mutexes

Threads can increase the performance of the systems requiring especially heavy computations. Race is a concept that arises when more than one threads perform operations on shared resources. For instance, assume that a global variable is processed by two threads, and one thread increments the value of global variable and the other decrements the global variable, so what will be the result?

In this case, a race condition arises, and the result that arrives to the digital gate outputs will be the winner. Race arises from propagation delays of the electronic devices.

The race problem appearing in threads is eliminated using mutexes in C language. Mutexes have two basic functions: lock and unlock. Before a thread accesses a shared resource, lock function is called, which prevents other threads from using the shared resource; and when the thread is done with the shared resource, it calls the unlock function, and the other threads can use the shared resource.

If a mutex is unlocked and a thread calls lock, the mutex locks and the thread continues. If, however, the mutex is locked, the thread is blocked until the thread holding the lock calls unlocks.

The mutex variable is defined as

```
pthread_mutex_t mtx
```

the mutex variable is initialized using `pthread_mutex_init` whose prototype is

```
int pthread_mutex_init(
    pthread_mutex_t *mp,
    const pthread_mutexattr_t *attr
);
```

which can be used for the defined mutex variable as

```
pthread_mutex_init(&mtx, NULL);
```

and this expression is equivalent to

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

The prototypes of the lock, trylock, unlock, and destroy functions are

```
int pthread_mutex_lock(pthread_mutex_t* mtx);
int pthread_mutex_trylock(pthread_mutex_t* mtx);
int pthread_mutex_unlock(pthread_mutex_t* mtx);
int pthread_mutex_destroy(pthread_mutex_t* mtx);
```

Lock function is used for a mutex variable as

```
int rc = pthread_mutex_lock(&mtx);
if (rc)
{
    perror("Error in locking mutex");
    pthread_exit(NULL);
}
```

Unlock function is used for a mutex variable as

```

rc = pthread_mutex_unlock(&mtx);
if(rc)
{
    perror("Error in unlocking mutex ");
    pthread_exit(NULL);
}

```

Destroy function is used for a mutex variable as

```

rc = pthread_mutex_destroy(&mtx);
if(rc)
{
    perror("Error in destroying mutex ");
    pthread_exit(NULL);
}

```

`pthread_mutex_destroy()` returns zero after completing successfully. Any other returned value indicates that an error occurred.

Example 15.17 In this example, we show how synchronization is achieved using the mutex data type `pthread_mutex_t`. First, let us introduce a global variable that will be used by two functions.

```

#include <stdio.h>
#include <pthread.h>

int glb_var = 0;

int main()
{
}

```

**Code
15.22**

Let us add the function `myFunc1` to the code; it increments the value of the global variable.

```

#include <stdio.h>
#include <pthread.h>

int glb_var = 0;

void* myFunc1(void* ptr);

int main()
{
}

void* myFunc1(void* ptr)
{
    for(long unsigned indx = 0; indx < 100; indx++)
    {
        glb_var++;
    }
}

```

**Code
15.23**

Let us add the function `myFunc2` to the code as shown in Code 15.24; it decrements the value of the global variable.

```
#include <stdio.h>
#include <pthread.h>

int glb_var = 0;

void* myFunc1(void* ptr);
void* myFunc2(void* ptr);

int main()
{
}

void* myFunc1(void* ptr)
{
    for(long unsigned indx = 0; indx < 100; indx++)
    {
        glb_var++;
    }
}

void* myFunc2(void* ptr)
{
    for(long unsigned indx = 0; indx < 100; indx++)
    {
        glb_var--;
    }
}
```

Code
15.24

The threads for these two functions are created in Code 15.25.

#include <stdio.h> #include <pthread.h> int glb_var = 0; void* myFunc1(void* ptr); void* myFunc2(void* ptr); int main() { pthread_t thrd1, thrd2; pthread_create(&thrd1, NULL, myFunc1, NULL); pthread_create(&thrd2, NULL, myFunc2, NULL); } void* myFunc1(void* ptr) { for(long unsigned indx = 0; indx < 100; indx++) { glb_var++; } } void* myFunc2(void* ptr) { for(long unsigned indx = 0; indx < 100; indx++) { glb_var--; } }	Code 15.25
---	---------------

pthread_join() functions are added to the program as shown in Code 15.26.

#include <stdio.h> #include <pthread.h> int glb_var = 0; void* myFunc1(void* ptr); void* myFunc2(void* ptr); int main() { pthread_t thrd1, thrd2; pthread_create(&thrd1, NULL, myFunc1, NULL); pthread_create(&thrd2, NULL, myFunc2, NULL); pthread_join(thrd1, NULL); pthread_join(thrd2, NULL); printf("Inside main \n"); printf("Glob num is %d \n", glb_var); } void* myFunc1(void* ptr) { for(long unsigned idx = 0; idx < 100; idx++) { glb_var++; } } void* myFunc2(void* ptr) { for(long unsigned idx = 0; idx < 100; idx++) { glb_var--; } }	Code 15.26
---	---------------

If we run the Code-15.26, we get the output

Inside main
Glob num is 0

If we change the loop iteration number from 100 to 1000000000 and run the program we get

Inside main
Glob num is 5572371

if we run it again we get

Inside main
Glob num is -904775

If we run the program over and over again, we see that every time we see a different number for global variable value. This is due to the race event.

To eliminate the race effect, we use pthread_mutex_t data as in Code 15.27. Initialization on mutex variable is performed and is destroyed when the program finishes.

```
#include <stdio.h>
#include <pthread.h>

int glb_var = 0;

void* myFunc1(void* ptr);
void* myFunc2(void* ptr);

pthread_mutex_t mtx;

int main()
{
    pthread_t thrd1, thrd2;

    pthread_create(&thrd1, NULL, myFunc1, NULL);
    pthread_create(&thrd2, NULL, myFunc2, NULL);

    pthread_join(thrd1, NULL);
    pthread_join(thrd2, NULL);

    if (pthread_mutex_init(&mtx, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    printf("Inside main \n");

    printf("Global number is : % d \n", glb_var);

    pthread_mutex_destroy(&mtx);
}

void* myFunc1(void* ptr)
{
    for(long unsigned indx = 0; indx < 1000000000; indx++)
    {
        glb_var++;
    }
}

void* myFunc2(void* ptr)
{
    for(long unsigned indx = 0; indx < 1000000000; indx++)
    {
        glb_var--;
    }
}
```

Code
15.27

Finally, we add lock and unlock functions to the beginning and the end of the functions as shown in Code 15.28.

```
#include <stdio.h>
#include <pthread.h>

int glb_var = 0;

void* myFunc1(void* ptr);
void* myFunc2(void* ptr);

pthread_mutex_t mtx;

int main()
{
    pthread_t thrd1, thrd2;

    pthread_create(&thrd1, NULL, myFunc1, NULL);
    pthread_create(&thrd2, NULL, myFunc2, NULL);

    pthread_join(thrd1, NULL);
    pthread_join(thrd2, NULL);

    if (pthread_mutex_init(&mtx, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    printf("Inside main \n");

    printf("Global number is : % d \n", glb_var);

    pthread_mutex_destroy(&mtx);
}

void* myFunc1(void* ptr)
{
    pthread_mutex_lock(&mtx);

    for(long unsigned indx = 0; indx < 1000000000; indx++)
    {
        glb_var++;
    }
    pthread_mutex_unlock(&mtx);
}

void* myFunc2(void* ptr)
{
    pthread_mutex_lock(&mtx);

    for(long unsigned indx = 0; indx < 1000000000; indx++)
    {
        glb_var--;
    }
    pthread_mutex_unlock(&mtx);
}
```

Code
15.28

When we run Code 15.28 several times, we always get the output

Inside main
Glob num is 0

Thus, race event is eliminated and synchronization is achieved.

Problems

- Fill the dots in Code 15.29.

```
void* myFunc(void* ptr);

int main ()
{
    .....// define a thread ID here
    .....// create a theread for the function myfunc

    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}

void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);
        printf("Inside myFunc \n");
    }
}
```

Code
15.29

- For the previous problem, using for-loop creates 10 threads for myFunc in Code 15.29.
- Why do we use the pthread_join() function? What happens if we do not use this function?
- Explain the use of the pthread_detach() function.
- Fill the dots in Code 15.30 such that msg is passed to myFunc.

```
#include <stdio.h>
#include <pthread.h>

void* myFunc(void* ptr);

int main ()
{
    pthread_t thrd;
    char* msg = "myFunc is called";
    pthread_create(&thrd, NULL, myFunc, ....);

    while(1)
    {
        printf("Inside main \n");
        sleep(1);
    }
}

void* myFunc(void* ptr)
{
    while(1)
    {
        sleep(1);

        ...
        printf("%s \n", msg);

        printf("Inside myFunc \n");
    }
}
```

Code
15.30

```
pthread_mutex_t mtx;

void* myFunc1(void* ptr)
{
    ..... // write mutex lock function here

    for(long unsigned indx = 0; indx < 1000000000; indx++)
    {
        glb_var++;
    }
    ..... // use mutex unlock function here
}
```

Code
15.31

Chapter 16

Atomic Data Types



16.1 How to Define an Atomic Data Type?

The type `_Atomic` is used to avoid the race conditions when more than one thread tries to update the value of a variable simultaneously. The atomic data types are defined in `<stdatomic.h>`.

Atomic data types are available since C11.

Example 16.1 Three different methods can be used to define an atomic variable as in Code 16.1.

```
#include <stdio.h>
#include <stdatomic.h>

int main()
{
    _Atomic int var_name1; // 1st method

    atomic_int var_name2; // 2nd method

    _Atomic(int) var_name3; // 3rd method
}
```

Code
16.1

Example 16.2 The size of atomic data type can be calculated using the `sizeof()` operator.

```
#include <stdio.h>
#include <stdatomic.h>

int main()
{
    printf("Size of int is : %lu \n", sizeof(int));
    printf("Size of atomic_int : %lu \n", sizeof(atomic_int));
    printf("Size of _Atomic(int) : %lu \n", sizeof(_Atomic(int)));
}
```

Code
16.2

Output(s)

Size of int is : 4
 Size of atomic_int : 4
 Size of _Atomic(int) : 4

Atomics are optional features of compilers. So how can we check whether the compiler we use supports atomic types or not? There is a macro

__STDC_NO_ATOMICS__

If this macro is defined, then the compiler does NOT support atomics.

Example 16.3 In this example, we show how to check whether the compiler supports atomic data types or not.

```
#include <stdio.h>
#include <stdatomic.h>

int main()
{
    #ifdef __STDC_NO_ATOMICS__
        printf("Atomics are NOT supported.");
    #else
        printf("Atomics are supported.");
    #endif
}
```

Code
16.3

Output(s)

Atomics are supported.

16.2 Atomic Integer Types

Atomic integer data types can be listed as

Atomic type name	Direct type
atomic_bool	_Atomic bool
atomic_char	_Atomic char
atomic_schar	_Atomic signed char
atomic_uchar	_Atomic unsigned char
atomic_short	_Atomic short
atomic_ushort	_Atomic unsigned short
atomic_int	_Atomic int
atomic_uint	_Atomic unsigned int
atomic_long	_Atomic long
atomic_ulong	_Atomic unsigned long
atomic_llong	_Atomic long long
atomic_ullong	_Atomic unsigned long long
atomic_char8_t	_Atomic char8_t
atomic_char16_t	_Atomic char16_t
atomic_char32_t	_Atomic char32_t
atomic_wchar_t	_Atomic wchar_t
atomic_int_least8_t	_Atomic int_least8_t
atomic_uint_least8_t	_Atomic uint_least8_t
atomic_int_least16_t	_Atomic int_least16_t
atomic_uint_least16_t	_Atomic uint_least16_t
atomic_int_least32_t	_Atomic int_least32_t
atomic_uint_least32_t	_Atomic uint_least32_t
atomic_int_least64_t	_Atomic int_least64_t
atomic_uint_least64_t	_Atomic uint_least64_t
atomic_int_fast8_t	_Atomic int_fast8_t
atomic_uint_fast8_t	_Atomic uint_fast8_t
atomic_int_fast16_t	_Atomic int_fast16_t
atomic_uint_fast16_t	_Atomic uint_fast16_t
atomic_int_fast32_t	_Atomic int_fast32_t
atomic_uint_fast32_t	_Atomic uint_fast32_t
atomic_int_fast64_t	_Atomic int_fast64_t
atomic_uint_fast64_t	_Atomic uint_fast64_t
atomic_intptr_t	_Atomic intptr_t
atomic_uintptr_t	_Atomic uintptr_t
atomic_size_t	_Atomic size_t
atomic_ptrdiff_t	_Atomic ptrdiff_t
atomic_intmax_t	_Atomic intmax_t
atomic_uintmax_t	_Atomic uintmax_t

Other atomic data types can be defined using the type qualifier as

_Atomic(know_data_type)

For instance, we can define atomic float, double, and long double as

_Atomic(float) **_Atomic(double)** **_Atomic(long double)**

Typedef can also be used for atomic data types, for instance,

```
typedef _Atomic(double) atomic_double
```

then an atomic double variable can be defined as

```
atomic_double var_name;
```

Example 16.4 In this example, we define an atomic structure data type.

<pre><code>#include <stdio.h> #include <stdatomic.h> int main(void) { struct myStruct { int x; double y; }; _Atomic(struct myStruct) s; }</code></pre>	Code 16.4
--	----------------------

16.3 Atomic Pointers

Pointers are data types. As with any other data type, we can define atomic pointer data types.

Pointer to an Atomic data

Pointer to an atomic float is defined as

```
_Atomic float a;

_Atomic float* fp; // pointer to an atomic float

fp = &a;
```

Atomic Pointer to an Ordinary Data

An atomic pointer to a float is defined as

```
float a;  
  
float* __Atomic fp; // atomic pointer to a float  
  
fp = &a; // OK!
```

Atomic Pointer to an Atomic Data

An atomic pointer to an atomic float is defined as

```
__Atomic float a;  
  
__Atomic float * __Atomic fp; // atomic pointer to an atomic float  
  
fp = &a;
```

16.4 Race Prevention by Atomic Variables

Example 16.5 Atomic variables are used to prevent race conditions. In this example, we illustrate how the atomic variables eliminate the race condition. Let us define two global variables, one is atomic and the other is normal, as in Code 16.5.

```
#include <stdio.h>  
#include <pthread.h>  
#include <stdatomic.h>  
  
int glb_var1 = 0;  
  
__Atomic int glb_var2 = 0;  
  
int main()  
{  
}
```

Code
16.5

We add one thread function that increments the global variables as in Code 16.6.

```
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>

int glb_var1 = 0;
_Atomic int glb_var2 = 0;

void* myFunc1(void* ptr);

int main()
{
}

void* myFunc1(void* ptr)
{
    for(long unsigned indx = 0; indx < 10000000; indx++)
    {
        glb_var1++;
        glb_var2++;
    }
}
```

Code
16.6

We add a second thread function that decrements the global variables as in Code 16.7.

```
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>

int glb_var1 = 0;
_Atomic int glb_var2 = 0;

void* myFunc1(void* ptr);
void* myFunc2(void* ptr);

int main()
{
}

void* myFunc1(void* ptr)
{
    for(long unsigned indx = 0; indx < 10000000; indx++)
    {
        glb_var1++;
        glb_var2++;
    }
}
void* myFunc2(void* ptr)
{
    for(long unsigned indx = 0; indx < 10000000; indx++)
    {
        glb_var1--;
        glb_var2--;
    }
}
```

Code
16.7

We create two threads as in Code 16.8.

```
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>

int glb_var1 = 0;
_Atomic int glb_var2 = 0;

void* myFunc1(void* ptr);
void* myFunc2(void* ptr);

int main()
{
    pthread_t thrd1, thrd2;

    pthread_create(&thrd1, NULL, myFunc1, NULL);
    pthread_create(&thrd2, NULL, myFunc2, NULL);

}

void* myFunc1(void* ptr)
{
    for(long unsigned indx = 0; indx < 10000000; indx++)
    {
        glb_var1++;
        glb_var2++;
    }
}

void* myFunc2(void* ptr)
{
    for(long unsigned indx = 0; indx < 10000000; indx++)
    {
        glb_var1--;
        glb_var2--;
    }
}
```

Code
16.8

We add the pthread_join() functions as in Code 16.9.

```
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>

int glb_var1 = 0;
_Atomic int glb_var2 = 0;

void* myFunc1(void* ptr);
void* myFunc2(void* ptr);

int main()
{
    pthread_t thrd1, thrd2;

    pthread_create(&thrd1, NULL, myFunc1, NULL);
    pthread_create(&thrd2, NULL, myFunc2, NULL);

    pthread_join(thrd1, NULL);
    pthread_join(thrd2, NULL);

    printf("Inside main \n");
    printf("glb_var1 is % d \n", glb_var1);
    printf("glb_var2 is % d \n", glb_var2);
}

void* myFunc1(void* ptr)
{
    for(long unsigned indx = 0; indx < 10000000; indx++)
    {
        glb_var1++;
        glb_var2++;
    }
}

void* myFunc2(void* ptr)
{
    for(long unsigned indx = 0; indx < 10000000; indx++)
    {
        glb_var1--;
        glb_var2--;
    }
}
```

Code
16.9

Output(s) When the program is run, we get the output.

```
glb_var1 is -112381
glb_var2 is 0
```

From these outputs, it is seen that for nonatomic global variable, race condition exists, whereas, for atomic global variable, race condition is eliminated.

Example 16.6 This example is an improved version of the previous example. In this example, we create 20 threads that use the same function. We have one global atomic integer and a normal global integer variable. The thread function increments both variables.

<pre>#include<stdio.h> #include<stdatomic.h> #include<pthread.h> void* myFunc(void* ptr); _Atomic int a_glb = 0; int glb = 0; int main() { pthread_t thrd[20]; for(int i = 0; i < 20; i++) pthread_create(&thrd[i], NULL, myFunc, NULL); for(int i = 0; i < 20; i++) pthread_join(thrd[i], NULL); printf("The value of atomic global variable a_glb is %d\n", a_glb); printf("The value of global variable glb is %d \n", glb); } void* myFunc(void* ptr) { for(int indx = 0; indx < 10000; indx++) { a_glb++; // atomic operation glb++; } pthread_exit(NULL); }</pre>	Code 16.10
--	--

Output(s) When the program is run, we get the outputs.

The value of atomic global variable a_glb is 200000

The value of global variable glb is 197760

The for-loop is run 10000 times, and there are 20 threads. This means that global variables are incremented $20 \times 10000 = 200000$, and this value is displayed for atomic global variable, whereas, for nonatomic global variable, a smaller number is displayed.

From the outputs, it is seen that race condition exists for nonatomic global variable.

Example 16.7 If **a** is an atomic integer, then **a++** is an atomic operation, whereas **a = a+1** is not an atomic operation. If the thread function in the previous example is replaced by the function in Code 16.11, race condition is not eliminated for atomic variable.

```

void* myFunc(void* ptr)
{
    for(int indx = 0; indx < 10000; indx++)
    {
        a_glb = a_glb + 1; // NOT atomic operation

        glb = glb + 1;
    }
    pthread_exit(NULL);
}

```

Code
16.11

If the thread function in Code.16.10 is replaced by the function in Code 16.11, we get the outputs

The value of atomic global variable a_glb is 111918

The value of global variable glb is 133081

16.5 Lock-Free Atomic Types

Some hardware structures do not support atomic types, and when atomic types are met, parallel processing operations are performed using lock and unlock functions as in mutex lock and unlock functions. Atomic operations are fast, but parallel processing performed by lock and unlock functions are slower compared to atomic operations.

Some atomic types are forced to be lock-free; for instance, atomic flags are all lock-free.

To understand whether an atomic data type is lock-free or not, we can check its corresponding macro value. The macros of some atomic types are listed as follows:

Atomic Type	Lock Free Macro
atomic_bool	ATOMIC_BOOL_LOCK_FREE
atomic_char	ATOMIC_CHAR_LOCK_FREE
atomic_char16_t	ATOMIC_CHAR16_T_LOCK_FREE
atomic_char32_t	ATOMIC_CHAR32_T_LOCK_FREE
atomic_wchar_t	ATOMIC_WCHAR_T_LOCK_FREE
atomic_short	ATOMIC_SHORT_LOCK_FREE
atomic_int	ATOMIC_INT_LOCK_FREE
atomic_long	ATOMIC_LONG_LOCK_FREE
atomic_llong	ATOMIC_LLONG_LOCK_FREE
atomic_intptr_t	ATOMIC_POINTER_LOCK_FREE

Macros can have three different values:

- 0** which indicates that atomic type is never lock-free
- 1** which indicates that atomic type is sometimes lock-free
- 2** which indicates that atomic type is always lock-free

Example 16.8 We can check whether atomic integer type is lock-free or not in our computer using Code 16.12.

```
#include<stdio.h>
#include<stdatomic.h>
#include<pthread.h>

int main()
{
    if(ATOMIC_INT_LOCK_FREE==0)
    {
        printf("atomic_int type is never lock-free");
    }
    else if(ATOMIC_INT_LOCK_FREE==1)
    {
        printf("atomic_int type is sometimes lock-free");
    }
    else if(ATOMIC_INT_LOCK_FREE==2)
    {
        printf("atomic_int type is always lock-free");
    }
}
```

Code
16.12

Output(s) atomic_int type is always lock-free

16.6 Atomic Assignments, Operators, and Functions

Some operators operating on atomic variables are race-free, that is, they are atomic operators. Not all the operators are atomic ones.

For instance;

```
_Atomic int a = 0;
a++; // this is an atomic operation
a = a + 1; // this is NOT an atomic operation
```

The atomic operators are listed as

a++	a--	--a	++a	
a += b	a -= b	a *= b	a /= b	a %= b
a &= b	a = b	a ^= b	a >= b	a <= b

16.7 Atomic Functions

In this section, we explain some of the atomic functions.

16.7.1 *atomic_is_lock_free()* Function

The prototype of this function is

```
bool atomic_is_lock_free(const volatile A *obj);
```

This function returns true if the atomic operations on all objects of type A are lock-free.

Example 16.9

<pre>#include<stdio.h> #include<stdatomic.h> int main() { _Atomic int a; printf("_Atomic int a is %s \n", \ atomic_is_lock_free(&a) ? "lock-free" : "not lock-free"); }</pre>	Code 16.13
---	---

Output(s) _Atomic int a is lock-free

16.7.2 *atomic_fetch_key()* Function

The prototype of this function is

```
C atomic_fetch_key( volatile A* obj, M arg );
```

where C, nonatomic type, is the value held previously by the atomic object pointed to by obj, and arg is the argument supplied.

This function performs arithmetic and bitwise computations. The operations are applicable to atomic integer type. The operations that can be performed are

key	op	computation
add	+	addition
sub	-	subtraction
or		bitwise inclusive or
xor		bitwise exclusive or
and	&	bitwise and

If key = add, then we have the addition function

```
C atomic_fetch_add( volatile A* obj, M arg );
```

Example 16.10 This example illustrates the use of the atomic_fetch_add() function.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    _Atomic int a = 6;

    int ret;

    ret=atomic_fetch_add(&a, 8); // a=a+8

    printf("a = %d    ", a);
    printf("ret = %d", ret);
}
```

Code
16.14

Output(s) a = 14 ret = 6

Example 16.11 This example illustrates the use of the atomic_fetch_sub() function.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    _Atomic int a = 10;

    int ret;

    ret=atomic_fetch_sub(&a, 3); // a=a-3

    printf("a = %d    ", a);
    printf("ret = %d", ret);
}
```

Code
16.15

Output(s) a = 7 ret = 10

Example 16.12 This example illustrates the use of the atomic_fetch_xor() function.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    Atomic unsigned char a = 0x0F;
    Atomic unsigned char b = 0xF0;
    Atomic unsigned char ret;

    ret=atomic_fetch_xor(&a, b); // a = a XOR b

    printf("a = 0x%X    ", a);
    printf("ret = 0x%0X", ret);
}
```

**Code
16.16**

Output(s) a = 0xFF ret = 0x0F

Example 16.13 The function atomic_fetch_add() is race-free. This example shows that if we perform classical summation, race condition is not avoided, whereas performing summations using atomic_fetch_add() with an atomic variable avoids race problem.

```

#include<stdio.h>
#include<stdatomic.h>
#include<pthread.h>

void* myFunc(void* ptr);

Atomic int a = 0;
int b = 0;

int main()
{
    pthread_t thrd[20];

    for(int i = 0; i < 20; i++)
        pthread_create(&thrd[i], NULL, myFunc, NULL);

    for(int i = 0; i < 20; i++)
        pthread_join(thrd[i], NULL);

    printf("The value of atomic global a is %d\n", a);
    printf("The value of global variable b is %d \n", b);
}

void* myFunc(void* ptr)
{
    for(int idx = 0; idx < 10000; idx++)
    {
        atomic_fetch_add(&a, 8); // a = a+8, atomic operation
        b = b + 8; // not atomic operation
    }
    pthread_exit(NULL);
}

```

Code
16.17

Output(s)

The value of atomic global a is 1600000

The value of global variable b is 976200

The results show that when classical summation is performed by threads, race occurs, whereas the function `atomic_fetch_add()` avoids the race problem.

16.7.3 *atomic_store()* Function

The prototype of the function is

```
void atomic_store( volatile A* obj , C desired );
```

where A is an atomic data type and C is a normal, that is, nonatomic data type.

This is a write operation; it atomically, that is, race freely, performs the operation

***obj=desired**

Example 16.14 This example illustrates the use of the atomic_store() function.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    _Atomic int a;

    atomic_store(&a, 17); // atomic write operation

    printf("a = %d", a);
}
```

**Code
16.18**

Output(s) a = 17

16.7.4 atomic_load() Function

The prototype of the function is

C atomic_load(const volatile A* obj);

where A is an atomic data type and C nonatomic data type.

This is atomic read operation. It returns the current value of the atomic variable pointed by obj, that is, it performs

C=*obj

Example 16.15 This example illustrates the use of the atomic_load() function.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    _Atomic int a = 30;
    int ret;

    ret = atomic_load(&a); // atomic read operation

    printf("a = %d", a);
}
```

Code
16.19

Output(s) a = 30

16.7.5 atomic_exchange() Function

The prototype of the function is

```
C atomic_exchange( volatile A* obj, C desired );
```

where A is an atomic data type and C nonatomic data type.

This is atomic r read-modify-write operation. The value pointed by obj is replaced with desired, and the value obj held previously is returned.

Example 16.16 This example illustrates the use of the atomic_exchange() function.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    _Atomic int a = 30;
    int desired = 20;
    int c;

    printf("Value of a is : %d \n", a);

    c = atomic_exchange(&a, desired); // atomic operation

    printf("New value of a is : %d \n", a);
    printf("Old value of a is : %d", c);
}
```

Code
16.20

Output(s)

Value of a is : 30
 New value of a is : 20
 Old value of a is : 30

16.7.6 Comparison Functions

We have the atomic comparison functions

```
atomic_compare_exchange_weak()
atomic_compare_exchange_strong()
```

The prototypes of these functions are

```
_Bool atomic_compare_exchange_strong(volatile A* obj, \
                                      C* expected, C desired );
_Bool atomic_compare_exchange_weak(volatile A *obj, \
                                    C* expected, C desired );
```

The value pointed to by object is compared to the value pointed to by expected and

- If they are equal, the value pointed to by object is replaced with desired.
- If they are not equal, the value pointed to by expected is updated with the value pointed to by object.

Exchange weak is preferred in loops for better performance. Exchange weak sometimes returns false if even values are equal to each other.

Example 16.17 This example illustrates the use of atomic comparison functions.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    _Atomic int a = 23;
    _Atomic int b = 26;
    int expected = 23;
    int desired = 56;
    _Bool ret;

    printf("a is : %d \n", a);
    printf("expected is : %d \n", expected);
    printf("desired is : %d \n", desired);
    ret=atomic_compare_exchange_strong(&a, &expected, desired);
    printf("Comparison result is : %d \n",ret);
    printf("a is : %d \n", a);
    printf("expected is : %d \n\n", expected);

    printf("b is : %d \n", b);
    printf("expected is : %d \n", expected);
    printf("desired is : %d \n", desired);
    ret=atomic_compare_exchange_strong(&b, &expected, desired);
    printf("Comparison result is : %d \n",ret);
    printf("b is : %d \n", b);
    printf("expected is : %d \n\n", expected);
}
```

Code
16.21

Output(s)

```
a is : 23
expected is : 23
desired is : 56
Comparison result is : 1
a is : 56
expected is : 23

b is : 26
expected is : 23
desired is : 56
Comparison result is : 0
b is : 26
expected is : 26
```

16.7.7 *atomic_flag* Macro

It is an atomic Boolean type. This atomic type is guaranteed to be lock-free. The atomic type `atomic_bool` can perform load and store operations, whereas `atomic_flag` does not provide load or store operations. It has two states, set and clear.

The macro

`ATOMIC_FLAG_INIT`

initializes an `atomic_flag` variable to the clear state.

If an `atomic_flag` variable is not initialized, it gets an indeterminate state.

The function

`atomic_flag_test_and_set()`

with prototype

`_Bool atomic_flag_test_and_set(volatile atomic_flag *object);`

sets an `atomic_flag` variable to true, and it returns the value of variable before the update.

The function

`atomic_flag_clear()`

with prototype

`void atomic_flag_clear(volatile atomic_flag *object);`

sets an `atomic_flag` variable to false. There is no return value for this function.

Example 16.18 This example illustrates the use of the atomic flag functions.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    atomic_flag flg = ATOMIC_FLAG_INIT;
    _Bool ret;

    printf("Initial flag value is : %d\n", flg);

    ret=atomic_flag_test_and_set(&flg);

    printf("After set, flag value is : %d\n", flg);
    printf("After set, returned value is : %d\n", ret);

    atomic_flag_clear(&flg);

    printf("After clear, flag value is : %d\n", flg);
}
```

Code
16.22

Output(s)

Initial flag value is : 0
 After set, flag value is : 1
 After set, returned value is : 0
 After clear, flag value is : 0

16.7.8 *atomic_init()* Function

The prototype of the function is

```
void atomic_init(volatile A *obj, C value);
```

where A is an atomic data type and C nonatomic data type.

This function initializes the atomic variable A by the value of V. Although this function initializes an atomic variable, it is **NOT** a race-free function.

Example 16.19 This example illustrates the use of the atomic_init() function.

```
#include <stdio.h>
#include <stdatomic.h>

int main(void)
{
    _Atomic int a;

    int b = 18;

    atomic_init(&a,b); // the same as: atomic_init(&a,18)

    printf("a = %d\n", a);
}
```

Code
16.23

Output(s) a = 18

ATOMIC_VAR_INIT Macro

This macro is defined in C11 as

```
#define ATOMIC_VAR_INIT(C value)
```

However, it is deprecated in C17 and is removed in C23. It is NOT race-free.

Example

```
_Atomic int a = ATOMIC_VAR_INIT(50);
```

16.8 Memory Order in C

Before studying the memory order in C, let us see some terminology.

16.8.1 Acquire, Release, and Consume

Reading an atomic variable is called **acquire operation**, that is, acquire operation is atomic read operation, and writing an atomic variable is called **release operation**, that is, atomic write operation.

The word **load** can be used for read operation. In C, we have race-free atomic load function. In a similar manner, we can use the word **store** for write operation.

The sentence

“Let’s say that **a** is an atomic variable, and when a thread acquires **a**, it can see value of the **a** released in another thread.”

means that

“Let’s say that **a** is an atomic variable, and when a thread reads the value of atomic **a**, it can see value of the atomic **a** written in another thread.”

In a C program, besides atomic read and write operations, nonatomic read and write operations also exist.

Consume is a less-strict version of acquire. You cannot reorder anything before acquire operation. However, before consume operation you cannot reorder only those that depend on the loaded atomic value.

Fences

The word **fence** is used for nonatomic variables. Acquire of a nonatomic variable is called **fence acquire**, and similarly release of a nonatomic variable is called **fence release**.

16.8.2 *Memory Order*

The enumerated type **memory_order** is used to order memory accesses of regular nonatomic operations around an atomic operation. It is used to synchronize operations on distinct threads.

It is defined in <stdatomic.h> as

```
typedef enum
{
    memory_order_relaxed = __ATOMIC_RELAXED,
    memory_order_consume = __ATOMIC_CONSUME,
    memory_order_acquire = __ATOMIC_ACQUIRE,
    memory_order_release = __ATOMIC_RELEASE,
    memory_order_acq_rel = __ATOMIC_ACQ_REL,
    memory_order_seq_cst = __ATOMIC_SEQ_CST
} memory_order;
```

By default, for atomic operations, sequentially consistent memory ordering is used by the compiler. However, default choice can hurt performance.

The compiler can reorder memory access using selected member of **memory_order**.

memory_order_relaxed

No order is guaranteed concerning locking and normal memory accesses.

memory_order_consume

Assume that an atomic load operation is performed, no reads or writes in the current thread that depend on the value being loaded can be done.

This is almost the same as `memory_order_acquire`, except for that the ordering is guaranteed only for dependent data.

memory_order_acquire

Assume that an atomic load operation is performed, no reads or writes in the current thread that depend on the value being loaded can be done.

memory_order_release

Assume that an atomic store operation is performed, no reads or writes in the current thread can be done after this store operation. This prevents ordinary loads and stores from being reordered after the store operation.

memory_order_acq_rel

This is a hybrid of `memory_order_acquire` and `memory_order_release`. Assume inside a thread one variable is loaded and another one is stored, no memory reads or writes in the current thread can be reordered before the load, or after the store. That is, read and write operations cannot be reordered around the operation.

memory_order_seq_cst

A load operation with this memory order performs an acquire operation, a store performs a release operation, and read-modify-write performs both an acquire operation and a release operation (Table 16.1).

atomic_thread_fence

The prototype of this function is

```
void atomic_thread_fence(memory_order order);
```

This function achieves a memory synchronization order for nonatomic and relaxed atomic accesses without an associated atomic operation.

atomic_signal_fence

The prototype of this function is

```
void atomic_signal_fence(memory_order order);
```

Table 16.1 Supported memory orders for atomic functions

Functions	Supported memory orders				
	relaxed	acquire	release	acq_rel	seq_cst
load	✓	✓	✗	✗	✓
store	✓	✗	✓	✗	✓
Exchange, compare exchange, fetch	✓	✓	✓	✓	✓
fence	✓	✓	✓	✓	✓

The `atomic_signal_fence()` achieves synchronization of nonatomic and relaxed atomic accesses between a thread and a signal handler that are included in the same thread.

16.8.3 Atomic Functions with Memory Order

In the previous sections, we explained the functions

```
void atomic_store(volatile A *object, C desired);
C atomic_load(const volatile A *object);

C atomic_exchange(volatile A *object, C desired);

bool atomic_compare_exchange_strong(volatile A *object, C *expected,\n
                                    C desired);

bool atomic_compare_exchange_weak(volatile A *object, C *expected,\n
                                  C desired);

C atomic_fetch_key(volatile A *object, M operand);

bool atomic_flag_test_and_set(volatile atomic_flag *object);

void atomic_flag_clear(volatile atomic_flag *object);
```

These functions can be used with memory order property; the operation of the functions is the same, and the prototypes of these functions with memory order input are

```

void atomic_store_explicit(volatile A *object, C desired,\n
                           memory_order order);

C atomic_load_explicit(const volatile A *object, memory_order order);
C atomic_exchange_explicit(volatile A *object, C desired,\n
                           memory_order order);

bool atomic_compare_exchange_strong_explicit(volatile A *object,\n
                                              C *expected,\n
                                              C desired,\n
                                              memory_order success, memory_order failure);

bool atomic_compare_exchange_weak_explicit(volatile A *object,\n
                                            C *expected, C desired,\n
                                            memory_order success,\n
                                            memory_order failure);

C atomic_fetch_key_explicit(volatile A *object, M operand,\n
                           memory_order order);

bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object,\n
                                       memory_order order);

void atomic_flag_clear_explicit(volatile atomic_flag *object,\n
                                 memory_order order);

```

Problems

1. Why do we use atomic data types? Can we use them in ordinary C programs that do not contain threads?
2. Are there any differences between the use of atomic data types and mutex functions? Which one is preferable and why?
3. Define an atomic integer variable in three different ways.
4. Define an atomic double variable.
5. Assume that a is an atomic variable; is the operation `a=a+2` an atomic operation?
6. Write three operations that are atomic by default.
7. What does lock-free atomic type mean?
8. Write three lock-free atomic types.
9. Write a C code to check whether an atomic integer is lock-free or not.
10. Consider the function `atomic_fetch_key()`. What can be replaced for the word "key" in this function?
11. Explain the use of `atomic_store()` and `atomic_load()` functions.
12. What is the difference between `atomic_flag` and `atomic_bool`?
13. In which case is it beneficial to use memory orders?
14. How many memory order methods are available, and list them.

Chapter 17

File Operations in C



17.1 File Types

There are two types of files used in C programming:

- Text files
- Binary files

Text files have extension **.txt**, whereas binary files have extension **.bin**.

17.2 File Operations

File operations consist of

- Opening a file
- Reading or writing the file
- Closing the file

Every opened file after reading or writing must be closed before a program is terminated.

17.2.1 *Opening a File*

Files can be opened for reading and writing using the `fopen()` function. The prototype of the `fopen()` function is

```
FILE* fopen(const char* fileName, const char* operationMode);
```

For text files, `operationMode` can be one of these:

- r** The text file is opened just for reading. If the file does not exist, NULL is returned by `fopen()`.
- w** The text file is opened just for writing. If the file exists, its contents are overwritten. If the file does not exist, a new file is created.
- a** The text file is opened for appending only. Appending means concatenating data to the end of the file. If the file does not exist, a new file is created.
- r+** The text file is opened for both reading and writing. If the file does not exist, NULL is returned by `fopen()`.
- w+** The text file is opened for both reading and writing. If the file exists, its contents are overwritten. If the file does not exist, a new file is created.
- a+** The text file is opened for both reading and appending. If the file does not exist, a new file is created.

For binary files, `operationMode` can be one of these:

- rb** The binary file is opened just for reading. If the file does not exist, NULL is returned by `fopen()`.
- wb** The binary file is opened just for writing. If the file exists, its contents are overwritten. If the file does not exist, a new file is created.
- ab** The binary file is opened for appending only. Appending means concatenating data to the end of the file. If the file does not exist, a new file is created.
- rb+** The binary file is opened for both reading and writing. If the file does not exist, NULL is returned by `fopen()`.
- wb+** The binary file is opened for both reading and writing. If the file exists, its contents are overwritten. If the file does not exist, a new file is created.
- ab+** The binary file is opened for both reading and appending. If the file does not exist, a new file is created.

17.2.2 Closing a File

An opened file is closed using the `fclose()` function; the prototype of the `fclose()` function is

```
fclose(FILE* fp);
```

Example 17.1 This example illustrates how to open a text for writing file.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE* fp; // file pointer

    fp = fopen("myFile.txt", "w");

    if(fp == NULL)
    {
        printf("File cannot be opened.");
        exit(1);
    }
    else
    {
        printf("File is opened successfully.");
    }
    fclose(fp); // close the file
}
```

**Code
17.1**

Output(s) myFile.txt

17.2.3 Reading and Writing of a Text File

17.2.3.1 Writing to a Text File

We can write data to a file in C using the fprintf(), fputs(), and fputc() functions.

fprintf()

fprintf() function is used to write data to a text file; its prototype is

```
int fprintf(FILE* fp, const char* format,... ); (before C99)
int fprintf(FILE* restrict fp, const char* restrict format,... ); (C99)
int fprintf_s(FILE* restrict fp, const char* restrict format,...); (C11)
```

It can also be used for strings without specifying format as

```
int fprintf(FILE* fp, const char* str);
```

Example 17.2 In this example, we open a file and write the text “Hello World!” to the file.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE* fp; // file pointer

    fp = fopen("myFile.txt", "w");

    fprintf(fp,"Hello World-1! \n"); // \n means go to next line
    fprintf(fp,"Hello World-2!");
}
```

Code
17.2

Output(s)

myFile.txt	File 17.3
Hello World-1! Hello World-2!	

In this example, for simplicity of coding, we did not check the return values of fopen() and fprintf() functions. Our aim here was to illustrate the use of fprintf() function; otherwise, they should always be checked.

In Code 17.2, the statement

```
fprintf(fp, "Hello World!");
```

can also be written as

```
fprintf(fp, "%s", "Hello World!");
```

Example 17.3 In this example, we show how to get an integer from the user and write it to a text file.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num;

    FILE *fp;

    fp = fopen("myFile.txt", "w");

    printf("Please enter an integer : ");

    scanf("%d", &num);

    fprintf(fp, "You entered : ");

    // fprintf(fp, "%s", "You entered 2: ");

    fprintf(fp, "%d", num);

    fclose(fp);
}
```

Code
17.4

Output(s) Please enter an integer : 38

myFile.txt
You entered : 38

File
17.5

fputs()

fputs() function can be used to write data to a text file; its prototype is

```
int fputs(const char* str, FILE* fp);  (before C99)

int fputs(const char* restrict str, FILE* restrict fp); (C99)
```

Example 17.4

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE* fp; // file pointer
    fp = fopen("myFile.txt", "w");
    fputs("Hello World!", fp);
    fclose(fp); // close the file
}
```

**Code
17.6**

Note the arguments of fprintf() and fputs() functions in Codes 17.4 and 17.6. File pointer is written as first argument in fprintf(), whereas it is written as second argument in fputs().

Output(s)

myFile.txt	File 17.7
Hello World!	

fputc()

fputc() function is used to write a single character to a text file; its prototype is

```
int fputc(int ch, FILE* fp);
```

where ch is converted to unsigned char just before being written.

Example 17.5

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE* fp; // file pointer

    fp = fopen("myFile.txt", "w");

    fputc('H',fp);
    fputc('e',fp);
    fputc('l',fp);
    fputc('l',fp);
    fputc('o',fp);
    fputc('!',fp);

    fclose(fp); // close the file
}
```

**Code
17.8****Output(s)**

myFile.txt	File 17.9
Hello!	

17.2.3.2 Reading a Text File

A text file in C can be read using the functions fscanf(), fgets(), and fgetc().

fscanf()

fscanf() function is used to read data from a text file; its prototype is

```
int fscanf(FILE* fp, const char* format,...); (before C99)
int fscanf(FILE* restrict fp, const char* restrict format,...); (C99)
```

fscanf() function reads text file until a space or new line '\n' is met or end of file is reached.

Example 17.6 Assume that we have the text file myFile.txt.

myFile.txt	File 17.10
Hello World-1! Hello World-2! Hello World-3!	

<pre>#include <stdio.h> int main() { FILE * fp; int num; fp = fopen("myFile.txt", "r"); char str[10]; fscanf(fp, "%s", str); printf("%s\n", str); }</pre>	Code 17.11
---	----------------------

Output(s) Hello

Let us add two more lines to the code as in Code 17.12.

<pre>#include <stdio.h> int main() { FILE * fp; fp = fopen("myFile.txt", "r"); char str[10]; fscanf(fp, "%s", str); printf("%s\n", str); fscanf(fp, "%s", str); printf("%s\n", str); }</pre>	Code 17.12
--	----------------------

Output(s)

Hello
World-1!

Instead of repeating fscanf() and printf() functions, we can write a loop as in Code 17.13 to read all the contents of the text file.

```
#include <stdio.h>
int main()
{
    FILE* fp;
    int ret;

    fp = fopen("myFile.txt", "r");

    char str[10];

    while(1)
    {
        ret = fscanf(fp,"%s", str);

        if(ret == EOF)
            break;

        printf("%s", str);
    }
}
```

Code
17.13

Output(s) HelloWorld-1!HelloWorld-2!HelloWorld-3!

The C function feof() can be used to check the end of file. The prototype of the feof() function is

```
int feof(FILE* fp);
```

The C function feof() returns zero if end of file is NOT reached.

Code 17.13 can be written using feof() as in Code 17.14.

```
#include <stdio.h>
Code
17.14
int main()
{
    FILE* fp;
    int ret;

    fp = fopen("myFile.txt", "r");

    char str[10];

    while(!feof(fp))
    {
        fscanf(fp,"%s", str);
        printf("%s", str);
    }
}
```

Output(s) HelloWorld-1!HelloWorld-2!HelloWorld-3!

Example 17.7 Assume that we have the text file myFile.txt.

```
#include <stdio.h>
Code
17.15
int main()
{
    printf("Value of EOF is: %d\n", EOF);
}
```

Output(s) Value of EOF is: -1

Example 17.8 Assume that we have the text file myFile.txt.

myFile.txt	File 17.16
25 789 Hello World!	

<pre>#include <stdio.h> int main() { FILE * fp; int a; fp = fopen("myFile.txt", "r"); char str[10]; fscanf(fp,"%d", &a); printf("%d ", a); }</pre>	Code 17.17
---	-----------------------

Output(s) 25

Let us add to the code two more fscanf() and printf() expressions as in Code 17.18.

<pre>#include <stdio.h> int main() { FILE * fp; int a; fp = fopen("myFile.txt", "r"); char str[10]; fscanf(fp,"%d", &a); printf("%d ", a); fscanf(fp,"%d", &a); printf("%d ", a); fscanf(fp,"%s", str); printf("%s ", str); }</pre>	Code 17.18
--	-----------------------

Output(s) 25 789 Hello

17.2.4 Reading and Writing of a Binary File

Binary files have usually extension .bin in your computer. Data is stored in binary, that is, 0 and 1, form.

17.2.4.1 Writing to Binary Files

Binary files can be written using the function fwrite() .

```
fwrite()
```

The prototype of the fwrite() function is

```
size_t fwrite(const void* ptr, size_t element_size,\n            size_t count, FILE* fp); (before C99)
```

```
size_t fwrite(const void* restrict ptr, size_t size,\n            size_t count, FILE* restrict fp); (C99)
```

where

- **ptr** is a pointer pointing to the data block to be written
- **element_size** is the size of an element of the data block
- **count** is the number of elements in the data block
- **fp** is the file pointer

Example 17.9 In this example, we write three integers to a binary file. Return value of the fwrite() function is the number of elements written to the file.

<pre>#include<stdio.h> int main() { FILE* fp; int a[3] = {34, 67, 89}; fp = fopen("myFile.txt" , "w"); unsigned long ret = fwrite(a, sizeof(int), 3, fp); printf("ret = %lu", ret); fclose(fp); }</pre>	Code 17.19
---	---

Output(s) ret = 3

Although the file name is myFile.txt, it is a binary file, and its contents cannot be viewed.

17.2.4.2 Reading Binary Files

Binary files can be read using the function `fread()`.

```
fread()
```

The prototype of the `fread()` function is

```
size_t fread(void* ptr, element_size, size_t count, FILE* fp);  
size_t fread(void* restrict ptr, element_size,\ (before C99  
size_t count, FILE* restrict fp); (C99)
```

where

- **ptr** is the head of the address where read data is to be written
- **element_size** is the size of an element to be read
- **count** is the number of elements to be read
- **fp** is the file pointer

Example 17.10 In this example, we first write three integers to a binary file, and then read these three integers from the file and print them. However, before reading the integers from file we close the file and change the file mode from write to read.

```
#include<stdio.h>  
  
int main()  
{  
    FILE* fp;  
  
    int a[3] = {34, 67, 89};  
  
    int b[3];  
  
    fp = fopen( "myFile.txt" , "w" );  
  
    fwrite(a, sizeof(int), 3 , fp );  
  
    fclose(fp);  
  
    fp = fopen( "myFile.txt" , "r" ); // we change file mode here  
  
    fread(b, sizeof(int), 3 , fp );  
  
    printf("Read array elements are : %d, %d, %d", a[0], a[1], a[2]);  
  
    fclose(fp);  
}
```

Code
17.20

Output(s) Read array elements are : 34, 67, 89

Example 17.11 In this example, we illustrate how to write structure objects to a binary file and read it. We first define a structure as a global data type as in Code 17.21.

```
#include <stdio.h>
struct student
{
    int st_id;
    char st_name[20];
};

int main()
{
    FILE* fp;
```

Code
17.21

We define two structure arrays as in Code 17.22. The first one is initialized. We open a binary file using the fopen() function.

```
#include <stdio.h>
struct student
{
    int st_id;
    char st_name[20];
};

int main()
{
    FILE* fp;

    struct student st1[2] = {
        {38729, "Ilhan Gazi"},  

        {68545, "Orhan Gazi"}  

    };

    struct student st2[2];

    fp = fopen("myFile.bin", "w");
}
```

Code
17.22

The first structure array is written to binary file as in Code 17.23. The file is closed using the fclose() function.

```
#include <stdio.h>

Code
17.23

struct student
{
    int st_id;
    char st_name[20];
};

int main()
{
    FILE* fp;

    struct student st1[2] = {
        {38729, "Ilhan Gazi"},  

        {68545, "Orhan Gazi"}  

    };

    struct student st2[2];

    fp = fopen("myFile.bin", "w");

    fwrite(st1, sizeof(struct student), 2, fp);

    fclose(fp);
}
```

The same binary file is opened for reading operation and read data is placed into the second structure array as in Code 17.24.

```
#include <stdio.h>

struct student
{
    int st_id;
    char st_name[20];
};

int main()
{
    FILE* fp;

    struct student st1[2] = {
        {38729, "Ilhan Gazi"},  

        {68545, "Orhan Gazi"}  

    };

    struct student st2[2];

    fp = fopen("myFile.bin", "w");

    fwrite(st1, sizeof(struct student), 2, fp);

    fclose(fp);

    fp = fopen("myFile.bin", "r");

    fread(st2, sizeof(struct student), 2, fp);
}
```

Code
17.24

Finally, the read data is displayed using the printf() function as in Code 17.25, and the opened file is closed using the fclose() function. In Code 17.25, we omitted conditional expressions to check the results of file opening, file writing, file reading, and file closing operations for the simplicity of the code to focus on the main steps. However, they should never be ignored in professional coding.

<pre>#include <stdio.h> struct student { int st_id; char st_name[20]; }; int main() { FILE* fp; struct student st1[2] = { {38729, "Ilhan Gazi"}, {68545, "Orhan Gazi"} }; struct student st2[2]; fp = fopen("myFile.bin", "w"); fwrite(st1, sizeof(struct student), 2, fp); fclose(fp); fp = fopen("myFile.bin", "r"); fread(st2, sizeof(struct student), 2, fp); printf("Student ID is %d \n", st2[0].st_id); printf("Student name is %s\n", st2[0].st_name); printf("Student ID is %d \n", st2[1].st_id); printf("Student name is %s", st2[1].st_name); fclose(fp); }</pre>	Code 17.25
---	---------------

Output(s)

Student ID is 38729
Student name is Ilhan Gazi
Student ID is 68545
Student name is Orhan Gazi

fseek()

When a file is opened, the file pointer points to the beginning of the file. The function fseek() is used to move file pointer to a specific location in the file. The prototype of the fseek() function is

```
int fseek(FILE* fp, long int offset, int position);
```

where

- fp** is the file pointer
- offset** is the number of bytes to offset from the position of the file pointer
- position** is the position of the file pointer from where the offset is added

It has three values

- SEEK_END denotes the end of the file
- SEEK_SET denotes the beginning of the file
- SEEK_CUR denotes the current position of the file pointer

rewind()

The function rewind() is used to move file pointer to the beginning of the file; its prototype is

```
void rewind(FILE*fp);
```

Problems

1. Write a C program that writes the string "I like C programming" to a text file. Write another C program that opens the previously written text file and displays each word of the text in the file separately.
2. Define a structure that contains, integer, double, and string data types. Define an array of structures and initialize the array. Write the array into a binary file and read the file and display the content of the file.

Bibliography

1. C99 – ISO/IEC 9899:1999
2. C11 – ISO/IEC 9899:2011
3. C17 – ISO/IEC 9899:2018
4. C23 – ISO/IEC 9899:2023

Index

A

Absolute value, 161, 162
Arithmetic, 43, 74, 158, 160, 311, 354
Array initialization, 177–183
Arrays, 81, 97, 175–191, 224, 227, 230–237, 382, 383
Array size, 177, 178
Array storage, 182
Atomic functions, 354–364, 367–368
Atomic integers, 344–346, 350, 351, 353, 354, 368
Atomic pointer, 346–347
Atomic types, 344, 352, 362, 368

B

Binary, 1, 3–6, 8, 10, 21, 22, 24, 27, 28, 38, 40, 43, 51–53, 58, 68, 71, 379
Binary files, 369, 370, 379–386
Bit fields, 91–92
Boolean condition, 99, 126

C

Char, 19–25, 38, 51–53, 63, 93, 94, 96, 176, 214, 234, 374
Complex number, 97, 157–173
Complex sine, 168
Conditional compilation, 259, 270–274
Conditional statements, 99–121, 142
Conjugate, 163
Const, 277–279
Conversion, 3–7, 67–76, 218, 219

D

Data types, 15–64, 71, 73–78, 83, 84, 87, 92–94, 102, 154, 157, 158, 194, 207, 209, 210, 215, 217, 218, 228, 241, 277, 278, 281, 289, 293, 295–301, 313, 334, 343–346, 352, 357–359, 363, 368, 382, 386
Dereferencing operator, 207–209
Designated initializer, 79–83
Directives, 17, 18, 94, 259–274
Double, 19, 39, 41, 63, 69, 72, 101, 158, 160–162, 165, 176, 207, 239, 277, 345, 346, 368, 386
Do-while loop, 147–149, 151, 154–156

E

Enumeration, 277–287, 313
Explicit, 67, 70–72, 79, 218, 219, 284

F

File close, 370, 381
File opening, 384
File reading, 384
Files, 18, 122, 265, 281, 298, 303, 316, 369
File writing, 384
Fixed width, 289, 298
Float, 19, 39, 41, 63, 67, 68, 71, 72, 96, 100, 158, 161, 162, 165, 176, 191, 207, 278, 345–347
For loop, 125–140, 149, 151–153, 156, 179, 184, 186, 340, 351

Function arguments, 92–93, 189, 190, 201, 222, 223, 230
 Function calling, 202
 Function prototype, 196, 314, 331
 Functions, 15, 69, 92, 104, 127, 222, 269, 280, 290, 303, 313, 375

H

Hexadecimal, 2, 4–8, 12, 13, 24, 27–29, 33, 35–38, 64, 181, 183, 299, 300
 Higher order, 73–74

I

If-else, 99, 110–112, 264
 Imaginary, 97, 158, 161, 173
 Implicit, 67–70
 Information loss, 73–76
 Int, 15, 16, 19, 20, 63, 68, 75, 93, 94, 354
 Integer, 1, 19, 57, 67, 96, 99, 156, 208, 267, 289, 303, 314, 328, 350, 372
 Interrupts, 280, 303–308, 310, 312
 int_leastN_t, 295–296, 299
 intN_t, 299

L

Ladder structure, 104–107, 273
 Little-endian, 12–13, 29, 181, 208
 Lock-free, 352–354, 362, 368
 Logical, 43, 47–50, 63, 103, 107, 134, 143, 147
 Loop statements, 125–156

M

Macros, 161, 259–274, 289–293, 295–298, 303, 304, 308, 313, 344, 352, 362, 364
 Memory order, 364–368
 Multiconditional, 107–110, 133, 144
 Multidimensional arrays, 185–188
 Multiline macros, 262–265
 Mutex, 332–340, 368
 Mutex lock, 333, 352
 Mutex unlock, 333, 334, 339, 341

N

Nested loops, 138, 144
 Nested structures, 86–87
 Number bases, 1–2

O

Octal, 2–4, 6, 12, 24, 28–29, 64, 299, 300
 Operators, 15–66, 107, 109, 112–116, 134, 158, 175, 207, 209, 215, 274, 284, 285, 314, 343, 353

P

Parallel processing, 313–320, 352
 Pointers, 88, 90, 200, 201, 207–252, 277–279, 285, 286, 305, 314, 321, 323, 346, 374, 380, 381, 385, 386
 Predefined macros, 267–270
 Preprocessor, 259–261, 264, 265

R

Race condition, 332, 343, 347, 350, 351, 356
 Race prevention, 347–352
 Raise function, 312
 Real, 19, 39, 40, 44, 45, 97, 157, 158, 160, 161, 173
 Register, 11, 12, 22, 29, 30, 36, 181, 210, 245, 279, 280, 284–287
 Restrict, 279, 287

S

Self-referential, 89–90
 SIGINT, 304–308, 310
 Signal handling, 304, 310
 Signals, 303–312, 367
 Signed, 19, 27, 38, 53, 74–76, 289, 290, 292–296, 299, 300
 SIGQUIT, 308–309
 sizeof, 22–23, 26, 63, 209, 215, 216, 314, 343
 Storage classes, 277–287
 Structures, 77–96, 99–103, 107–110, 112, 125, 126, 137, 139, 144, 145, 151, 194, 202, 314, 315, 346, 352, 382, 383, 386
 Switch statement, 116–123

T

Text files, 369–379, 386
 Thread creation, 313–314, 317
 Threads, 313–340, 343, 347–352, 357, 365–368
 Two's complement, 8–10
 Typedef, 83–85, 96, 345
 Type qualifiers, 277–286, 345

U

uint_leastN_t, 296
uintN_t, 293–295
Unions, 63, 94–97
Unsigned, 23, 24, 34–37, 51, 52, 54, 74, 75,
 248, 293, 299, 300, 374
Unsigned integer, 8, 34–39, 296, 297

V

Variable address, 229
Void pointer, 215–221, 226
Volatile, 279, 280

W

While-loop, 140–146, 148, 150, 151, 153, 154