

C Programming

Cookbook

Over 40 recipes exploring data structures, pointers, interprocess communication, and database in C



Packt

www.packt.com

B.M. Harwani

C Programming Cookbook

Over 40 recipes exploring data structures, pointers,
interprocess communication, and database in C

B.M. Harwani

Packt

BIRMINGHAM - MUMBAI

C Programming Cookbook

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Richa Tripathi

Acquisition Editor: Alok Dhuri

Content Development Editor: Afshaan Khan

Technical Editor: Mayank Dubey

Copy Editor: Safis Editing

Project Coordinator: Vaidehi Sawant

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Graphics: Alishon Mendonsa

Production Coordinator: Nilesh Mohite

First published: March 2019

Production reference: 1280319

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78961-745-0

www.packtpub.com

To Guido van Rossum, the creator of the Python language. Python is not only very commonly used for making real-time applications, web applications, and smartphone applications, but is also used in AI, data learning, Internet of things (IoT), and much more.

To my mom, Mrs Nita Harwani. She is next to God for me. Whatever I am today is because of the moral values she taught me.

– Bintu Harwani



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

B.M. Harwani is the founder of Microchip Computer Education, based in Ajmer, India, which provides computer literacy in programming and web development to learners of all ages. He further helps the community by sharing the knowledge and expertise he's gained over 20 years of teaching through writing books. His recent publications include *jQuery Recipes*, published by Apress, *Introduction to Python Programming and Developing GUI Applications with PyQT*, published by Cengage Learning, *The Android Tablet Developer's Cookbook*, published by Addison-Wesley Professional, *UNIX and Shell Programming*, published by Oxford University Press, and *Qt5 Python GUI Programming Cookbook*, published by Packt.

About the reviewer

Nibedit Dey is a software engineer turned entrepreneur with over 8 years of experience of building complex software-based products. Before starting his entrepreneurial journey, he worked for L&T and Tektronix in different research and development roles. He has reviewed *The Modern C++ Challenge*, *Hands-on GUI programming with C++ and Qt5*, *Getting Started with Qt5*, and *Hands-On High Performance Programming with Qt 5* for Packt.

I would like to thank the online programming communities, bloggers, and my peers from earlier organizations from whom I have learned a lot over the years.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
<hr/>	
Section 1: Arrays, Strings, and Functions	
<hr/>	
Chapter 1: Working with Arrays	8
Inserting an element in an array	8
How to do it...	9
How it works...	11
There's more...	13
Multiplying two matrices	15
How to do it...	15
How it works...	19
There's more...	21
Finding the common elements in two arrays	22
How to do it...	22
How it works...	25
Finding the difference between two sets or arrays	27
How to do it...	27
How it works...	29
Finding the unique elements in an array	31
How to do it...	31
How it works...	34
Finding whether a matrix is sparse	35
How to do it...	35
How it works...	37
There's more...	39
Merging two sorted arrays into a single array	42
How to do it...	42
How it works...	45
Chapter 2: Managing Strings	48
Determining whether the string is a palindrome	48
How to do it...	49
How it works...	50
Finding the occurrence of the first repetitive character in a string	52
How to do it...	53
How it works...	55
Displaying the count of each character in a string	57
How to do it...	58
How it works...	60

Counting vowels and consonants in a sentence	63
How to do it...	63
How it works...	65
Converting the vowels in a sentence to uppercase	66
How to do it...	67
How it works...	69
Chapter 3: Exploring Functions	70
What is a stack?	71
Finding whether a number is an Armstrong number	72
How to do it...	73
How it works...	75
Returning maximum and minimum values in an array	79
How to do it...	79
How it works...	82
Finding the greatest common divisor using recursion	84
How to do it...	84
How it works...	86
Converting a binary number into a hexadecimal number	88
How to do it...	88
How it works...	92
Finding whether a number is a palindrome	95
How to do it...	96
How it works...	99

Section 2: Pointers and Files

Chapter 4: Deep Dive into Pointers	103
What is a pointer?	103
Reversing a string using pointers	107
How to do it...	107
How it works...	109
Finding the largest value in an array using pointers	113
How to do it...	113
How it works...	115
Sorting a singly linked list	117
How to do it...	118
How it works...	122
Creating a singly linked list	122
Sorting the singly linked list	124
The first iteration	125
The second iteration	127
The third and fourth iterations	128
Finding the transpose of a matrix using pointers	129
How to do it...	130
How it works...	134

Accessing a structure using a pointer	136
How to do it...	136
How it works...	139
Chapter 5: File Handling	142
Functions used in file handling	143
Functions commonly used in sequential file handling	143
fopen()	143
fclose()	144
fgets()	144
fputs()	144
Functions commonly used in random files	144
fseek()	145
ftell()	145
rewind()	146
Reading a text file and converting all characters after the period into uppercase	146
How to do it...	146
How it works...	150
Displaying the contents of a random file in reverse order	151
How to do it...	152
How it works...	154
Counting the number of vowels in a file	156
How to do it...	156
How it works...	159
Replacing a word in a file with another word	161
How to do it...	161
How it works...	164
Encrypting a file	166
How to do it...	166
How it works...	169

Section 3: Concurrency, Networking, and Databases

Chapter 6: Implementing Concurrency	173
What are processes and threads?	173
Mutual exclusion	174
Performing a task with a single thread	176
How to do it...	176
How it works...	178
Performing multiple tasks with multiple threads	179
How to do it...	179
How it works...	181
Using mutex to share data between two threads	182
How to do it...	183
How it works...	186

Understanding how a deadlock is created	188
How to do it...	188
How it works...	192
Avoiding a deadlock	194
How to do it...	194
How it works...	198
Chapter 7: Networking and Interprocess Communication	200
 Communicating between processes using pipes	201
Creating and to connecting processes	201
pipe()	201
mkfifo()	202
write()	202
read()	202
perror()	203
fork()	203
One process, both writing and reading from the pipe	203
How to do it...	204
How it works...	205
One process writing into the pipe and another process reading from the pipe	206
How to do it...	206
How it works...	207
 Communicating between processes using FIFO	208
Writing data into a FIFO	208
How to do it...	208
Reading data from a FIFO	210
How to do it...	210
How it works...	211
 Communicating between the client and server using socket programming	211
Client-server model	212
struct sockaddr_in structure	212
socket()	213
memset()	214
htonl()	214
bind()	215
listen()	215
accept()	215
send()	216
connect()	216
recv()	216
Sending data to the client	217
How to do it...	217
How it works...	218
Reading data that's been sent from the server	219
How to do it...	219
How it works...	220

Communicating between processes using a UDP socket	221
Using a UDP socket for server-client communication	221
bzero()	221
INADDR_ANY	222
sendto()	222
recvfrom()	223
Await a message from the client and sending a reply using a UDP socket	223
How to do it...	223
How it works...	225
Sending a message to the server and receiving the reply from the server using the UDP socket	226
How to do it...	226
How it works...	227
Running Cygserver	229
Passing a message from one process to another using the message queue	230
Functions used in IPC using shared memory and message queues	230
ftok()	230
shmget()	230
shmat()	231
shmdt()	232
shmctl()	232
msgget()	232
msgrcv()	233
msgsnd()	234
Writing a message into the message queue	235
How to do it...	235
How it works...	236
Reading a message from the message queue	237
How to do it...	237
How it works...	238
Communicating between processes using shared memory	239
Writing a message into shared memory	239
How to do it...	240
How it works...	241
Reading a message from shared memory	241
How to do it...	241
How it works...	242
Chapter 8: Using MySQL Database Functions in MySQL	244
mysql_init()	244
mysql_real_connect()	245
mysql_query()	245
mysql_use_result()	246
mysql_fetch_row()	246
mysql_num_fields()	247
mysql_free_result()	247

mysql_close()	247
Creating a MySQL database and tables	247
Create database	248
Create table	249
Displaying all the built-in tables in a default mysql database	250
How to do it...	250
How it works...	252
Storing information in MySQL database	254
How to do it...	254
How it works...	256
Searching for the desired information in the database	258
How to do it...	259
How it works...	261
Updating information in the database	264
How to do it...	264
How it works...	268
Deleting data from the database using C	272
How to do it...	272
How it works...	276
Appendix A	280
Appendix B	295
Appendix C	303
Other Books You May Enjoy	317
Index	320

Preface

In this book, we will explore all the important elements of C such as strings, arrays, one- and two-dimensional arrays, functions, pointers, file handling, threads, interprocess communication, and database handling. Through the cookbook approach, you will find solutions to different problems that you will usually come across while making applications. By the end of the book, you will have sufficient knowledge to use some low- as well as high-level features of the C language and the ability to apply them for making real-time applications.

Who this book is for

This book is meant for basic to intermediate programmers and developers who want to make complex and real-time applications in C. This book can be of great use to trainers, teachers, and software developers who get stuck while making applications with arrays, pointers, functions, structures, files, databases, and interprocess communication and wish to see a running examples to find the way out of the problem.

What this book covers

Chapter 1, *Working with Arrays*, explains how to do some complex but essential operations with arrays. You will learn how to insert an element in an array, multiply two matrices, find the common elements in two arrays, and also find the difference between two sets or arrays. Also, you will learn how to find the unique elements in an array, how to know whether a matrix is a sparse matrix or not, and how to merge two sorted arrays into one array.

Chapter 2, *Managing Strings*, covers manipulating strings at the character level. You will learn how to work out whether the given string is a palindrome or not, how to find the first repetitive character in a string, and how to count each character in a string. You will also learn how to count the number of vowels and consonants in a string and the procedure of converting the vowels in a sentence into uppercase.

Chapter 3, *Exploring Functions*, covers functions, which play a major role in breaking down a big application into small, independent, manageable modules. In this chapter, you will learn how to make a function that finds whether the supplied argument is an Armstrong number. You will also learn how a function returns an array, and we make a function that finds the **greatest common divisor (GCD)** of two numbers using recursion. You will also learn how to make a function that converts a binary number into hexadecimal and how to make a function that determines whether the supplied number is a palindrome or not.

Chapter 4, *Deep Dive into Pointers*, explains how to use pointers to access content from specific memory locations. You will learn how to reverse a string using pointers, how to find the largest value in an array using pointers, and how to sort a singly linked list. Besides this, the chapter also explains how to find the transpose of a matrix and how to access a structure using pointers.

Chapter 5, *File Handling*, explains that file handling is very important for storing data for later use. In this chapter, you will learn how to read a text file and convert all the characters after full stops into uppercase. You will also learn how to display the content of a random file in reverse order and how to count the number of vowels in a file. This chapter will also explain how to replace a word in a file with another word and how to keep your file secure from unauthorized access. You will also learn how a file is encrypted.

Chapter 6, *Implementing Concurrency*, covers concurrency, which is implemented to increase the efficiency of the CPU. In this chapter, you will learn how to do a task using a single thread. Also, you will learn how to do multiple tasks with multiple threads and the technique of sharing data on two threads using a `mutex`. Besides this, you will learn how to recognize situations in which a deadlock can arise and how it can be avoided.

Chapter 7, *Networking and Interprocess Communication*, focuses on explaining how to establish communication between processes. You will learn how to communicate between processes using pipes, establish communication between processes using FIFO, and how communication is established between the client and server using socket programming. You will also learn how to do interprocess communication using the UDP socket, how a message is passed from one process to another using the message queue, and how two processes communicate using shared memory.

Chapter 8, *Using MySQL Database*, explains that no real-time application is possible without storing information in a database. The information in a database needs to be managed. In this chapter, you will learn how to display all the built-in tables in a default MySQL database. You will learn how to store information in a MySQL database and how to search for information in database tables. You will also learn how to update information in database tables and how to delete data from the database when it's no longer required.

Appendix A, explains how to create sequential and random files step by step. Most of the recipes in Chapter 5, *File Handling*, are about reading content from a file, and those recipes assume that the file already exists. This chapter explains how to create a sequential file and enter some text in it. You will also learn how to read content from a sequential file. Besides this, you will learn how to create a random file and enter some content in it, and how to read content from a random file and display it on the screen. Finally, you will also learn how to decrypt the content of an encrypted file.

Appendix B, explains how to install Cygwin.

Appendix C, explains how to install MySQL Community Server.

To get the most out of this book

You must have some preliminary knowledge of C programming. You will find it beneficial to have some prior basic knowledge of arrays, strings, functions, file handling, threads, and interprocess communication.

In addition, you must have some knowledge of basic SQL commands to handle databases.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/C-Programming-Cookbook>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/9781789617450_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, path names, dummy URLs, user input, and Twitter handles. Here is an example: "In the figure, 1000 represents the memory address of the *i* variable."

A block of code is set as follows:

```
for (i=0; i<2; i++)
{
    for (j=0; j<4; j++)
    {
        matR[i][j]=0;
        for (k=0; k<3; k++)
        {
            matR[i][j]=matR[i][j]+matA[i][k]*matB[k][j];
        }
    }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
printf("How many elements are there? ");
scanf("%d", &n);
```

Any command-line input or output is written as follows:

```
D:\CBook>reversestring
Enter a string: manish
Reverse string is hsinam
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Simply click the **Next** button to continue."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Sections

In this book, you will find several headings that appear frequently (*How to do it* and *How it works*).

To receive clear instructions on how to complete a recipe, use these sections as follows:

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section consists of a detailed explanation of the steps followed in the previous section.

There's more...

This section, when present, consists of additional information about the recipe in order to enhance your knowledge about the recipe.

See also

This section, when present, provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, select your book, click on the Errata Submission Form link, and enter the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1

Section 1: Arrays, Strings, and Functions

In this section, we will be covering various recipes that will teach you how to work with text and numbers, in arrays and strings. We will also learn how to create various functions to serve our needs.

The following chapters will be covered in this section:

- Chapter 1, *Working with Arrays*
- Chapter 2, *Managing Strings*
- Chapter 3, *Exploring Functions*

1

Working with Arrays

Arrays are an important construct of any programming language. To keep data of a similar type together, we need arrays. Arrays are heavily used in applications where elements have to be accessed at random. Arrays are also a prime choice when you need to sort elements, look for desired data in a collection, and find common or unique data between two sets.

Arrays are assigned contiguous memory locations and are a very popular structure for sorting and searching data collections because any element of an array can be accessed by simply specifying its subscript or index location. This chapter will cover recipes that include operations commonly applied to arrays.

In this chapter, we will learn how to make the following recipes using arrays:

- Inserting an element into a one-dimensional array
- Multiplying two matrices
- Finding the common elements in two arrays
- Finding the difference between two sets or arrays
- Finding the unique elements in an array
- Finding whether a matrix is sparse
- Merging two sorted arrays into one

Let's begin with the first recipe!

Inserting an element in an array

In this recipe, we will learn how to insert an element in-between an array. You can define the length of the array and also specify the location where you want the new value to be inserted. The program will display the array after the value has been inserted.

How to do it...

1. Let's assume that there is an array, **p**, with five elements, as follows:

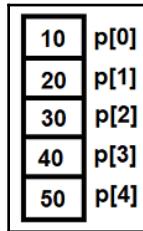


Figure 1.1

Now, suppose you want to enter a value, say **99**, at the third position. We will write a C program that will give the following output:

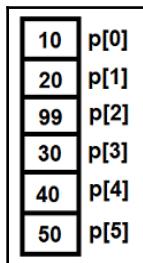


Figure 1.2

Here are the steps to follow to insert an element in an array:

1. Define a macro called **max** and initialize it to a value of **100**:

```
#define max 100
```

2. Define an array **p** of size **max** elements:

```
int p[max]
```

3. Enter the length of the array when prompted. The length you enter will be assigned to a variable **n**:

```
printf("Enter length of array:");
scanf("%d", &n);
```

4. A for loop will be executed prompting you to enter the elements of the array:

```
for(i=0;i<=n-1;i++)
    scanf("%d",&p[i]);
```

5. Specify the position in the array where the new value has to be inserted:

```
printf("\nEnter position where to insert:");
scanf("%d",&k);
```

6. Because the arrays in C are zero-based, the position you enter is decremented by 1:

```
k--;
```

7. To create space for the new element at the specified index location, all the elements are shifted one position down:

```
for(j=n-1;j>=k;j--)
    p[j+1]=p[j];
```

8. Enter the new value which will be inserted at the vacated index location:

```
printf("\nEnter the value to insert:");
scanf("%d",&p[k]);
```

Here is the `insertintoarray.c` program for inserting an element in between an array:

```
#include<stdio.h>
#define max 100
void main()
{
    int p[max], n,i,k,j;
    printf("Enter length of array:");
    scanf("%d",&n);
    printf("Enter %d elements of array\n",n);
    for(i=0;i<=n-1;i++)
        scanf("%d",&p[i]);
    printf("\nThe array is:\n");
    for(i = 0;i<=n-1;i++)
        printf("%d\n",p[i]);
    printf("\nEnter position where to insert:");
    scanf("%d",&k);
    k--;/*The position is always one value higher than the subscript,
so it is decremented by one*/
    for(j=n-1;j>=k;j--)
        p[j+1]=p[j];
```

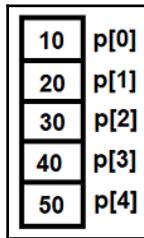
```
/* Shifting all the elements of the array one position down from
the location of insertion */
printf("\nEnter the value to insert:");
scanf("%d",&p[k]);
printf("\nArray after insertion of element: \n");
for(i=0;i<=n;i++)
    printf("%d\n",p[i]);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

Because we want to specify the length of the array, we will first define a macro called `max` and initialize it to a value of 100. I have defined the value of `max` as 100 because I assume that I will not need to enter more than 100 values in an array, but it can be any value as desired. An array, `p`, is defined of size `max` elements. You will be prompted to specify the length of the array. Let's specify the length of the array as 5. We will assign the value 5 to the variable `n`. Using a `for` loop, you will be asked to enter the elements of the array.

Let's say you enter the values in the array, as shown in *Figure 1.1* given earlier:



In the preceding diagram, the numbers, 0, 1, 2, and so on are known as index or subscript and are used for assigning and retrieving values from an array. Next, you will be asked to specify the position in the array where the new value has to be inserted. Suppose, you enter 3, which is assigned to the variable `k`. This means that you want to insert a new value at location 3 in the array.

Because the arrays in C are zero-based, position 3 means that you want to insert a new value at index location 2, which is `p[2]`. Hence, the position entered in `k` is decremented by 1.

To create space for the new element at index location **p[2]**, all the elements are shifted one position down. This means that the element at **p[4]** is moved to index location **p[5]**, the one at **p[3]** is moved to **p[4]**, and the element at **p[2]** is moved to **p[3]**, as follows:

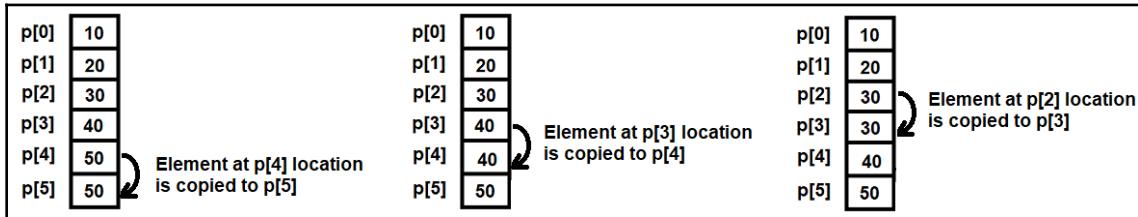
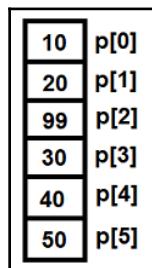


Figure 1.3

Once the element from the target index location is safely copied to the next location, you will be asked to enter the new value. Suppose you enter the new value as 99; that value will be inserted at index location **p[2]**, as shown in *Figure 1.2*, given earlier:



Let's use GCC to compile the `insertintoarray.c` program, as shown in this statement:

```
D:\CBook>gcc insertintoarray.c -o insertintoarray
```

Now, let's run the generated executable file, `insertintoarray.exe`, to see the program output:

```
D:\CBook>./insertintoarray
Enter length of array:5
Enter 5 elements of array
10
20
30
40
50
```

The array is:

```
10
20
30
40
50

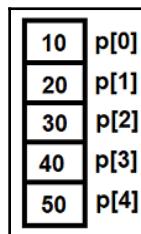
Enter target position to insert:3
Enter the value to insert:99
Array after insertion of element:
10
20
99
30
40
50
```

Voilà! We've successfully inserted an element in an array.

There's more...

What if we want to delete an element from an array? The procedure is simply the reverse; in other words, all the elements from the bottom of the array will be copied one place up to replace the element that was deleted.

Let's assume array **p** has the following five elements (*Figure 1.1*):



Suppose, we want to delete the third element, in other words, the one at **p[2]**, from this array. To do so, the element at **p[3]** will be copied to **p[2]**, the element at **p[4]** will be copied to **p[3]**, and the last element, which here is at **p[4]**, will stay as it is:

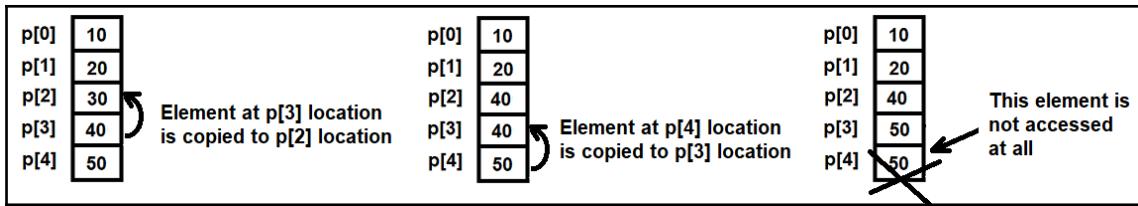


Figure 1.4

The `deletefromarray.c` program for deleting the array is as follows:

```
#include<stdio.h>
void main()
{
    int p[100],i,n,a;
    printf("Enter the length of the array: ");
    scanf("%d",&n);
    printf("Enter %d elements of the array \n",n);
    for(i=0;i<=n-1;i++)
        scanf("%d",&p[i]);
    printf("\nThe array is:\n");
    for(i=0;i<=n-1;i++)
        printf("%d\n",p[i]);
    printf("Enter the position/location to delete: ");
    scanf("%d",&a);
    a--;
    for(i=a;i<=n-2;i++)
    {
        p[i]=p[i+1];
        /* All values from the bottom of the array are shifted up till
         * the location of the element to be deleted */
    }
    p[n-1]=0;
    /* The vacant position created at the bottom of the array is set to
     * 0 */
    printf("Array after deleting the element is\n");
    for(i=0;i<= n-2;i++)
        printf("%d\n",p[i]);
}
```

Now, let's move on to the next recipe!

Multiplying two matrices

A prerequisite for multiplying two matrices is that the number of columns in the first matrix must be equal to the number of rows in the second matrix.

How to do it...

1. Create two matrices of orders 2×3 and 3×4 each.
2. Before we make the matrix multiplication program, we need to understand how matrix multiplication is performed manually. To do so, let's assume that the two matrices to be multiplied have the following elements:

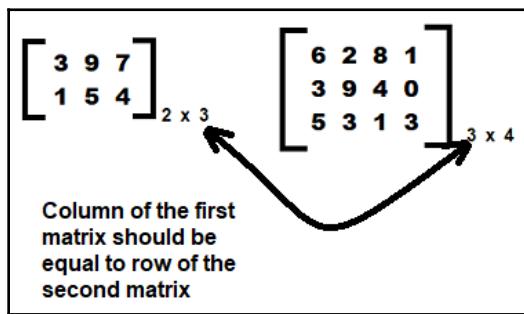


Figure 1.5

3. The resultant matrix will be of the order 2×4 , that is, the resultant matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix:

$$\begin{bmatrix} 3 & 9 & 7 \\ 1 & 5 & 4 \end{bmatrix}_{2 \times 3} \times \begin{bmatrix} 6 & 2 & 8 & 1 \\ 3 & 9 & 4 & 0 \\ 5 & 3 & 1 & 3 \end{bmatrix}_{3 \times 4} = \begin{bmatrix} \dots & \dots & \dots & \dots \end{bmatrix}_{2 \times 4}$$

Figure 1.6

Essentially, the resultant matrix of the order 2×4 will have the following elements:

first row, first column	first row, second column	first row, third column	first row, fourth column
second row, first column	second row, second column	second row, third column	second row, fourth column

2 x 4

Figure 1.7

4. The element **first row, first column** in the resultant matrix is computed using the following formula:

SUM(first element of the first row of the first matrix × first element of the first column of the second matrix), (second element of the first row... × second element of the first column...), (and so on...)

For example, let's assume the elements of the two matrices are as shown in *Figure 1.5*. The elements in the first row and the first column of the resultant matrix will be computed as follows:

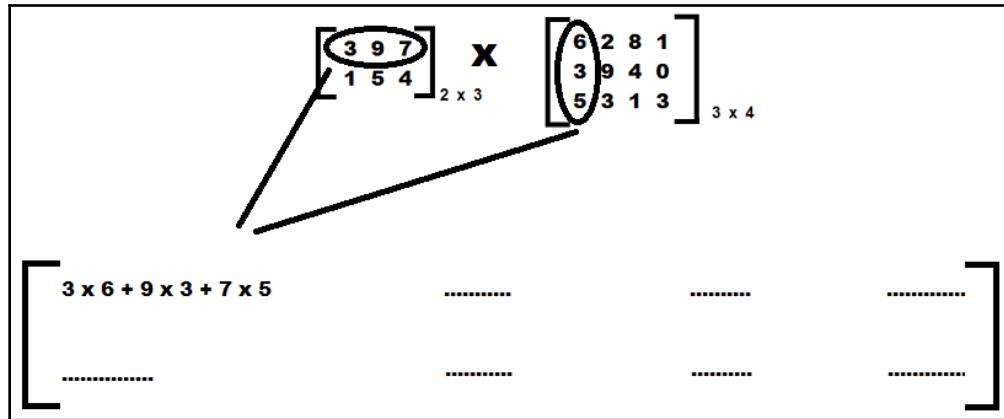


Figure 1.8

5. Hence, the element in **first row, first column** in the resultant matrix will be as follows:

$$\begin{aligned}
 &(3 \times 6) + (9 \times 3) + (7 \times 5) \\
 &= 18 + 27 + 35 \\
 &= 80
 \end{aligned}$$

Figure 1.9 explains how the rest of the elements are computed in the resultant matrix:

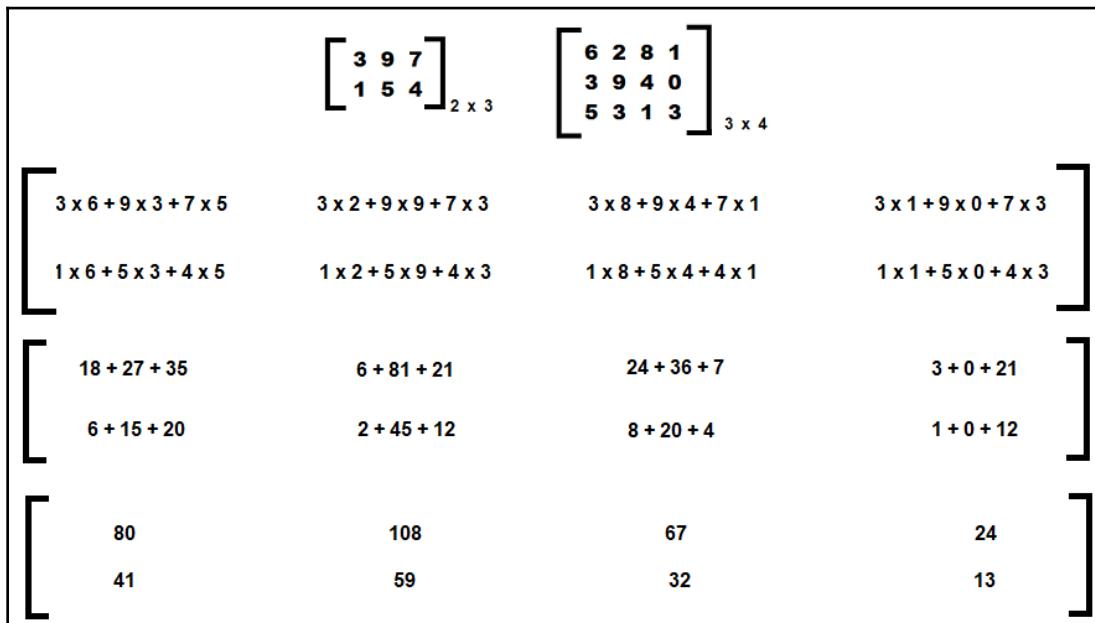


Figure 1.9

The `matrixmulti.c` program for multiplying the two matrices is as follows:

```
#include <stdio.h>
int main()
{
    int matA[2][3], matB[3][4], matR[2][4];
    int i, j, k;
    printf("Enter elements of the first matrix of order 2 x 3 \n");
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        {
            scanf("%d",&matA[i][j]);
        }
    }
    printf("Enter elements of the second matrix of order 3 x 4 \n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {

```

```
    scanf("%d", &matB[i][j]);
}
}
for(i=0;i<2;i++)
{
    for(j=0;j<4;j++)
    {
        matR[i][j]=0;
        for(k=0;k<3;k++)
        {
            matR[i][j]=matR[i][j]+matA[i][k]*matB[k][j];
        }
    }
}
printf("\nFirst Matrix is \n");
for(i=0;i<2;i++)
{
    for(j=0;j<3;j++)
    {
        printf("%d\t", matA[i][j]);
    }
    printf("\n");
}
printf("\nSecond Matrix is \n");
for(i=0;i<3;i++)
{
    for(j=0;j<4;j++)
    {
        printf("%d\t", matB[i][j]);
    }
    printf("\n");
}
printf("\nMatrix multiplication is \n");
for(i=0;i<2;i++)
{
    for(j=0;j<4;j++)
    {
        printf("%d\t", matR[i][j]);
    }
    printf("\n");
}
return 0;
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

The two matrices are defined `matA` and `matB` of the orders 2×3 and 3×4 , respectively, using the following statement:

```
int matA[2][3], matB[3][4]
```

You will be asked to enter the elements of the two matrices using the nested `for` loops. The elements in the matrix are entered in row-major order, in other words, all the elements of the first row are entered first, followed by all the elements of the second row, and so on.

In the nested loops, `for i` and `for j`, the outer loop, `for i`, represents the row and the inner loop, and `for j` represents the column.

While entering the elements of matrices `matA` and `matB`, the values entered in the two matrices will be assigned to the respective index locations of the two-dimensional arrays as follows:

The diagram illustrates the assignment of values to two matrices, `matA` and `matB`, using row-major traversal. The top part shows matrix `matA` (2 rows, 3 columns) and the bottom part shows matrix `matB` (3 rows, 4 columns). The assignment process is shown with boxes around specific cells.

		columns		
		0	1	2
rows	0	3 matA[0][0]	9 matA[0][1]	7 matA[0][2]
	1	1 matA[1][0]	5 matA[1][1]	4 matA[1][2]

		columns			
		0	1	2	3
rows	0	6 matB[0][0]	2 matB[0][1]	8 matB[0][2]	1 matB[0][3]
	1	3 matB[1][0]	9 matB[1][1]	4 matB[1][2]	0 matB[1][3]
	2	5 matB[2][0]	3 matB[2][1]	1 matB[2][2]	3 matB[2][3]

Figure 1.10

The nested loops that actually compute the matrix multiplication are as follows:

```
for(i=0;i<2;i++)
{
    for(j=0;j<4;j++)
    {
        matR[i][j]=0;
        for(k=0;k<3;k++)
        {
            matR[i][j]=matR[i][j]+matA[i][k]*matB[k][j];
        }
    }
}
```

The variable `i` represents the row of the resultant matrix, `j` represents the column of the resultant matrix, and `k` represents the common factor. The *common factor* here means the column of the first matrix and the row of the second matrix.

Recall that the prerequisite for matrix multiplication is that the column of the first matrix should have the same number of rows as the second matrix. Because the respective elements have to be added after multiplication, the element has to be initialized to 0 before addition.

The following statement initializes the elements of the resultant matrix:

```
matR[i][j]=0;
```

The `for k` loop inside the nested loops helps in selecting the elements in the rows of the first matrix and multiplying them by elements of the column of the second matrix:

```
matR[i][j]=matR[i][j]+matA[i][k]*matB[k][j];
```

Let's use GCC to compile the `matrixmulti.c` program as follows:

```
D:\CBook>gcc matrixmulti.c -o matrixmulti
```

Let's run the generated executable file, `matrixmulti.exe`, to see the output of the program:

```
D:\CBook\Chapters\1Arrays>./matrixmulti
```

```
Enter elements of the first matrix of order 2 x 3
3
9
7
1
5
```

4

```
Enter elements of the second matrix of order 3 x 4
6 2 8 1
3 9 4 0
5 3 1 3

First Matrix is
3 9 7
1 5 4

Second Matrix is
6 2 8 1
3 9 4 0
5 3 1 3

Matrix multiplication is
80 108 67 24
41 59 32 13
```

Voilà! We've successfully multiplied two matrices.

There's more...

One thing that you might notice while entering the elements of the matrix is that there are two ways of doing it.

1. The first method is that you press *Enter* after inputting each element as follows:

```
3
9
7
1
5
4
```

The values will be automatically assigned to the matrix in row-major order, in other words, 3 will be assigned to `matA[0][0]`, 9 will be assigned to `matA[0][1]`, and so on.

2. The second method of entering elements in the matrix is as follows:

```
6 2 8 1  
3 9 4 0  
5 3 1 3
```

Here, 6 will be assigned to `matB[0][0]`, 2 will be assigned to `matB[0][1]`, and so on.

Now, let's move on to the next recipe!

Finding the common elements in two arrays

Finding the common elements in two arrays is akin to finding the intersection of two sets. Let's learn how to do it.

How to do it...

1. Define two arrays of a certain size and assign elements of your choice to both the arrays. Let's assume that we created two arrays called **p** and **q**, both of size four elements:



Figure 1.11

2. Define one more array. Let's call it array **r**, to be used for storing the elements that are common between the two arrays.
3. If an element in array **p** exists in the array **q**, it is added to array **r**. For instance, if the element at the first location in array **p**, which is at **p[0]**, does not appear in array **q**, it is discarded, and the next element, at **p[1]**, is picked up for comparison.

4. And if the element at $p[0]$ is found anywhere in array q , it is added to array r , as follows:

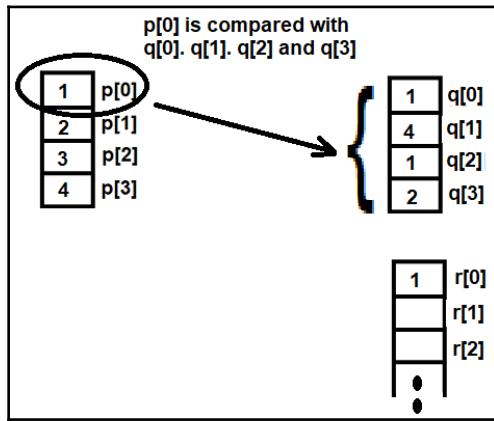


Figure 1.12

5. This procedure is repeated with other elements of array q . That is, $p[1]$ is compared with $q[0]$, $q[1]$, $q[2]$, and $q[3]$. If $p[1]$ is not found in array q , then before inserting it straightaway into array r , it is compared with the existing elements of array r to avoid repetitive elements.
6. Because the element at $p[1]$ appears in array q and is not already present in array r , it is added to array r as follows:

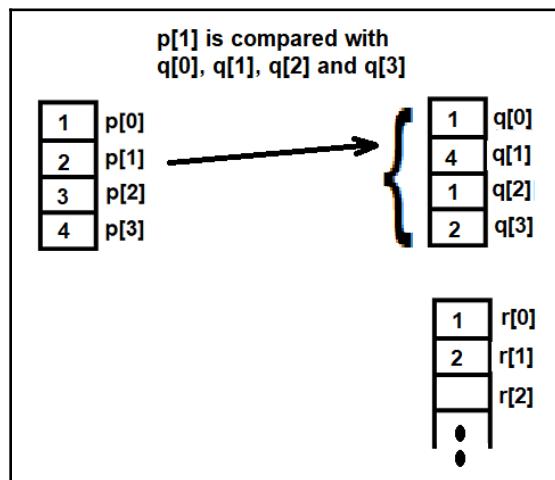


Figure 1.13

The commoninarray.c program for establishing common elements among the two arrays is as follows:

```
#include<stdio.h>
#define max 100

int ifexists(int z[], int u, int v)
{
    int i;
    if (u==0) return 0;
    for (i=0; i<=u;i++)
        if (z[i]==v) return (1);
    return (0);
}
void main()
{
    int p[max], q[max], r[max];
    int m,n;
    int i,j,k;
    k=0;
    printf("Enter the length of the first array:");
    scanf("%d",&m);
    printf("Enter %d elements of the first array\n",m);
    for(i=0;i<m;i++)
        scanf("%d",&p[i]);
    printf("\nEnter the length of the second array:");
    scanf("%d",&n);
    printf("Enter %d elements of the second array\n",n);
    for(i=0;i<n;i++)
        scanf("%d",&q[i]);
    k=0;
    for (i=0;i<m;i++)
    {
        for (j=0;j<n;j++)
        {
            if (p[i]==q[j])
            {
                if(!ifexists(r,k,p[i]))
                {
                    r[k]=p[i];
                    k++;
                }
            }
        }
    }
    if(k>0)
    {
        printf("\nThe common elements in the two arrays are:\n");
    }
}
```

```
        for(i = 0;i<k;i++)
            printf("%d\n",r[i]);
    }
else
    printf("There are no common elements in the two arrays\n");
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

A macro, `max`, is defined of size 100. A function, `ifexists()`, is defined that simply returns `true` (1) or `false` (0). The function returns `true` if the supplied value exists in the specified array, and `false` if it doesn't.

Two arrays are defined, called `p` and `q`, of size `max` (in other words, 100 elements). You will be prompted to specify the length of the array, `p`, and then asked to enter the elements in that array. After that, you will be asked to specify the length of array `q`, followed by entering the elements in array `q`.

Thereafter, `p[0]`, the first element in array `p`, is picked up, and by using the `for` loop, `p[0]` is compared with all the elements of array `q`. If `p[0]` is found in array `q`, then `p[0]` is added to the resulting array, `r`.

After a comparison of `p[0]`, the second element in array `p`, `p[1]`, is picked up and compared with all the elements of array `q`. The procedure is repeated until all the elements of array `p` are compared with all the elements of array `q`.

If any elements of array `p` are found in array `q`, then before adding that element to the resulting array, `r`, it is run through the `ifexists()` function to ensure that the element does not already exist in array `r`. This is because we don't want repetitive elements in array `r`.

Finally, all the elements in array `r`, which are the common elements of the two arrays, are displayed on the screen.

Let's use GCC to compile the `commoninarray.c` program as follows:

```
D:\CBook>gcc commoninarray.c -o commoninarray
```

Now, let's run the generated executable file, `commoninarray.exe`, to see the output of the program:

```
D:\CBook>./commoninarray
Enter the length of the first array:5
Enter 5 elements in the first array
1
2
3
4
5

Enter the length of the second array:4
Enter 4 elements in the second array
7
8
9
0

There are no common elements in the two arrays
```

Because there were no common elements between the two arrays entered previously, we can't quite say that we've truly tested the program. Let's run the program again, and this time, we will enter the array elements such that they have something in common.

```
D:\CBook>./commoninarray
Enter the length of the first array:4
Enter 4 elements in the first array
1
2
3
4

Enter the length of the second array:4
Enter 4 elements in the second array
1
4
1
2

The common elements in the two arrays are:
1
2
4
```

Voilà! We've successfully identified the common elements between two arrays.

Finding the difference between two sets or arrays

When we talk about the difference between two sets or arrays, we are referring to all the elements of the first array that don't appear in the second array. In essence, all the elements in the first array that are not common to the second array are referred to as the difference between the two sets. The difference in sets p and q , for example, will be denoted by $p - q$.

If array p , for example, has the elements $\{1, 2, 3, 4\}$, and array q has the elements $\{2, 4, 5, 6\}$, then the difference between the two arrays, $p - q$, will be $\{1, 3\}$. Let's find out how this is done.

How to do it...

1. Define two arrays, say p and q , of a certain size and assign elements of your choice to both the arrays.
2. Define one more array, say r , to be used for storing the elements that represent the difference between the two arrays.
3. Pick one element from array p and compare it with all the elements of the array q .
4. If the element of array p exists in array q , discard that element and pick up the next element of array p and repeat from step 3.
5. If the element of array p does not exist in array q , add that element in array r . Before adding that element to array r , ensure that it does not already exist in array r .
6. Repeat steps 3 to 5 until all the elements of array p are compared.
7. Display all the elements in array r , as these are the elements that represent the difference between arrays p and q .

The `differencearray.c` program to establish the difference between two arrays is as follows:

```
#include<stdio.h>
#define max 100

int ifexists(int z[], int u, int v)
{
    int i;
    if (u==0) return 0;
```

```
        for (i=0; i<=u;i++)
            if (z[i]==v) return (1);
        return (0);
    }

void main()
{
    int p[max], q[max], r[max];
    int m,n;
    int i,j,k;
    printf("Enter length of first array:");
    scanf("%d",&m);
    printf("Enter %d elements of first array\n",m);
    for(i=0;i<m;i++)
        scanf("%d",&p[i]);
    printf("\nEnter length of second array:");
    scanf("%d",&n);
    printf("Enter %d elements of second array\n",n);
    for(i=0;i<n;i++)
        scanf("%d",&q[i]);
    k=0;
    for (i=0;i<m;i++)
    {
        for (j=0;j<n;j++)
        {
            if (p[i]==q[j])
            {
                break;
            }
        }
        if(j==n)
        {
            if(!ifexists(r,k,p[i]))
            {
                r[k]=p[i];
                k++;
            }
        }
    }
    printf("\nThe difference of the two array is:\n");
    for(i = 0;i<k;i++)
        printf("%d\n",r[i]);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We defined two arrays called **p** and **q**. We don't want to fix the length of these arrays, so we should define a macro called `max` of value 100 and set the two arrays, **p** and **q**, to the size of `max`.

Thereafter, you will be prompted to specify the size of the first array and enter the elements in the first array, **p**. Similarly, you will be asked to specify the length of the second array, **q**, followed by entering the elements in the second array.

Let's assume you have specified the length of both arrays as 4 and have entered the following elements:

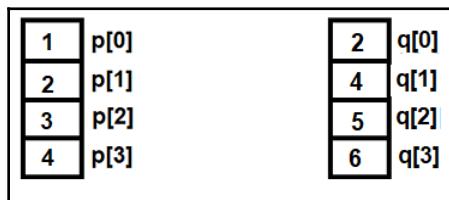


Figure 1.14

We need to pick up one element at a time from the first array and compare it with all the elements of the second array. If an element in array **p** does not appear in array **q**, it will be assigned to the third array we created, array **r**.

Array **r** will be used for storing the elements that define the difference between two arrays. As shown in *Figure 1.15*, the first element of array **p**, in other words, at **p[0]**, is compared with all the elements of array **q**, in other words, with **q[0]**, **q[1]**, **q[2]**, and **q[3]**.

Because the element at **p[0]**, which is **1**, does not appear in array **q**, it will be added to the array **r**, indicating the first element representing the difference between the two arrays:

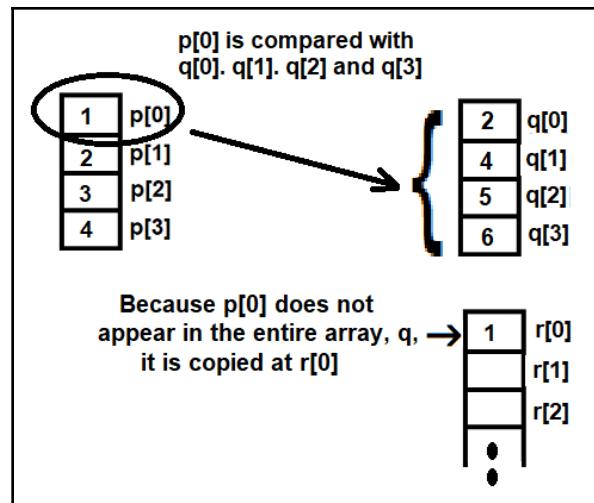


Figure 1.15

Because the element at **p[1]**, which is **2**, appears in array **q**, it is discarded, and the next element in array **p**, in other words, **p[2]**, is picked up and compared with all the elements in array **q**.

As the element at **p[2]** does not appear in array **q**, it is added to array **r** at the next available location, which is **r[1]** (see *Figure 1.16* as follows):

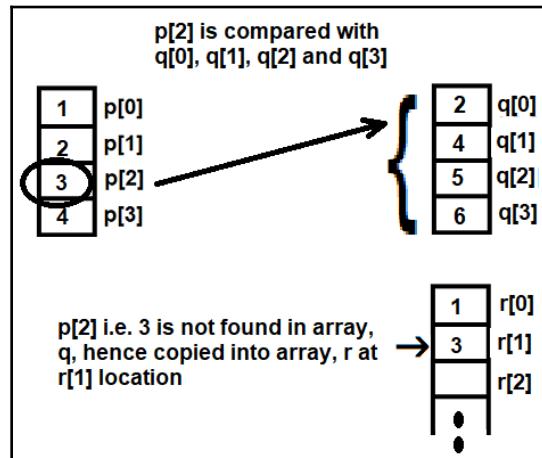


Figure 1.16

Continue the procedure until all the elements of array **p** are compared with all the elements of array **q**. Finally, we will have array **r**, with the elements showing the difference between our two arrays, **p** and **q**.

Let's use GCC to compile our program, `differencearray.c`, as follows:

```
D:\CBook>gcc differencearray.c -o differencearray
```

Now, let's run the generated executable file, `differencearray`, to see the output of the program:

```
D:\CBook>./differencearray
Enter length of first array:4
Enter 4 elements of first array
1
2
3
4
Enter length of second array:4
Enter 4 elements of second array
2
4
5
6
The difference of the two array is:
1
3
```

Voilà! We've successfully found the difference between two arrays. Now, let's move on to the next recipe!

Finding the unique elements in an array

In this recipe, we will learn how to find the unique elements in an array, such that the repetitive elements in the array will be displayed only once.

How to do it...

1. Define two arrays, **p** and **q**, of a certain size and assign elements only to array **p**. We will leave array **q** blank.
2. These will be our source and target arrays, respectively. The target array will contain the resulting unique elements of the source array.

3. After that, each of the elements in the source array will be compared with the existing elements in the target array.
4. If the element in the source array exists in the target array, then that element is discarded and the next element in the source array is picked up for comparison.
5. If the source array element does not exist in the target array, it is copied into the target array.
6. Let's assume that array **p** contains the following repetitive elements:

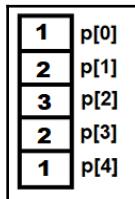


Figure 1.17

7. We will start by copying the first element of the source array, **p**, into the target array, **q**, in other words, **p[0]** into array **q[0]**, as follows:

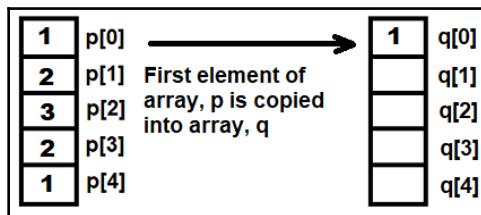


Figure 1.18

8. Next, the second array element of **p**, in other words, **p[1]**, is compared with all the existing elements of array **q**. That is, **p[1]** is compared with array **q** to check whether it already exists in array **q**, as follows:

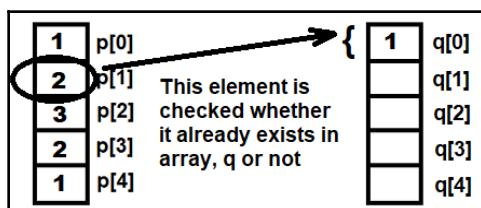


Figure 1.19

9. Because **p[1]** does not exist in array **q**, it is copied at **q[1]**, as shown in *Figure 1.20*:

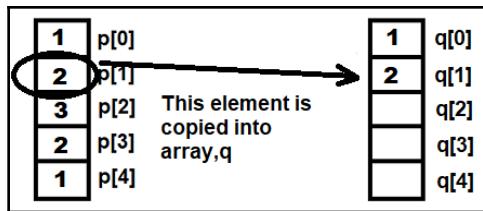


Figure 1.20

10. This procedure is repeated until all the elements of array **p** are compared with array **q**. In the end, we will have array **q**, which will contain the unique elements of array **p**.

Here is the `uniqueelements.c` program for finding the unique elements in the first array:

```
#include<stdio.h>
#define max 100

int ifexists(int z[], int u, int v)
{
    int i;
    for (i=0; i<u; i++)
        if (z[i]==v) return (1);
    return (0);
}

void main()
{
    int p[max], q[max];
    int m;
    int i,k;
    k=0;
    printf("Enter length of the array:");
    scanf("%d",&m);
    printf("Enter %d elements of the array\n",m);
    for(i=0;i<m;i++)
        scanf("%d",&p[i]);
    q[0]=p[0];
    k=1;
    for (i=1;i<m;i++)
    {
        if(!ifexists(q,k,p[i]))
        {
            q[k]=p[i];
            k++;
        }
    }
}
```

```
        k++;
    }
}
printf("\nThe unique elements in the array are:\n");
for(i = 0;i<k;i++)
    printf("%d\n",q[i]);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will define a macro called `max` of size 100. Two arrays, `p` and `q`, are defined of size `max`. Array `p` will contain the original elements, and array `q` will contain the unique elements of array `p`. You will be prompted to enter the length of the array and, thereafter, using the `for` loop, the elements of the array will be accepted and assigned to array `p`.

The following statement will assign the first element of array `p` to the first index location of our blank array, which we will name array `q`:

```
q[0]=p[0]
```

A `for` loop is again used to access the rest of the elements of array `p`, one by one. First, the foremost element of array `p`, which is at `p[0]`, is copied to array `q` at `q[0]`.

Next, the second array `p` element, `p[1]`, is compared with all the existing elements of array `q`. That is, `p[1]` is checked against array `q` to confirm whether it is already present there.

Because there is only a single element in array `q`, `p[1]` is compared with `q[0]`.

Because `p[1]` does not exist in array `q`, it is copied at `q[1]`.

This procedure is repeated for all elements in array `p`. Each of the accessed elements of array `p` is run through the `ifexists()` function to check whether any of them already exist in array `q`.

The function returns 1 if an element in array `p` already exists in array `q`. In that case, the element in array `p` is discarded and the next array element is picked up for comparison.

In case the `ifexists()` function returns 0, confirming that the element in array `p` does not exist in array `q`, the array `p` element is added to array `q` at the next available index/subscript location.

When all the elements of array p are checked and compared, array q will have only the unique elements of array p.

Let's use GCC to compile the `uniqueelements.c` program as follows:

```
D:\CBook>gcc uniqueelements.c -o uniqueelements
```

Now, let's run the generated executable file, `uniqueelements.exe`, to see the output of the program:

```
D:\CBook>./uniqueelements
Enter the length of the array:5
Enter 5 elements in the array
1
2
3
2
1

The unique elements in the array are:
1
2
3
```

Voilà! We've successfully identified the unique elements in an array. Now, let's move on to the next recipe!

Finding whether a matrix is sparse

A matrix is considered sparse when it has more zero values than non-zero values (and dense when it has more non-zero values). In this recipe, we will learn how to find out whether the specified matrix is sparse.

How to do it...

1. First, specify the order of the matrix. Then, you will be prompted to enter the elements in the matrix. Let's assume that you specified the order of the matrix as 4×4 . After entering the elements in the matrix, it might appear like this:

0	1	0	0
5	0	0	9
0	0	3	0
2	0	4	0

Figure 1.21

2. Once the elements of the matrix are entered, count the number of zeros in it. A counter for this purpose is initialized to **0**. Using nested loops, each of the matrix elements is scanned and, upon finding any zero elements, the value of the counter is incremented by 1.
3. Thereafter, the following formula is used for establishing whether the matrix is sparse.

If counter > [(the number of rows x the number of columns)/2] = Sparse Matrix

4. Depending on the result of the preceding formula, one of the following messages will be displayed on the screen as follows:

The given matrix is a sparse matrix

or

The given matrix is not a sparse matrix

The **sparsematrix.c** program for establishing whether the matrix is sparse is as follows:

```
#include <stdio.h>
#define max 100

/*A sparse matrix has more zero elements than nonzero elements */
void main ()
{
    static int arr[max][max];
    int i,j,r,c;
    int ctr=0;
    printf("How many rows and columns are in this matrix? ");
    scanf("%d %d", &r, &c);
    printf("Enter the elements in the matrix :\n");
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
```

```

    {
        scanf ("%d", &arr[i][j]);
        if (arr[i][j]==0)
            ++ctr;
    }
    if (ctr>((r*c)/2))
        printf ("The given matrix is a sparse matrix. \n");
    else
        printf ("The given matrix is not a sparse matrix.\n");
    printf ("There are %d number of zeros in the matrix.\n", ctr);
}

```

Now, let's go behind the scenes to understand the code better.

How it works...

Because we don't want to fix the size of the matrix, we will define a macro called `max` of value 100. A matrix, or a two-dimensional array called `arr`, is defined of the order `max x max`. You will be prompted to enter the order of the matrix, for which you can again enter any value up to 100.

Let's assume that you've specified the order of the matrix as 4×4 . You will be prompted to enter elements in the matrix. The values entered in the matrix will be in row-major order. After entering the elements, the matrix `arr` should look like *Figure 1.22*, as follows:

		0	1	2	3
		0	1	2	3
0	0	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]
	1	5 arr[1][0]	0 arr[1][1]	0 arr[1][2]	9 arr[1][3]
2	0	0 arr[2][0]	0 arr[2][1]	3 arr[2][2]	0 arr[2][3]
	3	2 arr[3][0]	0 arr[3][1]	4 arr[3][2]	0 arr[3][3]

Figure 1.22

A counter called `ctr` is created and is initialized to 0. Using nested loops, each element of matrix `arr` is checked and the value of `ctr` is incremented if any element is found to be 0. Thereafter, using the `if else` statement, we will check whether the count of zero values is more than non-zero values. If the count of zero values is more than non-zero values, then the message will be displayed on the screen as follows:

```
The given matrix is a sparse matrix
```

However, failing that, the message will be displayed on the screen as follows:

```
The given matrix is not a sparse matrix
```

Let's use GCC to compile the `sparsematrix.c` program as follows:

```
D:\CBook>gcc sparsematrix.c -o sparsematrix
```

Let's run the generated executable file, `sparsematrix.exe`, to see the output of the program:

```
D:\CBook>./sparsematrix
How many rows and columns are in this matrix? 4 4
Enter the elements in the matrix :
0 1 0 0
5 0 0 9
0 0 3 0
2 0 4 0
The given matrix is a sparse matrix.
There are 10 zeros in the matrix.
```

Okay. Let's run the program again to see the output when the count of non-zero values is higher:

```
D:\CBook>./sparsematrix
How many rows and columns are in this matrix? 4 4
Enter the elements in the matrix:
1 0 3 4
0 0 2 9
8 6 5 1
0 7 0 4
The given matrix is not a sparse matrix.
There are 5 zeros in the matrix.
```

Voilà! We've successfully identified a sparse and a non-sparse matrix.

There's more...

How about finding an identity matrix, in other words, finding out whether the matrix entered by the user is an identity matrix or not. Let me tell you—a matrix is said to be an identity matrix if it is a square matrix and all the elements of the principal diagonal are ones and all other elements are zeros. An identity matrix of the order 3×3 may appear as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad 3 \times 3$$

Figure 1.23

In the preceding diagram, you can see that the principal diagonal elements of the matrix are 1's and the rest of them are 0's. The index or subscript location of the principal diagonal elements will be `arr[0][0]`, `arr[1][1]`, and `arr[2][2]`, so the following procedure is followed to find out whether the matrix is an identity matrix or not:

- Checks that if the index location of the row and column is the same, in other words, if the row number is 0 and the column number, too, is 0, then at that index location, `[0][0]`, the matrix element must be 1. Similarly, if the row number is 1 and the column number, too, is 1, that is, at the `[1][1]` index location, the matrix element must be 1.
- Verify that the matrix element is 0 at all the other index locations.

If both the preceding conditions are met, then the matrix is an identity matrix, or else it is not.

The `identitymatrix.c` program to establish whether the entered matrix is an identity matrix or not is given as follows:

```
#include <stdio.h>
#define max 100
/* All the elements of the principal diagonal of the Identity matrix are
ones and rest all are zero elements */
void main ()
{
    static int arr[max][max];
    int i,j,r,c, bool;
    printf("How many rows and columns are in this matrix ? ");
    scanf("%d %d", &r, &c);
```

```
if (r !=c)
{
    printf("An identity matrix is a square matrix\n");
    printf("Because this matrix is not a square matrix, so it is not an
          identity matrix\n");
}
else
{
    printf("Enter elements in the matrix :\n");
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            scanf("%d",&arr[i][j]);
        }
    }
    printf("\nThe entered matrix is \n");
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            printf("%d\t",arr[i][j]);
        }
        printf("\n");
    }
    bool=1;
    for(i=0;i<r;i++)
    {
        for(j=0;j<c;j++)
        {
            if(i==j)
            {
                if(arr[i][j] !=1)
                {
                    bool=0;
                    break;
                }
            }
            else
            {
                if(arr[i][j] !=0)
                {
                    bool=0;
                    break;
                }
            }
        }
    }
}
```

```
        if(bool)
            printf("\nMatrix is an identity matrix\n");
        else
            printf("\nMatrix is not an identity matrix\n");
    }
}
```

Let's use GCC to compile the `identitymatrix.c` program as follows:

```
D:\CBook>gcc identitymatrix.c -o identitymatrix
```

No error is generated. This means the program is compiled perfectly and an executable file is generated. Let's run the generated executable file. First, we will enter a non-square matrix:

```
D:\CBook>./identitymatrix
How many rows and columns are in this matrix ? 3 4
An identity matrix is a square matrix
Because this matrix is not a square matrix, so it is not an identity matrix
```

Now, let's run the program again; this time, we will enter a square matrix

```
D:\CBook>./identitymatrix
How many rows and columns are in this matrix ? 3 3
Enter elements in the matrix :
1 0 1
1 1 0
0 0 1

The entered matrix is
1      0      1
1      1      0
0      0      1

Matrix is not an identity matrix
```

Because a non-diagonal element in the preceding matrix is 1, it is not an identity matrix.

Let's run the program again:

```
D:\CBook>./identitymatrix
How many rows and columns are in this matrix ? 3 3
Enter elements in the matrix :
1 0 0
0 1 0
0 0 1
The entered matrix is
1      0      0
0      1      0
```

```
0      0      1  
Matrix is an identity matrix
```

Now, let's move on to the next recipe!

Merging two sorted arrays into a single array

In this recipe, we will learn to merge two sorted arrays into a single array so that the resulting merged array is also in sorted form.

How to do it...

1. Let's assume there are two arrays, **p** and **q**, of a certain length. The length of the two arrays can differ. Both have some sorted elements in them, as shown in *Figure 1.24*:



Figure 1.24

2. The merged array that will be created from the sorted elements of the preceding two arrays will be called array **r**. Three subscripts or index locations will be used to point to the respective elements of the three arrays.
3. Subscript **i** will be used to point to the index location of array **p**. Subscript **j** will be used to point to the index location of array **q** and subscript **k** will be used to point to the index location of array **r**. In the beginning, all three subscripts will be initialized to 0.

4. The following three formulas will be applied to get the merged sorted array:
1. The element at $p[i]$ is compared with the element at $q[j]$. If $p[i]$ is less than $q[j]$, then $p[i]$ is assigned to array r , and the indices of arrays p and r are incremented so that the following element of array p is picked up for the next comparison as follows:

```
r[k]=p[i];
i++;
k++
```

2. If $q[j]$ is less than $p[i]$, then $q[j]$ is assigned to array r , and the indices of arrays q and r are incremented so that the following element of array q is picked up for the next comparison as follows:

```
r[k]=q[j];
i++;
k++
```

3. If $p[i]$ is equal to $q[j]$, then both the elements are assigned to array r . $p[i]$ is added to $r[k]$. The values of the i and k indices are incremented. $q[j]$ is also added to $r[k]$, and the indices of the q and r arrays are incremented. Refer to the following code snippet:

```
r[k]=p[i];
i++;
k++
r[k]=q[j];
i++;
k++
```

5. The procedure will be repeated until either of the arrays gets over. If any of the arrays is over, the remainder of the elements of the other array will be simply appended to the array r .

The `mergetwosortedarrays.c` program for merging two sorted arrays is as follows:

```
#include<stdio.h>
#define max 100

void main()
{
    int p[max], q[max], r[max];
    int m,n;
    int i,j,k;
    printf("Enter length of first array:");
    scanf("%d",&m);
```

```
printf("Enter %d elements of the first array in sorted order\n",m);
for(i=0;i<m;i++)
    scanf("%d",&p[i]);
printf("\nEnter length of second array:");
scanf("%d",&n);
printf("Enter %d elements of the second array in sorted
order\n",n);
for(i=0;i<n;i++)
    scanf("%d",&q[i]);
i=j=k=0;
while ((i<m) && (j <n))
{
    if(p[i] < q[j])
    {
        r[k]=p[i];
        i++;
        k++;
    }
    else
    {
        if(q[j]< p[i])
        {
            r[k]=q[j];
            k++;
            j++;
        }
        else
        {
            r[k]=p[i];
            k++;
            i++;
            r[k]=q[j];
            k++;
            j++;
        }
    }
}
while(i<m)
{
    r[k]=p[i];
    k++;
    i++;
}
while(j<n)
{
    r[k]=q[j];
    k++;
}
```

```

        j++;
    }
    printf("\nThe combined sorted array is:\n");
    for(i = 0; i < k; i++)
        printf("%d\n", r[i]);
}

```

Now, let's go behind the scenes to understand the code better.

How it works...

A macro called `max` is defined of size 100. Three arrays, `p`, `q`, and `r`, are defined of size `max`. You will first be asked to enter the size of the first array, `p`, followed by the sorted elements for array `p`. The process is repeated for the second array `q`.

Three indices, `i`, `j` and `k`, are defined and initialized to 0. The three indices will point to the elements of the three arrays, `p`, `q`, and `r`, respectively.

The first elements of arrays `p` and `q`, in other words, `p[0]` and `q[0]`, are compared and the smaller one is assigned to array `r`.

Because `q[0]` is smaller than `p[0]`, `q[0]` is added to array `r`, and the indices of arrays `q` and `r` are incremented for the next comparison as follows:

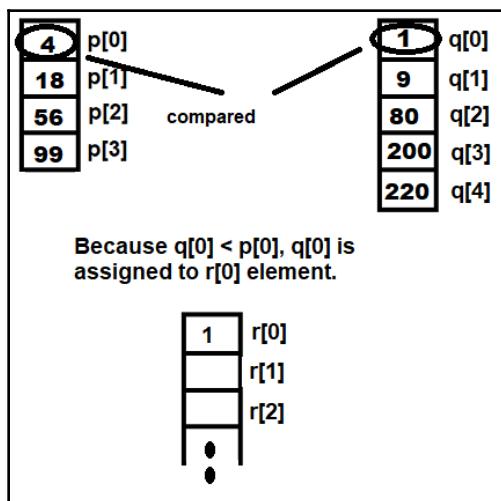


Figure 1.25

Next, $p[0]$ will be compared with $q[1]$. Because $p[0]$ is smaller than $q[1]$, the value at $p[0]$ will be assigned to array r at $r[1]$:

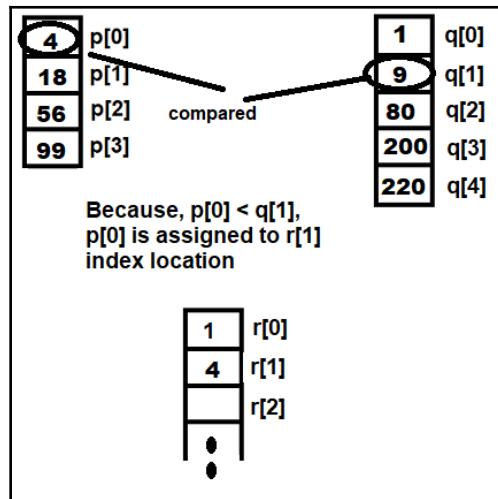


Figure 1.26

Then, $p[1]$ will be compared with $q[1]$. Because $q[1]$ is smaller than $p[1]$, $q[1]$ will be assigned to array r , and the indices of the q and r arrays will be incremented for the next comparisons (refer to the following diagram):

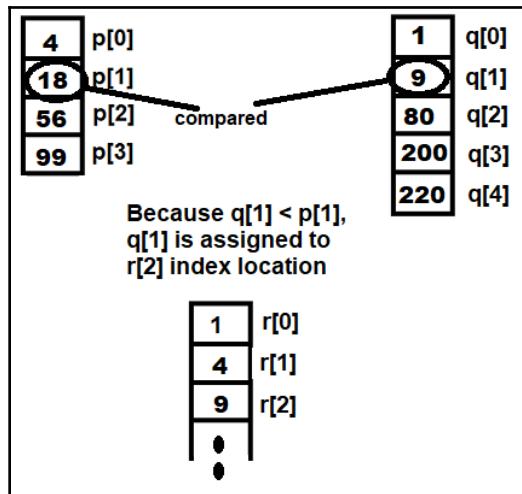


Figure 1.27

Let's use GCC to compile the `mergetwosortedarrays.c` program as follows:

```
D:\CBook>gcc mergetwosortedarrays.c -o mergetwosortedarrays
```

Now, let's run the generated executable file, `mergetwosortedarrays.exe`, in order to see the output of the program:

```
D:\CBook>./mergetwosortedarrays
Enter length of first array:4
Enter 4 elements of the first array in sorted order
4
18
56
99

Enter length of second array:5
Enter 5 elements of the second array in sorted order
1
9
80
200
220

The combined sorted array is:
1
4
9
18
56
80
99
200
220
```

Voilà! We've successfully merged two sorted arrays into one.

2

Managing Strings

Strings are nothing but arrays that store characters. Since strings are character arrays, they utilize less memory and lead to efficient object code, making programs run faster. Just like numerical arrays, strings are zero-based, that is, the first character is stored at index location 0. In C, strings are terminated by a null character, `\0`.

The recipes in this chapter will enhance your understanding of strings and will acquaint you with string manipulation. Strings play a major role in almost all applications. You will learn how to search strings (which is a very common task), replace a string with another string, search for a string that contains a specific pattern, and more.

In this chapter, you will learn how to create the following recipes using strings:

- Determining whether the string is a palindrome
- Finding the occurrence of the first repetitive character in a string
- Displaying the count of each character in a string
- Counting the vowels and consonants in a string
- Converting the vowels in a sentence to uppercase

Determining whether the string is a palindrome

A palindrome is a string that reads the same regardless of whether it is in a forward or backwards order. For example, the word *radar* is a palindrome because it reads the same way forwards and backwards.

How to do it...

1. Define two 80-character strings called `str` and `rev`(assuming your string will not exceed 79 characters). Your string can be of any length, but remember that the last position in the string is fixed for the null character `\0`:

```
char str[80], rev[80];
```

2. Enter characters that will be assigned to the `str` string:

```
printf("Enter a string: ");
scanf("%s", str);
```

3. Compute the length of the string using the `strlen` function and assign this to the `n` variable:

```
n=strlen(str);
```

4. Execute a `for` loop in reverse order to access the characters in the `str` string in reverse order, and then assign them to the `rev` string:

```
for(i=n-1;i >=0; i--)
{
    rev[x]=str[i];
    x++;
}
rev[x]='\0';
```

5. Compare the two strings, `str` and `rev`, using `strcmp`:

```
if(strcmp(str, rev)==0)
```

6. If `str` and `rev` are the same, then the string is a palindrome.



In C, the functionality of specific built-in functions is specified in the respective libraries, also known as header files. So, while writing C programs, whenever built-in functions are used, we need to use their respective header files in the program at the top. The header files usually have the extension .h. In the following program, I am using a built-in function called `strlen`, which finds out the length of a string. Therefore, I need to use its library, `string.h`, in the program.

The `palindrome.c` program for finding out whether the specified string is a palindrome is as follows:

```
#include<stdio.h>
#include<string.h>
void main()
{
    char str[80],rev[80];
    int n,i,x;
    printf("Enter a string: ");
    scanf("%s",str);
    n=strlen(str);
    x=0;
    for(i=n-1;i >=0; i--)
    {
        rev[x]=str[i];
        x++;
    }
    rev[x]='\0';
    if(strcmp(str,rev)==0)
        printf("The %s is palindrome",str);
    else
        printf("The %s is not palindrome",str);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

To ensure that a string is a palindrome, we first need to ensure that the original string and its reverse form are of the same length.

Let's suppose that the original string is `sanjay` and it is assigned to a string variable, `str`. The string is a character array, where each character is stored individually as an array element and the last element in the string array is a null character. The null character is represented as `\0` and is always the last element in a string variable in C, as shown in the following diagram:

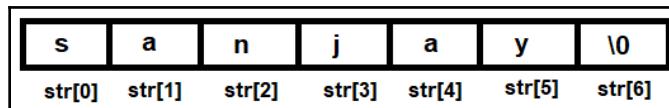


Figure 2.1

As you can see, the string uses zero-based indexing, that is, the first character is placed at index location **str[0]**, followed by the second character at **str[1]**, and so on. In regards to the last element, the null character is at **str[6]**.

Using the **strlen** library function, we will compute the length of the entered string and assign it to the **n** variable. By executing the **for** loop in reverse order, each of the characters of the **str** string is accessed in reverse order, that is, from **n-1** to **0**, and assigned to the **rev** string.

Finally, a null character, **\0**, is added to the **rev** string to make it a complete string. Therefore, **rev** will contain the characters of the **str** string, but in reverse order:

y	a	j	n	a	s	\0
rev[0]	rev[1]	rev[2]	rev[3]	rev[4]	rev[5]	rev[6]

Figure 2.2

Next, we will run the **strcmp** function. If the function returns **0**, it means that the content in the **str** and **rev** strings is exactly the same, which means that **str** is a palindrome. If the **strcmp** function returns a value other than **0**, it means that the two strings are not the same; hence, **str** is not a palindrome.

Let's use GCC to compile the **palindrome.c** program, as follows:

```
D:\CBook>gcc palindrome.c -o palindrome
```

Now, let's run the generated executable file, **palindrome.exe**, to see the output of the program:

```
D:\CBook>./palindrome
Enter a string: sanjay
The sanjay is not palindrome
```

Now, suppose that **str** is assigned another character string, **sanas**. To ensure that the word in **str** is a palindrome, we will again reverse the character order in the string.

So, once more, we will compute the length of `str`, execute a `for` loop in reverse order, and access and assign each character in `str` to `rev`. The null character `\0` will be assigned to the last location in `rev`, as follows:

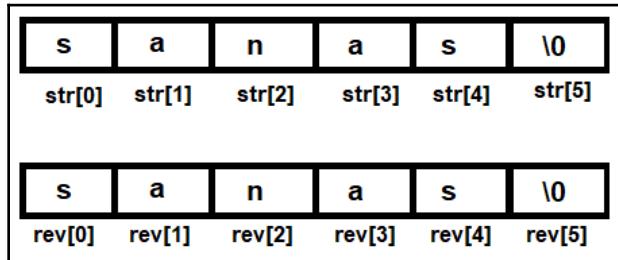


Figure 2.3

Finally, we will invoke the `strcmp` function again and supply both strings.

After compiling, let's run the program again with the new string:

```
D:\CBook>palindrome
Enter a string: sanas
The sanas is palindrome
```

Voilà! We have successfully identified whether our character strings are palindromes. Now, let's move on to the next recipe!

Finding the occurrence of the first repetitive character in a string

In this recipe, you will learn how to create a program that displays the first character to be repeated in a string. For example, if you enter the string `racecar`, the program should give the output as **The first repetitive character in the string racecar is c**. The program should display **No character is repeated in the string** if a string with no repetitive characters is entered.

How to do it...

1. Define two strings called `str1` and `str2`. Your strings can be of any length, but the last position in the string is fixed for the null character `\0`:

```
char str1[80],str2[80];
```

2. Enter characters to be assigned to `str1`. The characters will be assigned to the respective index locations of the string, beginning with `str1[0]`:

```
printf("Enter a string: ");
scanf("%s",str1);
```

3. Compute the length of `str1` using the `strlen` library function. Here, the first character of `str1` is assigned to `str2`:

```
n=strlen(str1);
str2[0]=str1[0];
```

4. Use a `for` loop to access all of the characters of `str1` one by one and pass them to the `ifexists` function to check whether that character already exists in `str2`. If the character is found in `str2`, this means it is the first repetitive character of the string, and so it is displayed on the screen:

```
for(i=1;i < n; i++)
{
    if(ifexists(str1[i], str2, x))
    {
        printf("The first repetitive character in %s is %c",
str1,
str1[i]);
        break;
    }
}
```

5. If the character of `str1` does not exist in `str2`, then it is simply added to `str2`:

```
else
{
    str2[x]=str1[i];
    x++;
}
```

The `repetitive.c` program for finding the occurrence of the first repetitive character in a string is as follows::

```
#include<stdio.h>
#include<string.h>
int ifexists(char u, char z[], int v)
{
    int i;
    for (i=0; i<v; i++)
        if (z[i]==u) return (1);
    return (0);
}

void main()
{
    char str1[80],str2[80];
    int n,i,x;
    printf("Enter a string: ");
    scanf("%s",str1);
    n=strlen(str1);
    str2[0]=str1[0];
    x=1;
    for(i=1;i < n; i++)
    {
        if(ifexists(str1[i], str2, x))
        {
            printf("The first repetitive character in %s is %c", str1,
                   str1[i]);
            break;
        }
        else
        {
            str2[x]=str1[i];
            x++;
        }
    }
    if(i==n)
        printf("There is no repetitive character in the string %s", str1);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

Let's assume that we have defined a string, **str1**, of some length, and have entered the following characters—racecar.

Each of the characters of the string racecar will be assigned to the respective index locations of **str1**, that is, r will be assigned to **str1[0]**, a will be assigned to **str1[1]**, and so on. Because every string in C is terminated by a null character, \0, the last index location of **str1** will have the null character \0, as follows:

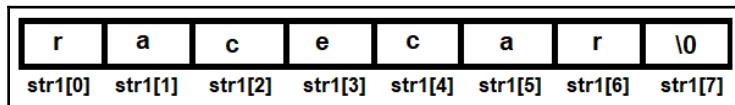


Figure 2.4

Using the library function **strlen**, the length of **str1** is computed and a **for** loop is used to access all of the characters of **str1**, one by one, except for the first character. The first character is already assigned to **str2**, as shown in the following diagram:

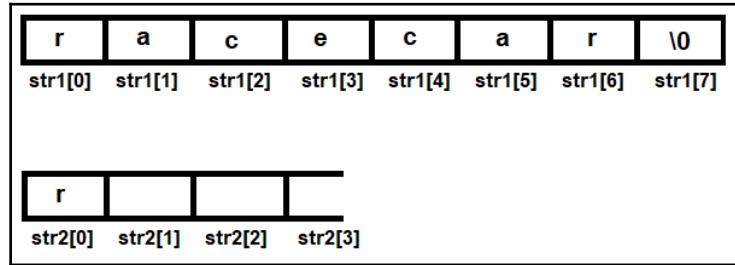


Figure 2.5

Each character that is accessed from **str1** is passed through the **ifeexists** function. The **ifeexists** function will check whether the supplied character already exists in **str2** and will return a Boolean value accordingly. The function returns 1, that is, **true**, if the supplied character is found in **str2**. The function returns 0, that is, **false**, if the supplied character is not found in **str2**.

If `ifexists` returns 1, this means that the character is found in `str2`, and hence, the first repetitive character of the string is displayed on the screen. If the `ifexists` function returns 0, this means that the character does not exist in `str2`, so it is simply added to `str2` instead.

Since the first character is already assigned, the second character of `str1` is picked up and checked to see if it already exists in `str2`. Because the second character of `str1` does not exist in `str2`, it is added to the latter string, as follows:

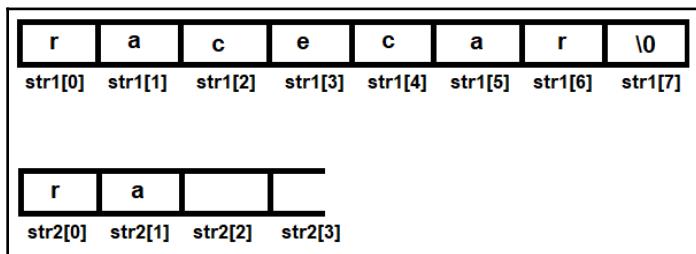


Figure 2.6

The procedure is repeated until all of the characters of `str1` are accessed. If all the characters of `str1` are accessed and none of them are found to exist in `str2`, this means that all of the characters in `str1` are unique and none of them are repeated.

The following diagram shows strings `str1` and `str2` after accessing the first four characters of `str1`. You can see that the four characters are added to `str2`, since none of them already exist in `str2`:

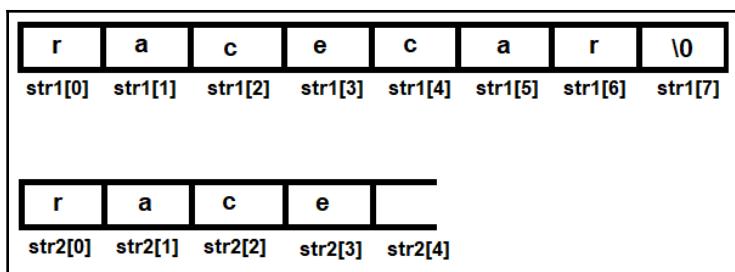


Figure 2.7

The next character to be accessed from **str1** is **c**. Before adding it to **str2**, it is compared with all the existing characters of **str2** to determine if it already exists there. Because the **c** character already exists in **str2**, it is not added to **str2** and is declared as the first repeating character in **str1**, as follows:

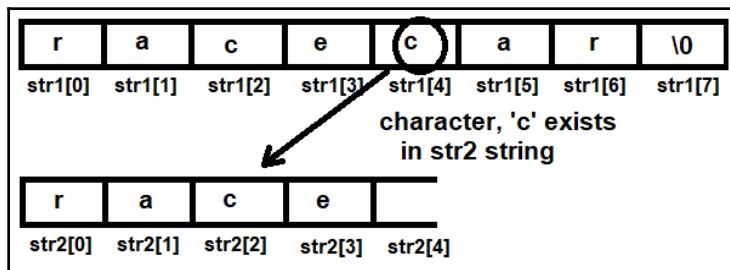


Figure 2.8

Let's use GCC to compile the `repetitive.c` program, as follows:

```
D:\CBook>gcc repetitive.c -o repetitive
```

Let's run the generated executable file, `repetitive.exe`, to see the output of the program:

```
D:\CBook>./repetitive
Enter a string: education
There is no repetitive character in the string education
```

Let's run the program again:

```
D:\CBook>repetitive
Enter a string: racecar
The first repetitive character in racecar is c
```

Voilà! We've successfully found the first repeating character in a string.

Now, let's move on to the next recipe!

Displaying the count of each character in a string

In this recipe, you will learn how to create a program that displays the count of each character in a string in a tabular form.

How to do it...

1. Create a string called `str`. The last element in the string will be a null character, `\0`.
2. Define another string called `chr` of matching length, to store the characters of `str`:

```
char str[80], chr[80];
```

3. Prompt the user to enter a string. The entered string will be assigned to the `str` string:

```
printf("Enter a string: ");
scanf("%s", str);
```

4. Compute the length of the string array, `str`, using `strlen`:

```
n=strlen(str);
```

5. Define an integer array called `count` to display the number of times the characters have occurred in `str`:

```
int count[80];
```

6. Execute `chr[0]=str[0]` to assign the first character of `str` to `chr` at index location `chr[0]`.
7. The count of the character that's assigned in the `chr[0]` location is represented by assigning `1` at the `count[0]` index location:

```
chr[0]=str[0];
count[0]=1;
```

8. Run a `for` loop to access each character in `str`:

```
for(i=1; i < n; i++)
```

9. Run the `ifexists` function to find out whether the character of `str` exists in the `chr` string or not. If the character does not exist in the `chr` string, it is added to the `chr` string at the next index location and the respective index location in the `count` array is set to 1:

```
if(!ifexists(str[i], chr, x, count))
{
    x++;
    chr[x]=str[i];
    count[x]=1;
}
```

10. If the character exists in the `chr` string, the value in the respective index location in the `count` array is incremented by 1 in the `ifexists` function. The `p` and `q` arrays in the following snippet represent the `chr` and `count` arrays, respectively, since the `chr` and `count` arrays are passed and assigned to the `p` and `q` parameters in the `ifexists` function:

```
if (p[i]==u)
{
    q[i]++;
    return (1);
}
```

The `countofeach.c` program for counting each character in a string is as follows::

```
#include<stdio.h>
#include<string.h>
int ifexists(char u, char p[], int v, int q[])
{
    int i;
    for (i=0; i<=v; i++)
    {
        if (p[i]==u)
        {
            q[i]++;
            return (1);
        }
    }
    if(i>v) return (0);
}
void main()
{
    char str[80],chr[80];
    int n,i,x,count[80];
    printf("Enter a string: ");
```

```
scanf("%s", str);
n=strlen(str);
chr[0]=str[0];
count[0]=1;
x=0;
for(i=1;i < n; i++)
{
    if(!ifexists(str[i], chr, x, count))
    {
        x++;
        chr[x]=str[i];
        count[x]=1;
    }
}
printf("The count of each character in the string %s is \n", str);
for (i=0;i<=x;i++)
    printf("%c\t%d\n", chr[i], count[i]);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

Let's assume that the two string variables you have defined, `str` and `chr`, are of the size 80 (you can always increase the size of the strings if you wish).

We will assign the character string `racecar` to the `str` string. Each of the characters will be assigned to the respective index locations of `str`, that is, `r` will be assigned to index location `str[0]`, `a` will be assigned to `str[1]`, and so on. As always, the last element in the string will be a null character, as shown in the following diagram:



Figure 2.9

Using the `strlen` function, we will first compute the length of the string. Then, we will use the string array `chr` for storing characters of the `str` array individually at each index location. We will execute a `for` loop beginning from 1 until the end of the string to access each character of the string.

The integer array we defined earlier, that is, **count**, will represent the number of times the characters from **str** have occurred, which is represented by the index locations in the **chr** array. That is, if **r** is at index location **chr[0]**, then **count[0]** will contain an integer value (1, in this case) to represent the number of times the **r** character has occurred in the **str** string so far:

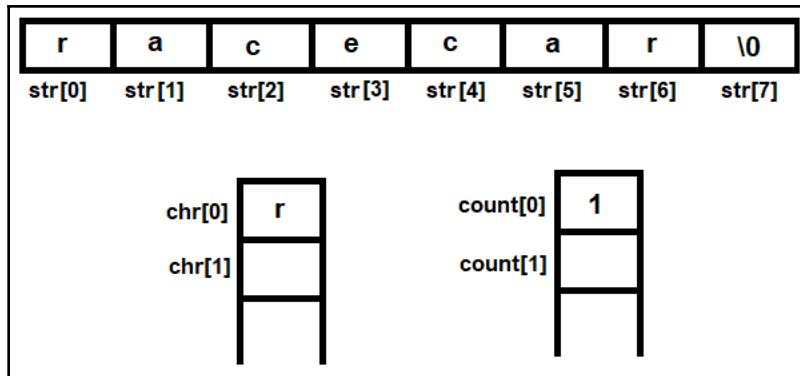


Figure 2.10

One of the following actions will be applied to every character that's accessed from the string:

- If the character exists in the **chr** array, the value in the respective index location in the **count** array is incremented by 1. For example, if the character of the string is found at the **chr[2]** index location, then the value in the **count[2]** index location is incremented by 1.
- If the character does not exist in the **chr** array, it is added to the **chr** array at the next index location, and the respective index location is found when the count array is set to 1. For example, if the character **a** is not found in the **chr** array, it is added to the **chr** array at the next available index location. If the character **a** is added at the **chr[1]** location, then a value of **1** is assigned at the **count[1]** index location to indicate that the character shown in **chr[1]** has appeared once up until now.

When the `for` loop completes, that is when all of the characters in the string are accessed. The `chr` array will have individual characters of the string and the `count` array will have the count, or the number of times the characters represented by the `chr` array have occurred in the string. All of the elements in the `chr` and `count` arrays are displayed on the screen.

Let's use GCC to compile the `countofeach.c` program, as follows:

```
D:\CBook>gcc countofeach.c -o countofeach
```

Let's run the generated executable file, `countofeach.exe`, to see the output of the program:

```
D:\CBook>./countofeach
Enter a string: bintu
The count of each character in the string bintu is
b      1
i      1
n      1
t      1
u      1
```

Let's try another character string to test the results:

```
D:\CBook>./countofeach
Enter a string: racecar
The count of each character in the string racecar is
r      2
a      2
c      2
e      1
```

Voilà! We've successfully displayed the count of each character in a string.

Now, let's move on to the next recipe!

Counting vowels and consonants in a sentence

In this recipe, you will learn how to count the number of vowels and consonants in an entered sentence. The vowels are *a*, *e*, *i*, *o*, and *u*, and the rest of the letters are consonants. We will use ASCII values to identify the letters and their casing:

ASCII	
A	65
B	66
C	67
•	•
•	•
•	•
Z	90
ASCII	
a	97
b	98
c	99
•	•
•	•
z	122

Figure 2.11

The blank spaces, numbers, special characters, and symbols will simply be ignored.

How to do it...

1. Create a string array called `str` to input your sentence. As usual, the last character will be a null character:

```
char str[255];
```

2. Define two variables, `ctrV` and `ctrC`:

```
int ctrV,ctrC;
```

3. Prompt the user to enter a sentence of your choice:

```
printf("Enter a sentence: ");
```

4. Execute the `gets` function to accept the sentence with blank spaces between the words:

```
gets(str);
```

5. Initialize `ctrV` and `ctrC` to 0. The `ctrV` variable will count the number of vowels in the sentence, while the `ctrC` variable will count the number of consonants in the sentence:

```
ctrV=ctrC=0;
```

6. Execute a `while` loop to access each letter of the sentence one, by one until the null character in the sentence is reached.
7. Execute an `if` block to check whether the letters are uppercase or lowercase, using ASCII values. This also confirms that the accessed character is not a white space, special character or symbol, or number.
8. Once that's done, execute a nested `if` block to check whether the letter is a lowercase or uppercase vowel, and wait for the `while` loop to terminate:

```
while(str[i]!='\0')  
{  
    if((str[i] >=65 && str[i]<=90) || (str[i] >=97 && str[i]<=122))  
    {  
        if(str[i]=='A' ||str[i]=='E' ||str[i]=='I' ||str[i]=='O'  
        ||str[i]=='U' ||str[i]=='a' ||str[i]=='e' ||str[i]=='i'  
        ||str[i]=='o'||str[i]=='u')  
            ctrV++;  
        else  
            ctrC++;  
    }  
    i++;  
}
```

The `countvowelsandcons.c` program for counting vowels and consonants in a string is as follows:

```
#include <stdio.h>  
void main()  
{  
    char str[255];  
    int ctrV,ctrC,i;  
    printf("Enter a sentence: ");  
    gets(str);  
    ctrV=ctrC=i=0;  
    while(str[i]!='\0')  
    {
```

```
if((str[i] >=65 && str[i]<=90) || (str[i] >=97 && str[i]<=122))
{
    if(str[i]=='A' ||str[i]=='E' ||str[i]=='I' ||str[i]=='O'
    ||str[i]=='U' ||str[i]=='a' ||str[i]=='e' ||str[i]=='i'
    ||str[i]=='o'||str[i]=='u')
        ctrV++;
    else
        ctrC++;
}
i++;
}
printf("Number of vowels are : %d\nNumber of consonants are :
%d\n",ctrV,ctrC);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We are assuming that you will not enter a sentence longer than 255 characters, so we have defined our string variable accordingly. When prompted, enter a sentence that will be assigned to the `str` variable. Because a sentence may have blank spaces between the words, we will execute the `gets` function to accept the sentence.

The two variables that we've defined, that is, `ctrV` and `ctrC`, are initialized to 0. Because the last character in a string is always a null character, `\0`, a `while` loop is executed, which will access each character of the sentence one by one until the null character in the sentence is reached.

Every accessed letter from the sentence is checked to confirm that it is either an uppercase or lowercase character. That is, their ASCII values are compared, and if the ASCII value of the accessed character is a lowercase or uppercase character, only then it will execute the nested `if` block. Otherwise, the next character from the sentence will be accessed.

Once you have ensured that the accessed character is not a blank space, any special character or symbol, or a numerical value, then an `if` block will be executed, which checks whether the accessed character is a lowercase or uppercase vowel. If the accessed character is a vowel, then the value of the `ctrV` variable is incremented by 1. If the accessed character is not a vowel, then it is confirmed that it is a consonant, and so the value of the `ctrC` variable is incremented by 1.

Once all of the characters of the sentence have been accessed, that is, when the null character of the sentence is reached, the `while` loop terminates and the number of vowels and consonants stored in the `ctrV` and `ctrC` variables is displayed on the screen.

Let's use GCC to compile the `countvowelsandcons.c` program, as follows:

```
D:\CBook>gcc countvowelsandcons.c -o countvowelsandcons
```

Let's run the generated executable file, `countvowelsandcons.exe`, to see the output of the program:

```
D:\CBook>./countvowelsandcons
Enter a sentence: Today it might rain. Its a hot weather. I do like rain
Number of vowels are : 18
Number of consonants are : 23
```

Voilà! We've successfully counted all of the vowels and consonants in our sentence.

Now, let's move on to the next recipe!

Converting the vowels in a sentence to uppercase

In this recipe, you will learn how to convert all of the lowercase vowels in a sentence to uppercase. The remaining characters in the sentence, including consonants, numbers, special symbols, and special characters, are simply ignored and will be left as they are.

Converting the casing of any letter is done by simply changing the ASCII value of that character, using the following formulas:

- Subtract 32 from the ASCII value of a lowercase character to convert it to uppercase
- Add 32 to the ASCII value of an uppercase character to convert it to lowercase

The following diagram shows the ASCII values of the uppercase and lowercase vowels:

ASCII	
A	65
E	69
I	73
O	79
U	85
ASCII	
a	97
e	101
i	105
o	111
u	117

Figure 2.12

The ASCII value of the uppercase letters is lower than that of lowercase letters, and the difference between the values is 32.

How to do it...

1. Create a string called `str` to input your sentence. As usual, the last character will be a null character:

```
char str[255];
```

2. Enter a sentence of your choice:

```
printf("Enter a sentence: ");
```

3. Execute the `gets` function to accept the sentence with blank spaces between the words, and initialize the `i` variable to 0, since each character of the sentence will be accessed through `i`:

```
gets(str);
i=0
```

4. Execute a `while` loop to access each letter of the sentence one by one, until the null character in the sentence is reached:

```
while(str[i]!='\0')
{
    {
        ...
    }
}
i++;
```

5. Check each letter to verify whether it is a lowercase vowel. If the accessed character is a lowercase vowel, then the value 32 is subtracted from the ASCII value of that vowel to convert it to uppercase:

```
if(str[i]=='a' ||str[i]=='e' ||str[i]=='i' ||str[i]=='o'
||str[i]=='u')
    str[i]=str[i]-32;
```

6. When all of the letters of the sentence have been accessed, then simply display the entire sentence.

The `convertvowels.c` program for converting the lowercase vowels in a sentence to uppercase is as follows:

```
#include <stdio.h>
void main()
{
    char str[255];
    int i;
    printf("Enter a sentence: ");
    gets(str);
    i=0;
    while(str[i]!='\0')
    {
        if(str[i]=='a' ||str[i]=='e' ||str[i]=='i' ||str[i]=='o'
        ||str[i]=='u')
            str [i] = str [i] -32;
        i++;
    }
    printf("The sentence after converting vowels into uppercase
is:\n");
    puts(str);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

Again, we will assume that you will not enter a sentence longer than 255 characters. Therefore, we have defined our string array, `str`, to be of the size 255. When prompted, enter a sentence to assign to the `str` array. Because a sentence may have blank spaces between the words, instead of `scanf`, we will use the `gets` function to accept the sentence.

To access each character of the sentence, we will execute a `while` loop that will run until the null character is reached in the sentence. After each character of the sentence, it is checked whether it is a lowercase vowel. If it is not a lowercase vowel, the character is ignored and the next character in the sentence is picked up for comparison.

If the character that's accessed is a lowercase vowel, then a value of 32 is subtracted from the ASCII value of the character to convert it to uppercase. Remember that the difference in the ASCII values of lowercase and uppercase letters is 32. That is, the ASCII value of lowercase `a` is 97 and that of uppercase `A` is 65. So, if you subtract 32 from 97, which is the ASCII value of lowercase `a`, the new ASCII value will become 65, which is the ASCII value of uppercase `A`.

The procedure of converting a lowercase vowel to an uppercase vowel is to first find the vowel in a sentence by using an `if` statement, and then subtract the value 32 from its ASCII value to convert it to uppercase.

Once all of the characters of the string are accessed and all of the lowercase vowels of the sentence are converted to uppercase, the entire sentence is displayed using the `puts` function.

Let's use GCC to compile the `convertvowels.c` program, as follows:

```
D:\CBook>gcc convertvowels.c -o convertvowels
```

Let's run the generated executable file, `convertvowels.exe`, to see the output of the program:

```
D:\CBook>./convertvowels
Enter a sentence: It is very hot today. Appears as if it might rain. I like
rain
The sentence after converting vowels into uppercase is:
It Is vEry hOt tOdAy. AppEArs As If It mIght rAIIn. I lIkE rAIIn
```

Voilà! We've successfully converted the lowercase vowels in a sentence to uppercase.

3

Exploring Functions

Whenever you need to create a large application, it is a wise decision to divide it into manageable chunks, called **functions**. Functions are small modules that represent tasks that can be executed independently. The code written inside a function can be invoked several times, which helps to avoid repetitive statements.

Functions help in the teamwork, debugging, and scaling of any application. Whenever you want to add more features to an application, simply add a few functions to it. When calling functions, the caller function may pass certain arguments, called **actual arguments**; these are then assigned to the parameters of the function. The parameters are also known as formal parameters.

The following recipes will help you understand how functions can be used to make complex applications easier and more manageable. Normally, a function can return only a single value. But in this chapter, we will learn a technique to return more than one value from a function. We will also learn how to apply recursion in functions.

In this chapter, we will cover the following recipes on strings:

- Determining whether a number is an Armstrong number
- Returning the maximum and minimum values of an array
- Finding GCD using recursion
- Converting a binary number into a hexadecimal number
- Determining whether a number is a palindrome

As I will be using a stack structure in the recipes in this chapter, let's have a quick introduction to stack.

What is a stack?

A Stack is a structure that can be implemented with arrays as well as linked lists. It is a sort of a bucket where the value you enter will be added at the bottom. The next item that you add to a stack will be kept just above the item that was added previously. The procedure of adding a value to the stack is called a **push** operation and the procedure of getting a value out of the stack is called a **pop** operation. The location where the value can be added or taken out of the stack is pointed at by a pointer called **top**. The value of the **top** pointer is **-1** when the stack is empty:

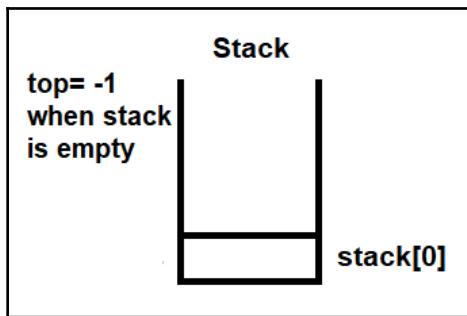


Figure 3.1

When the **push** operation is executed, the value of **top** is incremented by **1**, so that it can point to the location in the stack where the value can be pushed:

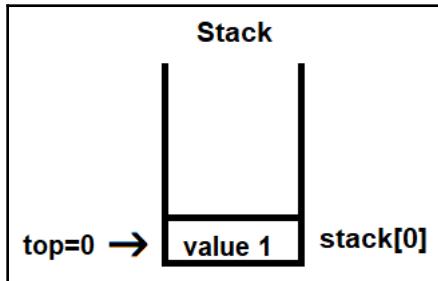


Figure 3.2

Now, the next value that will be pushed will be kept above value 1. More precisely, the value of the **top** pointer will be incremented by 1, making its value 1, and the next value will be pushed to the **stack[1]** location, as follows:

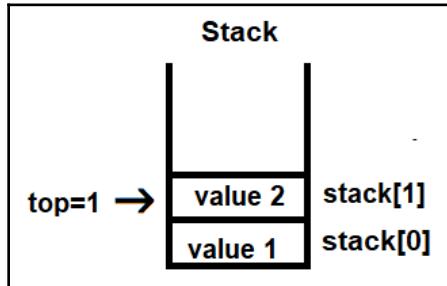


Figure 3.3

So, you can see that the stack is a **Last In First Out (LIFO)** structure; that is, the last value that was pushed sits at the top.

Now, when we execute a **pop** operation, the value at the top, that is, value **2**, will be popped out first, followed by the popping out of value **1**. Basically, in the **pop** operation, the value pointed at by the **top** pointer is taken out, and then the value of **top** is decremented by 1 so that it can point at the next value to be popped out.

Now, that we've understood stacks, let's begin with the first recipe.

Finding whether a number is an Armstrong number

An Armstrong number is a three-digit integer that is the sum of the cubes of its digits. This simply means that if $xyz = x^3 + y^3 + z^3$, it is an Armstrong number. For example, 153 is an Armstrong number because $1^3 + 5^3 + 3^3 = 153$.

Similarly, a number that comprises four digits is an Armstrong number if the sum of its digits raised to the power of four results in that number. For example, $pqrs = p^4 + q^4 + r^4 + s^4$.

How to do it...

1. Enter a number to assign to the `n` variable:

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

2. Invoke the `findarmstrong` function. The value assigned to `n` will get passed to this function:

```
findarmstrong(n)
```

3. In the function, the passed argument, `n`, is assigned to the `numb` parameter. Execute a `while` loop to separate out all the digits in the `numb` parameter:

```
while (numb >0)
```

4. In the `while` loop, apply the mod 10 (`%10`) operator on the number assigned to the `numb` variable. The mod operator divides a number and returns the remainder:

```
remainder=numb%10;
```

5. Push the remainder to the stack:

```
push(remainder);
```

6. Remove the last digit of the number in the `numb` variable by dividing the `numb` variable by 10:

```
numb=numb/10;
```

7. Repeat steps 4 to 6 until the number in the `numb` variable becomes 0. In addition, create a `count` counter to count the number of digits in the number. Initialize the counter to 0 and it will get incremented during the `while` loop:

```
count++;
```

8. Pop all the digits from the stack and raise it to the given power. To pop all the digits from the stack, execute a `while` loop that will execute until `top` is greater than or equal to 0, that is, until the stack is empty:

```
while (top >=0)
```

9. Inside the `while` loop, pop off a digit from the stack and raise it to the power of `count`, which is the count of the number of digits in the selected number. Then, add all the digits to the `value`:

```
j=pop();
value=value+pow(j,count);
```

10. Compare the number in the `value` variable with the number in the `numb` variable, and code it to return the value of 1 if both the compared numbers match:

```
if(value==numb) return 1;
```

If the numbers in the `numb` and `value` variables are the same, returning the Boolean value of 1, that means the number is an Armstrong number.

Here is the `armstrong.c` program for finding out whether the specified number is an Armstrong number:

```
/* Finding whether the entered number is an Armstrong number */
# include <stdio.h>
# include <math.h>

#define max 10

int top=-1;
int stack[max];
void push(int);
int pop();
int findarmstrong(int );
void main()
{
    int n;
    printf("Enter a number ");
    scanf("%d",&n);
    if (findarmstrong(n))
        printf("%d is an armstrong number",n);
    else printf("%d is not an armstrong number", n);
}
int findarmstrong(int numb)
{
    int j, remainder, temp,count,value;
    temp=numb;
    count=0;
    while(numb >0)
    {
        remainder=numb%10;
```

```
push(remainder);
count++;
numb=numb/10;
}
numb=temp;
value=0;
while(top >=0)
{
    j=pop();
    value=value+pow(j,count);
}
if(value==numb) return 1;
else return 0;
}
void push(int m)
{
    top++;
    stack[top]=m;
}
int pop()
{
    int j;
    if(top== -1) return (top);
    else
    {
        j=stack[top];
        top--;
        return (j);
    }
}
```

Now, let's go behind the scenes.

How it works...

First, we will apply the mod **10** operator to separate our digits. Assuming the number entered by us is **153**, you can see that **153** is divided by **10** and the remaining **3** is pushed to the stack:

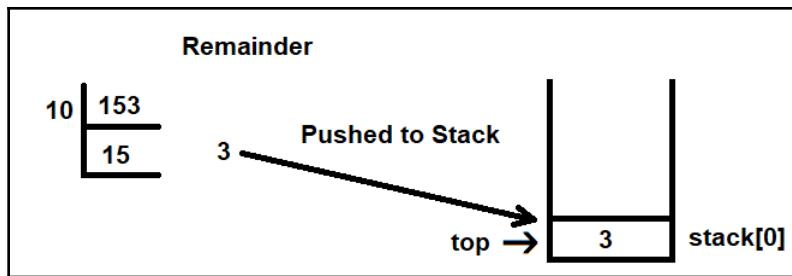


Figure 3.4

The value in the stack is pushed at the index location indicated by **top**. Initially, the value of **top** is -1. It is so because before the push operation, the value of **top** is incremented by 1, and the array is zero-based, that is, the first element in the array is placed at the 0 index location. Consequently, the value of **top** has to be initialized to -1. As mentioned, the value of **top** is incremented by 1 before pushing, that is, the value of **top** will become 0, and the remainder of 3 is pushed to **stack[0]**.



In the stack, the value of **top** is incremented by 1 to indicate the location in the stack where the value will be pushed.

We will again apply the mod 10 operator to the 15 quotient. The remainder that we will get is 5, which will be pushed to the stack. Again, before pushing to the stack, the value of **top**, which was 0, is incremented to 1. At **stack[1]**, the remainder is pushed:

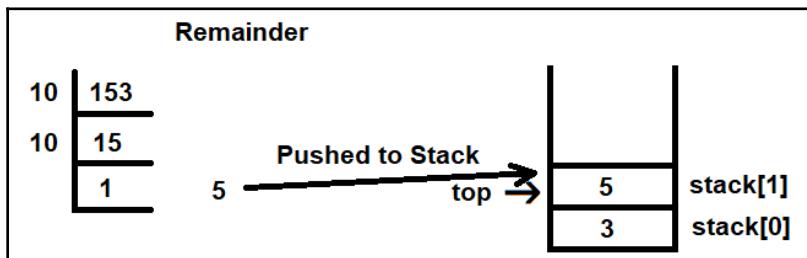


Figure 3.5

To the **1** quotient, we will again apply the mod **10** operator. But because **1** is not divisible by **10**, **1** itself will be considered as the remainder and will be pushed to the stack. The value of **top** will again be incremented by **1** and **1** will be pushed to **stack[2]**:

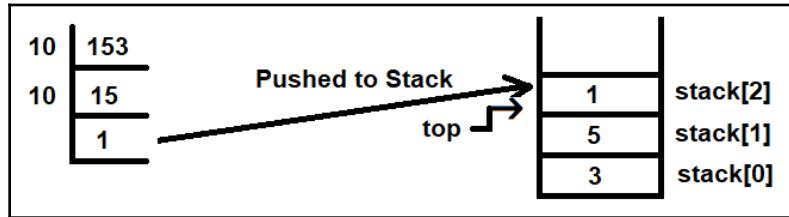


Figure 3.6

Once all the digits are separated and placed in the stack, we will pop them out one by one. Then, we will raise each digit to the power equal to the count of the digits. Because the number **153** consists of three digits, each digit is raised to the power of **3**.

While popping values out of the stack, the value indicated by the **top** pointer is popped out. The value of **top** is **2**, hence the value at **stack[2]**, that is, **1**, is popped out and raised to the power of **3**, as follows:

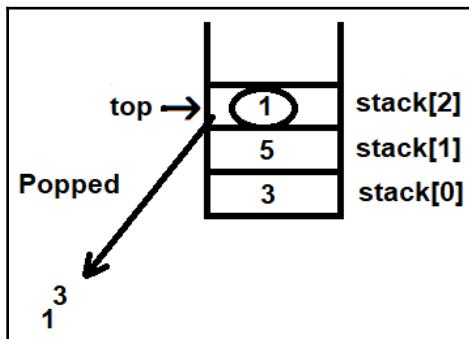


Figure 3.7

After the pop operation, the value of **top** will be decremented to **1** to indicate the next location to be popped out. Next, the value at **stack[1]** will be popped out and raised to the power of **3**. We will then add this value to our previous popped-out one:

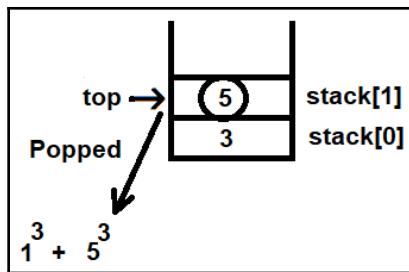


Figure 3.8

After the popping-out operation, the value of **top** is decremented by 1, now making its value **0**. So, the value at **stack[0]** is popped out and raised to the power of 3. The result is added to our earlier computation:

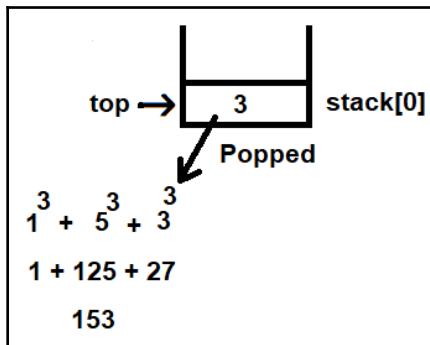


Figure 3.9

The result after computing $1^3 + 5^3 + 3^3$ is **153**, which is the same as the original number. This proves that **153** is an Armstrong number.

Let's use GCC to compile the `armstrong.c` program, as follows:

```
D:\CBook>gcc armstrong.c -o armstrong
```

Let's check whether **127** is an Armstrong number:

```
D:\CBook>./armstrong
Enter a number 127
127 is not an armstrong number
```

Let's check whether 153 is an Armstrong number:

```
D:\CBook> ./armstrong
Enter a number 153
153 is an armstrong number
```

Let's check whether 1634 is an Armstrong number:

```
D:\CBook> ./armstrong
Enter a number 1634
1634 is an armstrong number
```

Voilà! We've successfully made a function to find whether a specified number is an Armstrong number or not.

Now, let's move on to the next recipe!

Returning maximum and minimum values in an array

C functions cannot return more than one value. But what if you want a function to return more than one value? The solution is to store the values to be returned in an array and make the function return the array instead.

In this recipe, we will make a function return two values, the maximum and minimum values, and store them in another array. Thereafter, the array containing the maximum and minimum values will be returned from the function.

How to do it...

1. The size of the array whose maximum and minimum values have to be found out is not fixed, hence we will define a macro called `max` of size 100:

```
#define max 100
```

2. We will define an `arr` array of the `max` size, that is, 100 elements:

```
int arr[max];
```

3. You will be prompted to specify the number of elements in the array; the length you enter will be assigned to the `n` variable:

```
printf("How many values? ");
scanf("%d", &n);
```

4. Execute a `for` loop for `n` number of times to accept `n` values for the `arr` array:

```
for(i=0;i<n;i++)
    scanf("%d", &arr[i]);
```

5. Invoke the `maxmin` function to pass the `arr` array and its length, `n`, to it. The array that will be returned by the `maxmin` function will be assigned to the integer pointer, `*p`:

```
p=maxmin(arr,n);
```

6. When you look at the function definition, `int *maxmin(int ar[], int v) { }`, the `arr` and `n` arguments passed to the `maxmin` function are assigned to the `ar` and `v` parameters, respectively. In the `maxmin` function, define an `mm` array of two elements:

```
static int mm[2];
```

7. To compare it with the rest of the elements, the first element of `ar` array is stored at `mm[0]` and `mm[1]`. A loop is executed from the `1` value till the end of the length of the array and within the loop, the following two formulas are applied:

- We will use `mm[0]` to store the minimum value of the `arr` array. The value in `mm[0]` is compared with the rest of the elements. If the value in `mm[0]` is greater than any of the array elements, we will assign the smaller element to `mm[0]`:

```
if(mm[0] > ar[i])
    mm[0]=ar[i];
```

- We will use `mm[1]` to store the maximum value of the `arr` array. If the value at `mm[1]` is found to be smaller than any of the rest of the array element, we will assign the larger array element to `mm[1]`:

```
if(mm[1]< ar[i])
    mm[1]= ar[i];
```

8. After the execution of the `for` loop, the `mm` array will have the minimum and maximum values of the `arr` array at `mm[0]` and `mm[1]`, respectively. We will return this `mm` array to the `main` function where the `*p` pointer is set to point at the returned array, `mm`:

```
return mm;
```

9. The `*p` pointer will first point to the memory address of the first index location, that is, `mm[0]`. Then, the content of that memory address, that is, the minimum value of the array, is displayed. After that, the value of the `*p` pointer is incremented by 1 to make it point to the memory address of the next element in the array, that is, the `mm[1]` location:

```
printf("Minimum value is %d\n", *p++);
```

10. The `mm[1]` index location contains the maximum value of the array. Finally, the maximum value pointed to by the `*p` pointer is displayed on the screen:

```
printf("Maximum value is %d\n", *p);
```

The `returnarray.c` program explains how an array can be returned from a function. Basically, the program returns the minimum and maximum values of an array:

```
/* Find out the maximum and minimum values using a function returning an
array */
# include <stdio.h>
#define max 100
int *maxmin(int ar[], int v);
void main()
{
    int arr[max];
    int n,i, *p;
    printf("How many values? ");
    scanf("%d",&n);
    printf("Enter %d values\n", n);
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    p=maxmin(arr,n);
    printf("Minimum value is %d\n", *p++);
    printf("Maximum value is %d\n", *p);
}
int *maxmin(int ar[], int v)
{
    int i;
    static int mm[2];
    mm[0]=ar[0];
```

```

mm[1]=ar[0];
for (i=1;i<v;i++)
{
    if (mm[0] > ar[i])
        mm[0]=ar[i];
    if (mm[1]< ar[i])
        mm[1]= ar[i];
}
return mm;
}

```

Now, let's go behind the scenes.

How it works...

We will use two arrays in this recipe. The first array will contain the values from which the maximum and minimum values have to be found. The second array will be used to store the minimum and maximum values of the first array.

Let's call the first array **arr** and define it to contain five elements with the following values:

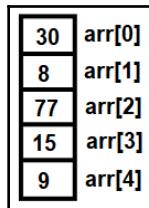


Figure 3.10

Let's call our second array **mm**. The first location, **mm[0]**, of the **mm** array will be used for storing the minimum value and the second location, **mm[1]**, for storing the maximum value of the **arr** array. To enable comparison of the elements of the **mm** array with the elements of the **arr** array, copy the first element of the **arr** array at **arr[0]** to both **mm[0]** and **mm[1]**:

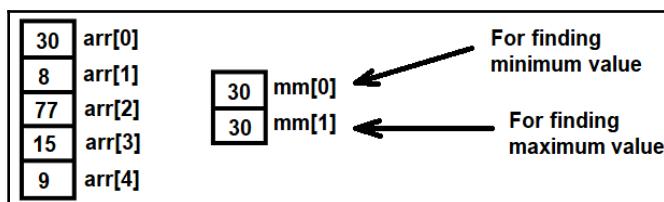


Figure 3.11

Now, we will compare the rest of the elements of the **arr** array with **mm[0]** and **mm[1]**. To keep the minimum value at **mm[0]**, any element smaller than the value at **mm[0]** will be assigned to **mm[0]**. Values larger than **mm[0]** are simply ignored. For example, the value at **arr[1]** is smaller than that at **mm[0]**, that is, $8 < 30$. So, the smaller value will be assigned to **mm[0]**:

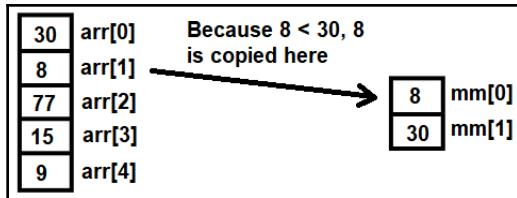


Figure 3.12

We will apply reverse logic to the element at **mm[1]**. Because we want the maximum value of the **arr** array at **mm[1]**, any element found larger than the value at **mm[1]** will be assigned to **mm[1]**. All smaller values will be simply ignored.

We will continue this process with the next element in the **arr** array, which is **arr[2]**. Because $77 > 8$, it will be ignored when compared with **mm[0]**. But $77 > 30$, so it will be assigned to **mm[1]**:

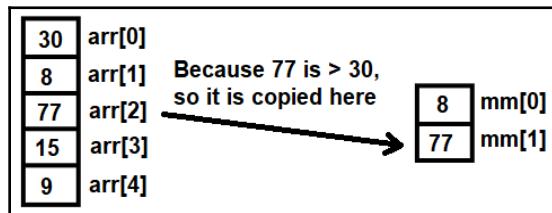


Figure 3.13

We will repeat this procedure with the rest of the elements of the **arr** array. Once all the elements of the **arr** array are compared with both the elements of the **mm** array, we will have the minimum and maximum values at **mm[0]** and **mm[1]**, respectively:

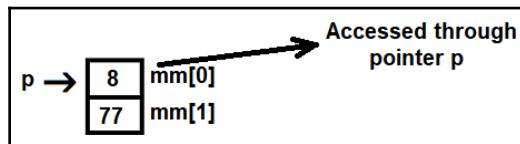


Figure 3.14

Let's use GCC to compile the `returnarray.c` program, as follows:

```
D:\CBook>gcc returnarray.c -o returnarray
```

Here is the output of the program:

```
D:\CBook>./returnarray
How many values? 5
Enter 5 values
30
8
77
15
9
Minimum value is 8
Maximum value is 77
```

Voilà! We've successfully returned the maximum and minimum values in an array.

Now, let's move on to the next recipe!

Finding the greatest common divisor using recursion

In this recipe, we will use recursive functions to find the **greatest common divisor (GCD)**, also known as the highest common factor) of two or more integers. The GCD is the largest positive integer that divides each of the integers. For example, the GCD of 8 and 12 is 4, and the GCD of 9 and 18 is 9.

How to do it...

The `int gcd(int x, int y)` recursive function finds the GCD of two integers, `x` and `y`, using the following three rules:

- If $y=0$, the GCD of x and y is x .
- If $x \bmod y$ is 0, the GCD of x and y is y .
- Otherwise, the GCD of x and y is $\gcd(y, (x \bmod y))$.

Follow the given steps to find the GCD of two integers recursively:

1. You will be prompted to enter two integers. Assign the integers entered to two variables, `u` and `v`:

```
printf("Enter two numbers: ");
scanf("%d %d",&x,&y);
```

2. Invoke the `gcd` function and pass the `x` and `y` values to it. The `x` and `y` values will be assigned to the `a` and `b` parameters, respectively. Assign the GCD value returned by the `gcd` function to the `g` variable:

```
g=gcd(x,y);
```

3. In the `gcd` function, `a % b` is executed. The `%` (mod) operator divides the number and returns the remainder:

```
m=a%b;
```

4. If the remainder is non-zero, call the `gcd` function again, but this time the arguments will be `gcd(b, a % b)`, that is, `gcd(b, m)`, where `m` stands for the mod operation:

```
gcd(b,m);
```

5. If this again results in a non-zero remainder, that is, if `b % m` is non-zero, repeat the `gcd` function with the new values obtained from the previous execution:

```
gcd(b,m);
```

6. If the result of `b % m` is zero, `b` is the GCD of the supplied arguments and is returned back to the `main` function:

```
return(b);
```

7. The result, `b`, that is returned back to the `main` function is assigned to the `g` variable, which is then displayed on the screen:

```
printf("Greatest Common Divisor of %d and %d is %d",x,y,g);
```

The `gcd.c` program explains how the greatest common divisor of two integers is computed through the recursive function:

```
#include <stdio.h>
int gcd(int p, int q);
void main()
{
    int x,y,g;
    printf("Enter two numbers: ");
    scanf("%d %d", &x, &y);
    g=gcd(x,y);
    printf("Greatest Common Divisor of %d and %d is %d",x,y,g);
}
int gcd(int a, int b)
{
    int m;
    m=a%b;
    if(m==0)
        return(b);
    else
        gcd(b,m);
}
```

Now, let's go behind the scenes.

How it works...

Let's assume we want to find the GCD of two integers, **18** and **24**. To do so, we will invoke the `gcd(x, y)` function, which in this case is `gcd(18, 24)`. Because **24**, that is, y , is not zero, Rule 1 is not applicable here. Next, we will use Rule 2 to check whether $18 \% 24$ ($x \% y$) is equal to 0. Because **18** cannot be divided by **24**, **18** will be the remainder:

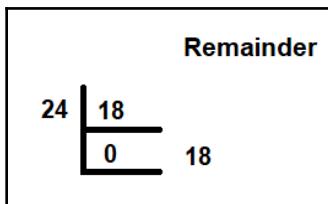


Figure 3.15

Since the parameters of Rule 2 were also not met, we will use Rule 3. We will invoke the `gcd` function with the `gcd(b, m)` argument, which is `gcd(24, 18%24)`. Now, `m` stands for the mod operation. At this stage, we will again apply Rule 2 and collect the remainder:

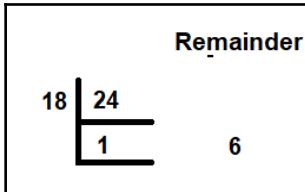


Figure 3.16

Because the result of `24%18` is a non-zero value, we will invoke the `gcd` function again with the `gcd(b, m)` argument, which is now `gcd(18, 24%18)`, since we were left with **18** and **6** from the previous execution. We will again apply Rule 2 to this execution. When **18** is divided by **6**, the remainder is **0**:

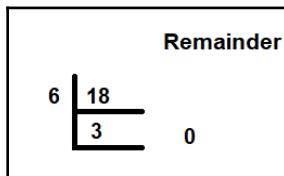


Figure 3.17

At this stage, we have finally fulfilled the requirements of one of the rules, Rule 2. If you recall, Rule 2 says that if $x \bmod y$ is 0, the GCD is y . Because the result of **18 mod 6** is 0, the GCD of **18** and **24** is **6**.

Let's use GCC to compile the `gcd.c` program, as follows:

```
D:\CBook>gcc gcd.c -o gcd
```

Here is the output of the program:

```
D:\CBook>./gcd
Enter two numbers: 18 24
Greatest Common Divisor of 18 and 24 is 6
D:\CBook>./gcd
Enter two numbers: 9 27
Greatest Common Divisor of 9 and 27 is 9
```

Voilà! We've successfully found the GCD using recursion.

Now, let's move on to the next recipe!

Converting a binary number into a hexadecimal number

In this recipe, we will learn how to convert a binary number into a hexadecimal number. A binary number comprises two bits, 0 and 1. To convert a binary number into a hexadecimal number, we first need to convert the binary number into a decimal number and then convert the resulting decimal number to hexadecimal.

How to do it...

1. Enter a binary number and assign it to the `b` variable:

```
printf("Enter a number in binary number ");
scanf("%d", &b);
```

2. Invoke the `intodecimal` function to convert the binary number into a decimal number, and pass the `b` variable to it as an argument. Assign the decimal number returned by the `intodecimal` function to the `d` variable:

```
d=intodecimal(b);
```

3. On looking at the `intodecimal` definition, `int intodecimal(int bin) { }`, we can see that the `b` argument is assigned to the `bin` parameter of the `intodecimal` function.

4. Separate all the binary digits and multiply them by 2 raised to the power of their position in the binary number. Sum the results to get the decimal equivalent. To separate each binary digit, we need to execute a `while` loop until the binary number is greater than 0:

```
while(bin >0)
```

5. Within the `while` loop, apply the mod 10 operator on the binary number and push the remainder to the stack:

```
remainder=bin%10;  
push(remainder);
```

6. Execute another `while` loop to get the decimal number of all the binary digits from the stack. The `while` loop will execute until the stack becomes empty (that is, until the value of `top` is greater than or equal to 0):

```
while (top >=0)
```

7. In the `while` loop, pop off all the binary digits from the stack and multiply each one by 2 raised to the power of `top`. Sum the results to get the decimal equivalent of the entered binary number:

```
j=pop();  
deci=deci+j*pow(2,exp);
```

8. Invoke the `intohexa` function and pass the binary number and the decimal number to it to get the hexadecimal number:

```
void intohexa(int bin, int deci)
```

9. Apply the mod 16 operator in the `intohexa` function on the decimal number to get its hexadecimal. Push the remainder that you get to the stack. Apply mod 16 to the quotient again and repeat the process until the quotient becomes smaller than 16:

```
remainder=deci%16;  
push(remainder);
```

10. Pop off the remainders that are pushed to the stack to display the hexadecimal number:

```
j=pop();
```

If the remainder that is popped off from the stack is less than 10, it is displayed as such. Otherwise, it is converted to its equivalent letter, as mentioned in the following table, and the resulting letter is displayed:

Decimal	Hexadecimal
10	A
11	B
12	C
13	D
14	E
15	F

```
if(j<10)printf("%d",j);
else printf("%c",prnhexa(j));
```

The `binarytohexa.c` program explains how a binary number can be converted into a hexadecimal number:

```
//Converting binary to hex
#include <stdio.h>
#include <math.h>
#define max 10
int top=-1;
int stack[max];
void push();
int pop();
char prnhexa(int);
int intodecimal(int);
void intohexa(int, int);
void main()
{
    int b,d;
    printf("Enter a number in binary number ");
    scanf("%d",&b);
    d=intodecimal(b);
    printf("The decimal of binary number %d is %d\n", b, d);
    intohexa(b,d);
}
int intodecimal(int bin)
{
    int deci, remainder,exp,j;
    while(bin >0)
    {
        remainder=bin%10;
        push(remainder);
        bin=bin/10;
```

```
        }
        deci=0;
        exp=top;
        while(top >=0)
        {
            j=pop();
            deci=deci+j*pow(2,exp);
            exp--;
        }
        return (deci);
    }
void intohexa(int bin, int deci)
{
    int remainder,j;
    while(deci >0)
    {
        remainder=deci%16;
        push(remainder);
        deci=deci/16;
    }
    printf("The hexa decimal format of binary number %d is ",bin);
    while(top >=0)
    {
        j=pop();
        if(j<10)printf("%d",j);
        else printf("%c",prnhexa(j));
    }
}
void push(int m)
{
    top++;
    stack[top]=m;
}
int pop()
{
    int j;
    if(top===-1) return (top);
    j=stack[top];
    top--;
    return(j);
}
char prnhexa(int v)
{
    switch(v)
    {
        case 10: return ('A');
                    break;
        case 11: return ('B');
```

```

        break;
    case 12: return ('C');
        break;
    case 13: return ('D');
        break;
    case 14: return ('E');
        break;
    case 15: return ('F');
        break;
}
}

```

Now, let's go behind the scenes.

How it works...

The first step is to convert the binary number into a decimal number. To do so, we will separate all the binary digits and multiply each by 2 raised to the power of their position in the binary number. We will then apply the mod **10** operator in order to separate the binary number into individual digits. Every time mod **10** is applied to the binary number, its last digit is separated and then pushed to the stack.

Let's assume that the binary number that we need to convert into a hexadecimal format is **110001**. We will apply the mod **10** operator to this binary number. The mod operator divides the number and returns the remainder. On application of the mod **10** operator, the last binary digit—in other words the rightmost digit will be returned as the remainder (as is the case with all divisions by **10**).

The operation is pushed in the stack at the location indicated by the **top** pointer. The value of **top** is initially -1. Before pushing to the stack, the value of **top** is incremented by 1. So, the value of **top** increments to 0 and the binary digit that appeared as the remainder (in this case, 1) is pushed to **stack[0]** (see *Figure 3.18*), and **11000** is returned as the quotient:

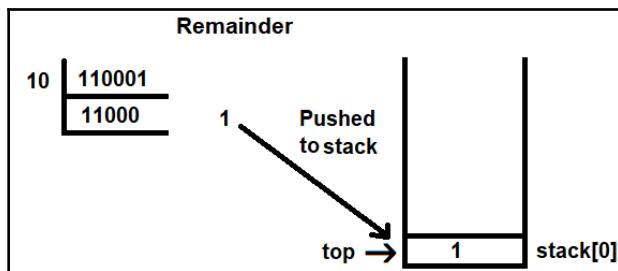


Figure 3.18

We will again apply the mod 10 operator to the quotient to separate the last digit of the present binary number. This time, 0 will be returned as the remainder and 1100 as the quotient on the application of the mod 10 operator. The remainder is again pushed to the stack. As mentioned before, the value of **top** is incremented before applying the push operation. As the value of **top** was 0, it is incremented to 1 and our new remainder, 0, is pushed to **stack[1]**:

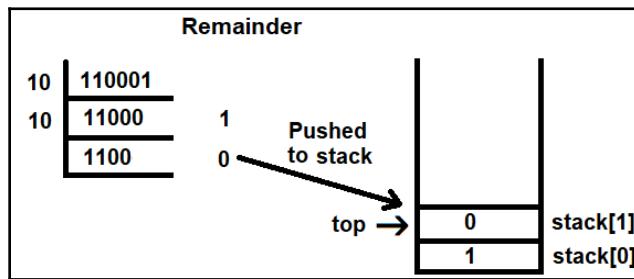


Figure 3.19

We will repeat this procedure until all the digits of the binary number are separated and pushed to the stack, as follows:

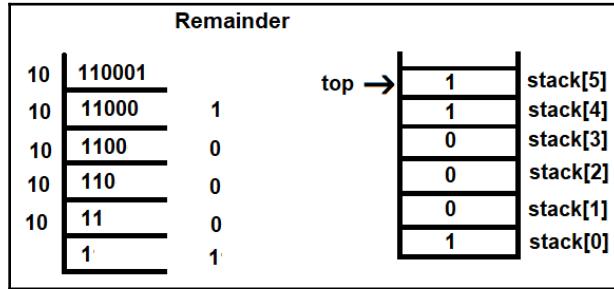


Figure 3.20

Once that's done, the next step is to pop the digits out one by one and multiply every digit by 2 raised to the power of **top**. For example, 2 raised to the power of **top** means 2 will be raised to the value of the index location from where the binary digit was popped off. The value from the stack is popped out from the location indicated by **top**.

The value of **top** is currently 5, hence the element at **stack[5]** will be popped out and multiplied by 2 raised to the power 5, as follows:

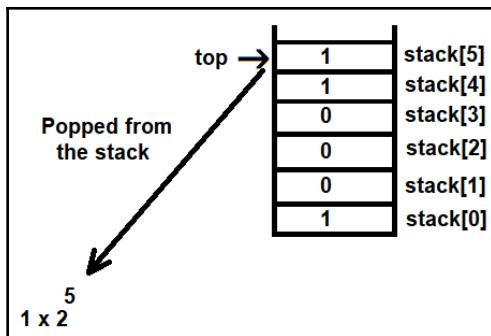


Figure 3.21

After popping a value from the stack, the value of `top` is decremented by 1 to point at the next element to be popped out. The procedure is repeated until every digit is popped out and multiplied by 2 raised to the power of its top location value. *Figure 3.19* shows how all the binary digits are popped from the stack and multiplied by 2 raised to the power of `top`:

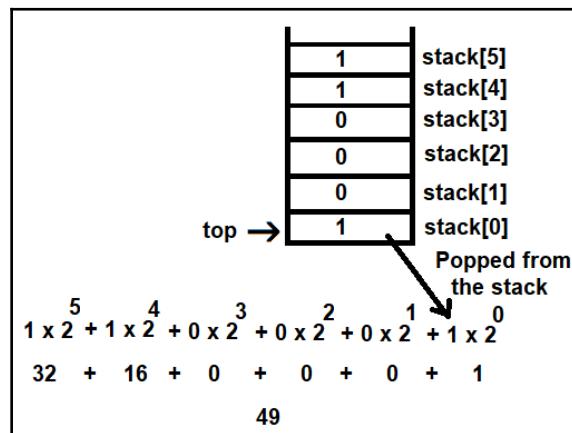


Figure 3.22

The resulting number we get is the decimal equivalent of the binary number that was entered by the user.

Now, to convert a decimal number into a hexadecimal format, we will divide it by 16. We need to keep dividing the number until the quotient becomes smaller than 16. The remainders of the division are displayed in LIFO order. If the remainder is below 10, it is displayed as is; otherwise, its equivalent letter is displayed. You can use the preceding table to find the equivalent letter if you get a remainder between 10 and 15.

In the following figure, you can see that the decimal number **49** is divided by **16**. The remainders are displayed in LIFO order to display the hexadecimal, hence 31 is the hexadecimal of the binary number **110001**. You don't need to apply the preceding table as both the remainders are less than 10:

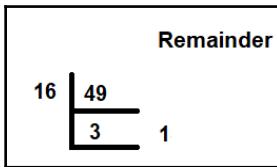


Figure 3.23

Let's use GCC to compile the `binaryintohexa.c` program, as follows:

```
D:\CBook>gcc binaryintohexa.c -o binaryintohexa
```

Here is one output of the program:

```
D:\CBook>./binaryintohexa
Enter a number in binary number 110001
The decimal of binary number 110001 is 49
The hexa decimal format of binary number 110001 is 31
```

Here is another output of the program:

```
D:\CBook>./binaryintohexa
Enter a number in binary number 11100
The decimal of binary number 11100 is 28
The hexa decimal format of binary number 11100 is 1C
```

Voilà! We've successfully converted a binary number into a hexadecimal number.

Now, let's move on to the next recipe!

Finding whether a number is a palindrome

A palindrome number is one that appears the same when read forward and backward. For example, 123 is not a palindrome but 737 is. To find out whether a number is a palindrome, we need to split it into separate digits and convert the unit of the original number to hundred and the hundred to unit.

For example, a pqr number will be called a **palindrome number** if $pqr=rqp$. And pqr will be equal to rqp only if the following is true:

$$p \times 100 + q \times 10 + r = r \times 100 + q \times 10 + p$$

In other words, we will have to multiply the digit in the unit place by 10^2 to convert it into the hundreds and convert the digit in the hundreds place to unit by multiplying it by 1. If the result matches the original number, it is a palindrome.

How to do it...

1. Enter a number to assign to the n variable:

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

2. Invoke the findpalindrome function and pass the number in the n variable to it as an argument:

```
findpalindrome(n)
```

3. The n argument is assigned to the numb parameter in the findpalindrome function. We need to separate each digit of the number; to do so, we will execute a while loop for the time the value in the numb variable is greater than 0:

```
while (numb >0)
```

4. Within the while loop, we will apply mod 10 on the number. On application of the mod 10 operator, we will get the remainder, which is basically the last digit of the number:

```
remainder=numb%10;
```

5. Push that remainder to the stack:

```
push(remainder);
```

6. Because the last digit of the number is separated, we need to remove the last digit from the existing number. That is done by dividing the number by 10 and truncating the fraction. The while loop will terminate when the number is individually divided into separate digits and all the digits are pushed to the stack:

```
numb=numb/10;
```

7. The number at the top of the stack will be the hundred and the one at the bottom of the stack will be the unit of the original number. Recall that we need to convert the hundred of the original number to the unit and vice versa. Pop all the digits out from the stack one by one and multiply each of them by 10 raised to a power. The power will be 0 for the first digit that is popped off. The power will be incremented with every value that is popped off. After being multiplied by 10 raised to the respective power, the digits are added into a separate variable, called value:

```
j=pop();  
value=value+j*pow(10,count);  
count++;
```

8. If the numbers in the numb and value variables match, that means the number is a palindrome. If the number is a palindrome, the `findpalindrome` function will return a value of 1, otherwise it will return a value of 0:

```
if(numb==value) return (1);  
else return (0);
```

The `findpalindrome.c` program determines whether the entered number is a palindrome number:

```
//Find out whether the entered number is a palindrome or not  
# include <stdio.h>  
#include <math.h>  
#define max 10  
int top=-1;  
int stack[max];  
void push();  
int pop();  
int findpalindrome(int);  
void main()  
{  
    int n;  
    printf("Enter a number ");  
    scanf("%d",&n);  
    if(findpalindrome(n))
```

```
        printf("%d is a palindrome number",n);
    else
        printf("%d is not a palindrome number", n);
}
int findpalindrome(int numb)
{
    int j, value, remainder, temp,count;
    temp=numb;
    while(numb >0)
    {
        remainder=numb%10;
        push(remainder);
        numb=numb/10;
    }
    numb=temp;
    count=0;
    value=0;
    while(top >=0)
    {
        j=pop();
        value=value+j*pow(10,count);
        count++;
    }
    if(numb==value) return (1);
    else return (0);
}
void push(int m)
{
    top++;
    stack[top]=m;
}
int pop()
{
    int j;
    if(top== -1) return (top);
    else
    {
        j=stack[top];
        top--;
        return(j);
    }
}
```

Now, let's go behind the scenes.

How it works...

Let's assume that the number we entered is 737. Now, we want to know whether 737 is a palindrome. We will start by applying the mod 10 operator on 737. On application, we will receive the remainder, 7, and the quotient, 73. The remainder, 7, will be pushed to the stack. Before pushing to the stack, however, the value of the **top** pointer is incremented by 1. The value of **top** is -1 initially; it is incremented to 0 and the remainder of 7 is pushed to **stack[0]** (see *Figure 3.21*).

The mod 10 operator returns the last digit of the number as the remainder. The quotient that we get on the application of the mod 10 operator is the original number with its last digit removed. That is, the quotient that we will get on the application of mod 10 operator on 737 is 73:

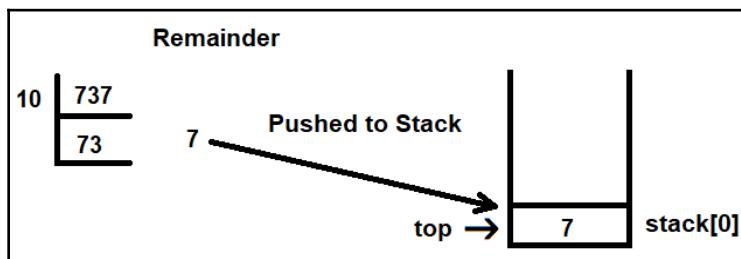


Figure 3.24

To the quotient, 73, we will apply the mod 10 operator again. The remainder will be the last digit, which is 3, and the quotient will be 7. The value of **top** is incremented by 1, making its value 1, and the remainder is pushed to **stack[1]**. To the quotient, 7, we will again apply the mod 10 operator. Because 7 cannot be divided by 10, 7 itself is returned and is pushed to the stack. Again, before the push operation, the value of **top** is incremented by 1, making its value 2. The value of 7 will be pushed to **stack[2]**:

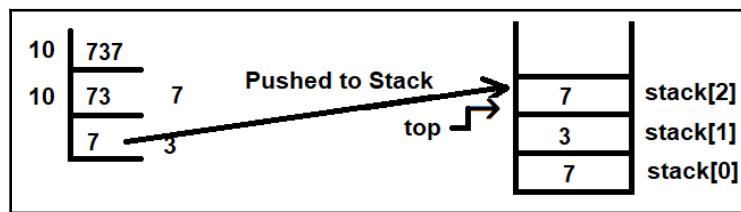


Figure 3.25

After separating the number into individual digits, we need to pop each digit from the stack one by one and multiply each one by **10** raised to a power. The power will be **0** for the topmost digit on the stack and will increment by 1 after every `pop` operation. The digit that will be popped from the stack will be the one indicated to by the `top` pointer. The value of `top` is **2**, so the digit on `stack[2]` is popped out and is multiplied by **10** raised to power of **0**:

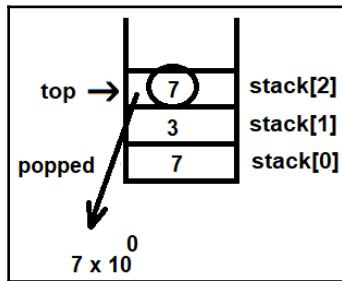


Figure 3.26

After every `pop` operation, the value of `top` is decremented by 1 and the value of the power is incremented by 1. The next digit that will be popped out from the stack is the one on `stack[1]`. That is, **3** will be popped out and multiplied by **10** raised to the power of **1**. Thereafter, the value of `top` will be decremented by 1, that is, the value of `top` will become **0**, and the value of the power will be incremented by 1, that is, the value of the power that was **1** will be incremented to **2**. The digit on `stack[0]` will be popped out and multiplied by **10** raised to the power of **2**:

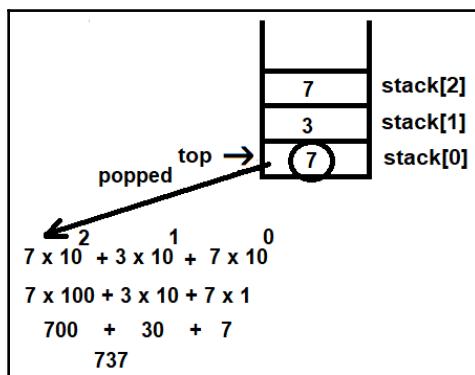


Figure 3.27

All the digits that are multiplied by **10** raised to the respective power are then summed. Because the result of the computation matches the original number, **737** is a palindrome.

Let's use GCC to compile the `findpalindrome.c` program, as shown in the following statement:

```
D:\CBook>gcc findpalindrome.c -o findpalindrome
```

Let's check whether 123 is a palindrome number:

```
D:\CBook>./findpalindrome
Enter a number 123
123 is not a palindrome number
```

Let's check whether 737 is a palindrome number:

```
D:\CBook>./findpalindrome
Enter a number 737
737 is a palindrome number
```

Voilà! We've successfully determined whether a number was a palindrome.

2

Section 2: Pointers and Files

In this section, we will take a closer look at the use of pointers and the various implementations available to us for using pointers effectively in C. We will also look at various recipes to better understand and handle files and their contents.

The following chapters will be covered in this section:

- Chapter 4, *Deep Dive into Pointers*
- Chapter 5, *File Handling*

4

Deep Dive into Pointers

Pointers have been the popular choice among programmers when it comes to using memory in an optimized way. Pointers have made it possible to access the content of any variable, array, or data type. You can use pointers for low-level access to any content and improve the overall performance of an application.

In this chapter, we will look at the following recipes on pointers:

- Reversing a string using pointers
- Finding the largest value in an array using pointers
- Sorting a singly linked list
- Finding the transpose of a matrix using pointers
- Accessing a structure using a pointer

Before we start with the recipes, I would like to discuss a few things related to how pointers work.

What is a pointer?

A pointer is a variable that contains the memory address of another variable, array, or string. When a pointer contains the address of something, it is said to be pointing at that thing. When a pointer points at something, it receives the right to access the content of that memory address. The question now is—why do we need pointers at all?

We need them because they do the following:

- Facilitate the dynamic allocation of memory
- Provide an alternative way to access a data type (apart from variable names, you can access the content of a variable through pointers)
- Make it possible to return more than one value from a function

For example, consider an *i* integer variable:

```
int i;
```

When you define an integer variable, two bytes will be allocated to it in memory. This set of two bytes can be accessed by a memory address. The value assigned to the variable is stored inside that memory location, as shown in the following diagram:

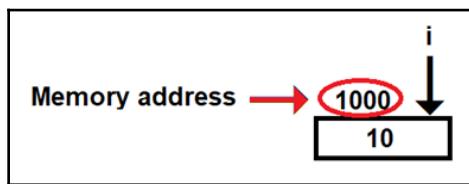


Figure 4.1

In the preceding diagram, **1000** represents the memory address of the *i* variable. Though, in reality, memory address is quite big and is in hex format, for the sake of simplicity, I am taking a small integer number, **1000**. The value of **10** is stored inside the memory address, **1000**.

Now, a *j* integer pointer can be defined as follows:

```
int *j;
```

This *j* integer pointer can point to the *i* integer through the following statement:

```
j=&i;
```

The **&** (ampersand) symbol represents the address, and the address of *i* will be assigned to the *j* pointer, as shown in the following diagram. The **2000** address is assumed to be the address of the *j* pointer and the address of the *i* pointer, that is, **1000**, is stored inside the memory location assigned to the *j* pointer, as shown in the following diagram:

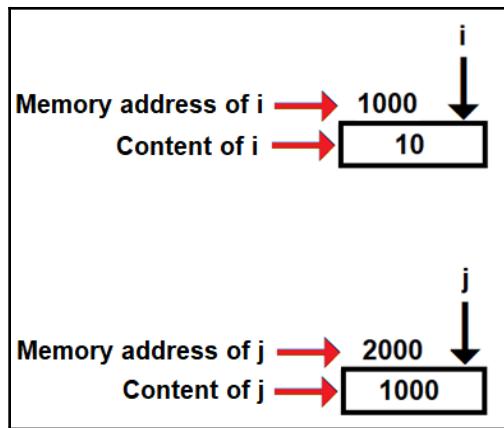


Figure 4.2

The address of the *i* integer can be displayed by the following statements:

```
printf("Address of i is %d\n", &i);  
printf("Address of i is %d\n", j);
```

To display the contents of *i*, we can use the following statements:

```
printf("Value of i is %d\n", i);  
printf("Value of i is %d\n", *j);
```



In the case of pointers, & (ampersand) represents the memory address and * (asterisk) represents content in the memory address.

We can also define a pointer to an integer pointer by means of the following statement:

```
int **k;
```

This pointer to a *k* integer pointer can point to a *j* integer pointer using the following statement:

```
k=&j;
```

Through the previous statement, the address of the **j** pointer will be assigned to the pointer to a **k** integer pointer, as shown in the following diagram. The value of **3000** is assumed to be the memory address of **k**:

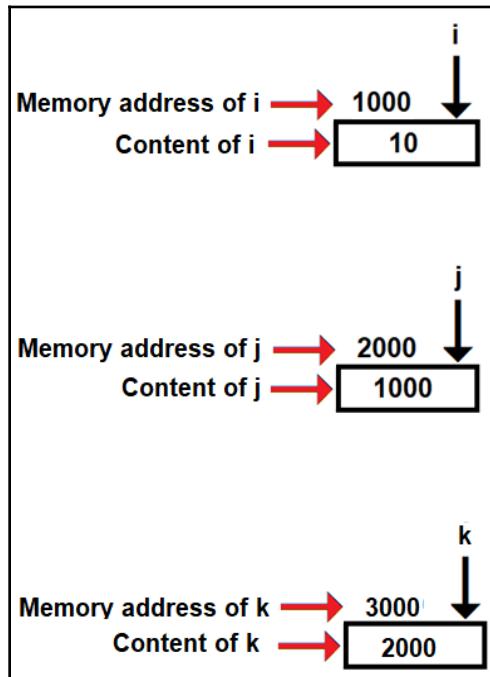


Figure 4.3

Now, when you display the value of **k**, it will display the address of **j**:

```
printf("Address of j = %d %d \n", &j, k);
```

To display the address of **i** through **k**, we need to use ***k**, because ***k** means that it will display the contents of the memory address pointed at by **k**. Now, **k** is pointing at **j** and the content in **j** is the address of **i**:

```
printf("Address of i = %d %d %d\n", &i, j, *k);
```

Similarly, to display the value of **i** through **k**, ****k** has to be used as follows:

```
printf("Value of i is %d %d %d %d \n", i, *(&i), *j, **k);
```

Using pointers enables us to access content precisely from desired memory locations. But allocating memory through pointers and not releasing it when the job is done may lead to a problem called **memory leak**. A memory leak is a sort of resource leak. A memory leak can allow unauthorized access of the memory content to hackers and may also block some content from being accessed even though it is present.

Now, let's begin with the first recipe of this chapter.

Reversing a string using pointers

In this recipe, we will learn to reverse a string using pointers. The best part is that we will not reverse the string and copy it onto another string, but we will reverse the original string itself.

How to do it...

1. Enter a string to assign to the `str` string variable as follows:

```
printf("Enter a string: ");
scanf("%s", str);
```

2. Set a pointer to point at the string, as demonstrated in the following code. The pointer will point at the memory address of the string's first character:

```
ptr1=str;
```

3. Find the length of the string by initializing an `n` variable to 1. Set a `while` loop to execute when the pointer reaches the null character of the string as follows:

```
n=1;
while(*ptr1 !='\0')
{
```

4. Inside the `while` loop, the following actions will be performed:

- The pointer is moved one character forward.
- The value of the `n` variable is incremented by 1:

```
ptr1++;
n++;
```

5. The pointer will be at the null character, so move the pointer one step back to make it point at the last character of the string as follows:

```
ptr1--;
```

6. Set another pointer to point at the beginning of the string as follows:

```
ptr2=str;
```

7. Exchange the characters equal to half the length of the string. To do that, set a while loop to execute for $n/2$ times, as demonstrated in the following code snippet:

```
m=1;  
while (m<=n/2)
```

8. Within the while loop, the first exchange operations take place; that is, the characters pointed at by our pointers are exchanged:

```
temp=*ptr1;  
*ptr1=*ptr2;  
*ptr2=temp;
```

9. After the character exchange, set the second pointer to move forward to point at its next character, that is, at the second character of the string, and move the first pointer backward to make it point at the second to last character as follows:

```
ptr1--;  
ptr2++;
```

10. Repeat this procedure for $n/2$ times, where n is the length of the string. When the while loop is finished, we will have the reverse form of the original string displayed on the screen:

```
printf("Reverse string is %s", str);
```

The `reversestring.c` program for reversing a string using pointers is as follows:

```
#include <stdio.h>  
void main()  
{  
    char str[255], *ptr1, *ptr2, temp ;  
    int n,m;  
    printf("Enter a string: ");  
    scanf("%s", str);  
    ptr1=str;  
    n=1;
```

```
while(*ptr1 != '\0')
{
    ptr1++;
    n++;
}
ptr1--;
ptr2=str;
m=1;
while(m<=n/2)
{
    temp=*ptr1;
    *ptr1=*ptr2;
    *ptr2=temp;
    ptr1--;
    ptr2++;
    m++;
}
printf("Reverse string is %s", str);
```

Now, let's go behind the scenes.

How it works...

We will be prompted to enter a string that will be assigned to the `str` variable. A string is nothing but a character array. Assuming we enter the name `manish`, each character of the name will be assigned to a location in the array one by one (see *Figure 4.4*). We can see that the first character of the string, the letter `m`, is assigned to the `str[0]` location, followed by the second string character being assigned to the `str[1]` location, and so on. The null character, as usual, is at the end of the string, as shown in the following diagram:

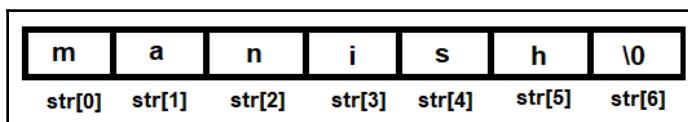


Figure 4.4

To reverse the string, we will seek the help of two pointers: one will be set to point at the first character of the string, and the other at the final character of the string. So, the first **ptr1** pointer is set to point at the first character of the string as follows:

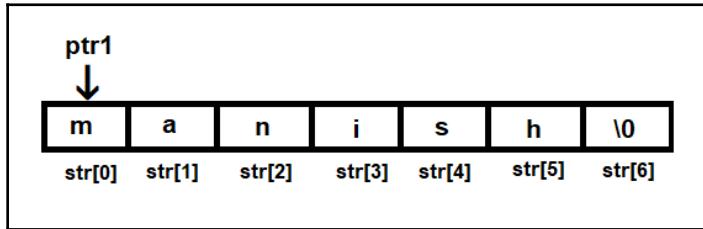


Figure 4.5

The exchanging of the characters has to be executed equal to half the length of the string; therefore, the next step will be to find the length of the string. After finding the string's length, the **ptr1** pointer will be set to move to the final character of the string.

In addition, another **ptr2** pointer is set to point at `m`, the first character of the string, as shown in the following diagram:

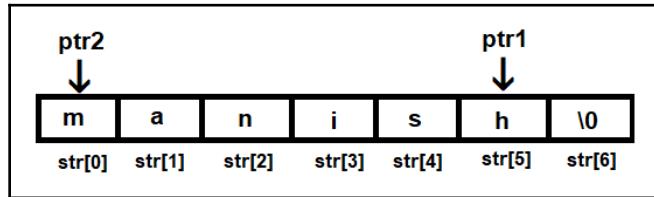


Figure 4.6

The next step is to interchange the first and last characters of the string that are being pointed at by the **ptr1** and **ptr2** pointers (see *Figure 4.7 (a)*). After interchanging the characters pointed at by the **ptr1** and **ptr2** pointers, the string will appear as shown in *Figure 4.7 (b)*:

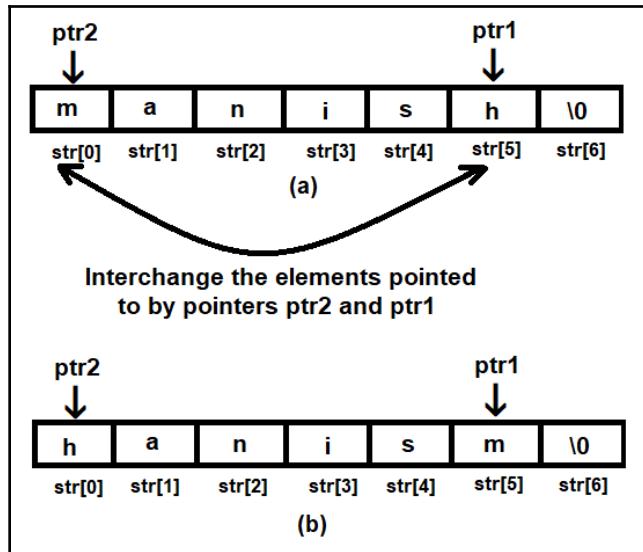


Figure 4.7

After interchanging the first and last characters, we will interchange the second and the second to last characters of the string. To do so, the **ptr2** pointer will be moved forward and set to point at the next character in line, and the **ptr1** pointer will be moved backward and set to point at the second to last character.

You can see in the following *Figure 4.8 (a)* that the **ptr2** and **ptr1** pointers are set to point at the **a** and **s** characters. Once this is done, another interchanging of the characters pointed at by **ptr2** and **ptr1** will take place. The string will appear as (*Figure 4.8 (b)*) after the interchanging of the **a** and **s** characters:

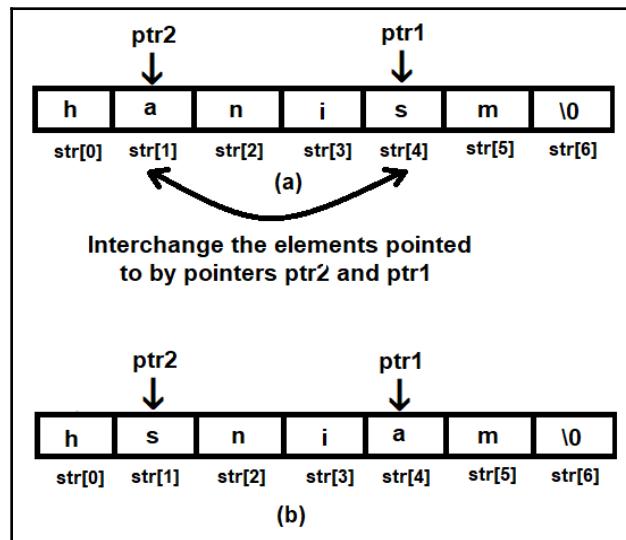


Figure 4.8

The only task now left in reversing the string is to interchange the third and the third to last character. So, we will repeat the relocation process of the **ptr2** and **ptr1** pointers. Upon interchanging the **n** and **i** characters of the string, the original **str** string will have been reversed, as follows:

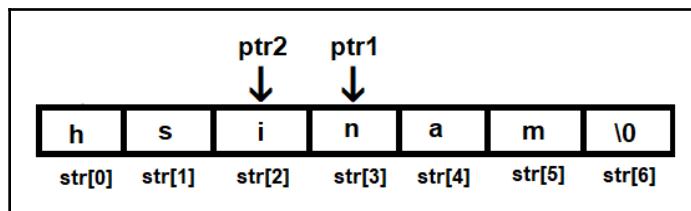


Figure 4.9

After applying the preceding steps, if we print the **str** string, it will appear in reverse.

Let's use GCC to compile the `reversestring.c` program as follows:

```
D:\CBook>gcc reversestring.c -o reversestring
```

If you get no errors or warnings, that means the `reversestring.c` program has been compiled into an executable file, called `reversestring.exe`. Let's run this executable file as follows:

```
D:\CBook>./reversestring  
Enter a string: manish  
Reverse string is hsinam
```

Voilà! We've successfully reversed a string using pointers. Now, let's move on to the next recipe!

Finding the largest value in an array using pointers

In this recipe, all the elements of the array will be scanned using pointers.

How to do it...

1. Define a macro by the name `max` with a size of 100 as follows:

```
#define max 100
```

2. Define a `p` integer array of a `max` size, as demonstrated in the following code:

```
int p[max]
```

3. Specify the number of elements in the array as follows:

```
printf("How many elements are there? ");  
scanf("%d", &n);
```

4. Enter the elements for the array as follows:

```
for(i=0;i<n;i++)  
    scanf("%d",&p[i]);
```

5. Define two `mx` and `ptr` pointers to point at the first element of the array as follows:

```
mx=p;
ptr=p;
```

6. The `mx` pointer will always point at the maximum value of the array, whereas the `ptr` pointer will be used for comparing the remainder of the values of the array. If the value pointed to by the `mx` pointer is smaller than the value pointed at by the `ptr` pointer, the `mx` pointer is set to point at the value pointed at by `ptr`. The `ptr` pointer will then move to point at the next array element as follows:

```
if (*mx < *ptr)
    mx=ptr;
```

7. If the value pointed at by the `mx` pointer is larger than the value pointed to by the `ptr` pointer, the `mx` pointer is undisturbed and is left to keep pointing at the same value and the `ptr` pointer is moved further to point at the next array element for the following comparison:

```
ptr++;
```

8. This procedure is repeated until all the elements of the array (pointed to by the `ptr` pointer) are compared with the element pointed to by the `mx` pointer. Finally, the `mx` pointer will be left pointing at the maximum value in the array. To display the maximum value of the array, simply display the array element pointed to by the `mx` pointer as follows:

```
printf("Largest value is %d\n", *mx);
```

The `largestinarray.c` program for finding out the largest value in an array using pointers is as follows:

```
#include <stdio.h>
#define max 100
void main()
{
    int p[max], i, n, *ptr, *mx;
    printf("How many elements are there? ");
    scanf("%d", &n);
    printf("Enter %d elements \n", n);
    for(i=0;i<n;i++)
        scanf("%d",&p[i]);
    mx=p;
    ptr=p;
```

```

for(i=1;i<n;i++)
{
    if (*mx < *ptr)
        mx=ptr;
    ptr++;
}
printf("Largest value is %d\n", *mx);
}

```

Now, let's go behind the scenes.

How it works...

Define an array of a certain size and enter a few elements in it. These will be the values among which we want to find the largest value. After entering a few elements, the array might appear as follows:

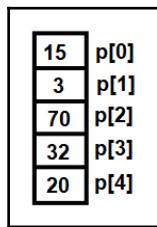


Figure 4.10

We will use two pointers for finding the largest value in the array. Let's name the two pointers **mx** and **ptr**, where the **mx** pointer will be used to point at the maximum value of the array, and the **ptr** pointer will be used for comparing the rest of the array elements with the value pointed at by the **mx** pointer. Initially, both the pointers are set to point at the first element of the array, **p[0]**, as shown in the following diagram:

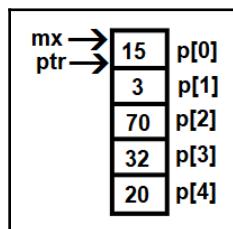


Figure 4.11

The **ptr** pointer is then moved to point at the next element of the array, **p[1]**. Then, the values pointed at by the **mx** and **ptr** pointers are compared. This process continues until all the elements of the array have been compared as follows:

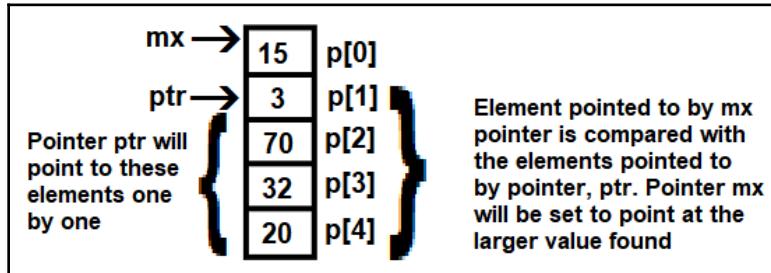


Figure 4.12

Recall that we want the **mx** pointer to keep pointing at the larger value. Since 15 is greater than 3 (see *Figure 4.13*), the position of the **mx** pointer will be left undisturbed, and the **ptr** pointer will be moved to point at the next element, **p[2]**, as follows:

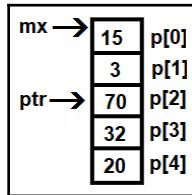


Figure 4.13

Again, the values pointed at by the **mx** and **ptr** pointers, which are the values 15 and 70 respectively, will be compared. Now, the value pointed at by the **mx** pointer is smaller than the value pointed at by the **ptr** pointer. So, the **mx** pointer will be set to point at the same array element as **ptr** as follows:

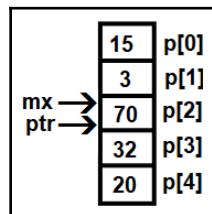


Figure 4.14

The comparison of the array elements will continue. The idea is to keep the **mx** pointer pointing at the largest element in the array, as shown in the following diagram:

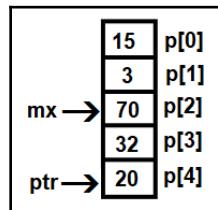


Figure 4.15

As shown in *Figure 4.15*, **70** is greater than **20**, so the **mx** pointer will remain at **p[2]**, and the **ptr** pointer will move to the next element, **p[4]**. Now, the **ptr** pointer is pointing at the last array element. So, the program will terminate, displaying the last value pointed at by the **mx** pointer, which also happens to be the largest value in the array.

Let's use GCC to compile the `largestinarray.c` program as the following statement:

```
D:\CBook>gcc largestinarray.c -o largestinarray
```

If you get no errors or warnings, that means that the `largestinarray.c` program has been compiled into an executable file, `largestinarray.exe`. Let's now run this executable file as follows:

```
D:\CBook>./largestinarray
How many elements are there? 5
Enter 5 elements
15
3
70
35
20
Largest value is 70
You can see that the program displays the maximum value in the array
```

Voilà! We've successfully found the largest value in an array using pointers. Now, let's move on to the next recipe!

Sorting a singly linked list

In this recipe, we will learn how to create a singly linked list comprising integer elements, and then we will learn how to sort this linked list in ascending order.

A singly linked list consists of several nodes that are connected through pointers. A node of a singly linked list might appear as follows:

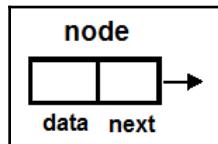


Figure 4.16

As you can see, a node of a singly linked list is a structure composed of two parts:

- **Data:** This can be one or more variables (also called members) of integer, float, string, or any data type. To keep the program simple, we will take **data** as a single variable of the integer type.
- **Pointer:** This will point to the structure of the type node. Let's call this pointer **next** in this program, though it can be under any name.

We will use bubble sort for sorting the linked list. Bubble sort is a sequential sorting technique that sorts by comparing adjacent elements. It compares the first element with the second element, the second element with the third element, and so on. The elements are interchanged if they are not in the preferred order. For example, if you are sorting elements into ascending order and the first element is larger than the second element, their values will be interchanged. Similarly, if the second element is larger than the third element, their values will be interchanged too.

This way, you will find that, by the end of the first iteration, the largest value will *bubble* down towards the end of the list. After the second iteration, the second largest value will be *bubbled* down to the end of the list. In all, $n-1$ iterations will be required to sort the n elements using bubble sort algorithm.

Let's understand the steps in creating and sorting a singly linked list.

How to do it...

1. Define a node comprising two members—`data` and `next`. The `data` member is for storing integer values and the `next` member is a pointer to link the nodes as follows:

```
struct node  
{
```

```
    int data;
    struct node *next;
};
```

2. Specify the number of elements in the linked list. The value entered will be assigned to the n variable as follows:

```
printf("How many elements are there in the linked list ?");
scanf("%d", &n);
```

3. Execute a for loop for n number of times. Within the for loop, a node is created by the name newNode. When asked, enter an integer value to be assigned to the data member of newNode as follows:

```
newNode=(struct node *)malloc(sizeof(struct node));
scanf("%d", &newNode->data);
```

4. Two pointers, startList and temp1, are set to point at the first node. The startList pointer will keep pointing at the first node of the linked list. The temp1 pointer will be used to link the nodes as follows:

```
startList = newNode;
temp1=startList;
```

5. To connect the newly created nodes, the following two tasks are performed:

- The next member of temp1 is set to point at the newly created node.
- The temp1 pointer is shifted to point at the newly created node as follows:

```
temp1->next = newNode;
temp1=newNode;
```

6. When the for loop gets over, we will have a singly linked list with its first node pointed at by startList, and the next pointer of the last node pointing at NULL. This linked list is ready to undergo the sorting procedure. Set a for loop to execute from 0 until n-2 that is equal to n-1 iterations as follows:

```
for (i=n-2;i>=0;i--)
```

7. Within the for loop, to compare values, use two pointers, temp1 and temp2. Initially, temp1 and temp2 will be set to point at the first two nodes of the linked list, as shown in the following code snippet:

```
temp1=startList;
temp2=temp1->next;
```

8. Compare the nodes pointed at by `temp1` and `temp2` in the following code:

```
if(temp1->data > temp2->data)
```

9. After comparing the first two nodes, the `temp1` and `temp2` pointers will be set to point at the second and third nodes, and so on:

```
temp1=temp2;
temp2=temp2->next;
```

10. The linked list has to be arranged in ascending order, so the data member of `temp1` must be smaller than the data member of `temp2`. In case the data member of `temp1` is larger than the data member of `temp2`, the interchanging of the values of the data members will be done with the help of a temporary variable, `k`, as follows:

```
k=temp1->data;
temp1->data=temp2->data;
temp2->data=k;
```

11. After $n-1$ performing iterations of comparing and interchanging consecutive values, if the first value in the pair is larger than the second, all the nodes in the linked list will be arranged in ascending order. To traverse the linked list and to display the values in ascending order, a temporary `t` pointer is set to point at the node pointed at by `startList`, that is, at the first node of the linked list, as follows:

```
t=startList;
```

12. A `while` loop is executed until the `t` pointer reaches `NULL`. Recall that the next pointer of the last node is set to `NULL`, so the `while` loop will execute until all the nodes of the linked list are traversed as follows:

```
while(t!=NULL)
```

13. Within the `while` loop, the following two tasks will be performed:

- The data member of the node pointed to by the `t` pointer is displayed.
- The `t` pointer is moved further to point at its next node:

```
printf("%d\t",t->data);
t=t->next;
```

The `sortlinkedlist.c` program for creating a singly linked list, followed by sorting it in ascending order, is as follows:

```
/* Sort the linked list by bubble sort */
#include<stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
void main()
{
    struct node *temp1,*temp2, *t,*newNode, *startList;
    int n,k,i,j;
    startList=NULL;
    printf("How many elements are there in the linked list ?");
    scanf("%d",&n);
    printf("Enter elements in the linked list\n");
    for(i=1;i<=n;i++)
    {
        if(startList==NULL)
        {
            newNode=(struct node *)malloc(sizeof(struct node));
            scanf("%d",&newNode->data);
            newNode->next=NULL;
            startList = newNode;
            temp1=startList;
        }
        else
        {
            newNode=(struct node *)malloc(sizeof(struct node));
            scanf("%d",&newNode->data);
            newNode->next=NULL;
            temp1->next = newNode;
            temp1=newNode;
        }
    }
    for(i=n-2;i>=0;i--)
    {
        temp1=startList;
        temp2=temp1->next;
        for(j=0;j<=i;j++)
        {
            if(temp1->data > temp2->data)
            {
                k=temp1->data;
                temp1->data=temp2->data;
```

```
        temp2->data=k;
    }
    temp1=temp2;
    temp2=temp2->next;
}
printf("Sorted order is: \n");
t=startList;
while(t!=NULL)
{
    printf("%d\t",t->data);
    t=t->next;
}
}
```

Now, let's go behind the scenes.

How it works...

This program is performed in two parts—the first part is the creation of a singly linked list, and the second part is the sorting of the linked list.

Let's start with the first part.

Creating a singly linked list

We will start by creating a new node by the name of **newNode**. When prompted, we will enter the value for its data member and then set the next **newNode** pointer to **NULL** (as shown in *Figure 4.17*). This next pointer will be used for connecting with other nodes (as we will see shortly):

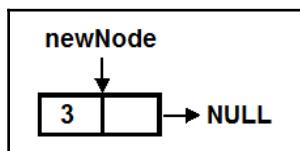


Figure 4.17

After the first node is created, we will make the following two pointers point at it as follows:

- **startList**: To traverse the singly linked list, we will need a pointer that points at the first node of the list. So, we will define a pointer called **startList** and set it to point at the first node of the list.
- **temp1**: In order to connect with the next node, we will need one more pointer. We will call this pointer **temp1**, and set it to point at the **newNode** (see *Figure 4.18*):

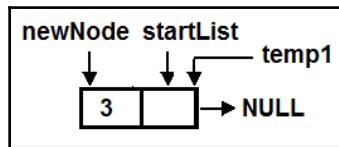


Figure 4.18

We will now create another node for the linked list and call that **newNode** as well. The pointer can point to only one structure at a time. So, the moment we create a new node, the **newNode** pointer that was pointing at the first node will now point at the recently created node. We will be prompted to enter a value for the data member of the new node, and its next pointer will be set to **NULL**.

You can see in the following diagram that the two pointers, **startList** and **temp1**, are pointing at the first node and the **newNode** pointer is pointing at the newly created node. As stated earlier, **startList** will be used for traversing the linked list and **temp1** will be used for connecting with the newly created node as follows:

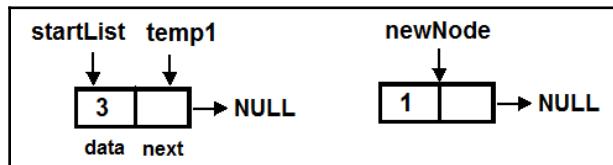


Figure 4.19

To connect the first node with **newNode**, the next pointer of **temp1** will be set to point at **newNode** (see *Figure 4.20 (a)*). After connecting with **newNode**, the **temp1** pointer will be moved further and set to point at **newNode** (see *Figure 4.20 (b)*) so that it can be used again for connecting with any new nodes that may be added to the linked list in future:

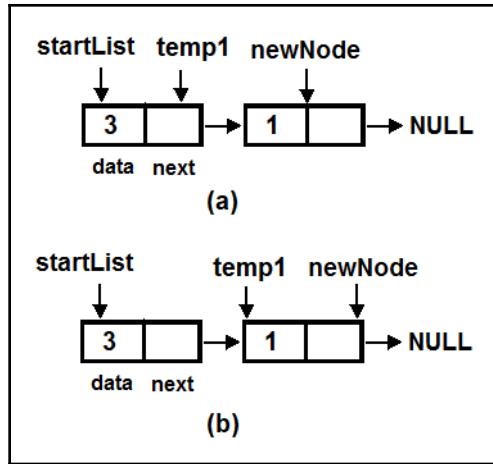


Figure 4.20

Steps three and four will be repeated for the rest of the nodes of the linked list. Finally, the singly linked list will be ready and will look something like this:

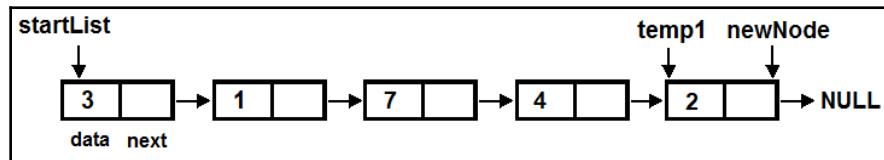


Figure 4.21

Now that we have created the singly linked list, the next step is to sort the linked list in ascending order.

Sorting the singly linked list

We will use the bubble sort algorithm for sorting the linked list. In the bubble sort technique, the first value is compared with the second value, the second is compared with the third value, and so on. If we want to sort our list in ascending order, then we will need to keep the smaller values toward the top when comparing the values.

Therefore, while comparing the first and second values, if the first value is larger than the second value, then their places will be interchanged. If the first value is smaller than the second value, then no interchanging will happen, and the second and third values will be picked up for comparison.

There will be $n-1$ iterations of such comparisons, meaning if there are five values, then there will be four iterations of such comparisons; and after every iteration, the last value will be left out—that is, it will not be compared as it reaches its destination. The destination here means the location where the value must be kept when arranged in ascending order.

The first iteration

To sort the linked list, we will employ the services of two pointers—**temp1** and **temp2**. The **temp1** pointer is set to point at the first node, and **temp2** is set to point at the next node as follows:

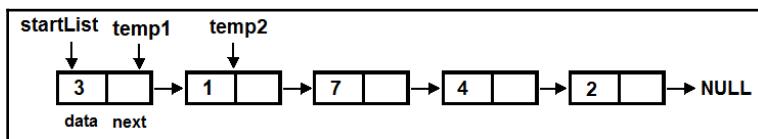


Figure 4.22

We will be sorting the linked list in ascending order, so we will keep the smaller values toward the beginning of the list. The data members of **temp1** and **temp2** will be compared. Because **temp1->data** is greater than **temp2->data**, that is, the data member of **temp1** is larger than the data member of **temp2**, their places will be interchanged (see the following diagram). After interchanging the data members of the nodes pointed at by **temp1** and **temp2**, the linked list will appear as follows:

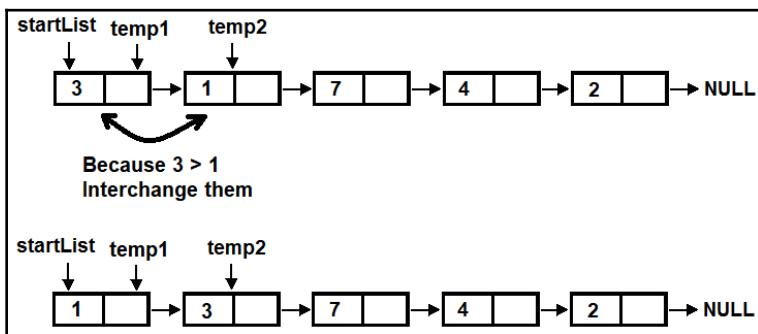


Figure 4.23

After this, the two pointers will shift further, that is, the **temp1** pointer will be set to point at **temp2**, and the **temp2** pointer will be set to point at its next node. We can see in *Figure 4.24 (a)* that the **temp1** and **temp2** pointers are pointing at the nodes with the values 3 and 7, respectively. We can also see that **temp1->data** is less than **temp2->data**, that is, $3 < 7$. Since the data member of **temp1** is already smaller than the data member of **temp2**, no interchanging of values will take place and the two pointers will simply move one step further (see *Figure 4.24 (b)*).

Now, because $7 > 4$, their places will be interchanged. The values of data members pointed at by **temp1** and **temp2** will interchange as follows (*Figure 4.24 (c)*):

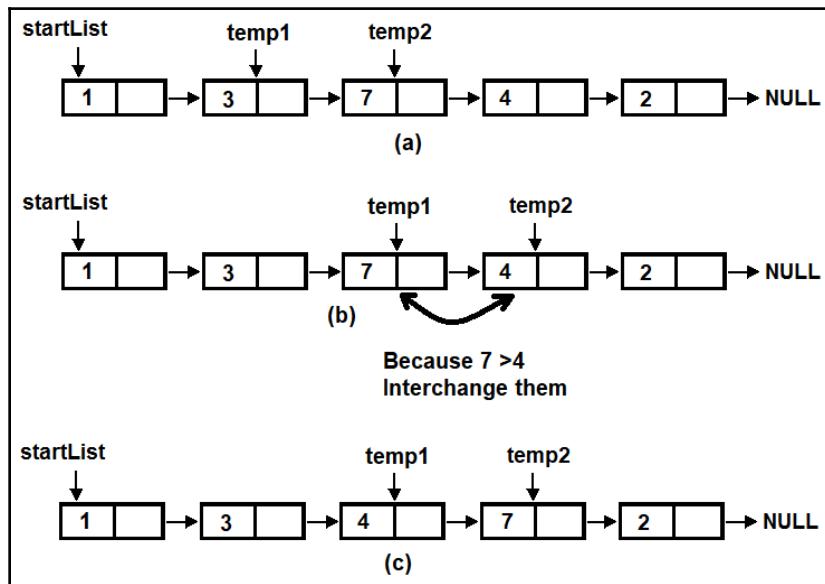


Figure 4.24

After that, the **temp1** and **temp2** pointer will be shifted one step further, that is, **temp1** will point at **temp2**, and **temp2** will move onto its next node. We can see in the following *Figure 4.25 (a)* that **temp1** and **temp2** are pointing at the nodes with the values 7 and 2, respectively. Again, the data members of **temp1** and **temp2** will be compared. Because **temp1->data** is greater than **temp2->data**, their places will be interchanged. *Figure 4.25 (b)* shows the linked list after interchanging values of the data members:

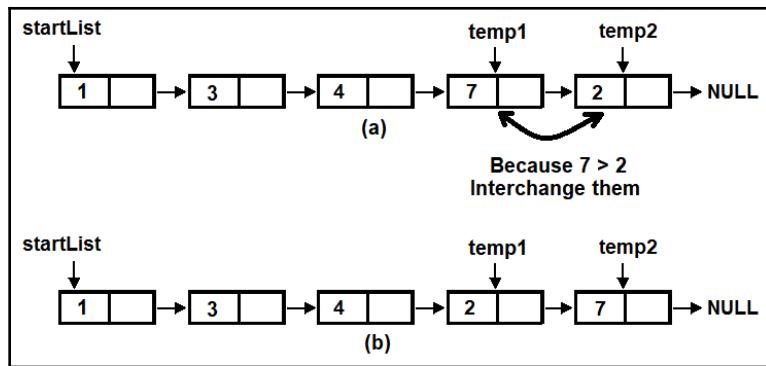


Figure 4.25

This was the first iteration, and you can notice that after this iteration, the largest value, 7, has been set to our desired location—at the end of the linked list. This also means that in the second iteration, we will not have to compare the last node. Similarly, after the second iteration, the second highest value will reach or is set to its actual location. The second highest value in the linked list is 4, so after the second iteration, the four node will just reach the seven node. How? Let's look at the second iteration of bubble sort.

The second iteration

We will begin the comparison by comparing first two nodes, so the **temp1** and **temp2** pointers will be set to point at the first and second nodes of the linked list, respectively (see *Figure 4.26 (a)*). The data members of **temp1** and **temp2** will be compared. As is clear, **temp1->data** is less than **temp2->data** (that is, $1 < 7$), so their places will not be interchanged. Thereafter, the **temp1** and **temp2** pointers will shift one step further. We can see in *Figure 4.26 (b)* that the **temp1** and **temp2** pointers are set to point at nodes of the values 3 and 4, respectively:

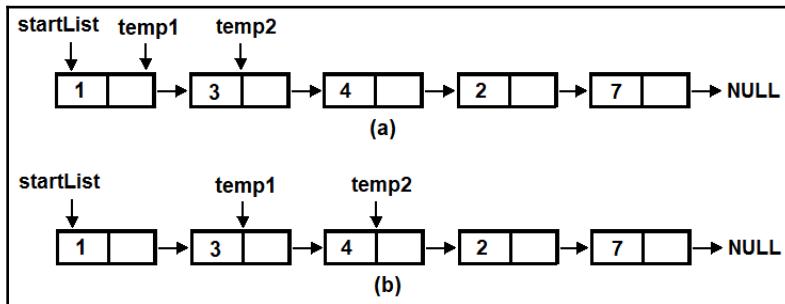


Figure 4.26

Once again, the data members of the **temp1** and **temp2** pointers will be compared. Because **temp1->data** is less than **temp2->data**, that is, $3 < 4$, their places will again not be interchanged and the **temp1** and **temp2** pointers will, again, shift one step further. That is, the **temp1** pointer will be set to point at **temp2**, and **temp2** will be set to point at its next node. You can see in *Figure 4.27 (a)* that the **temp1** and **temp2** pointers are set to point at nodes with the values 4 and 2, respectively. Because $4 > 2$, their places will be interchanged. After interchanging the place of these values, the linked list will appear as follows in *Figure 4.27 (b)*:

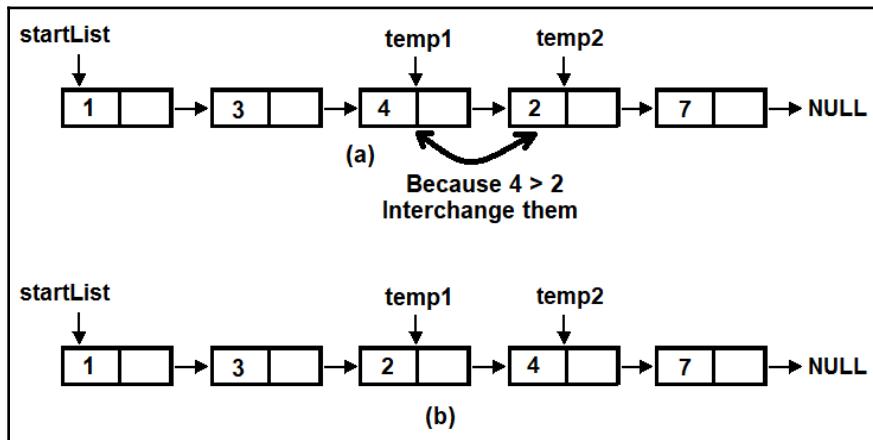


Figure 4.27

This is the end of the second iteration, and we can see that the second largest value, four, is set to our desired location as per ascending order. So, with every iteration, one value is being set at the required location. Accordingly, the next iteration will require one comparison less.

The third and fourth iterations

In the third iteration, we will only need to do the following comparisons:

1. Compare the first and second nodes
2. Compare the second and third nodes

After the third iteration, the third largest value, that is, three, will be set at our desired location, that is, just before node four.

In the fourth, and final, iteration, only the first and second nodes will be compared. The linked list will be sorted in ascending order as follows after the fourth iteration:

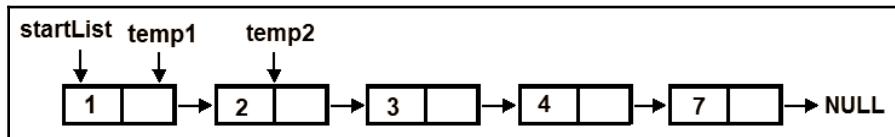


Figure 4.28

Let's use GCC to compile the `sortlinkedlist.c` program as follows:

```
D:\CBook>gcc sortlinkedlist.c -o sortlinkedlist
```

If you get no errors or warnings, that means that the `sortlinkedlist.c` program has been compiled into an executable file, `sortlinkedlist.exe`. Let's run this executable file as follows:

```
D:\CBook>./sortlinkedlist
How many elements are there in the linked list ?5
Enter elements in the linked list
3
1
7
4
2
Sorted order is:
1      2      3      4      7
```

Voilà! We've successfully created and sorted a singly linked list. Now, let's move on to the next recipe!

Finding the transpose of a matrix using pointers

The best part of this recipe is that we will not only display the transpose of the matrix using pointers, but we will also create the matrix itself using pointers.

The transpose of a matrix is a new matrix that has rows equal to the number of columns of the original matrix and columns equal to the number of rows. The following diagram shows you a matrix of order 2×3 and its transpose, of order 3×2 :

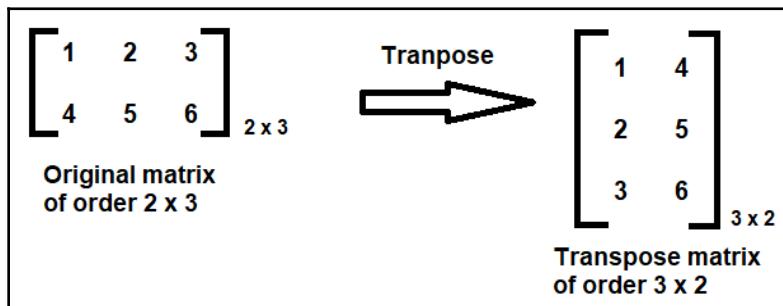


Figure 4.29

Basically, we can say that, upon converting the rows into columns and columns into rows of a matrix, you get its transpose.

How to do it...

1. Define a matrix of 10 rows and 10 columns as follows (you can have a bigger matrix if you wish):

```
int a[10][10]
```

2. Enter the size of the rows and columns as follows:

```
printf("Enter rows and columns of matrix: ");
scanf("%d %d", &r, &c);
```

3. Allocate memory locations equal to $r * c$ quantity for keeping the matrix elements as follows:

```
ptr = (int *)malloc(r * c * sizeof(int));
```

4. Enter elements of the matrix that will be assigned sequentially to each allocated memory as follows:

```
for(i=0; i<r; ++i)
{
    for(j=0; j<c; ++j)
    {
```

```

        scanf ("%d", &m);
        *(ptr + i*c + j) = m;
    }
}

```

5. In order to access this matrix via a pointer, set a `ptr` pointer to point at the first memory location of the allocated memory block, as shown in *Figure 4.30*. The moment that the `ptr` pointer is set to point at the first memory location, it will automatically get the address of the first memory location, so 1000 will be assigned to the `ptr` pointer:

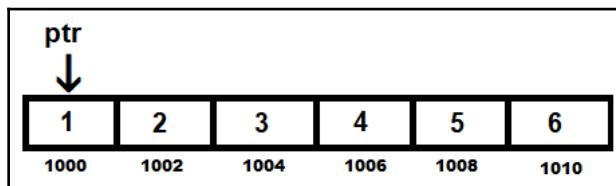


Figure 4.30

6. To access these memory locations and display their content, use the `* (ptr + i*c + j)` formula within the nested loop, as shown in this code snippet:

```

for(i=0; i<r; ++i)
{
    for(j=0; j<c; ++j)
    {
        printf("%d\t", * (ptr + i*c + j));
    }
    printf("\n");
}

```

7. The value of the `r` row is assumed to be two, and that of column `c` is assumed to be three. With values of `i=0` and `j=0`, the formula will compute as follows:

```

*(ptr + i*c + j);
*(1000+0*3+0)
*1000

```

It will display the content of the memory address, 1000.

When the value of $i=0$ and $j=1$, the formula will compute as follows:

```
* (ptr + i*c + j);
*(1000+0*3+1)
*(1000+1)
*(1002)
```

We will first get $*(1000+1)$, because the `ptr` pointer is an integer pointer, and it will jump two bytes every time we add the value 1 to it at every memory location, from which we will get $*(1002)$, and it will display the content of the memory location 1002.

Similarly, the value of $i=0$ and $j=2$ will lead to $*(1004)$; that is, the content of the memory location 1004 will be displayed. Using this formula, the value of $i=1$ and $j=0$ will lead to $*(1006)$; the value of $i=1$ and $j=1$ will lead to $*(1008)$; and the value of $i=1$ and $j=2$ will lead to $*(1010)$. So, when the aforementioned formula is applied within the nested loops, the original matrix will be displayed as follows:

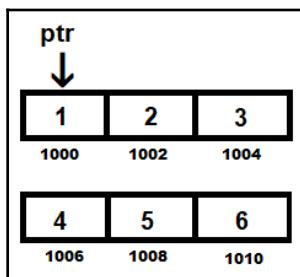


Figure 4.31

- To display the transpose of a matrix, apply the following formula within the nested loops:

```
* (ptr + j*c + i))
```

Again, assuming the values of row ($r=2$) and column ($c=3$), the following content of memory locations will be displayed:

i	j	Memory address
0	0	1000
0	1	1006
1	0	1002

1	1	1008
2	0	1004
2	1	1010

So, upon applying the preceding formula, the content of the following memory address will be displayed as the following in *Figure 4.32*. And the content of these memory addresses will comprise the transpose of the matrix:

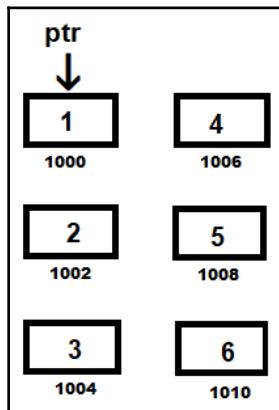


Figure 4.32

Let's see how this formula is applied in a program.

The `transposemat.c` program for displaying the transpose of a matrix using pointers is as follows:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int a[10][10], r, c, i, j, *ptr, m;
    printf("Enter rows and columns of matrix: ");
    scanf("%d %d", &r, &c);
    ptr = (int *)malloc(r * c * sizeof(int));
    printf("\nEnter elements of matrix:\n");
    for(i=0; i<r; ++i)
    {
        for(j=0; j<c; ++j)
        {
            scanf("%d", &m);
            *(ptr + i*c + j) = m;
        }
    }
}
```

```
    }
    printf("\nMatrix using pointer is: \n");
    for(i=0; i<r; ++i)
    {
        for(j=0; j<c; ++j)
        {
            printf("%d\t", *(ptr +i*c + j));
        }
        printf("\n");
    }
    printf("\nTranspose of Matrix:\n");
    for(i=0; i<c; ++i)
    {
        for(j=0; j<r; ++j)
        {
            printf("%d\t", *(ptr +j*c + i));
        }
        printf("\n");
    }
}
```

Now, let's go behind the scenes.

How it works...

Whenever an array is defined, the memory allocated to it internally is a sequential memory. Now let's define a matrix of size 2×3 , as shown in the following diagram. In that case, the matrix will be assigned six consecutive memory locations of two bytes each (see *Figure 4.33*). Why two bytes each? This is because an integer takes two bytes. This also means that if we define a matrix of the float type that takes four bytes, each allocated memory location would consist of four bytes:

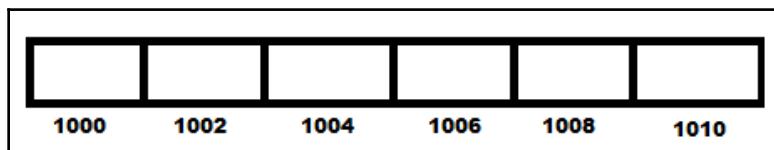


Figure 4.33

In reality, the memory address is long and is in hex format; but for simplicity, we will take the memory addresses of integer type and take easy-to-remember numbers, such as **1000**, as memory addresses. After memory address **1000**, the next memory address is **1002** (because an integer takes two bytes).

Now, to display the original matrix elements in row-major form using a pointer, we will need to display the elements of memory locations, **1000**, **1002**, **1004**, and so on:

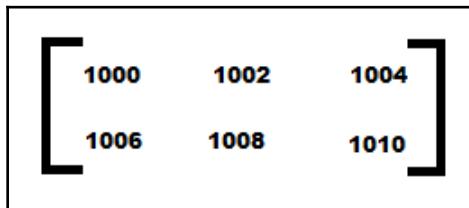


Figure 4.34

Similarly, in order to display the transpose of the matrix using a pointer, we will need to display the elements of memory locations; **1000**, **1006**, **1002**, **1008**, **1004**, and **1010**:

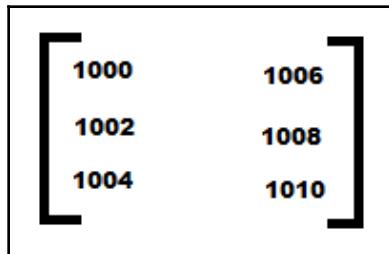


Figure 4.35

Let's use GCC to compile the `transposemat.c` program as follows:

```
D:\CBook>gcc transposemat.c -o transposemat
```

If you get no errors or warnings, that means that the `transposemat.c` program has been compiled into an executable file, `transposemat.exe`. Let's run this executable file with the following code snippet:

```
D:\CBook>./transposemat
Enter rows and columns of matrix: 2 3

Enter elements of matrix:
1
2
3
4
5
6
```

Matrix using pointer is:

```
1      2      3  
4      5      6
```

Transpose of Matrix:

```
1      4  
2      5  
3      6
```

Voilà! We've successfully found the transpose of a matrix using pointers. Now, let's move on to the next recipe!

Accessing a structure using a pointer

In this recipe, we will make a structure that stores the information of an order placed by a specific customer. A structure is a user-defined data type that can store several members of different data types within it. The structure will have members for storing the order number, email address, and password of the customer:

```
struct cart  
{  
    int orderno;  
    char emailaddress[30];  
    char password[30];  
};
```

The preceding structure is named `cart`, and comprises three members – `orderno` of the `int` type for storing the order number of the order placed by the customer, and `emailaddress` and `password` of the `string` type for storing the email address and password of the customer, respectively. Let's begin!

How to do it...

1. Define a `cart` structure by the name `mycart`. Also, define two pointers to structure of the `cart` structure, `ptrcart` and `ptrcust`, as shown in the following code snippet:

```
struct cart mycart;  
struct cart *ptrcart, *ptrcust;
```

2. Enter the order number, email address, and password of the customer, and these values will be accepted using the mycart structure variable. As mentioned previously, the dot operator (.) will be used for accessing the structure members, orderno, emailaddress, and password, through a structure variable as follows:

```
printf("Enter order number: ");
scanf("%d", &mycart.orderno);
printf("Enter email address: ");
scanf("%s", mycart.emailaddress);
printf("Enter password: ");
scanf("%s", mycart.password);
```

3. Set the pointer to the ptrcart structure to point at the mycart structure using the ptrcart=&mycart statement. Consequently, the pointer to the ptrcart structure will be able to access the members of the mycart structure by using the arrow (->) operator. By using ptrcart->orderno, ptrcart->emailaddress, and ptrcart->password, the values assigned to the orderno, emailaddress, and password structure members are accessed and displayed:

```
printf("\nDetails of the customer are as follows:\n");
printf("Order number : %d\n", ptrcart->orderno);
printf("Email address : %s\n", ptrcart->emailaddress);
printf("Password : %s\n", ptrcart->password);
```

4. We will also modify the email address and password of the customer by asking them to enter a new email address and password and accept the new details via the pointer to the ptrcart structure as follows. Because ptrcart is pointing to the mycart structure, the new email address and password will overwrite the existing values that were assigned to the structure members of mycart:

```
printf("\nEnter new email address: ");
scanf("%s", ptrcart->emailaddress);
printf("Enter new password: ");
scanf("%s", ptrcart->password);
/*The new modified values of orderno, emailaddress and password
members are displayed using structure variable, mycart using dot
operator (.)*/
printf("\nModified customer's information is:\n");
printf("Order number: %d\n", mycart.orderno);
printf("Email address: %s\n", mycart.emailaddress);
printf("Password: %s\n", mycart.password);
```

5. Then, define a pointer to the `*ptrcust` structure. Using the following `malloc` function, allocate memory for it. The `sizeof` function will find out the number of bytes consumed by each of the structure members and return the total number of bytes consumed by the structure as a whole:

```
ptrcust=(struct cart *)malloc(sizeof(struct cart));
```

6. Enter the order number, email address, and password of the customer, and all the values will be assigned to the respective structure members using a pointer to a structure as follows. Obviously, the arrow operator (`->`) will be used for accessing the structure members through a pointer to a structure:

```
printf("Enter order number: ");
scanf("%d", &ptrcust->orderno);
printf("Enter email address: ");
scanf("%s", ptrcust->emailaddress);
printf("Enter password: ");
scanf("%s", ptrcust->password);
```

7. The values entered by the user are then displayed through the pointer to the `ptrcust` structure again as follows:

```
printf("\nDetails of the second customer are as follows:\n");
printf("Order number : %d\n", ptrcust->orderno);
printf("Email address : %s\n", ptrcust->emailaddress);
printf("Password : %s\n", ptrcust->password);
```

The following `pointertostuct.c` program explains how to access a structure by using a pointer:

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

struct cart
{
    int orderno;
    char emailaddress[30];
    char password[30];
};

void main()
{
    struct cart mycart;
    struct cart *ptrcart, *ptrcust;
    ptrcart = &mycart;
    printf("Enter order number: ");
    scanf("%d", &mycart.orderno);
```

```
printf("Enter email address: ");
scanf("%s", mycart.emailaddress);
printf("Enter password: ");
scanf("%s", mycart.password);
printf("\nDetails of the customer are as follows:\n");
printf("Order number : %d\n", ptrcart->orderno);
printf("Email address : %s\n", ptrcart->emailaddress);
printf("Password : %s\n", ptrcart->password);

printf("\nEnter new email address: ");
scanf("%s", ptrcart->emailaddress);
printf("Enter new password: ");
scanf("%s", ptrcart->password);
printf("\nModified customer's information is:\n");
printf("Order number: %d\n", mycart.orderno);
printf("Email address: %s\n", mycart.emailaddress);
printf("Password: %s\n", mycart.password);

ptrcust=(struct cart *)malloc(sizeof(struct cart));
printf("\nEnter information of another customer:\n");
printf("Enter order number: ");
scanf("%d", &ptrcust->orderno);
printf("Enter email address: ");
scanf("%s", ptrcust->emailaddress);
printf("Enter password: ");
scanf("%s", ptrcust->password);
printf("\nDetails of the second customer are as follows:\n");
printf("Order number : %d\n", ptrcust->orderno);
printf("Email address : %s\n", ptrcust->emailaddress);
printf("Password : %s\n", ptrcust->password);
}
```

Now, let's go behind the scenes.

How it works...

When you define a variable of the type structure, that variable can access members of the structure in the following format:

structurevariable.structuremember

You can see a period (.) between the structure variable and the structure member. This period (.) is also known as a dot operator, or member access operator. The following example will make it clearer:

```
struct cart mycart;  
mycart.orderno
```

In the preceding code, you can see that `mycart` is defined as a structure variable of the `cart` structure. Now, the `mycart` structure variable can access the `orderno` member by making the member access operator (.).

You can also define a pointer to a structure. The following statement defines `ptrcart` as a pointer to the `cart` structure.

```
struct cart *ptrcart;
```

When the pointer to a structure points to a structure variable, it can access the structure members of the structure variable. In the following statement, the pointer to the `ptrcart` structure points at the address of the `mycart` structure variable:

```
ptrcart = &mycart;
```

Now, `ptrcart` can access the structure members, but instead of the dot operator (.), the arrow operator (->) will be used. The following statement accesses the `orderno` member of the structure using the pointer to a structure:

```
ptrcart->orderno
```

If you don't want a pointer to a structure to point at the structure variable, then memory needs to be allocated for a pointer to a structure to access structure members. The following statement defines a pointer to a structure by allocating memory for it:

```
ptrcust=(struct cart *)malloc(sizeof(struct cart));
```

The preceding code allocates memory equal to the size of a `cart` structure, typecasts that memory to be used by a pointer to a `cart` structure, and assigns that allocated memory to `ptrcust`. In other words, `ptrcust` is defined as a pointer to a structure, and it does not need to point to any structure variable, but can directly access the structure members.

Let's use GCC to compile the `pointertostruct.c` program as follows:

```
D:\CBook>gcc pointertostruct.c -o pointertostruct
```

If you get no errors or warnings, that means that the `pointertostruct.c` program has been compiled into an executable file, `pointertostruct.exe`. Let's run this executable file as follows:

```
D:\CBook>./pointertostruct
Enter order number: 1001
Enter email address: bmharwani@yahoo.com
Enter password: gold

Details of the customer are as follows:
Order number : 1001
Email address : bmharwani@yahoo.com
Password : gold

Enter new email address: harwanibm@gmail.com
Enter new password: diamond

Modified customer's information is:
Order number: 1001
Email address: harwanibm@gmail.com
Password: diamond

Enter information of another customer:
Enter order number: 1002
Enter email address: bintu@yahoo.com
Enter password: platinum

Details of the second customer are as follows:
Order number : 1002
Email address : bintu@yahoo.com
Password : platinum
```

Voilà! We've successfully accessed a structure using a pointer.

5

File Handling

Data storage is a mandatory feature in all applications. When we enter any data while running a program, that data is stored as RAM, which means that it is temporary in nature. We will not get that data back when running the program the next time. But what if we want the data to stay there so we can refer to it again when we need it? In this case, we have to store the data.

Basically, we want our data to be stored and to be accessible and available for reuse whenever required. In C, data storage can be done through traditional file handling techniques and through the database system. The following are the two types of file handling available in C:

- **Sequential file handling:** Data is written in a simple text format and can be read and written sequentially. To read the n th line, we have to first read $n-1$ lines.
- **Random file handling:** Data is written as bytes and can be read or written randomly. We can read or write any line randomly by positioning the file pointer at the desired location.

In this chapter, we will go through the following recipes using file handling:

- Reading a text file and converting all characters after a period into uppercase
- Displaying the contents of a random file in reverse order
- Counting the number of vowels in a file
- Replacing a word in a file with another word
- Encrypting a file

Before we start with the recipes, let's review some of the functions we will be using to create our recipes.

Functions used in file handling

I've divided this section into two parts. In the first part, we will look at the functions specific to the sequential file handling method. In the second, we will look at the functions used for random files.

Functions commonly used in sequential file handling

The following are some of the functions that are used to open, close, read, and write in a sequential file.

fopen()

The `fopen()` function is used for opening a file for reading, writing, and doing other operations. Here is its syntax:

```
FILE *fopen (const char *file_name, const char *mode)
```

Here, `file_name` represents the file that we want to work on and `mode` states the purpose for which we want to open the file. It can be any of the following:

- `r`: This opens the file in read mode and sets the file pointer at the first character of the file.
- `w`: This opens the file in write mode. If the file exists, it will be overwritten.
- `a`: Opens the file in append mode. Newly entered data will be added at the end of the file.
- `r+`: This opens the file in read and write mode. The file pointer is set to point at the beginning of the file. The file content will not be deleted if it already exists. It will not create a file if it does not already exist.
- `w+`: This also opens the file in read and write mode. The file pointer is set to point at the beginning of the file. The file content will be deleted if it already exists, but the file will be created if it does not already exist.
- `a+`: This opens a file for reading as well as for appending new content.

The `fopen` function returns a file descriptor that points to the file for performing different operations.

fclose()

The `fclose()` function is used for closing the file. Here is its syntax:

```
int fclose(FILE *file_pointer)
```

Here, `file_pointer` represents the file pointer that is pointing at the open file.

The function returns a 0 value if the file is successfully closed.

fgets()

The `fgets()` function is used for reading a string from the specified file. Here is its syntax:

```
char *fgets(char *string, int length, FILE *file_pointer)
```

This function has the following features:

- `string`: This represents the character array to which the data that is read from the file will be assigned.
- `length`: This represents the maximum number of characters that can be read from the file. The `length-1` number of characters will be read from the file. The reading of data from the file will stop either at `length-1` location or at the new line character, `\n`, whichever comes first.
- `file_pointer`: This represents the file pointer that is pointing at the file.

fputs()

The `fputs()` function is used for writing into the file. Here is its syntax:

```
int fputs (const char *string, FILE *file_pointer)
```

Here, `string` represents the character array containing the data to be written into the file. The `file_pointer` phrase represents the file pointer that is pointing at the file.

Functions commonly used in random files

The following functions are used to set the file pointer at a specified location in the random file, indicate the location where the file pointer is pointing currently, and rewind the file pointer to the beginning of the random file.

fseek()

The `fseek()` function is used for setting the file pointer at the specific position in the file. Here is its syntax:

```
fseek(FILE *file_pointer, long int offset, int location);
```

This function has the following features:

- `file_pointer`: This represents the file pointer that points at the file.
- `offset`: This represents the number of bytes that the file pointer needs to be moved from the position specified by the `location` parameter. If the value of `offset` is positive, the file pointer will move forward in the file, and if it is negative, the file pointer will move backward from the given position.
- `location`: This is the value that defines the position from which the file pointer needs to be moved. That is, the file pointer will be moved equal to the number of bytes specified by the `offset` parameter from the position specified by the `location` parameter. Its value can be 0, 1, or 2, as shown in the following table:

Value	Meaning
0	The file pointer will be moved from the beginning of the file
1	The file pointer will be moved from the current position
2	The file pointer will be moved from the end of the file

Let's look at the following example. Here, the file pointer will be moved 5 bytes forward from the beginning of the file:

```
fseek(fp, 5L, 0)
```

In the following example, the file pointer will be moved 5 bytes backward from the end of the file:

```
fseek(fp, -5L, 2)
```

f.tell()

The `f.tell()` function returns the byte location where `file_pointer` is currently pointing at the file. Here is its syntax:

```
long int ftell(FILE *file_pointer)
```

Here, `file_pointer` is a file pointer pointing at the file.

rewind()

The `rewind()` function is used for moving the file pointer back to the beginning of the specified file. Here is its syntax:

```
void rewind(FILE *file_pointer)
```

Here, `file_pointer` is a file pointer pointing at the file.

In this chapter, we will learn to use both types of file handling using recipes that make real-time applications.

Reading a text file and converting all characters after the period into uppercase

Say we have a file that contains some text. We think that there is an anomaly in the text—every first character after the period is in lowercase when it should be in uppercase. In this recipe, we will read that text file and convert each character after the period (.) that is, in lowercase into uppercase.



In this recipe, I assume that you know how to create a text file and how to read a text file. If you don't know how to perform these actions, you will find programs for both of them in *Appendix A*.

How to do it...

1. Open the sequential file in read-only mode using the following code:

```
fp = fopen (argv [1],"r");
```

2. If the file does not exist or does not have enough permissions, an error message will be displayed and the program will terminate. Set this up using the following code:

```
if (fp == NULL) {  
    printf("%s file does not exist\n", argv[1]);  
    exit(1);  
}
```

3. One line is read from the file, as shown in the following code:

```
fgets(buffer, BUFSIZE, fp);
```

4. Each character of the line is accessed and checked for the presence of periods, as shown in the following code:

```
for(i=0;i<n;i++)  
    if(buffer[i]=='.')
```

5. If a period is found, the character following the period is checked to confirm whether it is in uppercase, as shown in the following code:

```
if(buffer[i] >=97 && buffer[i] <=122)
```

6. If the character following the period is in lowercase, a value of 32 is subtracted from the ASCII value of the lowercase character to convert it into uppercase, as shown in the following code:

```
buffer[i]=buffer[i]-32;
```

7. If the line is not yet over, then the sequence from step 4 onward is repeated till step 6; otherwise, the updated line is displayed on the screen, as shown in the following code:

```
puts(buffer);
```

8. Check whether the end of file has been reached using the following code. If the file is not over, repeat the sequence from step 3:

```
while(!feof(fp))
```

The preceding steps are pictorially explained in the following diagram (*Figure 5.1*):

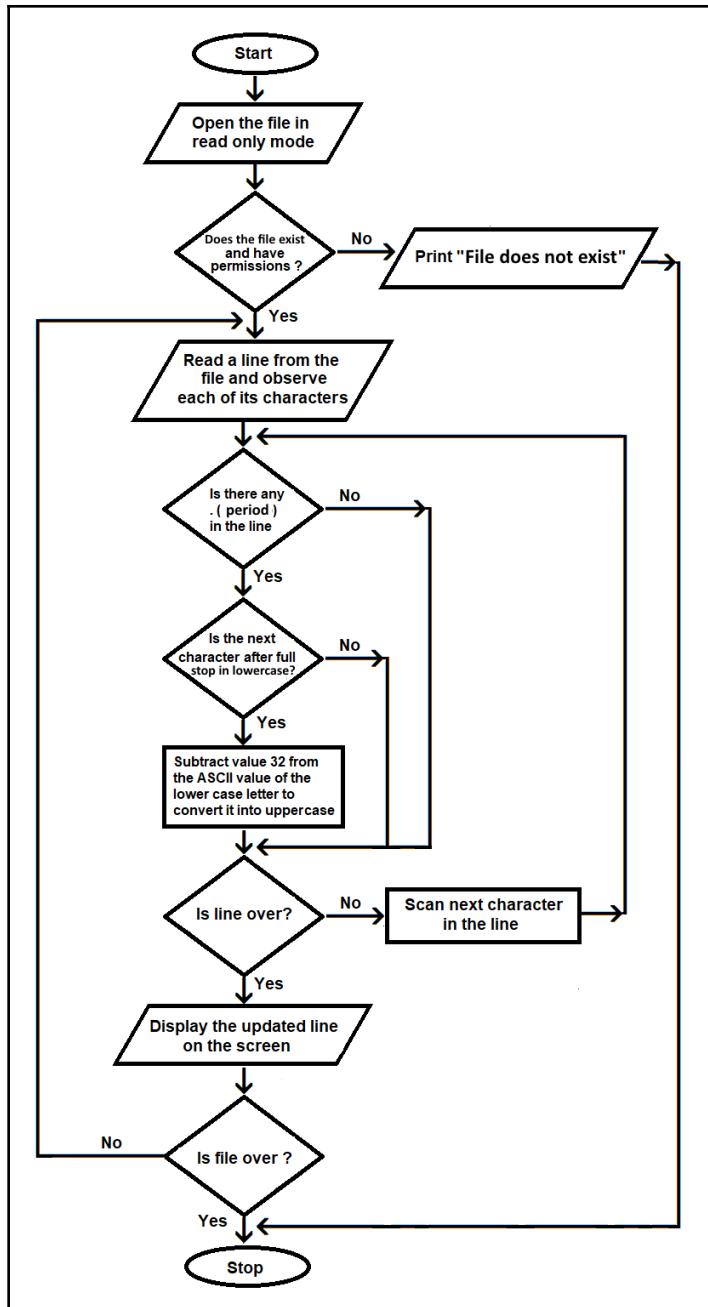


Figure 5.1

The `convertcase.c` program for converting a lowercase character found after a period in a file into uppercase is as follows:

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

#define BUFFSIZE 255

void main (int argc, char* argv[])
{
    FILE *fp;
    char buffer[BUFFSIZE];
    int i,n;

    fp = fopen (argv [1],"r");
    if (fp == NULL) {
        printf("%s file does not exist\n", argv[1]);
        exit(1);
    }
    while (!feof(fp))
    {
        fgets(buffer, BUFFSIZE, fp);
        n=strlen(buffer);
        for(i=0;i<n;i++)
        {
            if(buffer[i]=='.')
            {
                i++;
                while(buffer[i]==' ')
                {
                    i++;
                }
                if(buffer[i] >=97 && buffer[i] <=122)
                {
                    buffer[i]=buffer[i]-32;
                }
            }
            puts(buffer);
        }
        fclose(fp);
    }
}
```

Now, let's go behind the scenes.

How it works...

The file whose name is supplied as a command-line argument is opened in read-only mode and is pointed to by the file pointer, `fp`. This recipe is focused on reading a file and changing its case, so if the file does not exist or does not have read permission, an error will be displayed and the program will terminate.

A `while` loop will be set to execute until `feof` (the end of file) is reached. Within the `while` loop, each line of the file will be read one by one and assigned to the string named `buffer`. The `fgets()` function will be used to read one line at a time from the file. A number of characters will be read from the file until the newline character, `\n`, is reached, to a maximum of 254.

The following steps will be performed on each of the lines assigned to the string buffer:

1. The length of the buffer string will be computed and a `for` loop will be executed to access each of the characters in the string buffer.
2. The string buffer will be checked to see whether there are any periods in it.
3. If one is found, the character following it will be checked to see whether it is into lowercase. ASCII values will be used to then convert the lowercase characters into uppercase (refer to Chapter 2, *Managing Strings*, for more information on the ASCII values that correspond to the letters of the alphabet). If the character following the period is in lowercase, a value of 32 will be deducted from the ASCII value of the lowercase character to convert it into uppercase. Remember, the ASCII value of uppercase characters is lower by a value of 32 than their corresponding lowercase characters.
4. The updated string `buffer` with the character following the period converted into uppercase will be displayed on the screen.

When all the lines of the file are read and displayed, the file pointed to by the `fp` pointer will be closed.

Let's use GCC to compile the `convertcase.c` program as follows:

```
D:\CBook>gcc convertcase.c -o convertcase
```

If you get no errors or warnings, this means that the `convertcase.c` program has been compiled into an executable file, `convertcase.exe`.

Let's say that I have created a file called `textfile.txt` with the following content:

```
D:\CBook>type textfile.txt
I am trying to create a sequential file. It is through C programming. It is
very hot today. I have a cat. Do you like animals? It might rain. Thank
you. Bye
```



The preceding command is executed in Windows' Command Prompt.

Let's run the executable file, `convertcase.exe`, and then pass the `textfile.txt` file to it, as shown in the following code:

```
D:\CBook>./convertcase textfile.txt
I am trying to create a sequential file. It is through C programming. It is
very hot today. I have a cat. Do you like animals? It might rain. Thank
you. Bye
```

You can see in the preceding output that the characters that were in lowercase after the period are now converted into uppercase.

Let's move on to the next recipe!

Displaying the contents of a random file in reverse order

Let's say that we have a random file that contains some lines of text. Let's find out how to reverse the contents of this file.



This program will not give the correct output if a random file does not exist. Please read *Appendix A* to learn how to create a random file.

How to do it...

1. Open the random file in read-only mode using the following code:

```
fp = fopen (argv[1], "rb");
```

2. If the file does not exist or does not have enough permissions, an error message will be displayed and the program will terminate, as shown in the following code:

```
if (fp == NULL) {  
    perror ("An error occurred in opening the file\n");  
    exit(1);  
}
```

3. To read the random file in reverse order, execute a loop equal to the number of lines in the file. Every iteration of the loop will read one line beginning from the bottom of the file. The following formula will be used to find out the number of lines in the file:

total number of bytes used in the file/size of one line in bytes

The code for doing this is as follows:

```
fseek(fp, 0L, SEEK_END);  
n = ftell(fp);  
nol=n/sizeof(struct data);
```

4. Because the file has to be read in reverse order, the file pointer will be positioned at the bottom of the file, as shown in the following code:

```
fseek(fp, -sizeof(struct data)*i, SEEK_END);
```

5. Set a loop to execute that equals the number of lines in the file computed in step 3, as shown in the following code:

```
for (i=1;i<=nol;i++)
```

6. Within the loop, the file pointer will be positioned as follows:

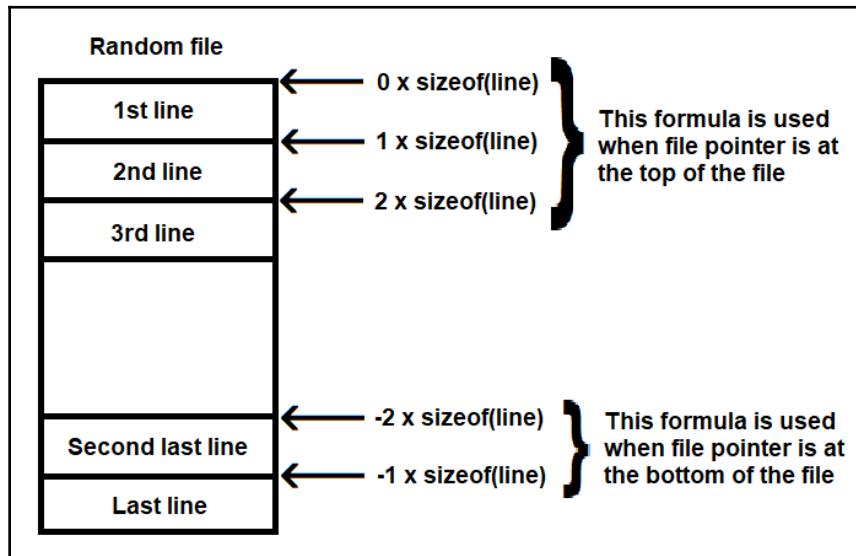


Figure 5.2

7. To read the last line, the file pointer will be positioned at the byte location where the last line begins, at the **-1 x sizeof(line)** byte location. The last line will be read and displayed on the screen, as shown in the following code:

```
 fread(&line,sizeof(struct data),1,fp);  
 puts(line.str);
```

8. Next, the file pointer will be positioned at the byte location from where the second last line begins, at the **-2 x sizeof(line)** byte location. Again, the second last line will be read and displayed on the screen.
9. The procedure will be repeated until all of the lines in the file have been read and displayed on the screen.

The `readrandominreverse.c` program for reading the random file in reverse order is as follows:

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

struct data{
    char str[ 255 ];
};

void main (int argc, char* argv[])
{
    FILE *fp;
    struct data line;
    int n,nol,i;
    fp = fopen (argv[1], "rb");
    if (fp == NULL) {
        perror ("An error occurred in opening the file\n");
        exit(1);
    }
    fseek(fp, 0L, SEEK_END);
    n = ftell(fp);
    nol=n/sizeof(struct data);
    printf("The content of random file in reverse order is :\n");
    for (i=1;i<=nol;i++)
    {
        fseek(fp, -sizeof(struct data)*i, SEEK_END);
        fread(&line,sizeof(struct data),1,fp);
        puts(line.str);
    }
    fclose(fp);
}
```

Now, let's go behind the scenes.

How it works...

We will open the chosen file in read-only mode. If the file opens successfully, it will be pointed at by the file pointer `fp`. Next, we will find out the total number of lines in the file using the following formula:

$$\text{total number of bytes used by the file}/\text{number of bytes used by one line}$$

To know the total number of bytes used by the file, the file pointer will be positioned at the bottom of the file and we will invoke the `ftell` function. The `ftell` function finds the current location of the file pointer. Because the file pointer is at the end of the file, using this function will tell us the total number of bytes used by the file. To find the number of bytes used by one line, we will use the `sizeof` function. We will apply the preceding formula to compute the total number of lines in the file; this will be assigned to the variable, `nol`.

We will set a `for` loop to execute for `nol` number of times. Within the `for` loop, the file pointer will be positioned at the end of the last line so that all of the lines from the file can be accessed in reverse order. So, the file pointer is first set at the `(-1 * size of one line)` location at the bottom of the file. Once the file pointer is positioned at this location, we will use the `fread` function to read the last line of the file and assign it to the structure variable `line`. The string in `line` will then be displayed on the screen.

After displaying the last line on the screen, the file pointer will be set at the byte position of the second last line at `(-2 * size of one line)`. We will again use the `fread` function to read the second last line and display it on the screen.

This procedure will be executed for the number of times that the `for` loop executes, and the `for` loop will execute the same number of times as there are lines in the file. Then the file will be closed.

Let's use GCC to compile the `readrandominreverse.c` program, as follows:

```
D:\CBook>gcc readrandominreverse.c -o readrandominreverse
```

If you get no errors or warnings, this means that the `readrandominreverse.c` program has been compiled into an executable file, `readrandominreverse.exe`.

Let's assume that we have a random file, `random.data`, with the following text:

```
This is a random file. I am checking if the code is working
perfectly well. Random file helps in fast accessing of
desired data. Also you can access any content in any order.
```

Let's run the executable file, `readrandominreverse.exe`, to display the random file, `random.data`, in reverse order using the following code:

```
D:\CBook>./readrandominreverse random.data
The content of random file in reverse order is :
desired data. Also you can access any content in any order.
perfectly well. Random file helps in fast accessing of
This is a random file. I am checking if the code is working
```

By comparing the original file with the preceding output, you can see that the file content is displayed in reverse order.

Now, let's move on to the next recipe!

Counting the number of vowels in a file

In this recipe, we will open a sequential text file and count the number of vowels (both uppercase and lowercase) that it contains.



In this recipe, I will assume that a sequential file already exists. Please read *Appendix A* to learn how to create a sequential file.

How to do it...

1. Open the sequential file in read-only mode using the following code:

```
fp = fopen (argv [1], "r");
```

2. If the file does not exist or does not have enough permissions, an error message will be displayed and the program will terminate, as shown in the following code:

```
if (fp == NULL) {
    printf("%s file does not exist\n", argv[1]);
    exit(1);
}
```

3. Initialize the counter that will count the number of vowels in the file to 0, as shown in the following code:

```
count=0;
```

4. One line is read from the file, as shown in the following code:

```
fgets(buffer, BUFSIZE, fp);
```

5. Each character of the line is accessed and checked for any lowercase or uppercase vowels, as shown in the following code:

```
if(buffer[i]=='a' || buffer[i]=='e' || buffer[i]=='i' ||  
buffer[i]=='o' || buffer[i]=='u' || buffer[i]=='A' ||  
buffer[i]=='E' || buffer[i]=='I' || buffer[i]=='O' ||  
buffer[i]=='U')
```

6. If any vowel is found, the value of the counter is incremented by 1, as shown in the following code:

```
count++;
```

7. Step 5 will be repeated until the end of the line has been reached. Check whether the end of the file has been reached. Repeat from step 4 until the end of the file, as shown in the following code:

```
while (!feof(fp))
```

8. Display the count of the number of vowels in the file by printing the value in the counter variable on the screen, as shown in the following code:

```
printf("The number of vowels are %d\n", count);
```

The preceding steps are shown in the following diagram:

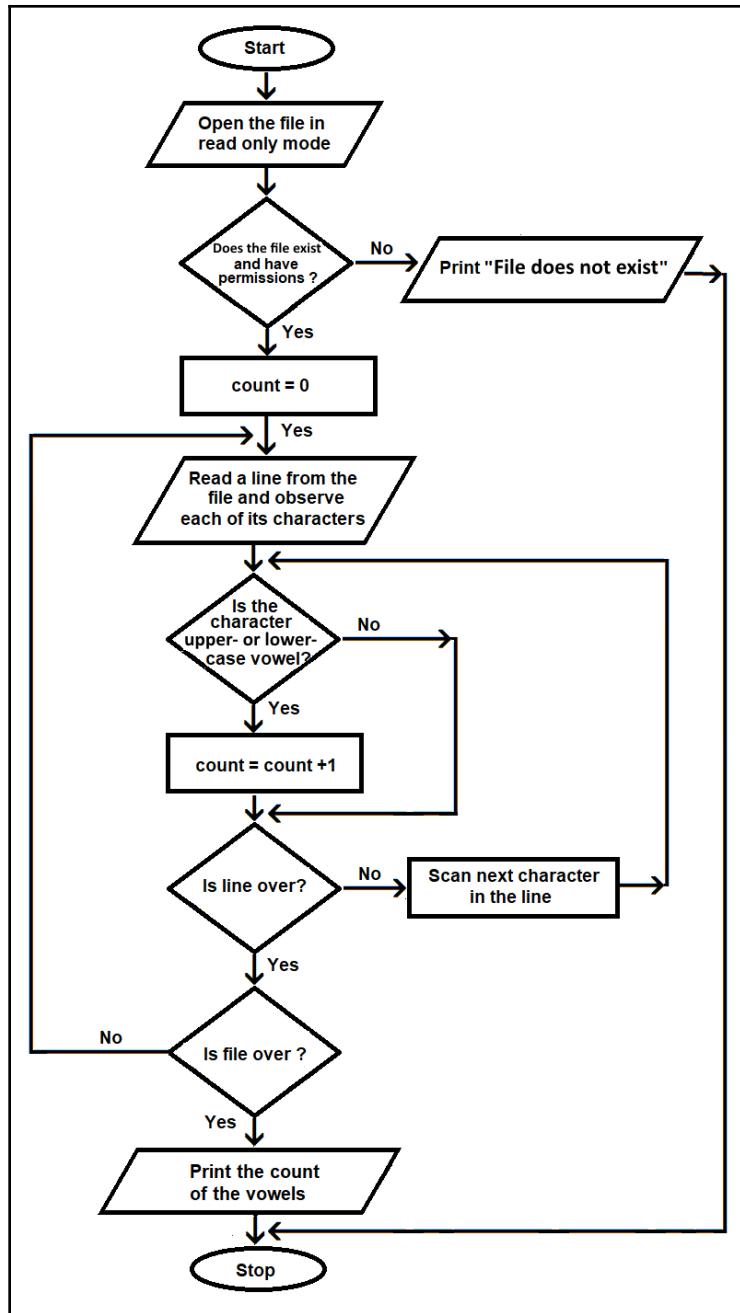


Figure 5.3

The `countvowels.c` program to count the number of vowels in a sequential text file is as follows:

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

#define BUFFSIZE 255

void main (int argc, char* argv[])
{
    FILE *fp;
    char buffer[BUFFSIZE];
    int n, i, count=0;
    fp = fopen (argv [1],"r");
    if (fp == NULL) {
        printf("%s file does not exist\n", argv[1]);
        exit(1);
    }
    printf("The file content is :\n");
    while (!feof(fp))
    {
        fgets(buffer, BUFFSIZE, fp);
        puts(buffer);
        n=strlen(buffer);
        for(i=0;i<n;i++)
        {
            if(buffer[i]=='a' || buffer[i]=='e' || buffer[i]=='i' ||
               buffer[i]=='o' || buffer[i]=='u' || buffer[i]=='A' ||
               buffer[i]=='E' || buffer[i]=='I' || buffer[i]=='O' ||
               buffer[i]=='U') count++;
        }
    }
    printf("The number of vowels are %d\n",count);
    fclose(fp);
}
```

Now, let's go behind the scenes.

How it works...

We will open the chosen sequential file in read-only mode. If the file opens successfully, it will be pointed at by the file pointer, `fp`. To count the number of vowels in the file, we will initialize a counter from 0.

We will set a `while` loop to execute until the file pointer, `fp`, reaches the end of the file. Within the `while` loop, each line in the file will be read using the `fgets` function.

The `fgets` function will read the `BUFFSIZE` number of characters from the file. The value of the `BUFFSIZE` variable is 255, so `fgets` will read either 254 characters from the file or will read characters until the newline character, `\n`, is reached, whichever comes first.

The line read from the file is assigned to the `buffer` string. To display the file contents along with the count of the vowels, the content in the `buffer` string is displayed on the screen. The length of the `buffer` string will be computed and a `for` loop will be set to execute equaling the length of the string.

Each of the characters in the `buffer` string will be checked in the `for` loop. If any lowercase or uppercase vowels appear in the line, the value of the counter variable will be incremented by 1. When the `while` loop ends, the counter variable will have the total count of the vowels present in the file. Finally, the value in the counter variable will be displayed on the screen.

Let's use GCC to compile the `countvowels.c` program as follows:

```
D:\CBook>gcc countvowels.c -o countvowels
```

If you get no errors or warnings, then this means that the `countvowels.c` program has been compiled into an executable file called `countvowels.exe`.

Let's assume that we have a text file called `textfield.txt` with some content. We will run the executable file, `countvowels.exe`, and supply the `textfield.txt` file to it to count the number of vowels in it, as shown in the following code:

```
D:\CBook>./countvowelstextfield.txt
The file content is :
I am trying to create a sequential file. it is through C programming. It is
very hot today. I have a cat. do you like animals? It might rain. Thank
you. bye
The number of vowels are 49
```

You can see from the output of the program that the program not only displays the count of the vowels, but also the complete content of the file.

Now, let's move on to the next recipe!

Replacing a word in a file with another word

Let's say that you want to replace all occurrences of the word `is` with the word `was` in one of your files. Let's find out how to do this.



In this recipe, I will assume that a sequential file already exists. Please read *Appendix A* to learn how to create a sequential file.

How to do it...

1. Open the file in read-only mode using the following code:

```
fp = fopen (argv [1], "r");
```

2. If the file does not exist or does not have enough permissions, an error message will be displayed and the program will terminate, as shown in the following code:

```
if (fp == NULL) {  
    printf("%s file does not exist\n", argv[1]);  
    exit(1);  
}
```

3. Enter the word to be replaced using the following code:

```
printf("Enter a string to be replaced: ");  
scanf("%s", str1);
```

4. Enter the new word that will replace the old word using the following code:

```
printf("Enter the new string ");  
scanf("%s", str2);
```

5. Read a line from the file using the following code:

```
fgets(line, 255, fp);
```

6. Check whether the word to be replaced appears anywhere in the line using the following code:

```
if(line[i]==str1[w])
{
    oldi=i;
    while(w<ls1)
    {
        if(line[i] != str1[w])
            break;
        else
        {
            i++;
            w++;
        }
    }
}
```

7. If the word appears in the line, then simply replace it with the new word using the following code:

```
if(w==ls1)
{
    i=oldi;
    for (k=0;k<ls2;k++)
    {
        nline[x]=str2[k];
        x++;
    }
    i=i+ls1-1;
}
```

8. If the word does not appear anywhere in the line, then move on to the next step. Print the line with the replaced word using the following code:

```
puts(nline);
```

9. Check whether the end of the file has been reached using the following code:

```
while (!feof(fp))
```

10. If the end of the file has not yet been reached, go to step 4. Close the file using the following code:

```
fclose(fp);
```

The `replaceword.c` program replaces the specified word in a file with another word and displays the modified content on the screen:

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

void main (int argc, char* argv[])
{
    FILE *fp;
    char line[255], nline[300], str1[80], str2[80];
    int i,ll, ls1,ls2, x,k, w, oldi;

    fp = fopen (argv [1],"r");
    if (fp == NULL) {
        printf("%s file does not exist\n", argv[1]);
        exit(1);
    }
    printf("Enter a string to be replaced: ");
    scanf("%s", str1);
    printf("Enter the new string ");
    scanf("%s", str2);
    ls1=strlen(str1);
    ls2=strlen(str2);
    x=0;
    while (!feof(fp))
    {
        fgets(line, 255, fp);
        ll=strlen(line);
        for(i=0;i<ll;i++)
        {
            w=0;
            if(line[i]==str1[w])
            {
                oldi=i;
                while(w<ls1)
                {
                    if(line[i] != str1[w])
                        break;
                    else
                    {
                        i++;
                        w++;
                    }
                }
                if(w==ls1)
                {
                    i=oldi;
                    for(j=i;j<ll;j++)
                        line[j]=nline[j];
                    w=ls1;
                }
            }
        }
    }
}
```

```
        for (k=0;k<ls2;k++)
        {
            nline[x]=str2[k];
            x++;
        }
        i=i+ls1-1;
    }
else
{
    i=oldi;
    nline[x]=line[i];
    x++;
}
else
{
    nline[x]=line[i];
    x++;
}
nline[x]='\0';
puts(nline);
}
fclose(fp);
}
```

Now, let's go behind the scenes.

How it works...

Open the chosen file in read-only mode. If the file opens successfully, then the file pointer, `fp`, will be set to point at it. Enter the word to be replaced and assign it to the string variable, `str1`. Similarly, enter the new string that will be assigned to another string variable, `str2`. The length of the two strings, `str1` and `str2`, will be computed and assigned to the variables, `ls1` and `ls2`, respectively.

Set a `while` loop to execute until the file pointed at by `fp` pointer gets over. Within the `while` loop, one line from the file will be read using the `fgets` function. The `fgets` function reads the file until the maximum length that is specified or the new line character, `\n`, is reached, whichever comes first. Because strings are terminated with a mandatory null character, `\0`, a maximum of 254 characters will be read from the file.

The string that is read from the file will be assigned to the `line` variable. The length of the `line` string will be computed and assigned to the `l1` variable. Using a `for` loop, each of the characters in the `line` variable will be accessed to check whether they match with `str1[0]`—that is, with the first character of the string to be replaced. The characters in the `line` variable that don't match with the string to be replaced will be assigned to another string, called `nline`. The `nline` string will contain the desired content—that is, all of the characters of the `line` variable and the new string. If it exists in `line`, then the string will be replaced with the new string and the entire modified content will be assigned to the new string, `nline`.

If the first character of the string to be replaced matches with any of the characters in `line`, then the `while` loop will be used to match all of the successive characters of the string that is to be replaced with the successive characters in `line`. If all of the characters of the string that is to be replaced match with successive characters in `line`, then all of the characters of the string to be replaced are replaced with the new string and assigned to the new string, `nline`. That way, the `while` loop will read one line of text at a time from the file, searching for occurrences of the string to be replaced. If it is found, it replaces it with the new string and assigns the modified line of text to another string, `nline`. The null character, `\0`, is added to the modified string, `nline`, and is displayed on the screen. Finally, the file pointed to by the file pointer, `fp`, is closed.



In this recipe, I am replacing the desired word and another string and displaying the updated content on the screen. If you want the updated content to be written into another file, then you can always open another file in write mode and execute the `fputs` function to write the updated content in it.

Let's use GCC to compile the `replaceword.c` program, as follows:

```
D:\CBook>gcc replaceword.c -o replaceword
```

If you get no errors or warnings, then this means that the `replaceword.c` program has been compiled into an executable file, `replaceword.exe`. Let's run the executable file, `replaceword.exe`, and supply a text file to it. We will assume that a text file called `textfile.txt` exists and has the following content:

```
I am trying to create a sequential file. it is through C programming. It is  
very hot today. I have a cat. do you like animals? It might rain. Thank  
you. bye
```

Now, let's use this file to replace one of its words with another word using the following code:

```
D:\CBook>./replaceword textfile.txt
Enter a string to be replaced: is
Enter the new string was
I am trying to create a sequential file. it was through C programming. It
was very hot today. I have a cat. do you like animals? It might rain. Thank
you. Bye
```

You can see that all occurrences of the word `is` are replaced by `was` in `textfile.txt`, and the modified content is displayed on the screen. We've successfully replaced the words of our choice.

Now, let's move on to the next recipe!

Encrypting a file

Encryption means converting content into a coded format so that unauthorized persons will be unable to see or access the original content of the file. A text file can be encrypted by applying a formula to the ASCII value of the content.

The formula or code can be of your choosing, and it can be as simple or complex as you want. For example, let's say that you have chosen to replace the current ASCII value of all letters by moving them forward 15 values. In this case, if the letter is a lowercase `a` that has the ASCII value of 97, then the forward shift of the ASCII values by 15 will make the *encrypted* letter a lowercase `p`, which has the ASCII value of 112 ($97 + 15 = 112$).



In this recipe, I assume that a sequential file that you want to encrypt already exists. Please read *Appendix A* to learn how to create a sequential file. You can also refer to *Appendix A* if you want to know how an encrypted file is decrypted.

How to do it...

1. Open the source file in read-only mode using the following code:

```
fp = fopen (argv [1], "r");
```

2. If the file does not exist or does not have enough permissions, an error message will be displayed and the program will terminate, as shown in the following code:

```
if (fp == NULL) {  
    printf("%s file does not exist\n", argv[1]);  
    exit(1);  
}
```

3. Open the destination file, the file where the encrypted text will be written, in write-only mode using the following code:

```
fq = fopen (argv[2], "w");
```

4. Read a line from the file and access each of its characters using the following code:

```
fgets(buffer, BUFFSIZE, fp);
```

5. Using the following code, subtract a value of 45 from the ASCII value of each of the characters in the line to encrypt that character:

```
for(i=0;i<n;i++)  
    buffer[i]=buffer[i]-45;
```

6. Repeat step 5 until the line is over. Once all of the characters in the line are encrypted, write the encrypted line into the destination file using the following code:

```
fputs(buffer,fq);
```

7. Check whether the end of the file has been reached using the following code:

```
while (!feof(fp))
```

8. Close the two files using the following code:

```
fclose (fp);  
fclose (fq);
```

The preceding steps are shown in the following diagram:

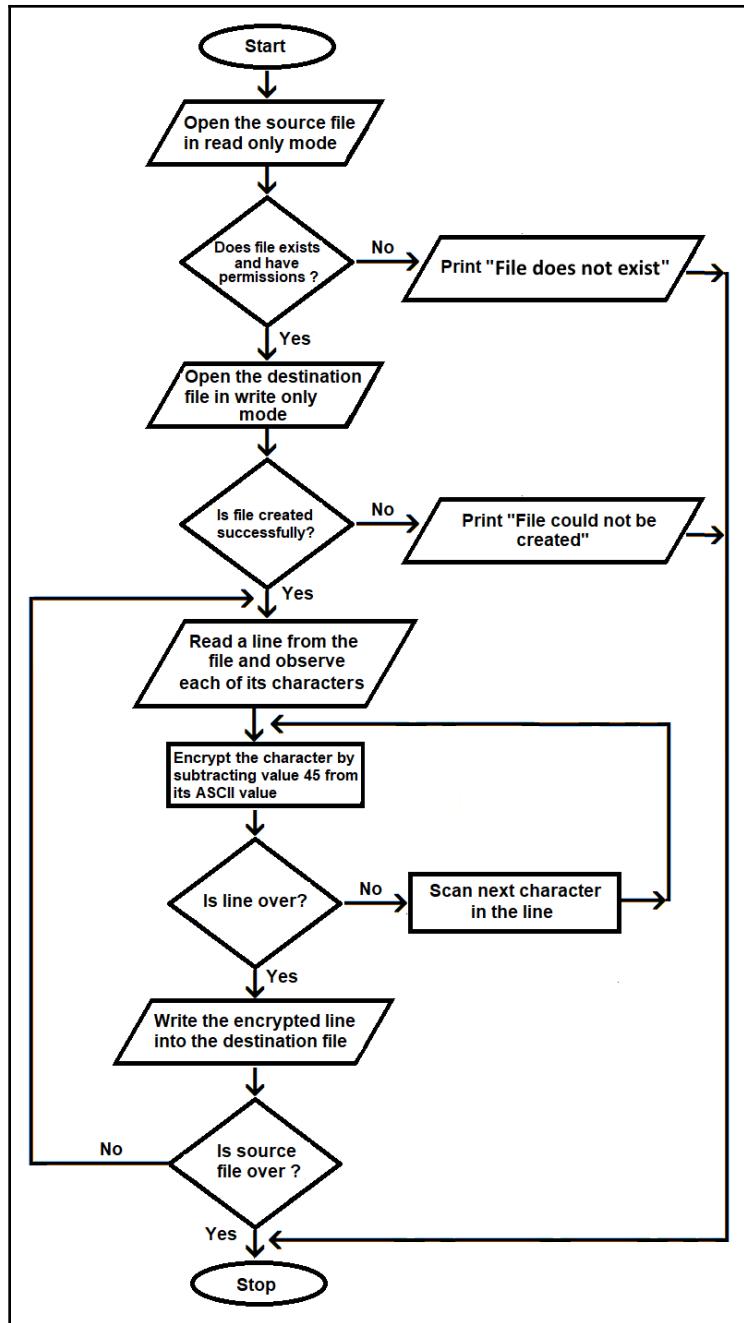


Figure 5.4

The encryptfile.c program to encrypt a file is as follows:

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

#define BUFFSIZE 255
void main (int argc, char* argv[])
{
    FILE *fp,*fq;
    int i,n;
    char buffer[BUFFSIZE];

    /* Open the source file in read mode */
    fp = fopen (argv [1],"r");
    if (fp == NULL) {
        printf("%s file does not exist\n", argv[1]);
        exit(1);
    }
    /* Create the destination file.  */
    fq = fopen (argv[2], "w");
    if (fq == NULL) {
        perror ("An error occurred in creating the file\n");
        exit(1);
    }
    while (!feof(fp))
    {
        fgets(buffer, BUFFSIZE, fp);
        n=strlen(buffer);
        for(i=0;i<n;i++)
            buffer[i]=buffer[i]-45;
        fputs(buffer,fq);
    }
    fclose (fp);
    fclose (fq);
}
```

Now, let's go behind the scenes.

How it works...

The first file name that is passed through the command-line arguments is opened in read-only mode. The second file name that is passed through the command-line arguments is opened in write-only mode. If both files are opened correctly, then the `fp` and `fq` pointers, respectively, will point at the read-only and write-only files.

We will set a `while` loop to execute until it reaches the end of the source file. Within the loop, one line from the source file will be read using the `fgets` function. The `fgets` function reads the specified number of bytes from the file or until the new line character, `\n`, is reached. If the new line character does not appear in the file, then the `BUFFSIZE` constant limits the bytes to be read from the file to 254.

The line read from the file is assigned to the `buffer` string. The length of the string `buffer` is computed and assigned to the variable, `n`. We will then set a `for` loop to execute until it reaches the end of the length of the `buffer` string, and within the loop, the ASCII value of each character will be changed.

To encrypt the file, we will subtract a value of 45 from the ASCII value of each of the characters, although we can apply any formula we like. Just ensure that you remember the formula, as we will need to reverse it in order to decrypt the file.

After applying the formula to all of the characters, the encrypted line will be written into the target file. In addition, to display the encrypted version on the screen, the encrypted line will be displayed on the screen.

When the `while` loop is finished, all of the lines from the source file will be written into the target file after they are encrypted. Finally, the two files will be closed.

Let's use GCC to compile the `encryptfile.c` program, as follows:

```
D:\CBook>gcc encryptfile.c -o encryptfile
```

If you get no errors or warnings, this means that the `encryptfile.c` program has been compiled into an executable file, `encryptfile.exe`. Let's run this executable file.

Before running the executable file, though, let's take a look at the text file, `textfile.txt`, which will be encrypted using this program. The contents of this text file are as follows:

```
I am trying to create a sequential file. it is through C programming. It is  
very hot today. I have a cat. do you like animals? It might rain. Thank  
you. bye
```

Let's run the executable file, `encryptfile.exe`, on `textfile.txt` and put the encrypted content into another file named `encrypted.txt` using the following code:

```
D:\CBook>./encryptfile textfile.txt encrypted.txt
```

The normal content in `textfile.txt` is encrypted and the encrypted content is written into another file named `encrypted.txt`. The encrypted content will appear as follows:

```
D:\CBook>type encrypted.txt
≤4@≤GEL<A:≤GB≤6E84G8≤4≤F8DH8AG<4?≤9<?8≤<G≤<F≤G;EBH: ;≤≤CEB:E4@@<A:<≤≤G≤<F≤I8E
L≤;BG≤GB74L≤;4I8≤4≤64G≤≤7B≤LBH≤?>>8≤4A<@4?F≤≤≤G≤@< :;G≤E4<A';4A>≤LBH≤5L8
```

The preceding command is executed in Windows' Command Prompt.



Voila! We've successfully encrypted the file!

3

Section 3: Concurrency, Networking, and Databases

In this section, we will learn how to implement concurrency in threads to reduce memory usage and speed up the CPU. We will then learn how to establish communication between processes and create a network between the server and the client. Finally, we will learn how to use a database to store and manage our data.

The following chapters will be covered in this section:

- Chapter 6, *Implementing Concurrency*
- Chapter 7, *Networking and Interprocess Communication*
- Chapter 8, *Using the MySQL Database*

6

Implementing Concurrency

Multitasking is a key feature in almost all operating systems; it increases the efficiency of the CPU and utilizes resources in a better manner. Threads are the best way to implement multitasking. A process can contain more than one thread to implement multitasking.

In this chapter, we will cover the following recipes involving threads:

- Performing a task with a single thread
- Performing multiple tasks with multiple threads
- Using `mutex` to share data between two threads
- Understanding how a deadlock is created
- Avoiding a deadlock

The terms process and thread can be confusing, so first, we'll make sure that you understand them.

What are processes and threads?

Whenever we run a program, the moment that it is loaded from the hard disk (or any other storage) into the memory, it becomes a process. A **process** is executed by a processor, and for its execution, it requires a **program counter (PC)** to keep track of the next instruction to be executed, the CPU registers, the signals, and so on.

A **thread** refers to a set of instructions within a program that can be executed independently. A thread has its own PC and set of registers, among other things. In that way, a process is comprised of several threads. Two or more threads can share their code, data, and other resources, but special care must be taken when sharing resources among threads, as it might lead to ambiguity and deadlock. An operating system also manages a thread pool.

A **thread pool** contains a collection of threads that are waiting for tasks to be allocated to them for concurrent execution. Using threads from the thread pool instead of instantiating new threads helps to avoid the delay that is caused by creating and destroying new threads; hence, it increases the overall performance of the application.

Basically, threads enhance the efficiency of an application through parallelism, that is, by running two or more independent sets of code simultaneously. This is called **multithreading**.

Multithreading is not supported by C, so to implement it, POSIX threads (`Pthreads`) are used. GCC allows for the implementation of a `pthread`.

While using a `pthread`, a variable of the type `pthread_t` is defined to store the thread identifier. A **thread identifier** is a unique integer, that is, assigned to a thread in the system.

You must be wondering which function is used for creating a thread. The `pthread_create` function is invoked to create a thread. The following four arguments are passed to the `pthread_create` function:

- A pointer to the thread identifier, which is set by this function
- The attributes of the thread; usually, `NULL` is provided for this argument to use the default attributes
- The name of the function to execute for the creation of the thread
- The arguments to be passed to the thread, set to `NULL` if no arguments need to be passed to the thread

When two or more threads operate on the same data, that is, when they share the same resources, certain check measures must be applied so that only one thread is allowed to manipulate the shared resource at a time; other threads' access must be blocked. One of the methods that helps to avoid ambiguity when a resource is shared among threads is mutual exclusion.

Mutual exclusion

To avoid ambiguity when two or more threads access the same resource, **mutual exclusion** implements serializing access to the shared resources. When one thread is using a resource, no other thread is allowed to access the same resource. All of the other threads are blocked from accessing the same resource until the resource is free again.

A `mutex` is basically a lock that is associated with the shared resource. To read or modify the shared resource, a thread must first acquire the lock for that resource. Once a thread acquires a lock (or `mutex`) for that resource, it can go ahead with processing that resource. All of the other threads that wish to work on it will be compelled to wait until the resource is unlocked. When the thread finishes its processing on the shared resource, it unlocks the `mutex`, enabling the other waiting threads to acquire a `mutex` for that resource. Aside from `mutex`, a semaphore is also used in process synchronization.

A **semaphore** is a concept that is used to avoid two or more processes from accessing a common resource in a concurrent system. It is basically a variable that is manipulated to only allow one process to have access to a common resource and implement process synchronization. A semaphore uses the signaling mechanism, that is, it invokes `wait` and `signal` functions, respectively, to inform that the common resource has been acquired or released. A `mutex`, on the other hand, uses the locking mechanism—the process has to acquire the lock on the `mutex` object before working on the common resource.

Although `mutex` helps to manage shared resources among threads, there is a problem. An application of `mutex` in the wrong order may lead to a deadlock. A deadlock occurs in a situation when a thread that has `lock X` tries to acquire `lock Y` to complete its processing, while another thread that has `lock Y` tries to acquire `lock X` to finish its execution. In such a situation, a deadlock will occur, as both of the threads will keep waiting indefinitely for the other thread to release its lock. As no thread will be able to finish its execution, no thread will be able to free up its locks, either. One solution to avoid a deadlock is to let threads acquire locks in a specific order.

The following functions are used to create and manage threads:

- `pthread_join`: This function makes the thread wait for the completion of all its spawned threads. If it is not used, the thread will exit as soon as it completes its task, ignoring the states of its spawned threads. In other words, `pthread_join` blocks the calling thread until the thread specified in this function terminates.
- `pthread_mutex_init`: This function initializes the `mutex` object with the specified attributes. If `NULL` is used for the attributes, the default `mutex` attributes are used for initializing the `mutex` object. When the `mutex` is initialized, it is in an unlocked state.

- `pthread_mutex_lock`: This function locks the specified `mutex` object. If the `mutex` is already locked by some other thread, the calling thread will get suspended, that is, it will be asked to wait until the `mutex` gets unlocked. This function returns the `mutex` object in a locked state. The thread that locks the `mutex` becomes its owner and remains the owner until it unlocks the `mutex`.
- `pthread_mutex_unlock`: This function releases the specified `mutex` object. The thread that has invoked the `pthread_mutex_lock` function and is waiting for the `mutex` to get unlocked will become unblocked and acquire the `mutex` object, that is, the waiting thread will be able to access and lock the `mutex` object. If there are no threads waiting for the `mutex`, the `mutex` will remain in the unlocked state without any owner thread.
- `pthread_mutex_destroy`: This function destroys a `mutex` object and frees up the resources allocated to it. The `mutex` must be in an unlocked state before invoking this method.



Depending on the operating system, a lock may be a **spinlock**. If any thread tries to acquire a lock but the lock is not free, a spinlock will make the thread wait in a loop until the lock becomes free. Such locks keep the thread busy while it's waiting for the lock to free up. They are efficient, as they avoid the consumption of time and resources in process rescheduling or context switching.

That is enough theory. Now, let's start with some practical examples!

Performing a task with a single thread

In this recipe, we will be creating a thread to perform a task. In this task, we will display the sequence numbers from 1 to 5. The focus of this recipe is to learn how a thread is created and how the main thread is asked to wait until the thread finishes its task.

How to do it...

1. Define a variable of the type `pthread_t` to store the thread identifier:

```
pthread_t tid;
```

2. Create a thread and pass the identifier that was created in the preceding step to the `pthread_create` function. The thread is created with the default attributes. Also, specify a function that needs to be executed to create the thread:

```
pthread_create(&tid, NULL, runThread, NULL);
```

3. In the function, you will be displaying a text message to indicate that the thread has been created and is running:

```
printf("Running Thread \n");
```

4. Invoke a `for` loop to display the sequence of numbers from 1 to 5 through the running thread:

```
for(i=1;i<=5;i++) printf("%d\n",i);
```

5. Invoke the `pthread_join` method in the main function to make the `main` method wait until the thread completes its task:

```
pthread_join(tid, NULL);
```

The `createthread.c` program for creating a thread and making it perform a task is as follows:

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

void *runThread(void *arg)
{
    int i;
    printf("Running Thread \n");
    for(i=1;i<=5;i++) printf("%d\n",i);
    return NULL;
}

int main()
{
    pthread_t tid;
    printf("In main function\n");
    pthread_create(&tid, NULL, runThread, NULL);
    pthread_join(tid, NULL);
    printf("Thread over\n");
    return 0;
}
```

Now, let's go behind the scenes.

How it works...

We will define a variable called `tid` of the type `pthread_t` to store the thread identifier. A **thread identifier** is a unique integer, that is, assigned to a thread in the system. Before creating a thread, the message `In main function` is displayed on the screen. We will create a thread and pass the identifier `tid` to the `pthread_create` function. The thread is created with the default attributes, and the `runThread` function is set to execute to create the thread.

In the `runThread` function, we will display the text message `Running Thread` to indicate that the thread was created and is running. We will invoke a `for` loop to display the sequence of numbers from 1 to 5 through the running thread. By invoking the `pthread_join` method, we will make the `main` method wait until the thread completes its task. It is essential to invoke the `pthread_join` here; otherwise, the `main` method will exit without waiting for the completion of the thread.

Let's use GCC to compile the `createthread.c` program, as follows:

```
D:\CBook>gcc createthread.c -o createthread
```

If you get no errors or warnings, that means the `createthread.c` program has been compiled into an executable file, `createthread.exe`. Let's run this executable file:

```
D:\Chap5>createthread
In main function
Running Thread
1
2
3
4
5
Thread over
```

Voila! We've successfully completed a task with a single thread. Now, let's move on to the next recipe!

Performing multiple tasks with multiple threads

In this recipe, you will learn how to multitask by executing two threads in parallel. Both of the threads will do their tasks independently. As the two threads will not be sharing a resource, there will not be a situation of race condition or ambiguity. The CPU will execute any thread randomly at a time, but finally, both of the threads will finish the assigned task. The task that the two threads will perform is displaying the sequence of numbers from 1 to 5.

How to do it...

1. Define two variables of the type `pthread_t` to store two thread identifiers:

```
pthread_t tid1, tid2;
```

2. Invoke the `pthread_create` function twice to create two threads, and assign the identifiers that we created in the previous step. The two threads are created with the default attributes. Specify two respective functions that need to be executed for the two threads:

```
pthread_create(&tid1,NULL,runThread1,NULL);
pthread_create(&tid2,NULL,runThread2,NULL);
```

3. In the function of the first thread, display a text message to indicate that the first thread was created and is running:

```
printf("Running Thread 1\n");
```

4. To indicate the execution of the first thread, execute a `for` loop in the first function to display the sequence of numbers from 1 to 5. To distinguish from the second thread, the sequence of numbers that were generated by the first thread are prefixed by Thread 1:

```
for(i=1;i<=5;i++)
    printf("Thread 1 - %d\n",i);
```

5. Similarly, in the second thread, display a text message to inform that the second thread has also been created and is running:

```
printf("Running Thread 2\n");
```

6. Again, in the second function, execute a `for` loop to display the sequence of numbers from 1 to 5. To differentiate these numbers from the ones generated by `thread1`, this sequence of numbers will be preceded by the text `Thread 2`:

```
for(i=1;i<=5;i++)
    printf("Thread 2 - %d\n",i);
```

7. Invoke the `pthread_join` twice, and pass the thread identifiers we created in step 1 to it. `pthread_join` will make the two threads, and the `main` method will wait until both of the threads have completed their tasks:

```
pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
```

8. When both of the threads are finished, a text message will be displayed to confirm this:

```
printf("Both threads are over\n");
```

The `twothreads.c` program for creating two threads and making them work on independent resources is as follows:

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

void *runThread1(void *arg) {
    int i;
    printf("Running Thread 1\n");
    for(i=1;i<=5;i++)
        printf("Thread 1 - %d\n",i);
}

void *runThread2(void *arg) {
    int i;
    printf("Running Thread 2\n");
    for(i=1;i<=5;i++)
        printf("Thread 2 - %d\n",i);
}

int main(){
    pthread_t tid1, tid2;
    pthread_create(&tid1,NULL,runThread1,NULL);
    pthread_create(&tid2,NULL,runThread2,NULL);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("Both threads are over\n");
}
```

```
pthread_create(&tid2,NULL,runThread2,NULL);
pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
printf("Both threads are over\n");
return 0;
}
```

Now, let's go behind the scenes.

How it works...

We will define two variables of the type `pthread_t`, by the names `tid1` and `tid2`, to store two thread identifiers. These thread identifiers uniquely represent the threads in the system. We will invoke the `pthread_create` function twice to create two threads and assign their identifiers to the two variables `tid1` and `tid2`, whose addresses are passed to the `pthread_create` function.

The two threads are created with the default attributes. We will execute the function `runThread1` to create the first thread, and then the `runThread2` function to create the second thread.

In the `runThread1` function, we will display the message `Running Thread 1` to indicate that the first thread was created and is running. In addition, we will invoke a `for` loop to display the sequence of numbers from 1 to 5 through the running thread. The sequence of numbers that are generated by the first thread will be prefixed by `Thread 1`.

Similarly, in the `runThread2` function, we will display the message `Running Thread 2` to inform that the second thread was also created and is running. Again, we will invoke a `for` loop to display the sequence of numbers from 1 to 5. To differentiate these numbers from the ones generated by `thread1`, these numbers are preceded by the text `Thread 2`.

We will then invoke the `pthread_join` method twice and pass our two thread identifiers, `tid1` and `tid2`, to it. The `pthread_join` is invoked to make the two threads, and the `main` method waits until both of the threads have completed their respective tasks. When both of the threads are over, that is, when the functions `runThread1` and `runThread2` are over, a message saying that `Both threads are over` will be displayed in the `main` function.

Let's use GCC to compile the `twothreads.c` program, as follows:

```
D:\CBook>gcc twothreads.c -o twothreads
```

If you get no errors or warnings, that means the `twothreads.c` program has been compiled into an executable file, `twothreads.exe`. Let's run this executable file:

```
D:\Chap5>twothreads
Running Thread 1
Running Thread 2
Thread 1 - 1
Thread 2 - 1
Thread 1 - 2
Thread 2 - 2
Thread 1 - 3
Thread 2 - 3
Thread 1 - 4
Thread 2 - 4
Thread 1 - 5
Thread 2 - 5
Both threads are over
```

You may not get exactly the same output, as it depends on the CPU, but it is certain that both threads will exit simultaneously.

Voila! We've successfully completed multiple tasks with multiple threads. Now, let's move on to the next recipe!

Using mutex to share data between two threads

Running two or more threads independently, where each accesses its own resources, is quite convenient. However, sometimes, we want the threads to share and process the same resource simultaneously so that we can finish a task faster. Sharing a common resource may lead to problems, as one thread might read the data before the other thread writes the updated data, leading to an ambiguous situation. To avoid such a situation, `mutex` is used. In this recipe, you will learn how to share common resources between two threads.

How to do it...

1. Define two variables of the `pthread_t` type to store two thread identifiers. Also, define a `mutex` object:

```
pthread_t tid1,tid2;  
pthread_mutex_t lock;
```

2. Invoke the `pthread_mutex_init` method to initialize the `mutex` object with the default `mutex` attributes:

```
pthread_mutex_init(&lock, NULL)
```

3. Invoke the `pthread_create` function twice to create two threads, and assign the identifiers that we created in step 1. Execute a function for creating the two threads:

```
pthread_create(&tid1, NULL, &runThread, NULL);  
pthread_create(&tid2, NULL, &runThread, NULL);
```

4. In the function, the `pthread_mutex_lock` method is invoked and the `mutex` object is passed to it to lock it:

```
pthread_mutex_lock(&lock);
```

5. Invoke the `pthread_self` method and assign the ID of the calling thread to a variable of the `pthread_t` type. Invoke the `pthread_equal` method and compare it with the variable to find out which thread is currently executing. If the first thread is being executed, display the message `First thread is running` on the screen:

```
pthread_t id = pthread_self();  
if(pthread_equal(id,tid1))  
    printf("First thread is running\n");
```

6. To indicate that the thread is executing a common resource, display the text message `Processing the common resource` on the screen:

```
printf("Processing the common resource\n");
```

7. Invoke the `sleep` method to make the first thread sleep for 5 seconds:

```
sleep(5);
```

8. After a duration of 5 seconds, display the message `First thread is over` on the screen:

```
printf("First thread is over\n\n");
```

9. The `pthread_mutex_unlock` function will be invoked, and the `mutex` object that we created in the first step will be passed to it to unlock it:

```
pthread_mutex_unlock(&lock);
```

10. The `thread` function will be invoked by the second thread. Lock the `mutex` object again:

```
pthread_mutex_lock(&lock);
```

11. To indicate that the second thread is running at the moment, display the message `Second thread is running` on the screen:

```
printf("Second thread is running\n");
```

12. Again, to indicate that the common resource is being accessed by the thread, display the message `Processing the common resource` on the screen:

```
printf("Processing the common resource\n");
```

13. Introduce a delay of 5 seconds. Then, display the message `second thread is over` on the screen:

```
sleep(5);
printf("Second thread is over\n\n");
```

14. Unlock the `mutex` object:

```
pthread_mutex_unlock(&lock);
```

15. Invoke the `pthread_join` method twice and pass the thread identifiers to it:

```
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
```

16. Invoke the `pthread_mutex_destroy` method to destroy the `mutex` object:

```
pthread_mutex_destroy(&lock);
```

The `twothreadsmutex.c` program for creating two threads that share common resources is as follows:

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
pthread_t tid1,tid2;
pthread_mutex_t lock;

void* runThread(void *arg)
{
    pthread_mutex_lock(&lock);
    pthread_t id = pthread_self();
    if(pthread_equal(id,tid1))
        printf("First thread is running\n");
    else
        printf("Second thread is running\n");
    printf("Processing the common resource\n");
    sleep(5);
    if(pthread_equal(id,tid1))
        printf("First thread is over\n\n");
    else
        printf("Second thread is over\n\n");
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void)
{
    if (pthread_mutex_init(&lock, NULL) != 0)
        printf("\n mutex init has failed\n");
    pthread_create(&tid1, NULL, &runThread, NULL);
    pthread_create(&tid2, NULL, &runThread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Now, let's go behind the scenes.

How it works...

We will first define a `mutex` object by the name `lock`. Recall that a `mutex` is basically a lock associated with a shared resource. To read or modify the shared resource, a thread needs to first acquire the lock for that resource. We will define two variables of the `pthread_t` type, with the names `tid1` and `tid2`, to store two thread identifiers.

We will invoke the `pthread_mutex_init` method that initializes the `lock` object with the default `mutex` attributes. When it's initialized, the `lock` object is in an unlocked state. We then invoke the `pthread_create` function twice to create two threads and assign their identifiers to the two variables `tid1` and `tid2`, whose addresses are passed to the `pthread_create` function. The two threads are created with the default attributes.

Next, we will execute the `runThread` function to create the two threads. In the `runThread` function, we will invoke the `pthread_mutex_lock` method and pass the `mutex` object `lock` to it to lock it. Now, the rest of the threads (if any) will be asked to wait until the `mutex` object `lock` is unlocked. We will invoke the `pthread_self` method and assign the ID of the calling thread to the variable `id` of the `pthread_t` type. We will then invoke the `pthread_equal` method to ensure that if the calling thread is the one with the identifier assigned to the `tid1` variable, then the message `First thread is running` will display on the screen.

Next, the message `Processing the common resource` is displayed on the screen. We will invoke the `sleep` method to make the first thread sleep for 5 seconds. After a duration of 5 seconds, the message `First thread is over` will be displayed on the screen to indicate that the first thread is over. We will then invoke `pthread_mutex_unlock` and pass the `mutex` object `lock` to it to unlock it. Unlocking the `mutex` object is an indication to the other threads that the common resource can be used by other threads, too.

The `runThread` method will be invoked by the second thread, with the identifier `tid2`. Again, the `mutex` object `lock` is locked, and the `id` of the calling thread, that is, the second thread, is assigned to the variable `id`. The message `Second thread is running` is displayed on the screen, followed by the message `Processing the common resource`.

We will introduce a delay of 5 seconds to indicate that the second thread is processing the common resource. Then, the message `second thread is over` will be displayed on the screen. The `mutex` object `lock` is now unlocked. We will invoke the `pthread_join` method twice and pass the `tid1` and `tid2` thread identifiers to it. `pthread_join` is invoked to make the two threads and the `main` method wait until both of the threads have completed their tasks.

When both of the threads are over, we will invoke the `pthread_mutex_destroy` method to destroy the `mutex` object `lock` and free up the resources allocated to it.

Let's use GCC to compile the `twothreadsmutex.c` program, as follows:

```
D:\CBook>gcc twothreadsmutex.c -o twothreadsmutex
```

If you get no errors or warnings, that means the `twothreadsmutex.c` program has been compiled into an executable file, `twothreadsmutex.exe`. Let's run this executable file:

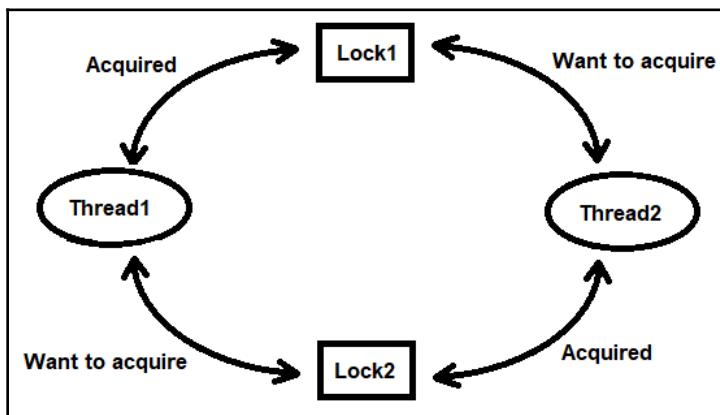
```
D:\Chap5>twothreadsmutex
First thread is running
Processing the common resource
First thread is over

Second thread is running
Processing the common resource
Second thread is over
```

Voila! We've successfully used `mutex` to share data between two threads. Now, let's move on to the next recipe!

Understanding how a deadlock is created

Locking a resource helps in non-ambiguous results, but locking can also lead to a deadlock. A **deadlock** is a situation wherein a thread has acquired the lock for one resource and wants to acquire the lock for a second resource. However, at the same time, another thread has acquired the lock for the second resource, but wants the lock for the first resource. Because the first thread will keep waiting for the second resource lock to be free and the second thread will keep waiting for the first resource lock to be free, the threads will not be able to proceed further, and the application will hang (as the following diagram illustrates):



In this recipe, we will use a stack. A stack requires two operations—push and pop. To make only one thread execute a push or pop operation at a time, we will use two `mutex` objects—`pop_mutex` and `push_mutex`. The thread needs to acquire locks on both of the objects to operate on the stack. To create a situation of deadlock, we will make a thread acquire one lock and ask it to acquire another lock, which was already acquired by another thread.

How to do it...

1. Define a macro of the value 10, and define an array of an equal size:

```
#define max 10  
int stack[max];
```

2. Define two `mutex` objects; one will be used while popping from the stack (`pop_mutex`), and the other will be used while pushing a value to the stack (`push_mutex`):

```
pthread_mutex_t pop_mutex;  
pthread_mutex_t push_mutex;
```

3. To use the `stack`, initialize the value of `top` to `-1`:

```
int top=-1;
```

4. Define two variables of the type `pthread_t` to store two thread identifiers:

```
pthread_t tid1,tid2;
```

5. Invoke the `pthread_create` function to create the first thread; the thread will be created with the default attributes. Execute the `push` function to create this thread:

```
pthread_create(&tid1,NULL,&push,NULL);
```

6. Invoke the `pthread_create` function again to create the second thread; this thread will also be created with the default attributes. Execute the `pop` function to create this thread:

```
pthread_create(&tid2,NULL,&pop,NULL);
```

7. In the `push` function, invoke the `pthread_mutex_lock` method and pass the `mutex` object for the `push` operation (`push_mutex`) to lock it:

```
pthread_mutex_lock(&push_mutex);
```

8. Then, the `mutex` object for the `pop` operation (`pop_mutex`) will be locked by the first thread:

```
pthread_mutex_lock(&pop_mutex);
```

9. The user is asked to enter the value to be pushed to the `stack`:

```
printf("Enter the value to push: ");  
scanf("%d",&n);
```

10. The value of `top` is incremented to 0. The value that was entered in the previous step is pushed to the location `stack[0]`:

```
top++;
stack[top]=n;
```

11. Invoke `pthread_mutex_unlock` and unlock the `mutex` objects meant for the pop (`pop_mutex`) and push operations (`push_mutex`):

```
pthread_mutex_unlock(&pop_mutex);
pthread_mutex_unlock(&push_mutex);
```

12. At the bottom of the push function, display a text message indicating that the value is pushed to the stack:

```
printf("Value is pushed to stack \n");
```

13. In the pop function, invoke the `pthread_mutex_lock` function to lock the `mutex` object `pop_mutex`. It will lead to a deadlock:

```
pthread_mutex_lock(&pop_mutex);
```

14. Again, try to lock the `push_mutex` object, too (although it is not possible, as it is always acquired by the first thread):

```
sleep(5);
pthread_mutex_lock(&push_mutex);
```

15. The value in the stack, that is, pointed to by the `top` pointer is popped:

```
k=stack[top];
```

16. Thereafter, the value of `top` is decremented by 1 to make it -1 again. The value, that is, popped from the stack is displayed on the screen:

```
top--;
printf("Value popped is %d \n", k);
```

17. Then, unlock the `mutex` object `push_mutex` and the `pop_mutex` object:

```
pthread_mutex_unlock(&push_mutex);
pthread_mutex_unlock(&pop_mutex);
```

18. In the main function, invoke the `pthread_join` method and pass the thread identifiers that were created in step 1 to it:

```
pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
```

The `deadlockstate.c` program for creating two threads and understanding how a deadlock occurs while acquiring locks is as follows:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

#define max 10
pthread_mutex_t pop_mutex;
pthread_mutex_t push_mutex;
int stack[max];
int top=-1;

void * push(void *arg) {
    int n;
    pthread_mutex_lock(&push_mutex);
    pthread_mutex_lock(&pop_mutex);
    printf("Enter the value to push: ");
    scanf("%d",&n);
    top++;
    stack[top]=n;
    pthread_mutex_unlock(&pop_mutex);
    pthread_mutex_unlock(&push_mutex);
    printf("Value is pushed to stack \n");
}

void * pop(void *arg) {
    int k;
    pthread_mutex_lock(&pop_mutex);
    pthread_mutex_lock(&push_mutex);
    k=stack[top];
    top--;
    printf("Value popped is %d \n",k);
    pthread_mutex_unlock(&push_mutex);
    pthread_mutex_unlock(&pop_mutex);
}

int main() {
    pthread_t tid1,tid2;
    pthread_create(&tid1,NULL,&push,NULL);
    pthread_create(&tid2,NULL,&pop,NULL);
    printf("Both threads are created\n");
```

```
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    return 0;
}
```

Now, let's go behind the scenes.

How it works...

We will first define a macro called `max` of the value 10, along with an array stack of the size `max`. Then, we will define two `mutex` objects with the names `pop_mutex` and `push_mutex`. To use the stack, we will initialize the value of `top` to -1. We will also define two variables of the type `pthread_t`, with the names `tid1` and `tid2`, to store two thread identifiers.

We will invoke the `pthread_create` function to create the first thread, and we will assign the identifier returned by the function to the variable `tid1`. The thread will be created with the default attributes, and we will execute the `push` function to create this thread.

We will invoke the `pthread_create` function again to create the second thread, and we will assign the identifier returned by the function to the variable `tid2`. This thread is also created with the default attributes, and we will execute the `pop` function to create this thread. On the screen, we will display the message Both threads are created.

In the `push` function, we will invoke the `pthread_mutex_lock` method and pass the `mutex` object `push_mutex` to it to lock it. Now, if any other thread asks for the `push_mutex` object, it will need to wait until the object is unlocked.

Then, the `mutex` object `pop_mutex` will be locked by the first thread. We will be asked to enter the value to be pushed to the stack. The entered value will be assigned to the variable `n`. The value of `top` will be incremented to 0. The value that we enter will be pushed to the location `stack[0]`.

Next, we will invoke the `pthread_mutex_unlock` and pass the `mutex` object `pop_mutex` to it to unlock it. Also, the `mutex` object `push_mutex` will be unlocked. At the bottom of the `push` function, we will display the message Value is pushed to stack.

In the `pop` function, the `mutex` object `pop_mutex` will be locked, and then it will try to lock the `push_mutex` object that is already locked by first thread. The value in the stack, that is, pointed at by the pointer `top` will be popped. Because the value of `top` is 0, the value at the stack [0] location will be picked up and assigned to the variable `k`. Thereafter, the value of `top` will decrement by 1 to make it -1 again. The value, that is, popped from the stack will be displayed on the screen. Then, the `mutex` object `push_mutex` will be unlocked, followed by unlocking the `pop_mutex` object.

In the `main` function, we will invoke the `pthread_join` method twice and pass the `tid1` and `tid2` thread identifiers to it. The reason that we invoke the `pthread_join` method is to make the two threads and the `main` method wait until both of the threads have completed their tasks.

In this program, a deadlock has occurred because in the `push` function, the first thread locked the `push_mutex` object and tried to get the lock of the `pop_mutex` object, which was already locked by the second thread in the `pop` function. In the `pop` function, the thread locked the `mutex` object `pop_mutex` and tried to lock the `push_mutex` object, which was already locked by the first thread. So, neither of the threads will be able to finish, and they will keep waiting indefinitely for the other thread to release its `mutex` object.

Let's use GCC to compile the `deadlockstate.c` program, as follows:

```
D:\CBook>gcc deadlockstate.c -o deadlockstate
```

If you get no errors or warnings, that means the `deadlockstate.c` program is compiled into an executable file, `deadlockstate.exe`. Let's run this executable file:

```
D:\Chap5>deadlockstate
Enter the value to push: Value popped is 0
Both threads are created
```

You've now seen how a deadlock can occur. Now, let's move on to the next recipe!

Avoiding a deadlock

A deadlock can be avoided if the threads are allowed to acquire the locks in a sequence. Let's suppose that a thread acquires the lock for a resource and wants to acquire the lock for a second resource. Any other thread that tries to acquire the first lock will be asked to wait, as it was already acquired by the first thread. Therefore, the second thread will not be able to acquire the lock for the second resource either, since it can only acquire locks in a sequence. However, our first thread will be allowed to acquire the lock to the second resource without waiting.

Applying a sequence to the locking of resources is the same as allowing only one thread to acquire resources at a time. The other threads will only be able to acquire the resources after the previous thread is over. This way, we will not have a deadlock on our hands.

How to do it...

1. Define an array of 10 elements:

```
#define max 10  
int stack[max];
```

2. Define two `mutex` objects—one to indicate the `pop` operation of the stack (`pop_mutex`), and another to represent the `push` operation of the stack (`push_mutex`):

```
pthread_mutex_t pop_mutex;  
pthread_mutex_t push_mutex;
```

3. To use the `stack`, the value of `top` is initialized to -1:

```
int top=-1;
```

4. Define two variables of the type `pthread_t`, to store two thread identifiers:

```
pthread_t tid1,tid2;
```

5. Invoke the `pthread_create` function to create the first thread. The thread is created with the default attributes, and the `push` function is executed to create the thread:

```
pthread_create(&tid1,NULL,&push,NULL);
```

6. Invoke the `pthread_create` function again to create the second thread. The thread is created with the default attributes, and the `pop` function is executed to create this thread:

```
pthread_create(&tid2, NULL, &pop, NULL);
```

7. To indicate that the two threads were created, display the message Both threads are created:

```
printf("Both threads are created\n");
```

8. In the `push` function, invoke the `pthread_mutex_lock` method and pass the `mutex` object `push_mutex`, related to the `push` operation, to it, in order to lock it:

```
pthread_mutex_lock(&push_mutex);
```

9. After a sleep of 2 seconds, the `mutex` object, that is, meant to invoke the `pop` operation `pop_mutex` will be locked by the first thread:

```
sleep(2);
pthread_mutex_lock(&pop_mutex);
```

10. Enter the value to be pushed to the stack:

```
printf("Enter the value to push: ");
scanf("%d", &n);
```

11. The value of `top` is incremented to 0. To `stack[0]` location, the value, that is, entered by the user is pushed:

```
top++;
stack[top]=n;
```

12. Invoke `pthread_mutex_unlock` and pass the `mutex` object `pop_mutex` to it to unlock it. Also, the `mutex` object `push_mutex` will be unlocked:

```
pthread_mutex_unlock(&pop_mutex);
pthread_mutex_unlock(&push_mutex);
```

13. At the bottom of the `push` function, display the message Value is pushed to stack:

```
printf("Value is pushed to stack \n");
```

14. In the `pop` function, the `pthread_mutex_lock` function is invoked to lock the `mutex` object `push_mutex`:

```
pthread_mutex_lock(&push_mutex);
```

15. After a sleep (or delay) of 5 seconds, the `pop` function will try to lock the `pop_mutex` object, too. However, the `pthread_mutex_lock` function will not be invoked, as the thread is kept waiting for the `push_mutex` object to be unlocked:

```
sleep(5);
pthread_mutex_lock(&pop_mutex);
```

16. The value in the stack pointed to by the pointer `top` is popped. Because the value of `top` is 0, the value at the location `stack[0]` is picked up:

```
k=stack[top];
```

17. Thereafter, the value of `top` will be decremented by 1 to make it -1 again. The value, that is, popped from the stack will be displayed on the screen:

```
top--;
printf("Value popped is %d \n",k);
```

18. Then, the `mutex` object `pop_mutex` will be unlocked, followed by the `push_mutex` object:

```
pthread_mutex_unlock(&pop_mutex);
pthread_mutex_unlock(&push_mutex);
```

19. In the `main` function, invoke the `pthread_join` method twice and pass the thread identifiers that were created in step 1 to it:

```
pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
```

The `avoiddeadlockst.c` program for creating two threads and understanding how a deadlock can be avoided while acquiring locks is as follows:

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

#define max 10
pthread_mutex_t pop_mutex;
```

```
pthread_mutex_t push_mutex;
int stack[max];
int top=-1;

void * push(void *arg) {
    int n;
    pthread_mutex_lock(&push_mutex);
    sleep(2);
    pthread_mutex_lock(&pop_mutex);
    printf("Enter the value to push: ");
    scanf("%d",&n);
    top++;
    stack[top]=n;
    pthread_mutex_unlock(&pop_mutex);
    pthread_mutex_unlock(&push_mutex);
    printf("Value is pushed to stack \n");
}

void * pop(void *arg) {
    int k;
    pthread_mutex_lock(&push_mutex);
    sleep(5);
    pthread_mutex_lock(&pop_mutex);
    k=stack[top];
    top--;
    printf("Value popped from stack is %d \n",k);
    pthread_mutex_unlock(&pop_mutex);
    pthread_mutex_unlock(&push_mutex);
}

int main() {
    pthread_t tid1,tid2;
    pthread_create(&tid1,NULL,&push,NULL);
    pthread_create(&tid2,NULL,&pop,NULL);
    printf("Both threads are created\n");
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    return 0;
}
```

Now, let's go behind the scenes.

How it works...

We will start by defining a macro called `max` of the value 10. Then, we will define an array `stack` of the size `max`. We will define two `mutex` objects with the names `pop_mutex` and `push_mutex`.

To use the stack, the value of `top` will be initialized to `-1`. We will define two variables of the type `pthread_t`, with the names `tid1` and `tid2`, to store two thread identifiers.

We will invoke the `pthread_create` function to create the first thread and assign the identifier returned by the function to the variable `tid1`. The thread will be created with the default attributes, and the `push` function will be executed to create this thread.

We will invoke the `pthread_create` function a second time to create the second thread, and we'll assign the identifier returned by the function to the variable `tid2`. The thread will be created with the default attributes and the `pop` function will be executed to create this thread. On the screen, we will display the message Both threads are created.

In the `push` function, the `pthread_mutex_lock` method is invoked, and the `mutex` object `push_mutex` is passed to it to lock it. Now, if any other thread asks for the `pop_mutex` object, it will need to wait until the object is unlocked. After a sleep of 2 seconds, the `mutex` object `pop_mutex` is locked by the first thread.

We will be prompted to enter the value to be pushed to the stack. The entered value will be assigned to the variable `n`. The value of `top` will increment to 0. The value that we enter will be pushed to the location `stack[0]`. Now, the `pthread_mutex_unlock` will be invoked, and the `mutex` object `pop_mutex` will be passed to it to unlock it. Also, the `mutex` object `push_mutex` will be unlocked. At the bottom of the `push` function, the message Value is pushed to stack will be displayed.

In the `pop` function, it will try to lock the `mutex` object `push_mutex`, but because it is already locked by the first thread, this thread will be asked to wait. After a sleep or delay of 5 seconds, it will also try to lock the `pop_mutex` object. The value in the stack, that is, pointed at by the pointer `top` will be popped. Because the value of `top` is 0, the value at `stack[0]` is picked up and assigned to the variable `k`. Thereafter, the value of `top` will decrement by 1 to make it `-1` again. The value, that is, popped from the stack will be displayed on the screen. Then, the `mutex` object `pop_mutex` will be unlocked, followed by the `push_mutex` object.

In the `main` function, the `pthread_join` method is invoked twice, and the `tid1` and `tid2` thread identifiers are passed to it. The `pthread_join` is invoked to make the two threads and the `main` method wait until both of the threads have completed their tasks.

Here, we avoided a deadlock because the locking and unlocking of the `mutex` objects was done in a sequence. In the `push` function, the first thread locked the `push_mutex` object and tried to get a lock on the `pop_mutex` object. The `pop_mutex` was kept free because the second thread in the `pop` function first tried to lock the `push_mutex` object, followed by the `pop_mutex` object. Since the first thread had already locked the `push_mutex` object, the second thread was asked to wait. Consequently, both of the `mutex` objects, `push_mutex` and `pop_mutex`, were in an unlocked state, and the first thread was able to easily lock both of the `mutex` objects and use the common resource. After finishing its task, the first thread will unlock both of the `mutex` objects, enabling the second thread to lock both of the `mutex` objects and access the common resource thread.

Let's use GCC to compile the `avoiddeadlockst.c` program, as follows:

```
D:\CBook>gcc avoiddeadlockst.c -o avoiddeadlockst
```

If you get no errors or warnings, that means the `avoiddeadlockst.c` program has been compiled into an executable file, `avoiddeadlockst.exe`. Let's run this executable file:

```
D:\Chap5>avoiddeadlockst
Both threads are created
Enter the value to push: 5
Value is pushed to stack
Value popped from stack is 5
```

Voila! We've successfully avoided a deadlock.

7

Networking and Interprocess Communication

Processes run individually and work independently in their respective address spaces. However, they sometimes need to communicate with each other to pass on information. For processes to cooperate, they need to be able to communicate with each other as well as synchronize their actions. Here are the types of communication that take place between processes:

- **Synchronous communication:** Such communication doesn't allow the process to continue with any other work until the communication is over
- **Asynchronous communication:** In this communication, the process can continue doing other tasks, and so it supports multitasking and results in better efficiency
- **Remote Procedure Call (RPC):** This is a protocol that uses client service techniques for communication where the client cannot do anything, that is, it is suspended until it gets a response from the server

These communications can be unidirectional or bidirectional. To enable any form of communication between processes, the following popular **interprocess communication (IPC)** mechanisms are used: pipes, FIFOs (named pipes), sockets, message queues, and shared memory. Pipes and FIFO enable unidirectional communication, whereas sockets, message queues, and shared memory enable bidirectional communication.

In this chapter, we will learn how to make the following recipes so that we can establish communication between processes:

- Communicating between processes using pipes
- Communicating between processes using FIFO
- Communicating between the client and server using socket programming
- Communicating between processes using a UDP socket
- Passing a message from one process to another using the message queue
- Communicating between processes using shared memory

Let's begin with the first recipe!

Communicating between processes using pipes

In this recipe, we will learn how to write data into a pipe from its writing end and then how to read that data from its reading end. This can happen in two ways:

- One process, both writing and reading from the pipe
- One process writing and another process reading from the pipe

Before we begin with the recipes, let's quickly review the functions, structures, and terms that are used in successful interprocess communication.

Creating and connecting processes

The most commonly used functions and terms for communication between processes are `pipe`, `mkfifo`, `write`, `read`, `perror`, and `fork`.

pipe()

A pipe is used for connecting two processes. The output from one process can be sent as an input to another process. The flow is unidirectional, that is, one process can write to the pipe and another process can read from the pipe. Writing and reading are done in an area of main memory, which is also known as a virtual file. Pipes have a **First in First out (FIFO)** or a queue structure, that is, what is written first will be read first.



A process should not try to read from the pipe before something is written into it, otherwise it will suspend until something is written into the pipe.

Here is its syntax:

```
int pipe(int arr[2]);
```

Here, `arr[0]` is the file descriptor for the read end of the pipe, and `arr[1]` is the file descriptor for the write end of the pipe.

The function returns `0` on success and `-1` on error.

mkfifo()

This function creates a new FIFO special file. Here is its syntax:

```
int mkfifo(const char *filename, mode_t permission);
```

Here, `filename` represents the filename, along with its complete path, and `permission` represents the permission bits of the new FIFO file. The default permissions are read and write permission for the owner, group, and others, that is, (0666).

The function returns `0` on successful completion; otherwise, it returns `-1`.

write()

This function is used for writing into the specified file or pipe whose descriptor is supplied. Here is its syntax:

```
write(int fp, const void *buf, size_t n);
```

It writes the `n` number of bytes into the file that's being pointed to by the file pointer, `fp`, from the buffer, `buf`.

read()

This function reads from the specified file or pipe whose descriptor is supplied in the method. Here is its syntax:

```
read(int fp, void *buf, size_t n);
```

It tries to read up to n number of bytes from a file that's being pointed to by a descriptor, `fp`. The bytes that are read are then assigned to the buffer, `buf`.

perror()

This displays an error message indicating the error that might have occurred while invoking a function or system call. The error message is displayed to `stderr`, that is, the standard error output stream. This is basically the console.

Here is its syntax:

```
void perror ( const char * str );
```

The error message that is displayed is optionally preceded by the message that's represented by `str`.

fork()

This is used for creating a new process. The newly created process is called the child process, and it runs concurrently with the parent process. After executing the `fork` function, the execution of the program continues and the instruction following the `fork` function is executed by the parent as well as the child process. If the system call is successful, it will return a process ID of the child process and returns a 0 to the newly created child process. The function returns a negative value if the child process is not created.

Now, let's start with the first recipe for enabling communication between processes using pipes.

One process, both writing and reading from the pipe

Here, we will learn how writing and reading from the pipe are done by a single process.

How to do it...

1. Define an array of size 2 and pass it as an argument to the `pipe` function.
2. Invoke the `write` function and write your chosen string into the pipe through the `write` end of the array. Repeat the procedure for the second message.
3. Invoke the `read` function to read the first message from the pipe. Invoke the `read` function again to read the second message.

The `readwritepipe.c` program for writing into the pipe and reading from it thereafter is as follows:

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

#define max 50

int main()
{
    char str[max];
    int pp[2];

    if (pipe(pp) < 0)
        exit(1);
    printf("Enter first message to write into pipe: ");
    gets(str);
    write(pp[1], str, max);
    printf("Enter second message to write into pipe: ");
    gets(str);
    write(pp[1], str, max);
    printf("Messages read from the pipe are as follows:\n");
    read(pp[0], str, max);
    printf("%s\n", str);
    read(pp[0], str, max);
    printf("%s\n", str);
    return 0;
}
```

Let's go behind the scenes.

How it works...

We defined a macro, `max`, of size of 50, a string, `str`, of size `max`, and an array, `pp`, with size 2. We will invoke the `pipe` function to connect two processes and pass the `pp` array to it. The index location, `pp[0]`, will get the file descriptor for the reading end of the pipe and `pp[1]` will get the file descriptor for the write end of the pipe. The program will exit if the `pipe` function does not execute successfully.

You will be prompted to enter the first message to be written into the pipe. The text that's entered by you will be assigned to the string variable, `str`. Invoke the `write` function and the string in `str` will be written into the pipe, `pp`. Repeat the procedure for the second message. The second text that's entered by you will also be written into the pipe.

Obviously, the second text will be written behind the first text in the pipe. Now, invoke the `read` function to read from the pipe. The text that was entered first in the pipe will be read and assigned to the string variable, `str`, and is consequently displayed on the screen. Again, invoke the `read` function and the second text message in the pipe will be read from its read end and assigned to the string variable, `str`, and then displayed on the screen.

Let's use GCC to compile the `readwritepipe.c` program, as follows:

```
$ gcc readwritepipe.c -o readwritepipe
```

If you get no errors or warnings, this means that the `readwritepipe.c` program has been compiled into an executable file, `readwritepipe.exe`. Let's run this executable file:

```
$ ./readwritepipe
Enter the first message to write into pipe: This is the first message for
the pipe
Enter the second message to write into pipe: Second message for the pipe
Messages read from the pipe are as follows:
This is the first message for the pipe
Second message for the pipe
```

In the preceding program, the main thread does the job of writing and reading from the pipe. But what if we want one process to write into the pipe and another process to read from the pipe? Let's find out how we can make that happen.

One process writing into the pipe and another process reading from the pipe

In this recipe, we will use the fork system call to create a child process. Then, we will write into the pipe using the child process and read from the pipe through the parent process, thereby establishing communication between two processes.

How to do it...

1. Define an array of size 2.
2. Invoke the `pipe` function to connect the two processes and pass the array we defined previously to it.
3. Invoke the `fork` function to create a new child process.
4. Enter the message that is going to be written into the pipe. Invoke the `write` function using the newly created child process.
5. The parent process invokes the `read` function to read the text that's been written into the pipe.

The `pipedemo.c` program for writing into the pipe through a child process and reading from the pipe through the parent process is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define max 50

int main()
{
    char wstr[max];
    char rstr[max];
    int pp[2];
    pid_t p;
    if(pipe(pp) < 0)
    {
        perror("pipe");
    }
    p = fork();
    if(p >= 0)
    {
        if(p == 0)
```

```
{  
    printf ("Enter the string : ");  
    gets(wstr);  
    write (pp[1] , wstr , strlen(wstr));  
    exit(0);  
}  
else  
{  
    read (pp[0] , rstr , sizeof(rstr));  
    printf("Entered message : %s\n " , rstr);  
    exit(0);  
}  
else  
{  
    perror("fork");  
    exit(2);  
}  
return 0;  
}
```

Let's go behind the scenes.

How it works...

Define a macro `max`, of size 50 and two string variables, `wstr` and `rstr`, of size `max`. The `wstr` string will be used for writing into the pipe and `rstr` will be used for reading from the pipe. Define an array, `pp`, of size 2, which will be used for storing the file descriptors of the read and write ends of the pipe. Define a variable, `p`, of the `pid_t` data type, which will be used for storing a process ID.

We will invoke the `pipe` function to connect the two processes and pass the `pp` array to it. The index location `pp[0]` will get the file descriptor for the reading end of the pipe, while `pp[1]` will get the file descriptor for the write end of the pipe. The program will exit if the `pipe` function does not execute successfully.

Then, we will invoke the `fork` function to create a new child process. You will be prompted to enter the message to be written into the pipe. The text you enter will be assigned to the string variable `wstr`. When we invoke the `write` function using the newly created child process, the string in the `wstr` variable will be written into the pipe, `pp`. Thereafter, the parent process will invoke the `read` function to read the text that's been written into the pipe. The text that's read from the pipe will be assigned to the string variable `rstr` and will consequently be displayed on the screen.

Let's use GCC to compile the `pipedemo.c` program, as follows:

```
$ gcc pipedemo.c -o pipedemo
```

If you get no errors or warnings, this means that the `pipedemo.c` program has been compiled into an executable file, `pipedemo.exe`. Let's run this executable file:

```
$ ./pipedemo
Enter the string : This is a message from the pipe
Entered message : This is a message from the pipe
```

Voila! We've successfully communicated between processes using pipes. Now, let's move on to the next recipe!

Communicating between processes using FIFO

In this recipe, we will learn how two processes communicate using a named pipe, also known as FIFO. This recipe is divided into the following two parts:

- Demonstrating how data is written into a FIFO
- Demonstrating how data is read from a FIFO

The functions and terms we learned in the previous recipe will also be applicable here.

Writing data into a FIFO

As the name suggests, we will learn how data is written into a FIFO in this recipe.

How to do it...

1. Invoke the `mkfifo` function to create a new FIFO special file.
2. Open the FIFO special file in write-only mode by invoking the `open` function.
3. Enter the text to be written into the FIFO special file.
4. Close the FIFO special file.

The `writefifo.c` program for writing into a FIFO is as follows:

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
    int fw;
    char str[255];
    mkfifo("FIFOPipe", 0666);
    fw = open("FIFOPipe", O_WRONLY);
    printf("Enter text: ");
    gets(str);
    write(fw, str, sizeof(str));
    close(fw);
    return 0;
}
```

Let's go behind the scenes.

How it works...

Let's assume we have defined a string called `str` of size 255. We will invoke the `mkfifo` function to create a new FIFO special file. We will create the FIFO special file with the name `FIFOPipe` with read and write permissions for owner, group, and others.

We will open this FIFO special file in write-only mode by invoking the `open` function. Then, we will assign the file descriptor of the opened FIFO special file to the `fw` variable. You will be prompted to enter the text that is going to be written into the file. The text you enter will be assigned to the `str` variable, which in turn will be written into the special FIFO file when you invoke the `write` function. Finally, close the FIFO special file.

Let's use GCC to compile the `writefifo.c` program, as follows:

```
$ gcc writefifo.c -o writefifo
```

If you get no errors or warnings, this means that the `writefifo.c` program has compiled into an executable file, `writefifo.exe`. Let's run this executable file:

```
$ ./writefifo
Enter text: This is a named pipe demo example called FIFO
```

If your program does not prompts for the string that means it is waiting for the other end of the FIFO to open. That is, you need to run the next recipe, *Reading data from a FIFO*, on the second Terminal screen. Please press *Alt+F2* on Cygwin to open the next terminal screen.

Now, let's check out the other part of this recipe.

Reading data from a FIFO

In this recipe, we will see how we can read data from a FIFO.

How to do it...

1. Open the FIFO special file in read-only mode by invoking the `open` function.
2. Read the text from the FIFO special file using the `read` function.
3. Close the FIFO special file.

The `readfifo.c` program for reading from the named pipe (FIFO) is as follows:

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define BUFFSIZE 255

int main()
{
    int fr;
    char str[BUFFSIZE];
    fr = open("FIFOPipe", O_RDONLY);
    read(fr, str, BUFFSIZE);
    printf("Read from the FIFO Pipe: %s\n", str);
    close(fr);
    return 0;
}
```

Let's go behind the scenes.

How it works...

We will start by defining a macro called `BUFFSIZE` of size 255 and a string called `str` of size `BUFFSIZE`, that is, 255 characters. We will open the FIFO special file named `FIFOPipe` in read-only mode by invoking the `open` function. The file descriptor of the opened FIFO special file will be assigned to the `fr` variable.

Using the `read` function, the text from the FIFO special file will be read and assigned to the `str` string variable. The text that's read from the FIFO special file will then be displayed on the screen. Finally, the FIFO special file will be closed.

Now, press `Alt + F2` to open a second Terminal window. In the second Terminal window, let's use GCC to compile the `readfifo.c` program, as follows:

```
$ gcc readfifo.c -o readfifo
```

If you get no errors or warnings, this means that the `readfifo.c` program has compiled into an executable file, `readfifo.exe`. Let's run this executable file:

```
$ ./readfifo
Read from the FIFO Pipe: This is a named pipe demo example called FIFO
```

The moment you run the `readfifo.exe` file, you will find, that on the previous Terminal screen where `writefifo.c` program was run will prompt you to enter a string. The moment you enter a string on that Terminal and press `Enter` key, you will get the output from the `readfifo.c` program.

Voila! We've successfully communicated between processes using a FIFO. Now, let's move on to the next recipe!

Communicating between the client and server using socket programming

In this recipe, we will learn how data from the server process is sent to the client process. This recipe is divided into the following parts:

- Sending data to the client
- Reading data that's been sent from the server

Before we begin with the recipes, let's quickly review the functions, structures, and terms that are used in successful client-server communication.

Client-server model

Different models are used for IPC, but the most popular one is the client-server model. In this model, whenever the client needs some information, it connects to another process called the server. But before establishing the connection, the client needs to know whether the server already exists, and it should know the address of the server.

On the other hand, the server is meant to serve the needs of the client and does not need to know the address of the client prior to the connection. To establish a connection, a basic construct called a socket is required, and both the connecting processes must establish their own sockets. The client and the server need to follow certain procedures to establish their sockets.

To establish a socket on the client side, a socket is created with the `socket` function system call. Thereafter, that socket is connected to the server's address using the `connect` function system call, followed by sending and receiving data by invoking the `read` function and `write` function system calls.

To establish a socket on the server side, again, a socket is created with the `socket` function system call and then the socket is bonded to an address using the `bind` function system call. Thereafter, the `listen` function system call is invoked to listen for the connections. Finally, the connection is accepted by invoking the `accept` function system call.

struct sockaddr_in structure

This structure references the socket's elements that are used for keeping addresses. The following are the built-in members of this structure:

```
struct sockaddr_in {  
    short int sin_family;  
    unsigned short int sin_port;  
    struct in_addr sin_addr;  
    unsigned char sin_zero[8];  
};
```

Here, we have the following:

- `sin_family`: Represents an address family. The valid options are `AF_INET`, `AF_UNIX`, `AF_NS`, and `AF_IMPLINK`. In most applications, the address family that's used is `AF_INET`.
- `sin_port`: Represents the 16-bit service port number.
- `sin_addr`: Represents a 32-bit IP address.
- `sin_zero`: This is not used and is usually set to `NULL`.

`struct in_addr` comprise one member, as follows:

```
struct in_addr {  
    unsigned long s_addr;  
};
```

Here, `s_addr` is used to represent the address in network byte order.

socket()

This function creates an endpoint for communication. To establish communication, every process needs a socket at the end of the communication line. Also, the two communicating processes must have the same socket type and both should be in the same domain. Here is the syntax for creating a socket:

```
int socket(int domain, int type, int protocol);
```

Here, `domain` represents the communication domain in which a socket is to be created. Basically, the address family or protocol family is specified, which will be used in the communication.

A few of the popular address family are listed as follows:

- `AF_LOCAL`: This is used for local communication.
- `AF_INET`: This is used for IPv4 internet protocols.
- `AF_INET6`: This is used for IPv6 internet protocols.
- `AF_IPX`: This is used for protocols that use standard **IPX** (short for **Internetwork Packet Exchange**) socket addressing.
- `AF_PACKET`: This is used for packet interface.

- `type`: Represents the type of socket to be created. The following are the popular socket types:
 - `SOCK_STREAM`: Stream sockets communicate as a continuous stream of characters using a **Transmission Control Protocol (TCP)**. TCP is a reliable stream-oriented protocol. So, the `SOCK_STREAM` type provides reliable, bidirectional, and connection-based byte streams.
 - `SOCK_DGRAM`: Datagram sockets read the entire messages at once using a **User Datagram Protocol (UDP)**. UDP is an unreliable, connectionless, and message-oriented protocol. These messages are of a fixed maximum length.
 - `SOCK_SEQPACKET`: Provides reliable, bidirectional, and connection-based transmission paths for datagrams.
- `protocol`: Represents the protocol to be used with the socket. A `0` value is specified so that you can use the default protocol that's suitable for the requested socket type.



You can replace the `AF_` prefix in the preceding list with `PF_` for protocol family.

On successful execution, the `socket` function returns a file descriptor that can be used to manage sockets.

memset()

This is used to fill a block of memory with the specified value. Here is its syntax:

```
void *memset(void *ptr, int v, size_t n);
```

Here, `ptr` points at the memory address to be filled, `v` is the value to be filled in the memory block, and `n` is the number of bytes to be filled, starting at the location of the pointer.

hton()

This is used to convert the unsigned short integer from host to network byte order.

bind()

A socket that is created with the `socket` function remains in the assigned address family. To enable the socket to receive connections, an address needs to be assigned to it. The `bind` function assigns the address to the specified socket. Here is its syntax:

```
int bind(int fdsock, const struct sockaddr *structaddr, socklen_t  
lenaddr);
```

Here, `fdsock` represents the file descriptor of the socket, `structaddr` represents the `sockaddr` structure that contains the address to be assigned to the socket, and `lenaddr` represents the size of the address structure that's pointed to by `structaddr`.

listen()

It listens for connections on a socket in order to accept incoming connection requests. Here is its syntax:

```
int listen(int sockfd, int lenque);
```

Here, `sockfd` represents the file descriptor of the socket, and `lenque` represents the maximum length of the queue of pending connections for the given socket. An error will be generated if the queue is full.

If the function is successful it returns zero, otherwise it returns `-1`.

accept()

It accepts a new connection on the listening socket, that is, the first connection from the queue of pending connections is picked up. Actually, a new socket is created with the same socket type protocol and address family as the specified socket, and a new file descriptor is allocated for that socket. Here is its syntax:

```
int accept(int socket, struct sockaddr *address, socklen_t *len);
```

Here, we need to address the following:

- `socket`: Represents the file descriptor of the socket waiting for the new connection. This is the socket that is created when the `socket` function is bound to an address with the `bind` function, and has invoked the `listen` function successfully.

- **address:** The address of the connecting socket is returned through this parameter. It is a pointer to a `sockaddr` structure, through which the address of the connecting socket is returned.
- **len:** Represents the length of the supplied `sockaddr` structure. When returned, this parameter contains the length of the address returned in bytes.

send()

This is used for sending the specified message to another socket. The socket needs to be in a connected state before you can invoke this function. Here is its syntax:

```
ssize_t send(int fdsock, const void *buf, size_t length, int flags);
```

Here, `fdsock` represents the file descriptor of the socket through which a message is to be sent, `buf` points to the buffer that contains the message to be sent, `length` represents the length of the message to be sent in bytes, and `flags` specifies the type of message to be sent. Usually, its value is kept at 0.

connect()

This initiates a connection on a socket. Here is its syntax:

```
int connect(int fdsock, const struct sockaddr *addr, socklen_t len);
```

Here, `fdsock` represents the file descriptor of the socket onto which the connection is desired, `addr` represents the structure that contains the address of the socket, and `len` represents the size of the structure `addr` that contains the address.

recv()

This is used to receive a message from the connected socket. The socket may be in connection mode or connectionless mode. Here is its syntax:

```
ssize_t recv(int fdsock, void *buf, size_t len, int flags);
```

Here, `fdsock` represents the file descriptor of the socket from which the message has to be fetched, `buf` represents the buffer where the message that is received is stored, `len` specifies the length in bytes of the buffer that's pointed to by the `buf` argument, and `flags` specifies the type of message being received. Usually, its value is kept at 0.

We can now begin with the first part of this recipe – how to send data to the client.

Sending data to the client

In this part of the recipe, we will learn how a server sends desired data to the client.

How to do it...

1. Define a variable of type `sockaddr_in`.
2. Invoke the `socket` function to create a socket. The port number that's specified for the socket is 2000.
3. Call the `bind` function to assign an IP address to it.
4. Invoke the `listen` function.
5. Invoke the `accept` function.
6. Invoke the `send` function to send the message that was entered by the user to the socket.
7. The socket at the client end will receive the message.

The server program, `serverprog.c`, for sending a message to the client is as follows:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

int main(){
    int serverSocket, toSend;
    char str[255];
    struct sockaddr_in server_Address;
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    server_Address.sin_family = AF_INET;
    server_Address.sin_port = htons(2000);
    server_Address.sin_addr.s_addr = inet_addr("127.0.0.1");
    memset(server_Address.sin_zero, '\0', sizeof
    server_Address.sin_zero);
    bind(serverSocket, (struct sockaddr *) &server_Address,
    sizeof(server_Address));
    if(listen(serverSocket,5)==-1)
    {
        printf("Not able to listen\n");
        return -1;
    }
    printf("Enter text to send to the client: ");
    gets(str);
```

```
    toSend = accept(serverSocket, (struct sockaddr *) NULL, NULL);
    send(toSend,str, strlen(str),0);
    return 0;
}
```

Let's go behind the scenes.

How it works...

We will start by defining a string of size 255, and a `server_Address` variable of type `sockaddr_in`. This structure references the socket's elements. Then, we will invoke the `socket` function to create a socket by the name of `serverSocket`. A socket is an endpoint for communication. The address family that's supplied for the socket is `AF_INET`, and the socket type selected is the stream socket type, since the communication that we want is of a continuous stream of characters.

The address family that's specified for the socket is `AF_INET`, and is used for IPv4 internet protocols. The port number that's specified for the socket is 2000. Using the `htonl` function, the short integer 2000 is converted into the network byte order before being applied as a port number. The fourth parameter, `sin_zero`, of the `server_Address` structure is set to `NULL` by invoking the `memset` function.

To enable the created `serverSocket` to receive connections, call the `bind` function to assign an address to it. Using the `sin_addr` member of the `server_Address` structure, a 32-bit IP address will be applied to the socket. Because we are working on the local machine, the localhost address `127.0.0.1` will be assigned to the socket. Now, the socket can receive the connections. We will invoke the `listen` function to enable the `serverSocket` to accept incoming connection requests. The maximum pending connections that the socket can have is 5.

You will be prompted to enter the text that is to be sent to the client. The text you enter will be assigned to the `str` string variable. By invoking the `accept` function, we will enable the `serverSocket` to accept a new connection.

The address of the connection socket will be returned through the structure of type `sockaddr_in`. The socket that is returned and that is ready to accept a connection is named `toSend`. We will invoke the `send` function to send the message that's entered by you. The socket at the client end will receive the message.

Let's use GCC to compile the `serverprog.c` program, as follows:

```
$ gcc serverprog.c -o serverprog
```

If you get no errors or warnings, this means that the `serverprog.c` program has compiled into an executable file, `serverprog.exe`. Let's run this executable file:

```
$ ./serverprog
Enter text to send to the client: thanks and good bye
```

Now, let's look at the other part of this recipe.

Reading data that's been sent from the server

In this part of the recipe, we will learn how data that's been sent from the server is received and displayed on the screen.

How to do it...

1. Define a variable of type `sockaddr_in`.
2. Invoke the `socket` function to create a socket. The port number that's specified for the socket is 2000.
3. Invoke the `connect` function to initiate a connection to the socket.
4. Because we are working on the local machine, the localhost address `127.0.0.1` is assigned to the socket.
5. Invoke the `recv` function to receive the message from the connected socket. The message that's read from the socket is then displayed on the screen.

The client program, `clientprog.c`, for reading a message that's sent from the server is as follows:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>

int main(){
    int clientSocket;
    char str[255];
    struct sockaddr_in client_Address;
    socklen_t address_size;
    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    client_Address.sin_family = AF_INET;
    client_Address.sin_port = htons(2000);
    client_Address.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
    memset(client_Address.sin_zero, '\0', sizeof client_Address.sin_zero);
    address_size = sizeof server_Address;
    connect(clientSocket, (struct sockaddr *) &client_Address,
address_size);
    recv(clientSocket, str, 255, 0);
    printf("Data received from server: %s", str);
    return 0;
}
```

Let's go behind the scenes.

How it works...

So, we have defined a string of size 255 and a variable called `client_Address` of type `sockaddr_in`. We will invoke the `socket` function to create a socket by the name of `clientSocket`.

The address family that's supplied for the socket is `AF_INET` and is used for IPv4 internet protocols, and the socket type that's selected is stream socket type. The port number that's specified for the socket is 2000. By using the `htonl` function, the short integer 2000 is converted into the network byte order before being applied as a port number.

We will set the fourth parameter, `sin_zero`, of the `client_Address` structure to `NULL` by invoking the `memset` function. We will initiate the connection to the `clientSocket` by invoking the `connect` function. By using the `sin_addr` member of the `client_Address` structure, a 32-bit IP address is applied to the socket. Because we are working on the local machine, the localhost address `127.0.0.1` is assigned to the socket. Finally, we will invoke the `recv` function to receive the message from the connected `clientSocket`. The message that's read from the socket will be assigned to the `str` string variable, which will then be displayed on the screen.

Now, press `Alt + F2` to open a second Terminal window. Here, let's use `GCC` to compile the `clientprog.c` program, as follows:

```
$ gcc clientprog.c -o clientprog
```

If you get no errors or warnings, this means that the `clientprog.c` program has compiled into an executable file, `clientprog.exe`. Let's run this executable file:

```
$ ./clientprog
Data received from server: thanks and good bye
```

Voila! We've successfully communicated between the client and server using socket programming. Now, let's move on to the next recipe!

Communicating between processes using a UDP socket

In this recipe, we will learn how two-way communication is implemented between a client and a server using a UDP socket. This recipe is divided into the following parts:

- Awaiting a message from the client and sending a reply using a UDP socket
- Sending a message to the server and receiving the reply from the server using the UDP socket

Before we begin with these recipes, let's quickly review the functions, structures, and terms that are used in successful interprocess communication using a UDP socket.

Using a UDP socket for server-client communication

In the case of communication with UDP, the client does not establish a connection with the server but simply sends a datagram. The server does not have to accept a connection; it simply waits for datagrams to be sent from the client. Every datagram contains the address of the sender, enabling the server to identify the client on the basis of where the datagram is sent from.

For communication, the UDP server first creates a UDP socket and binds it to the server address. Then, the server waits until the datagram packet arrives from the client. Once it has arrived, the server processes the datagram packet and sends a reply to the client. This procedure keeps on repeating.

On the other hand, the UDP client, for communication, creates a UDP socket, sends a message to the server, and waits for the server's response. The client will keep repeating the procedure if they want to send more messages to the server, otherwise the socket descriptor will close.

bzero()

This places n zero-valued bytes in the specified area. Here is its syntax:

```
void bzero(void *r, size_t n);
```

Here, `r` is the area that's pointed to by `r` and `n` is the `n` number of zero values bytes that are placed in the area that was pointed to by `r`.

INADDR_ANY

This is an IP address that is used when we don't want to bind a socket to any specific IP. Basically, while implementing communication, we need to bind our socket to an IP address. When we don't know the IP address of our machine, we can use the special IP address `INADDR_ANY`. It allows our server to receive packets that have been targeted by any of the interfaces.

sendto()

This is used to send a message on the specified socket. The message can be sent in connection mode as well as in connectionless mode. In the case of connectionless mode, the message is sent to the specified address. Here it its syntax:

```
ssize_t sendto(int fdsock, const void *buff, size_t len, int flags, const  
               struct sockaddr *recv_addr, socklen_t recv_len);
```

Here, we need to address the following:

- `fdsock`: Specifies the file descriptor of the socket.
- `buff`: Points to a buffer that contains the message to be sent.
- `len`: Specifies the length of the message in bytes.
- `flags`: Specifies the type of the message that is being transmitted. Usually, its value is kept as 0.
- `recv_addr`: Points to the `sockaddr` structure that contains the receiver's address. The length and format of the address depends on the address family that's been assigned to the socket.
- `recv_len`: Specifies the length of the `sockaddr` structure that's pointed to by the `recv_addr` argument.

On successful execution, the function returns the number of bytes sent, otherwise it returns -1.

recvfrom()

This is used to receive a message from a connection-mode or connectionless-mode socket. Here it its syntax:

```
ssize_t recvfrom(int fdsock, void *buffer, size_t length, int flags, struct sockaddr *address, socklen_t *address_len);
```

Here, we need to address the following:

- `fdsock`: Represents the file descriptor of the socket.
- `buffer`: Represents the buffer where the message is stored.
- `length`: Represents the number of bytes of the buffer that are pointed to by the `buffer` parameter.
- `flags`: Represents the type of message that's received.
- `address`: Represents the `sockaddr` structure in which the sending address is stored. The length and format of the address depend on the address family of the socket.
- `address_len`: Represents the length of the `sockaddr` structure that's pointed to by the `address` parameter.

The function returns the length of the message that's written to the buffer, which is pointed to by the `buffer` argument.

Now, we can begin with the first part of this recipe: preparing a server to wait for and reply to a message from the client using a UDP socket.

Await a message from the client and sending a reply using a UDP socket

In this part of the recipe, we will learn how a server waits for the message from the client and how, on receiving a message from the client, it replies to the client.

How to do it...

1. Define two variables of type `sockaddr_in`. Invoke the `bzero` function to initialize the structure.
2. Invoke the `socket` function to create a socket. The address family that's supplied for the socket is `AF_INET`, and the socket type that's selected is datagram type.

3. Initialize the members of the `sockaddr_in` structure to configure the socket. The port number that's specified for the socket is 2000. Use `INADDR_ANY`, a special IP address, to assign an IP address to the socket.
4. Call the `bind` function to assign the address to it.
5. Call the `recvfrom` function to receive the message from the UDP socket, that is, from the client machine. A null character, `\0`, is added to the message that's read from the client machine and is displayed on the screen. Enter the reply that is to be sent to the client.
6. Invoke the `sendto` function to send the reply to the client.

The server program, `udps.c`, for waiting for a message from the client and sending a reply to it using a UDP socket is as follows:

```
#include <stdio.h>
#include <strings.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include<netinet/in.h>
#include <stdlib.h>

int main()
{
    char msgReceived[255];
    char msgforclient[255];
    int UDPSocket, len;
    struct sockaddr_in server_Address, client_Address;
    bzero(&server_Address, sizeof(server_Address));
    printf("Waiting for the message from the client\n");
    if ( (UDPSocket = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("Socket could not be created");
        exit(1);
    }
    server_Address.sin_addr.s_addr = htonl(INADDR_ANY);
    server_Address.sin_port = htons(2000);
    server_Address.sin_family = AF_INET;
    if ( bind(UDPSocket, (const struct sockaddr *)&server_Address,
    sizeof(server_Address)) < 0 )
    {
        perror("Binding could not be done");
        exit(1);
    }
    len = sizeof(client_Address);
    int n = recvfrom(UDPSocket, msgReceived, sizeof(msgReceived), 0,
    (struct sockaddr*)&client_Address,&len);
```

```
msgReceived[n] = '\0';
printf("Message received from the client: ");
puts(msgReceived);
printf("Enter the reply to be sent to the client: ");
gets(msgforclient);
sendto(UDPSocket, msgforclient, 255, 0, (struct
sockaddr*)&client_Address, sizeof(client_Address));
printf("Reply to the client sent \n");
}
```

Let's go behind the scenes.

How it works...

We start by defining two strings by the names of `msgReceived` and `msgforclient`, both of which are of size 255. These two strings will be used to receive the message from and send a message to the client, respectively. Then, we will define two variables, `server_Address` and `client_Address`, of type `sockaddr_in`. These structures will reference the socket's elements and store the server's and client's addresses, respectively. We will invoke the `bzero` function to initialize the `server_Address` structure, that is, zeros will be filled in for all of the members of the `server_Address` structure.

The server, as expected, waits for the datagram from the client. So, the following text message is displayed on the screen: Waiting for the message from the client. We invoke the `socket` function to create a socket by the name of `UDPSocket`. The address family that's supplied for the socket is `AF_INET`, and the socket type that's selected is `datagram`. The members of the `server_Address` structure are initialized to configure the socket.

Using the `sin_family` member, the address family that's specified for the socket is `AF_INET`, which is used for IPv4 internet protocols. The port number that's specified for the socket is 2000. Using the `htons` function, the short integer 2000 is converted into the network byte order before being applied as a port number. Then, we use a special IP address, `INADDR_ANY`, to assign an IP address to the socket. Using the `htonl` function, the `INADDR_ANY` will be converted into the network byte order before being applied as the address to the socket.

To enable the created socket, `UDPSocket`, to receive connections, we will call the `bind` function to assign the address to it. We will call the `recvfrom` function to receive the message from the UDP socket, that is, from the client machine. The message that's read from the client machine is assigned to the `msgReceived` string, which is supplied in the `recvfrom` function. A null character, `\0`, is added to the `msgReceived` string and is displayed on the screen. Thereafter, you will be prompted to enter the reply to be sent to the client. The reply that's entered is assigned to `msgforclient`. By invoking the `sendto` function, the reply is sent to the client. After sending the message, the following message is displayed to the screen: Reply to the client sent.

Now, let's look at the other part of this recipe.

Sending a message to the server and receiving the reply from the server using the UDP socket

As the name suggests, in this recipe we will show you how the client sends a message to the server and then receives a reply from the server using the UDP socket.

How to do it...

1. Execute the first three steps from the previous part of this recipe. Assign the localhost IP address, `127.0.0.1`, as the address to the socket.
2. Enter the message to be sent to the server. Invoke the `sendto` function to send the message to the server.
3. Invoke the `recvfrom` function to get the message from the server. The message that's received from the server is then displayed on the screen.
4. Close the descriptor of the socket.

The client program, `udpc.c`, to send a message to the server and to receive the reply using a UDP socket is as follows:

```
#include <stdio.h>
#include <strings.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#include<stdlib.h>
```

```
int main()
{
    char msgReceived[255];
    char msgforserver[255];
    int UDPSocket, n;
    struct sockaddr_in client_Address;
    printf("Enter the message to send to the server: ");
    gets(msgforserver);
    bzero(&client_Address, sizeof(client_Address));
    client_Address.sin_addr.s_addr = inet_addr("127.0.0.1");
    client_Address.sin_port = htons(2000);
    client_Address.sin_family = AF_INET;
    if ( (UDPSocket = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("Socket could not be created");
        exit(1);
    }
    if(connect(UDPSocket, (struct sockaddr *)&client_Address,
    sizeof(client_Address)) < 0)
    {
        printf("\n Error : Connect Failed \n");
        exit(0);
    }
    sendto(UDPSocket, msgforserver, 255, 0, (struct sockaddr*)NULL,
    sizeof(client_Address));
    printf("Message to the server sent. \n");
    recvfrom(UDPSocket, msgReceived, sizeof(msgReceived), 0, (struct
    sockaddr*)NULL, NULL);
    printf("Received from the server: ");
    puts(msgReceived);
    close(UDPSocket);
}
```

Now, let's go behind the scenes.

How it works...

In the first part of this recipe, we have already defined two strings by the names of `msgReceived` and `msgforclient`, both of which are of size 255. We have also defined two variables, `server_Address` and `client_Address`, of type `sockaddr_in`.

Now, you will be prompted to enter a message that is to be sent to the server. The message you enter will be assigned to the `msgforserver` string. Then, we will invoke the `bzero` function to initialize the `client_Address` structure, that is, zeros will be filled in for all the members of the `client_Address` structure.

Next, we will initialize the members of the `client_Address` structure to configure the socket. Using the `sin_family` member, the address family that's specified for the socket is `AF_INET`, which is used for IPv4 internet protocols. The port number that's specified for the socket is `2000`. By using the `htons` function, the short integer, `2000`, is converted into the network byte order before being applied as a port number. Then, we will assign the localhost IP address, `127.0.0.1`, as the address to the socket. We will invoke the `inet_addr` function on the localhost address to convert the string containing the address in standard IPv4 dotted decimal notation into an integer value (suitable to be used as an internet address) before it is applied to the `sin_addr` member of the `client_Address` structure.

We will invoke the `socket` function to create a socket by the name of `UDPSocket`. The address family that's supplied for the socket is `AF_INET`, and the socket type that's selected is datagram.

Next, we will invoke the `sendto` function to send the message that's been assigned to the `msgforserver` string to the server. Similarly, we will invoke the `recvfrom` function to get the message from the server. The message that's received from the server is assigned to the `msgReceived` string, which is then displayed on the screen. Finally, the descriptor of the socket is closed.

Let's use GCC to compile the `udps.c` program, as follows:

```
$ gcc udps.c -o udps
```

If you get no errors or warnings, this means that the `udps.c` program has compiled into an executable file, `udps.exe`. Let's run this executable file:

```
$ ./udps
Waiting for the message from the client
```

Now, press `Alt + F2` to open a second Terminal window. Here, let's use GCC again to compile the `udpc.c` program, as follows:

```
$ gcc udpc.c -o udpc
```

If you get no errors or warnings, this means that the `udpc.c` program has compiled into an executable file, `udpc.exe`. Let's run this executable file:

```
$ ./udpc
Enter the message to send to the server: Will it rain today?
Message to the server sent.
```

The output on the server will give us the following output:

```
Message received from the client: Will it rain today?  
Enter the reply to be sent to the client: It might  
Reply to the client sent
```

Once the reply is sent from the server, on the client window, you will get the following output:

```
Received from the server: It might
```

To run the recipes that demonstrate IPC using shared memory and message queue, we need to run Cygserver. If you are running these programs on Linux, then you can skip this section. Let's see how Cygserver is run.

Running Cygserver

Before executing the command to run the Cygwin server, we need to configure Cygserver and install it as a service. To do so, you need to run the `cygserver.conf` script on the Terminal. The following is the output you get by running the script:

```
$ ./bin/cygserver-config  
Generating /etc/cygserver.conf file  
Warning: The following function requires administrator privileges!  
Do you want to install cygserver as service? yes  
  
The service has been installed under LocalSystem account.  
To start it, call `net start cygserver` or `cygrunsrv -S cygserver'.  
  
Further configuration options are available by editing the configuration  
file /etc/cygserver.conf. Please read the inline information in that  
file carefully. The best option for the start is to just leave it alone.  
  
Basic Cygserver configuration finished. Have fun!
```

Now, Cygserver will have been configured and installed as a service. The next step is to run the server. To run Cygserver, you need to use the following command:

```
$ net start cygserver  
The CYGWIN cygserver service is starting.  
The CYGWIN cygserver service was started successfully.
```

Now that Cygserver is running, we can make a recipe to demonstrate IPC using shared memory and message queues.

Passing a message from one process to another using the message queue

In this recipe, we will learn how communication between two processes is established using the message queue. This recipe is divided into the following parts:

- Writing a message into the message queue
- Reading a message from the message queue

Before we begin with these recipes, let's quickly review the functions, structures, and terms that are used in successful interprocess communication using shared memory and message queues.

Functions used in IPC using shared memory and message queues

The most commonly used functions and terms for IPC using shared memory and message queues are `ftok`, `shmget`, `shmat`, `shmdt`, `shmctl`, `msgget`, `msgrcv`, and `msgsnd`.

ftok()

This generates an IPC key on the basis of the supplied filename and ID. The filename can be provided along with its complete path. The filename must refer to an existing file. Here is the syntax:

```
key_t ftok(const char *filename, int id);
```

The `ftok` function will generate the same key value if the same filename (with same path) and the same ID is supplied. Upon successful completion, `ftok` will return a key, otherwise it will return `-1`.

shmget()

This allocates a shared memory segment and returns the shared memory identifier that's associated with the key. Here is its syntax:

```
int shmget(key_t key, size_t size, int shmflg);
```

Here, we need to address the following:

- **key:** This is (usually) the value that is returned by invoking the `ftok` function. You can also set the value of the key as `IPC_PRIVATE` if you don't want the shared memory to be accessed by other processes.
- **size:** Represents the size of the desired shared memory segment.
- **shmflg:** This can be any of the following constants:
 - `IPC_CREAT`: This creates a new segment if no shared memory identifier exists for the specified key. If this flag is not used, the function returns the shared memory segment associated with the key.
 - `IPC_EXCL`: This makes the `shmget` function fail if the segment already exists with the specified key.

On successful execution, the function returns the shared memory identifier in the form of a non-negative integer, otherwise it returns `-1`.

shmat()

This is used to attach a shared memory segment to the given address space. That is, the shared memory identifier that's received by invoking the `shmgt` function needs to be associated with the address space of a process. Here is its syntax:

```
void *shmat(int shidtfr, const void *addr, int flag);
```

Here, we need to address the following:

- **shidtfr:** Represents the memory identifier of the shared memory segment.
- **addr:** Represents the address space where the segment needs to be attached. If `shmaddr` is a null pointer, the segment is attached at the first available address or selected by the system.
- **flag:** This is attached as a read-only memory if the flag is `SHM_RDONLY`; otherwise, it is readable and writable.

If successfully executed, the function attaches the shared memory segment and returns the segment's start address, otherwise it returns `-1`.

shmdt()

This detaches the shared memory segment. Here is its syntax:

```
int shmdt(const void *addr);
```

Here, `addr` represents the address at which the shared memory segment is located.

shmctl()

This is used for performing certain control operations on the specified shared memory segment. Here is its syntax:

```
int shmctl(int shidtr, int cmd, struct shmid_ds *buf);
```

Here, we have to address the following:

- `shidtr`: Represents the identifier of the shared memory segment.
- `cmd`: This can have any of the following constants:
 - `IPC_STAT`: This copies the content of the `shmid_ds` data structure associated with the shared memory segment represented by `shidtr` into the structure that's pointed to by `buf`
 - `IPC_SET`: This writes the content of the structure that's pointed to by `buf` into the `shmid_ds` data structure, which is associated with the memory segment that's represented by `shidtr`
 - `IPC_RMID`: This removes the shared memory identifier that's specified by `shidtr` from the system and destroys the shared memory segment and `shmid_ds` data structure associated with it
- `buf`: This is a pointer to a `shmid_ds` structure.

If successfully executed, the function returns 0, otherwise it returns -1.

msgget()

This is used for creating a new message queue, and for accessing an existing queue that is related to the specified key. If this is executed successfully, the function returns the identifier of the message queue:

```
int msgget(key_t key, int flag);
```

Here, we have to address the following:

- **key:** This is a unique key value that is retrieved by invoking the `ftok` function.
- **flag:** This can be any of the following constants:
 - **IPC_CREAT:** Creates the message queue if it doesn't already exist and returns the message queue identifier for the newly created message queue. If the message queue already exists with the supplied key value, it returns its identifier.
 - **IPC_EXCL:** If both `IPC_CREAT` and `IPC_EXCL` are specified and the message queue does not exist, then it is created. However, if it already exists, then the function will fail.

msgrcv()

This is used for reading a message from a specified message queue whose identifier is supplied. Here is its syntax:

```
int msgrcv(int msqid, void *msgstruc, int msgsize, long typemsg, int flag);
```

Here, we have to address the following:

- **msqid:** Represents the message queue identifier of the queue from which the message needs to be read.
- **msgstruc:** This is the user-defined structure into which the read message is placed. The user-defined structure must contain two members. One is usually named `mttype`, which must be of type `long int` that specifies the type of the message, and the second is usually called `mesg`, which should be of `char` type to store the message.
- **msgsize:** Represents the size of text to be read from the message queue in terms of bytes. If the message that is read is larger than `msgsize`, then it will be truncated to `msgsize` bytes.
- **typemsg:** Specifies which message on the queue needs to be received:
 - If `typemsg` is 0, the first message on the queue is received
 - If `typemsg` is greater than 0, the first message whose `mttype` field is equal to `typemsg` is received
 - If `typemsg` is less than 0, a message whose `mttype` field is less than or equal to `typemsg` is received

- **flag:** Determines the action to be taken if the desired message is not found in the queue. It keeps its value of 0 if you don't want to specify the flag. The flag can have any of the following values:
 - **IPC_NOWAIT:** This makes the `msgrecv` function fail if there is no desired message in the queue, that is, it will not make the caller wait for the appropriate message on the queue. If flag is not set to `IPC_NOWAIT`, it will make the caller wait for an appropriate message on the queue instead of failing the function.
 - **MSG_NOERROR:** This allows you to receive text that is larger than the size that's specified in the `msgsize` argument. It simply truncates the text and receives it. If this flag is not set, on receiving the larger text, the function will not receive it and will fail the function.

If the function is executed successfully, the function returns the number of bytes that were actually placed into the text field of the structure that is pointed to by `msgstruc`. On failure, the function returns a value of -1.

msgsnd()

This is used for sending or delivering a message to the queue. Here is its syntax:

```
int msgsnd ( int msqid, struct msgbuf *msgstruc, int msgsize, int flag );
```

Here, we have to address the following:

- **msqid:** Represents the queue identifier of the message that we want to send. The queue identifier is usually retrieved by invoking the `msgget` function.
- **msgstruc:** This is a pointer to the user-defined structure. It is the `msg` member that contains the message that we want to send to the queue.
- **msgsize:** Represents the size of the message in bytes.
- **flag:** Determines the action to be taken on the message. If the flag value is set to `IPC_NOWAIT` and if the message queue is full, the message will not be written to the queue, and the control is returned to the calling process. But if flag is not set and the message queue is full, then the calling process will suspend until a space becomes available in the queue. Usually, the value of flag is set to 0.

If this is executed successfully, the function returns 0, otherwise it returns -1.

We will now begin with the first part of this recipe: writing a message into the queue.

Writing a message into the message queue

In this part of the recipe, we will learn how a server writes a desired message into the message queue.

How to do it...

1. Generate an IPC key by invoking the `flock` function. A filename and ID are supplied while creating the IPC key.
2. Invoke the `msgget` function to create a new message queue. The message queue is associated with the IPC key that was created in step 1.
3. Define a structure with two members, `mtype` and `mesg`. Set the value of the `mtype` member to 1.
4. Enter the message that's going to be added to the message queue. The string that's entered is assigned to the `mesg` member of the structure that we defined in step 3.
5. Invoke the `msgsnd` function to send the entered message into the message queue.

The `messageqsend.c` program for writing the message to the message queue is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MSGSIZE      255

struct msgstruc {
    long mtype;
    char mesg[MSGSIZE];
};

int main()
{
    int msqid, msglen;
    key_t key;
    struct msgstruc msgbuf;
    system("touch messagefile");
```

```
if ((key = ftok("messagefile", 'a')) == -1) {
    perror("ftok");
    exit(1);
}
if ((msqid = msgget(key, 0666 | IPC_CREAT)) == -1) {
    perror("msgget");
    exit(1);
}
msgbuf.mtype = 1;
printf("Enter a message to add to message queue : ");
scanf("%s", msgbuf.mesg);
msglen = strlen(msgbuf.mesg);
if (msgsnd(msqid, &msgbuf, msglen, IPC_NOWAIT) < 0)
    perror("msgsnd");
printf("The message sent is %s\n", msgbuf.mesg);
return 0;
}
```

Let's go behind the scenes.

How it works...

We will start by generating an IPC key by invoking the `ftok` function. The filename and ID are supplied while creating the IPC key are `messagefile` and `a`, respectively. The generated key is assigned to the `key` variable. Thereafter, we will invoke the `msgget` function to create a new message queue. The message queue is associated with the IPC key we just created using the `ftok` function.

Next, we will define a structure by the name of `msgstruc` with two members, `mtype` and `mesg`. The `mtype` member helps in determining the sequence number of the message that is going to be sent or received from the message queue. The `mesg` member contains the message that is going to be read or written into the message queue. We will define a variable called `msgbuf` of the `msgstruc` structure type. The value of the `mtype` member is set to 1.

You will be prompted to enter the message that is going to be added to the message queue. The string you enter is assigned to the `mesg` member of the `msgbuf` structure. The `msgsnd` function is invoked to send the message you entered into the message queue. Once the message is written into the message queue, a text message is displayed on the screen as confirmation.

Now, let's move on to the other part of this recipe.

Reading a message from the message queue

In this part of the recipe, we will learn how the message that was written into the message queue is read and displayed on the screen.

How to do it...

1. Invoke the `ftok` function to generate an IPC key. The filename and ID are supplied while creating the IPC key. These must be the same as what were applied while generating the key for writing the message in the message queue.
2. Invoke the `msgget` function to access the message queue that is associated with the IPC key. The message queue that's associated with this key already contains a message that we wrote through the previous program.
3. Define a structure with two members, `mtype` and `mesg`.
4. Invoke the `msgrcv` function to read the message from the associated message queue. The structure that was defined in Step 3 is passed to this function.
5. The read message is then displayed on the screen.

The `messageqrecv.c` program for reading a message from the message queue is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#define MSGSIZE      255

struct msgstruc {
    long mtype;
    char mesg[MSGSIZE];
};

int main()
{
    int msqid;
    key_t key;
    struct msgstruc rcvbuffer;

    if ((key = ftok("messagefile", 'a')) == -1) {
        perror("ftok");
        exit(1);
    }
```

```
if ((msqid = msgget(key, 0666)) < 0)
{
    perror("msgget");
    exit(1);
}
if (msgrecv(msqid, &rcvbuffer, MSGSIZE, 1, 0) < 0)
{
    perror("msgrecv");
    exit(1);
}
printf("The message received is %s\n", rcvbuffer.msg);
return 0;
}
```

Let's go behind the scenes.

How it works...

First, we will invoke the `ftok` function to generate an IPC key. The filename and ID that are supplied while creating the IPC key are `messagefile` and `a`, respectively. These filenames and ID must be the same as the ones that were applied while generating the key for writing the message in the message queue. The generated key is assigned to the `key` variable.

Thereafter, we will invoke the `msgget` function to access the message queue that is associated with the IPC key. The identifier of the accessed message queue is assigned to the `msqid` variable. The message queue that's associated with this key already contains the message that we wrote in the previous program.

Then, we will define a structure by the name `msgstruc` with two members, `mttype` and `msg`. The `mttype` member is for determining the sequence number of the message to be read from the message queue. The `msg` member will be used for storing the message that is read from the message queue. We will then define a variable called `rcvbuffer` of the `msgstruc` structure type. We will invoke the `msgrecv` function to read the message from the associated message queue.

The message identifier, `msqid`, is passed to the function, along with the `rcvbuffer` – the structure whose `msg` member will store the read message. After successful execution of the `msgrecv` function, the `msg` member of the `rcvbuffer` containing the message from the message queue will be displayed on screen.

Let's use GCC to compile the `messageqsend.c` program, as follows:

```
$ gcc messageqsend.c -o messageqsend
```

If you get no errors or warnings, this means that the `messageqsend.c` program has compiled into an executable file, `messageqsend.exe`. Let's run this executable file:

```
$ ./messageqsend
Enter a message to add to message queue : GoodBye
The message sent is GoodBye
```

Now, press *Alt + F2* to open a second Terminal screen. On this screen, you can compile and run the script for reading the message from the message queue.

Let's use GCC to compile the `messageqrecv.c` program, as follows:

```
$ gcc messageqrecv.c -o messageqrecv
```

If you get no errors or warnings, this means that the `messageqrecv.c` program has compiled into an executable file, `messageqrecv.exe`. Let's run this executable file:

```
$ ./messageqrecv
The message received is GoodBye
```

Voila! We've successfully passed a message from one process to another using the message queue. Let's move on to the next recipe!

Communicating between processes using shared memory

In this recipe, we will learn how communication between two processes is established using shared memory. This recipe is divided into the following parts:

- Writing a message into shared memory
- Reading a message from shared memory

We will start with the first one, that is, *Writing a message into shared memory*. The functions we learned in the previous recipe will also be applicable here.

Writing a message into shared memory

In this part of this recipe, we will learn how a message is written into shared memory.

How to do it...

1. Invoke the `ftok` function to generate an IPC key by supplying a filename and an ID.
2. Invoke the `shmget` function to allocate a shared memory segment that is associated with the key that was generated in step 1.
3. The size that's specified for the desired memory segment is 1024. Create a new memory segment with read and write permissions.
4. Attach the shared memory segment to the first available address in the system.
5. Enter a string that is then assigned to the shared memory segment.
6. The attached memory segment will be detached from the address space.

The `writememory.c` program for writing data into the shared memory is as follows:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *str;
    int shmid;

    key_t key = ftok("sharedmem", 'a');
    if ((shmid = shmget(key, 1024, 0666|IPC_CREAT)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    printf("Enter the string to be written in memory : ");
    gets(str);
    printf("String written in memory: %s\n", str);
    shmdt(str);
    return 0;
}
```

Let's go behind the scenes.

How it works...

By invoking the `ftok` function, we generate an IPC key with the filename `sharedmem` (you can change this) and an ID of `a`. The generated key is assigned to the `key` variable. Thereafter, invoke the `shmget` function to allocate a shared memory segment that is associated with the supplied key generated using the `ftok` function.

The size that's specified for the desired memory segment is `1024`. Create a new memory segment with read and write permissions and assign the shared memory identifier to the `shmid` variable. Then, attach the shared memory segment to the first available address in the system.

Once the memory segment is attached to the address space, the segment's start address is assigned to the `str` variable. You will be asked to enter a string. The string you enter will be assigned to the shared memory segment through the `str` variable. Finally, the attached memory segment is detached from the address space.

Let's move on to the next part of this recipe, *Reading a message from shared memory*.

Reading a message from shared memory

In this part of the recipe, we will learn how the message that was written into shared memory is read and displayed on screen.

How to do it...

1. Invoke the `ftok` function to generate an IPC key. The filename and ID that are supplied should be the same as those in the program for writing content into shared memory.
2. Invoke the `shmget` function to allocate a shared memory segment. The size that's specified for the allocated memory segment is `1024` and is associated with the IPC key that was generated in step 1. Create the memory segment with read and write permissions.
3. Attach the shared memory segment to the first available address in the system.
4. The content from the shared memory segment is read and displayed on screen.
5. The attached memory segment is detached from the address space.
6. The shared memory identifier is removed from the system, followed by destroying the shared memory segment.

The `readmemory.c` program for reading data from shared memory is as follows:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int shmid;
    char * str;
    key_t key = ftok("sharedmem", 'a');
    if ((shmid = shmget(key, 1024, 0666|IPC_CREAT)) < 0) {
        perror("shmget");
        exit(1);
    }
    if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }
    printf("Data read from memory: %s\n", str);
    shmdt(str);
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

Let's go behind the scenes.

How it works...

We will invoke the `ftok` function to generate an IPC key. The filename and ID that are supplied for generating the key are `sharedmem` (any name) and `a`, respectively. The generated key is assigned to the `key` variable. Thereafter, we will invoke the `shmget` function to allocate a shared memory segment. The size that's specified for the allocated memory segment is `1024` and is associated with the IPC key that was generated earlier.

We will create the new memory segment with read and write permissions and assign the fetched shared memory identifier to the `shmid` variable. The shared memory segment is then attached to the first available address in the system. This is done so that we can access the text that was written in the shared memory segment through the previous program.

So, after the memory segment is attached to the address space, the segment's start address is assigned to the `str` variable. Now, we can read the content that's been written in the shared memory through the previous program in the current program. The content from the shared memory segment is read through the `str` string and displayed on screen.

Thereafter, the attached memory segment is detached from the address space. Finally, the shared memory identifier `shmid` is removed from the system and the shared memory segment is destroyed.

Let's use GCC to compile the `writememory.c` program, as follows:

```
$ gcc writememory.c -o writememory
```

If you get no errors or warnings, this means that the `writememory.c` program has compiled into an executable file, `writememory.exe`. Let's run this executable file:

```
$ ./writememory
Enter the string to be written in memory : Today it might rain
String written in memory: Today it might rain
```

Now, press *Alt + F2* to open a second Terminal window. In this window, let's use GCC to compile the `readmemory.c` program, as follows:

```
$ gcc readmemory.c -o readmemory
```

If you get no errors or warnings, this means that the `readmemory.c` program has compiled into an executable file, `readmemory.exe`. Let's run this executable file:

```
$ ./readmemory
Data read from memory: Today it might rain
```

Voila! We've successfully communicated between processes using shared memory.

8

Using MySQL Database

MySQL is one of the most popular database management systems in recent times. Databases, as we all know, are used for storing data that's going to be used in the future when required. The data in a database can be secured through encryption and can be indexed for faster access. Where the volume of data is too high, a database management system is preferred over a traditional sequential and random file handling system. Storing data in a database is a very important task in any application.

This chapter is focused on understanding how table rows are managed in the database tables. In this chapter, you will learn about the following recipes:

- Displaying all the built-in tables in a default MySQL database
- Storing information into MySQL database
- Searching desired information in the database
- Updating information in the database
- Deleting data from the database using C

We will review the most commonly used functions in MySQL before we move on to the recipes. Also, ensure that you read *Appendix B* and *Appendix C* to install Cygwin and MySQL Server before implementing the recipes in this chapter.

Functions in MySQL

While accessing and working with MySQL database in C programming, we will have to use several functions. Let's go through them.

mysql_init()

This initializes a MySQL object that can be used in the `mysql_real_connect()` method. Here is its syntax:

```
MySQL *mysql_init (MySQL *object)
```

If the object parameter that's passed is `NULL`, then the function initializes and returns a new object; otherwise, the supplied object is initialized and the address of the object is returned.

mysql_real_connect()

This establishes a connection to a MySQL database engine running on the specified host. Here is its syntax:

```
MySQL *mysql_real_connect (MySQL *mysqlObject, const char *hostName, const  
char *userid, const char *password, const char *dbase, unsigned int port,  
const char *socket, unsigned long flag)
```

Here:

- `mysqlObject` represents the address of an existing MySQL object.
- `hostName` is where the hostname or IP address of the host is provided. To connect to a local host, either `NULL` or the string `localhost` is provided.
- `userid` represents a valid MySQL login ID.
- `password` represents the password of the user.
- `dbase` represents the database name to which the connection has to be established.
- `port` is where either value `0` is specified or the port number for the TCP/IP connection is supplied.
- `socket` is where either `NULL` is specified or the socket or named pipe is supplied.
- `flag` can be used to enable certain features, such as handling expired passwords and applying compression in the client/server protocol, but its value is usually kept at `0`.

The function returns a MySQL connection handler if the connection is established; otherwise, it returns `NULL`.

mysql_query()

This function executes the supplied SQL query. Here is its syntax:

```
int mysql_query(MYSQL *mysqlObject, const char *sqlstmt)
```

Here:

- `mysqlObject` represents the `MYSQL` object
- `sqlstmt` represents the null-terminated string that contains the SQL statement to be executed

The function returns `0` if the SQL statement executes successfully; otherwise, it returns a non-zero value.

mysql_use_result()

After successful execution of an SQL statement, this method is used to save the result set. This means that the result set is retrieved and returned. Here is its syntax:

```
MYSQL_RES *mysql_use_result(MYSQL *mysqlObject)
```

Here, `mysqlObject` represents the connection handler.

If no error occurs, the function returns a `MYSQL_RES` result structure. In case of any error, the function returns `NULL`.

mysql_fetch_row()

This function fetches the next row from a result set. The function returns `NULL` if there are no more rows in the result set to retrieve or if an error occurs. Here is its syntax:

```
MYSQL_ROW mysql_fetch_row(MYSQL_RES *resultset)
```

Here, the `resultset` parameter is the set from which the next row has to be fetched. You can access values in the column of the row by using the subscript `row[0]`, `row[1]`, and so on, where `row[0]` represents the data in the first column, `row[1]` represents the data in the second column, and so on.

mysql_num_fields()

This returns the number of values; that is, columns in the supplied row. Here is its syntax:

```
unsigned int mysql_num_fields(MYSQL_ROW row)
```

Here, the parameter `row` represents the individual row that is accessed from the resultset.

mysql_free_result()

This frees the memory allocated to a result set. Here is its syntax:

```
void mysql_free_result(MYSQL_RES *resultset)
```

Here, `resultset` represents the set whose memory we want to free up.

mysql_close()

This function closes the previously opened MySQL connection. Here is its syntax:

```
void mysql_close(MYSQL *mysqlObject)
```

It de-allocates the connection handler that's represented by the `mysqlObject` parameter. The function returns no value.

This covers the functions that we need to know for using the MySQL database for our recipes. From the second recipe onward, we will be working on a database table. So, let's get started and create a database and a table inside it.

Creating a MySQL database and tables

Open the Cygwin Terminal and open the MySQL command line by giving the following command. Through this command, we want to open MySQL through the user ID root and try to connect with the MySQL server running at the localhost (127.0.0.1):

```
$ mysql -u root -p -h 127.0.0.1
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 5.7.14-log MySQL Community Server (GPL)
```

```
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
MySQL [(none)]>
```

The preceding MySQL prompt that appears confirms that the `userid` and `password` have been entered correctly and that you are successfully connected to a running MySQL server. Now, we can go ahead and run SQL commands.

Create database

The `create database` statement creates the database with the specified name. Here is the syntax:

```
Create database database_name;
```

Here, `database_name` is the name of the new database to be created.

Let's create a database by the name `ecommerce` for our recipes:

```
MySQL [(none)]> create database ecommerce;
Query OK, 1 row affected (0.01 sec)
```

To confirm that our `ecommerce` database has been successfully created, we will use the `show databases` statement to see the list of existing databases on the MySQL server:

```
MySQL [(none)]> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| ecommerce      |
| mysql          |
| performance_schema |
| sakila         |
| sys            |
| world          |
+-----+
8 rows in set (0.00 sec)
```

In the preceding database listing, we can see the name `ecommerce`, which confirms that our database has been successfully created. Now, we will apply the `use` statement to access the `ecommerce` database, as shown here:

```
MySQL [(none)]> use ecommerce;
Database changed
```

Now, the `ecommerce` database is in use, so whatever SQL commands we will give will be applied to the `ecommerce` database only. Next, we need to create a table in our `ecommerce` database. For creating a database table, the `Create table` command is used. Let's discuss it next.

Create table

This creates a database table with the specified name. Here is its syntax:

```
CREATE TABLE table_name (column_name column_type, column_name
column_type,.....);
```

Here:

- `table_name` represents the name of the table that we want to create.
- `column_ name` represents the column names that we want in the table.
- `column_type` represents the data type of the column. Depending on the type of data we want to store in the column, the `column_type` can be `int`, `varchar`, `date`, `text`, and so on.

The `create table` statement creates a `users` table with three columns: `email_address`, `password`, and `address_of_delivery`. Assuming that this table will contain information of the users who have placed orders online, we will be storing their email address, password, and the location where the order has to be delivered:

```
MySQL [ecommerce]> create table users(email_address varchar(30), password
varchar(30), address_of_delivery text);
Query OK, 0 rows affected (0.38 sec)
```

To confirm that the table has been successfully created, we will use the `show tables` command to display the list of existing tables in the currently opened database, as shown here:

```
MySQL [ecommerce]> show tables;
+-----+
| Tables_in_ecommerce |
```

```
+-----+  
| users |  
+-----+  
1 row in set (0.00 sec)
```

The output of the `show tables` command displays the `users` table, thus confirming that the table has indeed been created successfully. To see the table structure (that is, its column names, column types, and column width), we will use the `describe` statement. The following statement displays the structure of the `users` table:

```
MySQL [ecommerce]> describe users;  
+-----+-----+-----+-----+-----+  
| Field | Type | Null | Key | Default | Extra |  
+-----+-----+-----+-----+-----+  
| email_address | varchar(30) | YES | | NULL | |  
| password | varchar(30) | YES | | NULL | |  
| address_of_delivery | text | YES | | NULL | |  
+-----+-----+-----+-----+-----+  
3 rows in set (0.04 sec)
```

So, now that we have learned about some basic commands to work with our database, we can begin with the first recipe of this chapter.

Displaying all the built-in tables in a default mysql database

The MySQL server, when installed, comes with certain default databases. One of those databases is `mysql`. In this recipe, we will learn to display all the table names that are available in the `mysql` database.

How to do it...

1. Create a MySQL object:

```
mysql_init(NULL);
```

2. Establish a connection to the MySQL server running at the specified host. Also, connect to the desired database:

```
mysql_real_connect(conn, server, user, password, database, 0, NULL, 0)
```

3. Create an execute SQL statement, comprised of show tables:

```
mysql_query(conn, "show tables")
```

4. Save the result of the executing SQL query (that is, the table information of the mysql database) into a resultset:

```
res = mysql_use_result(conn);
```

5. Fetch one row at a time from the resultset in a while loop and display only the table name from that row:

```
while ((row = mysql_fetch_row(res)) != NULL)
    printf("%s \n", row[0]);
```

6. Free up the memory that is allocated to the resultset:

```
mysql_free_result(res);
```

7. Close the opened connection handler:

```
mysql_close(conn);
```

The `mysql1.c` program for displaying all the tables in the built-in `mysql` database is as follows:

```
#include <mysql/mysql.h>
#include <stdio.h>
#include <stdlib.h>

void main() {
    MYSQL *conn;
    MYSQL_RES *res;
    MYSQL_ROW row;
    char *server = "127.0.0.1";
    char *user = "root";
    char *password = "Bintu2018$";
    char *database = "mysql";
    conn = mysql_init(NULL);
    if (!mysql_real_connect(conn, server,
        user, password, database, 0, NULL, 0)) {
        fprintf(stderr, "%s\n", mysql_error(conn));
        exit(1);
    }
    if (mysql_query(conn, "show tables")) {
        fprintf(stderr, "%s\n", mysql_error(conn));
        exit(1);
    }
```

```
res = mysql_use_result(conn);
printf("MySQL Tables in mysql database:\n");
while ((row = mysql_fetch_row(res)) != NULL)
    printf("%s\n", row[0]);
mysql_free_result(res);
mysql_close(conn);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will start by establishing a connection with the MySQL server and for that, we need to invoke the `mysql_real_connect` function. But we have to pass a `MYSQL` object to the `mysql_real_connect` function and we have to invoke the `mysql_init` function to create the `MYSQL` object. Hence, the `mysql_init` function is first invoked to initialize a `MYSQL` object by the name `conn`.

We will then supply the `MYSQL` object `conn` to the `mysql_real_connect` function, along with the valid user ID, password, and the host details. The `mysql_real_connect` function will establish a connection to the MySQL server running at the specified host. Besides this, the function will link to the supplied `mysql` database and will declare `conn` as the connection handler. This means that `conn` will be used in the rest of the program whenever we want to perform some action to the specified MySQL server and the `mysql` database.

If any error occurs in establishing the connection to the MySQL database engine, the program will terminate after displaying an error message. If the connection to the MySQL database engine is established successfully, the `mysql_query` function is invoked and the SQL statement `show tables` and the connection handler `conn` are supplied to it. The `mysql_query` function will execute the supplied SQL statement. To save the resulting table information of the `mysql` database, the `mysql_use_result` function is invoked. The table information that's received from the `mysql_use_result` function will be assigned to `resultset res`.

Next, we will invoke the `mysql_fetch_row` function in a `while` loop that will extract one row at a time from the `resultset res`; that is, one table detail will be fetched at a time from the `resultset` and assigned to the array `row`. The array `row` will contain complete information of one table at a time. The table name stored in the `row[0]` subscript is displayed on the screen. With every iteration of the `while` loop, the next piece of table information is extracted from `resultset res` and assigned to the array `row`. Consequently, all the table names in the `mysql` database will be displayed on the screen.

Then, we will invoke the `mysql_free_result` function to free up the memory that is allocated to resultset `res` and, finally, we will invoke the `mysql_close` function to close the opened connection handler `conn`.

Let's use GCC to compile the `mysql11.c` program, as shown here:

```
$ gcc mysql11.c -o mysql11 -I/usr/local/include/mysql -L/usr/local/lib/mysql  
-lmysqlclient
```

If you get no errors or warnings, that means the `mysql11.c` program has compiled into an executable file, `mysql11.exe`. Let's run this executable file:

```
$ ./mysql11  
MySQL Tables in mysql database:  
columns_priv  
db  
engine_cost  
event  
func  
general_log  
gtid_executed  
help_category  
help_keyword  
help_relation  
help_topic  
innodb_index_stats  
innodb_table_stats  
ndb_binlog_index  
plugin  
proc  
procs_priv  
proxies_priv  
server_cost  
servers  
slave_master_info  
slave_relay_log_info  
slave_worker_info  
slow_log  
tables_priv  
time_zone  
time_zone_leap_second  
time_zone_name  
time_zone_transition  
time_zone_transition_type  
user
```

Voila! As you can see, the output shows the list of built-in tables in the `mysql` database. Now, let's move on to the next recipe!

Storing information in MySQL database

In this recipe, we will learn how to insert a new row into the `users` table. Recall that at the beginning of this chapter, we created a database called `ecommerce`, and in that database, we created a table called `users` with the following columns:

```
email_address varchar(30)
password varchar(30)
address_of_delivery text
```

We will be inserting rows into this `users` table now.

How to do it...

1. Initialize a MySQL object:

```
conn = mysql_init(NULL);
```

2. Establish a connection to the MySQL server running at the localhost. Also, connect to the database that you want to work on:

```
mysql_real_connect(conn, server, user, password, database, 0, NULL, 0)
```

3. Enter the information of the new row that you want to insert into the `users` table in the `ecommerce` database, which will be for the new user's email address, password, and address of delivery:

```
printf("Enter email address: ");
scanf("%s", emailaddress);
printf("Enter password: ");
scanf("%s", upassword);
printf("Enter address of delivery: ");
getchar();
gets(deliveryaddress);
```

4. Prepare an SQL `INSERT` statement comprising this information; that is, the email address, password, and address of delivery of the new user:

```
strcpy(sqlquery, "INSERT INTO users(email_address, password,
address_of_delivery)VALUES ('");
strcat(sqlquery, emailaddress);
strcat(sqlquery, "','");
strcat(sqlquery, upassword);
strcat(sqlquery, "','");
strcat(sqlquery, deliveryaddress);
strcat(sqlquery, "')");
```

5. Execute the SQL `INSERT` statement to insert a new row into the `users` table in the `ecommerce` database:

```
mysql_query(conn, sqlquery)
```

6. Close the connection handler:

```
mysql_close(conn);
```

The `adduser.c` program for inserting a row into a MySQL database table is shown in the following code:

```
#include <mysql/mysql.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main() {
    MYSQL *conn;
    char *server = "127.0.0.1";
    char *user = "root";
    char *password = "Bintu2018$";
    char *database = "ecommerce";
    char emailaddress[30],
    upassword[30],deliveryaddress[255],sqlquery[255];
    conn = mysql_init(NULL);
    if (!mysql_real_connect(conn, server, user, password, database, 0,
    NULL, 0)) {
        fprintf(stderr, "%s\n", mysql_error(conn));
        exit(1);
    }
    printf("Enter email address: ");
    scanf("%s", emailaddress);
    printf("Enter password: ");
    scanf("%s", upassword);
    printf("Enter address of delivery: ");
```

```
getchar();
gets(deliveryaddress);
strcpy(sqlquery,"INSERT INTO users(email_address, password,
address_of_delivery)VALUES ('");
strcat(sqlquery,emailaddress);
strcat(sqlquery,"', '");
strcat(sqlquery,upassword);
strcat(sqlquery,"', '");
strcat(sqlquery,deliveryaddress);
strcat(sqlquery,"')");
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "Row could not be inserted into users
table\n");
    exit(1);
}
printf("Row is inserted successfully in users table\n");
mysql_close(conn);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We start by invoking the `mysql_init` function to initialize a `MYSQL` object by the name `conn`. The initialized `MYSQL` object `conn` is then supplied for invoking the `mysql_real_connect` function, along with the valid user ID and password, which in turn will establish a connection to the MySQL server running on the localhost. In addition, the function will link to our `ecommerce` database.

If any error occurs in establishing the connection to the MySQL database engine, an error message will be displayed and the program will terminate. If the connection to the MySQL database engine is established successfully, then `conn` will act as a connection handler for the rest of the program.

You will be prompted to enter information for the new row that you want to insert into the `users` table in the `ecommerce` database. You will be prompted to enter the information for the new row: the email address, password, and address of delivery. We will create an SQL `INSERT` statement comprising this information (email address, password, and address of delivery), which is supposed to be entered by users. Thereafter, we will invoke the `mysql_query` function and pass the MySQL object `conn` and the SQL `INSERT` statements to it to execute the SQL statement and insert a new row into the `users` table.

If any error occurs while executing the `mysql_query` function, an error message will be displayed on the screen and the program will terminate. If the new row is successfully inserted into the `users` table, the message Row is inserted successfully in `users` table will be displayed on the screen. Finally, we will invoke the `mysql_close` function and pass the connection handler `conn` to it to close the connection handler.

Let's open the Cygwin Terminal. We will require two Terminal windows; on one window, we will run SQL commands and on the other, we will compile and run C. Open another Terminal window by pressing *Alt+F2*. In the first Terminal window, invoke the MySQL command line by using the following command:

```
$ mysql -u root -p -h 127.0.0.1
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 27
Server version: 5.7.14-log MySQL Community Server (GPL)
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

To work with our `ecommerce` database, we need to make it the current database. So, open the `ecommerce` database by using the following command:

```
MySQL [(none)]> use ecommerce;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
```

Now, `ecommerce` is our current database; that is, whatever SQL commands we will execute will be applied to the `ecommerce` database only. Let's use the following SQL `SELECT` command to see the existing rows in the `users` database table:

```
MySQL [ecommerce]> select * from users;
Empty set (0.00 sec)
```

The given output confirms that the `users` table is currently empty. To compile the C program, switch to the second Terminal window. Let's use GCC to compile the `adduser.c` program, as shown here:

```
$ gcc adduser.c -o adduser -I/usr/local/include/mysql -
L/usr/local/lib/mysql -lmysqlclient
```

If you get no errors or warnings, that means the `adduser.c` program has compiled into an executable file, `adduser.exe`. Let's run this executable file:

```
$ ./adduser
Enter email address: bmharwani@yahoo.com
Enter password: gold
Enter address of delivery: 11 Hill View Street, New York, USA
Row is inserted successfully in users table
```

The given C program output confirms that the new row has been successfully added to the `users` database table. To confirm this, switch to the Terminal window where the MySQL command line is open and use the following command:

```
MySQL [ecommerce]> select * from users;
+-----+-----+-----+
| email_address | password | address_of_delivery |
+-----+-----+-----+
| bmharwani@yahoo.com | gold | 11 Hill View Street, New York, USA |
+-----+-----+
1 row in set (0.00 sec)
```

Voila! The given output confirms that the new row that was entered through C has been successfully inserted into the `users` database table.

Now, let's move on to the next recipe!

Searching for the desired information in the database

In this recipe, we will learn how to search for information in a database table. Again, we assume that a `users` table comprising three columns, `email_address`, `password`, and `address_of_delivery`, already exists (please see the section, *Creating a MySQL database and tables*, of this chapter, where we created an `ecommerce` database and a `users` table in it). On entering an email address, the recipe will search the entire `users` database table for it, and if any row is found that matches the supplied email address, that user's password and address of delivery will be displayed on the screen.

How to do it...

1. Initialize a MYSQL object:

```
mysql_init(NULL);
```

2. Establish a connection to the MySQL server running at the specified host. Also, establish a connection to the ecommerce database:

```
mysql_real_connect(conn, server, user, password, database, 0, NULL, 0)
```

3. Enter the email address of the user whose details you want to search for:

```
printf("Enter email address to search: ");
scanf("%s", emailaddress);
```

4. Create an SQL SELECT statement that searches the row in the users table that matches the email address that was entered by the user:

```
strcpy(sqlquery, "SELECT * FROM users where email_address like \'");
strcat(sqlquery, emailaddress);
strcat(sqlquery, "\'");
```

5. Execute the SQL SELECT statement. Terminate the program if the SQL query does not execute or some error occurs:

```
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "No row found in the users table with this email
address\n");
    exit(1);
}
```

6. If the SQL query executes successfully then the row(s) that matches the specified email address are retrieved and assigned to a resultset:

```
resultset = mysql_use_result(conn);
```

7. Use a while loop to extract one row at a time from the resultset and assign it to the array row:

```
while ((row = mysql_fetch_row(resultset)) != NULL)
```

8. The information of the entire row is shown by displaying the subscripts `row[0]`, `row[1]`, and `row[2]`, respectively:

```
printf("Email Address: %s \n", row[0]);
printf("Password: %s \n", row[1]);
printf("Address of delivery: %s \n", row[2]);
```

9. Memory that's allocated to the `resultset` is freed up:

```
mysql_free_result(resultset);
```

10. The opened connection handler is closed:

```
mysql_close(conn);
```

The `searchuser.c` program for searching in a specific row in a MySQL database table is shown in the following code:

```
#include <mysql/mysql.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main() {
    MYSQL *conn;
    MYSQL_RES *resultset;
    MYSQL_ROW row;
    char *server = "127.0.0.1";
    char *user = "root";
    char *password = "Bintu2018$";
    char *database = "ecommerce";
    char emailaddress[30], sqlquery[255];
    conn = mysql_init(NULL);
    if (!mysql_real_connect(conn, server, user, password, database, 0,
                           NULL, 0)) {
        fprintf(stderr, "%s\n", mysql_error(conn));
        exit(1);
    }
    printf("Enter email address to search: ");
    scanf("%s", emailaddress);
    strcpy(sqlquery, "SELECT * FROM users where email_address like \'");
    strcat(sqlquery, emailaddress);
    strcat(sqlquery, "\'");
    if (mysql_query(conn, sqlquery) != 0)
    {
        fprintf(stderr, "No row found in the users table with this
email address\n");
        exit(1);
    }
}
```

```
        }
        printf("The details of the user with this email address are as
follows:\n");
        resultset = mysql_use_result(conn);
        while ((row = mysql_fetch_row(resultset)) != NULL)
        {
            printf("Email Address: %s \n", row[0]);
            printf("Password: %s \n", row[1]);
            printf("Address of delivery: %s \n", row[2]);
        }
        mysql_free_result(resultset);
        mysql_close(conn);
    }
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will start by invoking the `mysql_init` function to initialize a `MYSQL` object by the name `conn`. Thereafter, we will invoke the `mysql_real_connect` function and pass the `MYSQL` object `conn` to it along with the valid user ID, password, and the host details. The `mysql_real_connect` function will establish a connection to the MySQL server running at the specified host and will also connect to the supplied database, `ecommerce`. The `MYSQL` object `conn` will act as the connection handler for the rest of the program. Wherever a connection to the MySQL server and `ecommerce` database is required, referring to `conn` will suffice.

If any error occurs in establishing a connection to the MySQL database engine or the `ecommerce` database, an error message will be displayed and the program will terminate. If a connection to the MySQL database engine is established successfully, you will be prompted to enter the email address of the user whose details you want to search for.

We will create an SQL `SELECT` statement that will search the row in the `users` table that matches the email address entered by the user. Then, we will invoke the `mysql_query` function and pass the created SQL `SELECT` statement to it, along with the connection handler `conn`. If the SQL query does not execute or some error occurs, the program will terminate after displaying an error message. If the query is successful, then the resulting row(s) that satisfy the condition (that is, the row(s) that match the supplied email address) will be retrieved by invoking the `mysql_use_result` function and will be assigned to the result set, `resultset`.

We will then invoke the `mysql_fetch_row` function in a `while` loop that will extract one row at a time from the `resultset`; that is, the first row from the `resultset` will be accessed and assigned to the array `row`.

Recall that the `users` table contains the following columns:

- `email_address` `varchar(30)`
- `password` `varchar(30)`
- `address_of_delivery` `text`

Consequently, the array `row` will contain complete information of the accessed row, where the subscript `row[0]` will contain the data of the `email_address` column, `row[1]` will contain the data of the column `password`, and `row[2]` will contain the data of the `address_of_delivery` column. The information of the entire row will be displayed by displaying the subscripts `row[0]`, `row[1]`, and `row[2]`, respectively.

At the end, we will invoke the `mysql_free_result` function to free up the memory that was allocated to the `resultset`. Then, we will invoke the `mysql_close` function to close the opened connection handler `conn`.

Let's open the Cygwin Terminal. We will require two Terminal windows; on one window, we will run SQL commands and on the other, we will compile and run C. Open another Terminal window by pressing *Alt+F2*. In the first Terminal window, invoke the MySQL command line by using the following command:

```
$ mysql -u root -p -h 127.0.0.1
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 27
Server version: 5.7.14-log MySQL Community Server (GPL)
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

To work with our `ecommerce` database, we need to make it the current database. So, open the `ecommerce` database by using the following command:

```
MySQL [(none)]> use ecommerce;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
```

Now, `ecommerce` is our current database; that is, whatever SQL commands we will execute will be applied to the `ecommerce` database only. Let's use the following SQL `SELECT` command to see the existing rows in the `users` database table:

```
MySQL [ecommerce]> select * from users;
+-----+-----+-----+
| email_address | password | address_of_delivery |
+-----+-----+-----+
| bmharwani@yahoo.com | gold | 11 Hill View Street, New York, USA
| harwanibm@gmail.com | diamond | House No. xyz, Pqr Apartments, Uvw Lane,
Mumbai, Maharashtra |
| bintu@gmail.com | platinum | abc Sea View, Ocean Lane, Opposite Mt.
Everest, London, UKg
+-----+-----+
3 rows in set (0.00 sec)
```

The given output shows that there are three rows in the `users` table.

To compile the C program, switch to the second Terminal window. Let's use GCC to compile the `searchuser.c` program, as shown here:

```
$ gcc searchuser.c -o searchuser -I/usr/local/include/mysql -
L/usr/local/lib/mysql -lmysqlclient
```

If you get no errors or warnings, that means the `searchuser.c` program has compiled into an executable file, `searchuser.exe`. Let's run this executable file:

```
$ ./searchuser
Enter email address to search: bmharwani@yahoo.com
The details of the user with this email address are as follows:
Email Address:bmharwani@yahoo.com
Password: gold
Address of delivery: 11 Hill View Street, New York, USA
```

Voila! We can see that complete information of the user with their email address, `bmharwani@yahoo.com`, is displayed on the screen.

Now, let's move on to the next recipe!

Updating information in the database

In this recipe, we will learn how to update information in a database table. We assume that a `users` database table already exists, comprising of three columns—`email_address`, `password`, and `address_of_delivery` (please see the beginning of this chapter, where we learned to create a database and a table in it). On entering an email address, all the current information of the user (that is, their password and address of delivery) will be displayed. Thereafter, the user will be prompted to enter a new password and address of delivery. This new information will be updated against the current information in the table.

How to do it...

1. Initialize a MySQL object:

```
mysql_init (NULL);
```

2. Establish a connection to the MySQL server running at the specified host. Also, generate a connection handler. The program will terminate if some error occurs in establishing the connection to the MySQL server engine or to the `ecommerce` database:

```
if (!mysql_real_connect(conn, server, user, password, database, 0,
    NULL, 0))
{
    fprintf(stderr, "%s\n", mysql_error(conn));
    exit(1);
}
```

3. Enter the email address of the user whose information has to be updated:

```
printf("Enter email address of the user to update: ");
scanf("%s", emailaddress);
```

4. Create an SQL SELECT statement that will search the row in the `users` table that matches the email address that was entered by the user:

```
strcpy(sqlquery,"SELECT * FROM users where email_address like
\\'%');
strcat(sqlquery,emailaddress);
strcat(sqlquery,"\\%');
```

5. Execute the SQL SELECT statement. The program will terminate if the SQL query does not execute successfully or some other error occurs:

```
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "No row found in the users table with this
email address\n");
    exit(1);
}
```

6. If the SQL query executes successfully, then the row(s) that match the supplied email address will be retrieved and assigned to the resultset:

```
resultset = mysql_store_result(conn);
```

7. Check if there is at least one row in the resultset:

```
if(mysql_num_rows(resultset) >0)
```

8. If there is no row in the resultset, then display the message that no row was found in the users table with the specified email address and exit from the program:

```
printf("No user found with this email address\n");
```

9. If there is any row in the resultset, then access it and assign it to the array row:

```
row = mysql_fetch_row(resultset)
```

10. Information about the user (that is, the email address, password, and address of delivery, which are assigned to the subscripts `row[0]`, `row[1]`, and `row[2]`, respectively) are displayed on the screen:

```
printf("Email Address: %s \n", row[0]);
printf("Password: %s \n", row[1]);
printf("Address of delivery: %s \n", row[2]);
```

11. The memory allocated to the resultset is freed:

```
mysql_free_result(resultset);
```

12. Enter the new updated information of the user; that is, the new password and the new address of delivery:

```
printf("Enter new password: ");
scanf("%s", upassword);
printf("Enter new address of delivery: ");
```

```
getchar();
gets(deliveryaddress);
```

13. An SQL UPDATE statement is prepared that contains the information of the newly entered password and address of delivery:

```
strcpy(sqlquery,"UPDATE users set password='");
strcat(sqlquery,upassword);
strcat(sqlquery,"\', address_of_delivery='");
strcat(sqlquery,deliveryaddress);
strcat(sqlquery,"\' where email_address like \'");
strcat(sqlquery,emailaddress);
strcat(sqlquery,"'');
```

14. Execute the SQL UPDATE statement. If any error occurs in executing the SQL UPDATE query, the program will terminate:

```
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "The desired row in users table could not be
updated\n");
    exit(1);
}
```

15. If the SQL UPDATE statement executes successfully, display a message on the screen informing that the user's information has been updated successfully:

```
printf("The information of user is updated successfully in users
table\n");
```

16. Close the opened connection handler:

```
mysql_close(conn);
```

The updateuser.c program for updating a specific row of a MySQL database table with new content is shown in the following code:

```
#include <mysql/mysql.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main() {
    MYSQL *conn;
    MYSQL_RES *resultset;
    MYSQL_ROW row;
    char *server = "127.0.0.1";
    char *user = "root";
```

```
char *password = "Bintu2018$";
char *database = "ecommerce";
char emailaddress[30], sqlquery[255],
upassword[30], deliveryaddress[255];
conn = mysql_init(NULL);
if (!mysql_real_connect(conn, server, user, password, database, 0,
NULL, 0)) {
    fprintf(stderr, "%s\n", mysql_error(conn));
    exit(1);
}
printf("Enter email address of the user to update: ");
scanf("%s", emailaddress);
strcpy(sqlquery,"SELECT * FROM users where email_address like \'");
strcat(sqlquery,emailaddress);
strcat(sqlquery,"\'");
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "No row found in the users table with this
email address\n");
    exit(1);
}
resultset = mysql_store_result(conn);
if(mysql_num_rows(resultset) >0)
{
    printf("The details of the user with this email address are as
follows:\n");
    while ((row = mysql_fetch_row(resultset)) != NULL)
    {
        printf("Email Address: %s \n", row[0]);
        printf("Password: %s \n", row[1]);
        printf("Address of delivery: %s \n", row[2]);
    }
    mysql_free_result(resultset);
    printf("Enter new password: ");
    scanf("%s", upassword);
    printf("Enter new address of delivery: ");
    getchar();
    gets(deliveryaddress);
    strcpy(sqlquery,"UPDATE users set password=\'");
    strcat(sqlquery,upassword);
    strcat(sqlquery,"\', address_of_delivery=\'");
    strcat(sqlquery,deliveryaddress);
    strcat(sqlquery,"\' where email_address like \'");
    strcat(sqlquery,emailaddress);
    strcat(sqlquery,"\'");
    if (mysql_query(conn, sqlquery) != 0)
    {
        fprintf(stderr, "The desired row in users table could not
```

```
        be updated\n");
        exit(1);
    }
    printf("The information of user is updated successfully in
users table\n");
}
else
    printf("No user found with this email address\n");
mysql_close(conn);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

In this program, we first ask the user to enter the email address they want to update. Then, we search the `users` table to see if there is any row with the matching email address. If we find it, we display the current information of the user; that is, the current email address, password, and address of delivery. Thereafter, we ask the user to enter a new password and new address of delivery. The new password and address of delivery will replace the old password and address of delivery, thereby updating the `users` table.

We will start by invoking the `mysql_init` function to initialize a `MYSQL` object by the name `conn`. Then, we will pass the `MYSQL` object `conn` to the `mysql_real_connect` function that we invoked to establish a connection to the MySQL server running at the specified host. Several other parameters will also be passed to the `mysql_real_connect` function, including a valid user ID, password, host details, and the database with which we want to work. The `mysql_real_connect` function will establish the connection to the MySQL server running at the specified host and will declare the `MYSQL` object `conn` as the connection handler. This means that `conn` can connect to the MySQL server and the `ecommerce` database wherever it is used.

The program will terminate after displaying an error message if some error occurs while establishing the connection to the MySQL server engine or to the `ecommerce` database. If the connection to the MySQL database engine is established successfully, you will be prompted to enter the email address of the user whose record you want to update.

As we mentioned earlier, we will first display the current information of the user. So, we will create an SQL `SELECT` statement and we will search the row in the `users` table that matches the email address that's entered by the user. Then, we will invoke the `mysql_query` function and pass the created SQL `SELECT` statement to it, along with the connection handler `conn`.

Again, the program will terminate after displaying an error message if the SQL query does not execute successfully or some other error occurs. If the query executes successfully, then the resulting row(s) (that is, the row(s) that match the supplied email address), will be retrieved by invoking the `mysql_use_result` function and will be assigned to the `resultset`.

We will then invoke the `mysql_num_rows` function to ensure that there is at least one row in the `resultset`. If there is no row in the `resultset`, this means that no row was found in the `users` table that matches the given email address. In this case, the program will terminate after informing that no row was found in the `users` table with the given email address. If there is even a single row in the `resultset`, we will invoke the `mysql_fetch_row` function on the `resultset`, which will extract one row from the `resultset` and assign it to the array `row`.

The `users` table contains the following three columns:

- `email_address` `varchar(30)`
- `password` `varchar(30)`
- `address_of_delivery` `text`

The array `row` will contain the information of the accessed row, where the subscripts `row[0]`, `row[1]`, and `row[2]` will contain the data of the columns `email_address`, `password`, and `address_of_delivery`, respectively. The current information of the user is displayed by displaying the information assigned to the aforementioned subscripts. Then, we will invoke the `mysql_free_result` function to free up the memory that is allocated to the `resultset`.

At this stage, the user will be asked to enter the new password and the new address of delivery. We will prepare an SQL UPDATE statement that contains the information of the newly entered password and address of delivery. The `mysql_query` function will be invoked and the SQL UPDATE statement will be passed to it, along with the connection handler `conn`.

If any error occurs in executing the SQL UPDATE query, again, an error message will be displayed and the program will terminate. If the SQL UPDATE statement executes successfully, a message informing that the user's information has been updated successfully will be displayed. Finally, we will invoke the `mysql_close` function to close the opened connection handler `conn`.

Let's open the Cygwin Terminal. We will require two Terminal windows; on one window, we will run SQL commands and on the other, we will compile and run C. Open another Terminal window by pressing *Alt+F2*. In the first Terminal window, invoke the MySQL command line by using the following command:

```
$ mysql -u root -p -h 127.0.0.1
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 27
Server version: 5.7.14-log MySQL Community Server (GPL)
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

To work with our ecommerce database, we need to make it the current database. So, open the ecommerce database by using the following command:

```
MySQL [(none)]> use ecommerce;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
```

Now, ecommerce is our current database; that is, whatever SQL commands we will execute will be applied to the ecommerce database only. Let's use the following SQL SELECT command to see the existing rows in the users database table:

```
MySQL [ecommerce]> select * from users;
+-----+-----+
| email_address | password | address_of_delivery |
+-----+-----+
| bmharwani@yahoo.com | gold | 11 Hill View Street, New York, USA |
| harwanimb@gmail.com | diamond | House No. xyz, Pqr Apartments, Uvw Lane, Mumbai, Maharashtra |
| bintu@gmail.com | platinum | abc Sea View, Ocean Lane, Opposite Mt. Everest, London, UKg |
+-----+-----+
3 rows in set (0.00 sec)
```

We can see from the preceding output that there are three rows in the users table. To compile the C program, switch to the second Terminal window. Let's use GCC to compile the updateuser.c program, as shown here:

```
$ gcc updateuser.c -o updateuser -I/usr/local/include/mysql -
L/usr/local/lib/mysql -lmysqlclient
```

If you get no errors or warnings, that means the `updateuser.c` program has compiled into an executable file, `updateuser.exe`. Let's run this executable file:

```
$ ./updateuser
Enter email address of the user to update: harwanibintu@gmail.com
No user found with this email address
```

Let's run the program again and enter an email address that already exists:

```
$ ./updateuser
Enter email address of the user to update: bmharwani@yahoo.com
The details of the user with this email address are as follows:
Email Address: bmharwani@yahoo.com
Password: gold
Address of delivery: 11 Hill View Street, New York, USA
Enter new password: coffee
Enter new address of delivery: 444, Sky Valley, Toronto, Canada
The information of user is updated successfully in users table
```

So, we have updated the row of the user with the email address, `bmharwani@yahoo.com`. To confirm that the row has been updated in the `users` database table too, switch to the Terminal window where the MySQL command line is running and issue the following SQL SELECT command:

```
MySQL [ecommerce]> MySQL [ecommerce]> select * from users;
+-----+-----+-----+
| email_address | password | address_of_delivery |
+-----+-----+-----+
| bmharwani@yahoo.com | coffee | 444, Sky Valley, Toronto, Canada
|
| harwanibm@gmail.com | diamond | House No. xyz, Pqr Apartments, Uvw Lane, Mumbai, Maharashtra
|
| bintu@gmail.com | platinum | abc Sea View, Ocean Lane, Opposite Mt. Everest, London, UKg
+-----+-----+-----+
```

Voila! We can see that the row of the `users` table with the email address `bmharwani@yahoo.com` has been updated and is showing the new information.

Now, let's move on to the next recipe!

Deleting data from the database using C

In this recipe, we will learn how to delete information from a database table. We assume that a `users` table comprising three columns, `email_address`, `password`, and `address_of_delivery`, already exists (please see the beginning of this chapter, where we created an `ecommerce` database and a `users` table in it). You will be prompted to enter the email address of the user whose row has to be deleted. On entering an email address, all the information of the user will be displayed. Thereafter, you will again be asked to confirm if the displayed row should be deleted or not. After your confirmation, the row will be permanently deleted from the table.

How to do it...

1. Initialize a MySQL object:

```
mysql_init(NULL);
```

2. Establish a connection to the MySQL server running at the specified host. Also, generate a connection handler. If any error occurs in establishing a connection to the MySQL server engine, the program will terminate:

```
if (!mysql_real_connect(conn, server, user, password, database,
0,
NULL, 0)) {
    fprintf(stderr, "%s\n", mysql_error(conn));
    exit(1);
}
```

3. If the connection to the MySQL database engine is established successfully, you will be prompted to enter the email address of the user whose record you want to delete:

```
printf("Enter email address of the user to delete: ");
scanf("%s", emailaddress);
```

4. Create an SQL SELECT statement that will search the row from the `users` table that matches the email address that's entered by the user:

```
strcpy(sqlquery, "SELECT * FROM users where email_address like
\\''");
strcat(sqlquery, emailaddress);
strcat(sqlquery, "\\'");
```

5. Execute the SQL SELECT statement. The program will terminate after displaying an error message if the SQL query does not execute successfully:

```
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "No row found in the users table with this
email
address\n");
    exit(1);
}
```

6. If the query executes successfully, then the resulting row(s) that match the supplied email address will be retrieved and assigned to the `resultset`:

```
resultset = mysql_store_result(conn);
```

7. Invoke the `mysql_num_rows` function to ensure that there is at least one row in the `resultset`:

```
if(mysql_num_rows(resultset) >0)
```

8. If there is no row in the `resultset`, that means no row was found in the `users` table that matches the given email address; hence, the program will terminate:

```
printf("No user found with this email address\n");
```

9. If there is any row in the result set, that row is extracted from the `resultset` and will be assigned to the array `row`:

```
row = mysql_fetch_row(resultset)
```

10. The information of the user is displayed by displaying the corresponding subscripts in the array `row`:

```
printf("Email Address: %s \n", row[0]);
printf("Password: %s \n", row[1]);
printf("Address of delivery: %s \n", row[2]);
```

11. The memory that's allocated to the `resultset` is freed up:

```
mysql_free_result(resultset);The user is asked whether he/she
really want to delete the shown record.
printf("Are you sure you want to delete this record yes/no: ");
scanf("%s", k);
```

12. If the user enters yes, an SQL DELETE statement will be created that will delete the row from the users table that matches the specified email address:

```
if (strcmp(k, "yes") == 0)
{
    strcpy(sqlquery, "Delete from users where email_address like
\''");
    strcat(sqlquery, emailaddress);
    strcat(sqlquery, "\'");
}
```

13. The SQL DELETE statement is executed. If there are any error occurs in executing the SQL DELETE query, the program will terminate:

```
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "The user account could not be deleted\n");
    exit(1);
}
```

14. If the SQL DELETE statement is executed successfully, a message informing that the user account with the specified email address is deleted successfully is displayed:

```
printf("The user with the given email address is successfully
deleted from the users table\n");
```

15. The opened connection handler is closed:

```
mysql_close(conn);
```

The `deleteuser.c` program for deleting a specific row from a MySQL database table is shown in the following code:

```
#include <mysql/mysql.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main() {
    MYSQL *conn;
    MYSQL_RES *resultset;
    MYSQL_ROW row;
    char *server = "127.0.0.1";
    char *user = "root";
    char *password = "Bintu2018$";
    char *database = "ecommerce";
    char emailaddress[30], sqlquery[255], k[10];
```

```
conn = mysql_init(NULL);
if (!mysql_real_connect(conn, server, user, password, database, 0, NULL,
0)) {
    fprintf(stderr, "%s\n", mysql_error(conn));
    exit(1);
}
printf("Enter email address of the user to delete: ");
scanf("%s", emailaddress);
strcpy(sqlquery,"SELECT * FROM users where email_address like \'");
strcat(sqlquery,emailaddress);
strcat(sqlquery,"\'");
if (mysql_query(conn, sqlquery) != 0)
{
    fprintf(stderr, "No row found in the users table with this email
address\n");
    exit(1);
}
resultset = mysql_store_result(conn);
if(mysql_num_rows(resultset) >0)
{
    printf("The details of the user with this email address are as
follows:\n");
    while ((row = mysql_fetch_row(resultset)) != NULL)
    {
        printf("Email Address: %s \n", row[0]);
        printf("Password: %s \n", row[1]);
        printf("Address of delivery: %s \n", row[2]);
    }
    mysql_free_result(resultset);
    printf("Are you sure you want to delete this record yes/no: ");
    scanf("%s", k);
    if(strcmp(k,"yes")==0)
    {
        strcpy(sqlquery, "Delete from users where email_address like
\'");
        strcat(sqlquery,emailaddress);
        strcat(sqlquery,"\'");
        if (mysql_query(conn, sqlquery) != 0)
        {
            fprintf(stderr, "The user account could not be deleted\n");
            exit(1);
        }
        printf("The user with the given email address is successfully
deleted from the users table\n");
    }
}
else
    printf("No user found with this email address\n");
```

```
    mysql_close(conn);  
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will start by invoking the `mysql_init` function to initialize a `MYSQL` object by the name `conn`. We will then pass the `MYSQL` object `conn` to the `mysql_real_connect` function that we invoked to establish a connection to the MySQL server running at the specified host. Several other parameters will also be passed to the `mysql_real_connect` function, including a valid user ID, password, host details, and the database with which we want to work. The `mysql_real_connect` function will establish a connection to the MySQL server running at the specified host and will declare a `MYSQL` object `conn` as the connection handler. This means that `conn` can connect to the MySQL server and the commerce database wherever it is used.

The program will terminate after displaying an error message if some error occurs while establishing a connection to the MySQL server engine or to the `ecommerce` database. If the connection to the MySQL database engine is established successfully, you will be prompted to enter the email address of the user whose record you want to delete.

We will first display the information of the user and thereafter will seek permission from the user as to whether they really want to delete that row or not. So, we will create an SQL `SELECT` statement that will search the row from the `users` table that matches the email address that was entered by the user. Then, we will invoke the `mysql_query` function and pass the created SQL `SELECT` statement to it, along with the connection handler `conn`.

Again, the program will terminate after displaying an error message if the SQL query does not execute successfully or some other error occurs. If the query executes successfully, then the resulting row(s) (that is, the row(s) that match the supplied email address) will be retrieved by invoking the `mysql_use_result` function and will be assigned to the `resultset`.

We will invoke the `mysql_num_rows` function to ensure that there is at least one row in the `resultset`. If there is no row in the `resultset`, that means no row was found in the `users` table that matches the given email address. In that case, the program will terminate after informing that no row was found in the `users` table with the given email address. If there is even a single row in the `resultset`, we will invoke the `mysql_fetch_row` function on the `resultset`, which will extract one row from the `resultset` and assign it to the array `row`.

The `users` table contains the following three columns:

- `email_address` `varchar(30)`
- `password` `varchar(30)`
- `address_of_delivery` `text`

The array `row` will contain information of the accessed row, where the subscripts `row[0]`, `row[1]`, and `row[2]` will contain the data of the columns `email_address`, `password`, and `address_of_delivery`, respectively. The current information of the user will be displayed by displaying the current email address, password, and address of delivery that's assigned to the subscripts `row[0]`, `row[1]`, and `row[2]`. Then, we will invoke the `mysql_free_result` function to free up the memory that is allocated to the `resultset`.

At this stage, the user will be asked to confirm whether they really want to delete the shown record. The user is supposed to enter `yes`, all in lowercase, to delete the record. If the user enters `yes`, an SQL `DELETE` statement will be created that will delete the row from the `users` table that matches the specified email address. The `mysql_query` function will be invoked and the SQL `DELETE` statement will be passed to it, along with the connection handler `conn`.

If any error occurs in executing the SQL `DELETE` query, again an error message will be displayed and the program will terminate. If the SQL `DELETE` statement executes successfully, a message informing that the user account with the specified mail address has been deleted successfully is displayed. Finally, we will invoke the `mysql_close` function to close the opened connection handler `conn`.

Let's open the Cygwin Terminal. We will require two Terminal windows; on one window, we will run MySQL commands and on the other, we will compile and run C. Open another Terminal window by pressing `Alt+F2`. In the first Terminal window, invoke the MySQL command line by giving the following command:

```
$ mysql -u root -p -h 127.0.0.1
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 27
Server version: 5.7.14-log MySQL Community Server (GPL)
Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

To work with our ecommerce database, we need to make it the current database. So, open the ecommerce database by using the following command:

```
MySQL [(none)]> use ecommerce;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Database changed
```

Now, ecommerce is our current database; that is, whatever SQL commands we will execute will be applied to the ecommerce database only. Let's use the following SQL SELECT command to see the existing rows in the users database table:

```
MySQL [ecommerce]> select * from users;
+-----+-----+
| email_address | password | address_of_delivery |
+-----+-----+
| bmharwani@yahoo.com | coffee | 444, Sky Valley, Toronto, Canada
|
| harwanibm@gmail.com | diamond | House No. xyz, Pqr Apartments, Uvw Lane,
Mumbai, Maharashtra |
| bintu@gmail.com | platinum | abc Sea View, Ocean Lane, Opposite Mt.
Everest, London, UKg
+-----+-----+
3 rows in set (0.00 sec)
```

From the preceding output, we can see that there are three rows in the users table. To compile the C program, switch to the second Terminal window. Let's use GCC to compile the deleteuser.c program, as shown here:

```
$ gcc deleteuser.c -o deleteuser -I/usr/local/include/mysql -
L/usr/local/lib/mysql -lmysqlclient
```

If you get no errors or warnings, that means the deleteuser.c program has compiled into an executable file, deleteuser.exe. Let's run this executable file:

```
$ ./deleteuser
Enter email address of the user to delete: harwanibintu@gmail.com
No user found with this email address
```

Now, let's run the program again with a valid email address:

```
$ ./deleteuser
Enter email address of the user to delete: bmharwani@yahoo.com
The details of the user with this email address are as follows:
Email Address: bmharwani@yahoo.com
Password: coffee
Address of delivery: 444, Sky Valley, Toronto, Canada
```

```
Are you sure you want to delete this record yes/no: yes
The user with the given email address is successfully deleted from the
users table
```

So, the row of the user with the email address `bmharwani@yahoo.com` will be deleted from the `users` table. To confirm that the row has been deleted from the `users` database table too, switch to the Terminal window where the MySQL command line is running and issue the following SQL SELECT command:

```
MySQL [ecommerce]> select * from users;
+-----+-----+-----+
| email_address | password | address_of_delivery
|             |
+-----+-----+-----+
| harwanibm@gmail.com | diamond | House No. xyz, Pqr Apartments, Uvw Lane,
Mumbai, Maharashtra
|
| bintu@gmail.com | platinum | abc Sea View, Ocean Lane, Opposite Mt.
Everest, London, UKg
+-----+-----+
```

Voila! We can see that now there are only two rows left in the `users` table, confirming that one row has been deleted from the `users` table.

Appendix A

In this section of the book, we will look at some additional recipes that were outside the scope of Chapter 3, *Exploring Functions*:

- Creating a sequential file and entering some text into it
- Reading content from a sequential file and displaying it onscreen
- Creating a random file and entering some data into it
- Reading content from a random file and displaying it onscreen
- Decrypting the contents of an encrypted file

Creating a sequential file and entering some data into it

In this recipe, we will be creating a sequential file and the user can enter any desired number of lines into it. The name of the file to be created is passed through command line arguments. You can enter as many lines in the file as you want and, when finished, you have to type `stop`, followed by pressing the *Enter* key.

How to do it...

1. Open a sequential file in write-only mode and point to it with a file pointer:

```
fp = fopen (argv[1], "w");
```

2. Enter the content for the file when prompted:

```
printf("Enter content for the file\n");
gets(str);
```

3. Enter `stop` when you are done entering the file content:

```
while(strcmp(str, "stop") !=0)
```

4. If the string you entered is not `stop`, the string is written into the file:

```
fputs(str, fp);
```

5. Close the file pointer to release all the resources allocated to the file:

```
fclose(fp);
```

The `createtextfile.c` program for creating a sequential file is as follows:

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

void main (int argc, char* argv[])
{
    char str[255];
    FILE *fp;

    fp = fopen (argv[1], "w");
    if (fp == NULL) {
        perror ("An error occurred in creating the file\n");
        exit(1);
    }
    printf("Enter content for the file\n");
    gets(str);
    while(strcmp(str, "stop") !=0) {
        fputs(str,fp);
        gets(str);
    }
    fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We define a file pointer by the name `fp`. We will open the sequential file, whose name is supplied through the command-line argument, in write-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in write-only mode, it might be because there are not enough permissions or disk space constraints. An error message will be displayed and the program will terminate.

If the file opens successfully in write-only mode, you will be prompted to enter the contents for the file. All the text you enter will be assigned to the `str` string variable, which is then written into the file. You should enter `stop` when you have completed entering the content into the file. Finally, we will close the file pointer.

Let's use GCC to compile the `createtextfile.c` program, as shown in the following statement:

```
D:\CBook>gcc createtextfile.c -o createtextfile
```

If you get no errors or warnings, it means that the `createtextfile.c` program has been compiled into an executable file, `createtextfile.exe`. Let's run this executable file:

```
D:\CBook>createtextfile myfile.txt
Enter content for the file
I am trying to create a sequential file. It is through C programming. It
is very hot today
I have a cat. do you like animals? It might rain
Thank you. bye
stop
```

Voila! We've successfully created a sequential file and entered data in it.

Now let's move on to the next recipe!

Reading content from a sequential file and displaying it onscreen

In this recipe, we assume that a sequential file already exists, so we will be reading the contents of that file and displaying it onscreen. The name of the file whose contents we want to read will be supplied through command-line arguments.

How to do it...

1. Open the sequential file in read-only mode and set the `fp` file pointer to point at it:

```
fp = fopen (argv [1], "r");
```

2. The program terminates if the file cannot be opened in read-only mode:

```
if (fp == NULL) {
    printf("%s file does not exist\n", argv[1]);
    exit(1);
}
```

3. Set a `while` loop to execute until the end of the file is reached:

```
while (!feof(fp))
```

4. Within the `while` loop, one line at a time is read from the file and is displayed onscreen:

```
fgets(buffer, BUFFSIZE, fp);
puts(buffer);
```

5. Close the file pointer to release all the resources allocated to the file:

```
fclose(fp);
```

The `readtextfile.c` program for reading a sequential file is as follows:

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

#define BUFFSIZE 255

void main (int argc, char* argv[])
{
    FILE *fp;
    char buffer[BUFFSIZE];

    fp = fopen (argv [1], "r");
    if (fp == NULL) {
        printf("%s file does not exist\n", argv[1]);
        exit(1);
    }
    while (!feof(fp))
    {
        fgets(buffer, BUFFSIZE, fp);
        puts(buffer);
    }
    fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will define a file pointer is defined by the name of `fp` and a string by the name of `buffer` of size 255. We will open the sequential file, whose name is supplied through the command-line argument, in read-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in read-only mode because the file does not exist or because of a lack of enough permissions, an error message will be displayed and the program will terminate.

If the file opens successfully in read-only mode, set a `while` loop to execute until the end of the file is reached. Use the `fgets` function to read one line at a time from the file; the line read from the file is assigned to the `buffer` string. Then, display the content in the `buffer` string onscreen. When the end of the file has been reached, the `while` loop will terminate and the file pointer is closed to release all the resources allocated to the file.

Let's use GCC to compile the `readtextfile.c` program, as shown in the following statement:

```
D:\CBook>gcc readtextfile.c -o readtextfile
```

If you get no errors or warnings, this means that the `readtextfile.c` program has been compiled into an executable file, `readtextfile.exe`. Assuming the file whose content we want to read is `textfield.txt`, let's run the executable file, `readtextfile.exe`:

```
D:\CBook>readtextfiletextfield.txt
I am trying to create a sequential file. it is through C programming. It
is very hot today. I have a cat. do you like animals? It might rain.
Thank you. bye
```

Voila! We've successfully read the content from our sequential file and displayed it onscreen.

Now, let's move on to the next recipe!

Creating a random file and entering some data into it

In this recipe, we will be creating a random file and we will enter some lines of text into it. The random files are structured, and the content in the random file is written via structures. The benefit of creating a file using structures is that we can compute the location of any structure directly and can access any content in the file randomly. The name of the file to be created is passed through command-line arguments.

How to do it...

The following are the steps to create a random file and enter a few lines of text in it. You can enter any number of lines as desired; simply type `stop`, followed by the *Enter* key, when you are done:

1. Define a structure consisting of a string member:

```
struct data{  
    char str[ 255 ];  
};
```

2. Open a random file in write-only mode and point to it with a file pointer:

```
fp = fopen (argv[1], "wb");
```

3. The program will terminate if the file cannot be opened in write-only mode:

```
if (fp == NULL) {  
    perror ("An error occurred in creating the file\n");  
    exit(1);  
}
```

4. Enter the file contents when prompted and store it into the structure members:

```
printf("Enter file content:\n");  
gets(line.str);
```

5. If the text entered is not `stop`, the structure containing the text is written into the file:

```
while(strcmp(line.str, "stop") !=0){  
    fwrite( &line, sizeof(struct data), 1, fp );
```

6. Steps 4 and 5 are repeated until you enter `stop`.

7. When you enter `stop`, the file pointed at by the file pointer is closed to release the resources allocated to the file:

```
fclose(fp);
```

The `createrandomfile.c` program for creating a random file is as follows:

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

struct data{
    char str[ 255 ];
};

void main (int argc, char* argv[])
{
    FILE *fp;
    struct data line;
    fp = fopen (argv[1], "wb");
    if (fp == NULL) {
        perror ("An error occurred in creating the file\n");
        exit(1);
    }
    printf("Enter file content:\n");
    gets(line.str);
    while(strcmp(line.str, "stop") !=0){
        fwrite( &line, sizeof(struct data), 1, fp );
        gets(line.str);
    }
    fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

Let's start by defining a structure by the name `data` consisting of a member `str`, which is a string variable of size 255. Then we will define a file pointer by the name `fp`, followed by a variable `line` as a type `data` structure, so the `line` becomes a structure with a member called `str`. We will open a random file, whose name is supplied through a command-line argument, in write-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in write-only mode for any reason, an error message will be displayed and the program will terminate.

You will be prompted to enter the file contents. The text you enter will be assigned to the `str` member of the line structure. Because you are supposed to enter `stop` to indicate that you have finished entering data in the file, the text you entered will be compared with the `stop` string. If the text entered is not `stop`, it is written into the file pointed at by the `fp` file pointer.

Because it is a random file, the text is written into the file through the structure line. The `fwrite` function writes the number of bytes equal to the size of the structure line into the file pointed at by the `fp` pointer at its current position. The text in the `str` member of the line structure is written into the file. When the text entered by the user is `stop`, the file pointed at by file pointer, the `fp` file pointer is closed.

Let's use GCC to compile the `createrandomfile.c` program, as shown in the following statement:

```
D:\CBook>gcc createrandomfile.c -o createrandomfile
```

If you get no errors or warnings, this means that the `createrandomfile.c` program has been compiled into an executable file, `createrandomfile.exe`. Assuming that we want to create a random file with the name `random.data`, let's run the executable file, `createrandomfile.exe`:

```
D:\CBook>createrandomfile random.data
Enter file content:
This is a random file. I am checking if the code is working
perfectly well. Random file helps in fast accessing of
desired data. Also you can access any content in any order.
stop
```

Voila! We've successfully created a random file and entered some data in it. Now let's move on to the next recipe!

Reading content from a random file and displaying it onscreen

In this recipe, we will be reading the contents of a random file and will be displaying it on screen. Because the content in the random file comprises records, where the size of the record is already known, any record from the random file can be picked up randomly; hence, this type of file gets the name of *random file*. To access the *n*th record from a random file, we don't have to read all the *n-1* records first as we would do in a sequential file. We can compute the location of that record and can access it directly. The name of the file to be read is passed through command-line arguments.

How to do it...

1. Define a structure consisting of a string member:

```
struct data{  
    char str[ 255 ];  
};
```

2. Open a random file in read-only mode and point at it with a file pointer:

```
fp = fopen (argv[1], "rb");
```

3. The program will terminate if the file cannot be opened in read-only mode:

```
if (fp == NULL) {  
    perror ("An error occurred in opening the file\n");  
    exit(1);  
}
```

4. Find the total number of bytes in the file. Divide the retrieved total number of bytes in the file by the size of one record to get the total number of records in the file:

```
fseek(fp, 0L, SEEK_END);  
n = ftell(fp);  
nol=n/sizeof(struct data);
```

5. Use a `for` loop to read one record at a time from the file:

```
for (i=1;i<=nol;i++)  
    fread(&line,sizeof(struct data),1,fp);
```

6. The content read from the random file is via the structure defined in step 1. Display the contents of the file by displaying the file content assigned to structure members:

```
puts(line.str);
```

7. The end of the file is reached when the `for` loop has finished. Close the file pointer to release the resources allocated to the file:

```
fclose(fp);
```

The `readrandomfile.c` program for reading the content from a random file is as follows:

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

struct data{
    char str[ 255 ];
};

void main (int argc, char* argv[])
{
    FILE *fp;
    struct data line;
    int n,nol,i;
    fp = fopen (argv[1], "rb");
    if (fp == NULL) {
        perror ("An error occurred in opening the file\n");
        exit(1);
    }
    fseek(fp, 0L, SEEK_END);
    n = ftell(fp);
    nol=n/sizeof(struct data);
    rewind(fp);
    printf("The content in file is :\n");
    for (i=1;i<=nol;i++)
    {
        fread(&line,sizeof(struct data),1,fp);
        puts(line.str);
    }
    fclose(fp);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will define a structure by the name, `data`, consisting of a member, `str`, which is a string variable of size 255. Then, a file pointer is defined by the name of `fp` and a variable line by the type data structure, so the line becomes a structure with a member called `str`. We will open a random file, whose name is supplied through the command-line argument, in read-only mode and set the `fp` file pointer to point at it. If the file cannot be opened in read-only mode for any reason, an error message will be displayed and the program will terminate. The file error can occur if any non-existing file is referred to, or if the file does not have enough permissions. If the file opens in read-only mode successfully, the next step is to find the total count of the number of records in the file. For this, the following formula is applied:

$$\text{Total number of bytes in the file}/\text{size of one record}$$

To find the total number of bytes in the file, first move the file pointer to the end of the file by invoking the `fseek` function. Thereafter, using the `ftell` function, retrieve the total number of bytes consumed by the file. We will then divide the total number of bytes in the file by the size of one record to determine the total count of records in the file.

Now, we are ready to read one record at a time from the file, and to do so, we will move the file pointer to the beginning of the file. We will set a `for` loop to execute the same amount of times as the number of records in the file. Within the `for` loop, we will invoke the `fread` function to read one record at a time from the file. The text read from the file is assigned to the `str` member of the line structure. The content in the `str` member of the line structure is displayed onscreen. When the `for` loop terminates, the file pointed to by the `fp` file pointer is closed to release the resources allocated to the file.

Let's use GCC to compile the `readrandomfile.c` program , as shown in the following statement:

```
D:\CBook>gcc readrandomfile.c -o readrandomfile
```

If you get no errors or warnings, this means that the `readrandomfile.c` program has been compiled into an executable file, `readrandomfile.exe`. Assuming that we want to create a random file with the name `random.data`, let's run the executable file, `readrandomfile.exe`:

```
D:\CBook>readrandomfile random.data
The content in file is :
This is a random file. I am checking if the code is working
perfectly well. Random file helps in fast accessing of
desired data. Also you can access any content in any order.
```

Voila! We've successfully read the content from a random file and displayed it onscreen.

Now let's move on to the next recipe!

Decrypting the contents of an encrypted file

In this recipe, we will be reading an encrypted file. We will be decrypting its content and writing the decrypted contents into another sequential file. Both filenames, the encrypted one and the one with which we will save the decrypted version, are supplied through command-line arguments.

How to do it...

Two files are used in this program. One is opened in read-only mode and the other is opened in write-only mode. The contents of one file is read and decrypted, and the decrypted content is stored in another file. The following are the steps to decrypt an existing encrypted file and save the decrypted version into another file:

1. Open two files, one in read-only and the other in write-only mode:

```
fp = fopen (argv [1], "r");
fq = fopen (argv[2], "w");
```

2. If either of the files cannot be opened in their respective modes, the program will terminate after displaying an error message:

```
if (fp == NULL) {
    printf("%s file does not exist\n", argv[1]);
    exit(1);
}
if (fq == NULL) {
    perror ("An error occurred in creating the file\n");
    exit(1);
}
```

3. Set a `while` loop to execute. It will read one line at a time from the file to be read:

```
while (!feof(fp))
fgets(buffer, BUFSIZE, fp);
```

4. The length of the line that is read from the file is computed:

```
n=strlen(buffer);
```

5. Set a `for` loop to execute. This will access all the characters of the line one by one:

```
for(i=0;i<n;i++)
```

6. Add the value of 45 to the ASCII value of each character to decrypt it. I assume that ASCII value 45 was deducted from each character to encrypt the file:

```
buffer[i]=buffer[i]+45;
```

7. The decrypted line is written into the second file:

```
fputs(buffer,fq);
```

8. When the `while` loop has finished (read, which is the file being decrypted), both the file pointers are closed to release the resources allocated to them:

```
fclose (fp);
fclose (fq);
```

The `decryptfile.c` program for decrypting an encrypted file is as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFFSIZE 255
void main (int argc, char* argv[])
{
    FILE *fp,*fq;
    int i,n;
    char buffer[BUFFSIZE];
    fp = fopen (argv [1], "r");
    if (fp == NULL) {
        printf ("%s file does not exist\n", argv[1]);
        exit(1);
    }
    fq = fopen (argv[2], "w");                                if (fq == NULL) {
        perror ("An error occurred in creating the file\n");
        exit(1);
    }
    while (!feof(fp))
    {
        fgets(buffer, BUFFSIZE, fp);
        n=strlen(buffer);
        for(i=0;i<n;i++)
            buffer[i]=buffer[i]+45;
        fputs(buffer,fq);
    }
}
```

```
        buffer[i]=buffer[i]+45;
        fputs (buffer,fq);
    }
fclose (fp);
fclose (fq);
}
```

Now, let's go behind the scenes to understand the code better.

How it works...

We will define two file pointers, `fp` and `fq`. We will open the first file that is supplied through the command-line argument in read-only mode, and the second file in write-only mode. If the files cannot be opened in the read-only mode and write-only mode, respectively, an error message will be displayed and the program will terminate. The file that opens in read-only mode is pointed at by the `fp` file pointer and the file that opens in write-only mode is pointed at by the file `fq` file pointer.

We will set a `while` loop to execute, which will read all the lines from the file pointed at by the `fp` pointer one by one. The `while` loop will continue to execute until the end of the file pointed at by `fp` is reached.

Within the `while` loop, a line is read and is assigned to the `buffer` string variable. The length of the line is computed. We will then set a `for` loop is set to execute up until the end of the line; that is, each character of the line is accessed. We will add the value 45 to the ASCII value of each character to encrypt it. Thereafter, we will write the decrypted line into the second file. The file is pointed at by the `fq` file pointer. When the file from which the content was read is finished with, both the file pointers are closed to release the resources allocated to both the files.

Let's use GCC to compile the `decryptfile.c` program, as shown in the following statement:

```
D:\CBook>gcc decryptfile.c -o decryptfile
```

Assume the encrypted file is named `encrypted.txt`. Let's see the encrypted text in this file:

```
D:\CBook>type encrypted.txt
≤4@≤GEL<A:≤GB≤6E84G8≤4≤F8DH8AG<4?≤9<?8≤<G≤<F≤G;EBH: ;≤≤CEB:E4@@<A:≤≤≤G≤<F≤I8E
I≤;BG≤GB74I≤;4I8≤4≤64G≤≤7B≤LBH≤?>>8≤4A<@4?F≤≤≤G≤@< :;G≤E4<A' ;4A>≤LBH≤5I8
```



The preceding command is executed in Windows' Command Prompt.

If you get no errors or warnings while compiling the file, this means that the `decryptfile.c` program has been compiled into an executable file, `decryptfile.exe`. Assume that an encrypted file by the name of `encrypted.txt` already exists and you want to decrypt it into another file, `originalfile.txt`. So, let's run the executable, `decryptfile.exe`, to decrypt the `encrypted.txt` file:

```
D:\CBook>decryptfile encrypted.txt originalfile.txt
```

Let's see the contents of `originalfile.txt` to see whether it contains the decrypted version of the file:

```
D:\CBook>type originalfile.txt
I am trying to create a sequential file. it is through C programming. It
is very hot today. I have a cat. do you like animals? It might rain.
Thank you. bye
```

Voila! You can see that `originalfile.txt` contains the decrypted file.

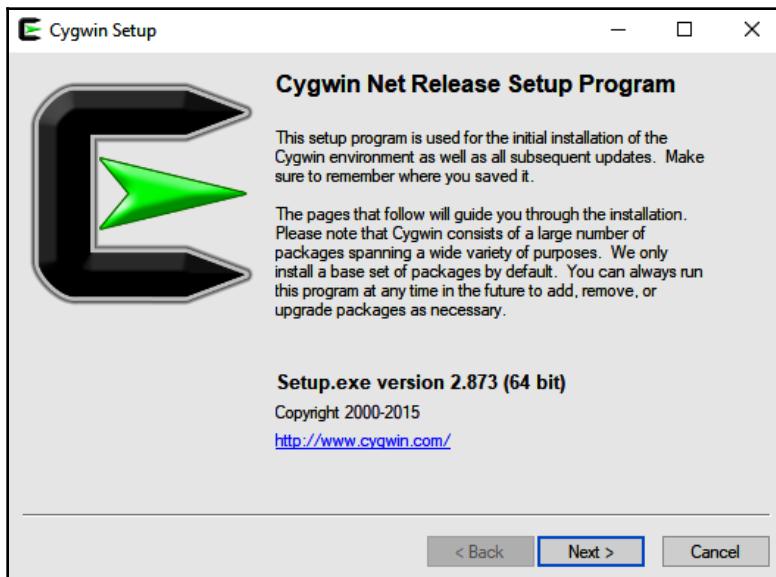
Appendix B

I am using Cygwin for compiling and running my C programs in this book. Cygwin is an excellent tool that provides an environment to compile and run UNIX or Linux applications on a Windows operating system. In this section of the book, we will learn how to install Cygwin.

Installing Cygwin

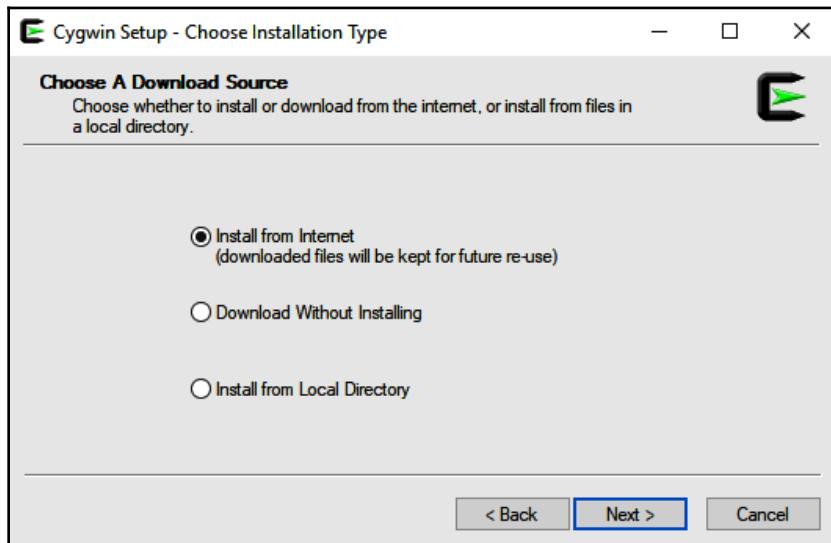
To download Cygwin, visit <https://www.cygwin.com/>. You will be provided with two setup files, `setup-x86_64.exe` for 64-bit installation and `setup-x86.exe` for 32-bit installation. Depending on the Windows operating system version that is installed on your machine, you can download a 64-bit or 32-bit setup file. Because I have 64-bit Windows 10 installed on my computer, I have downloaded the `setup-x86_64.exe` file. Perform the following steps:

1. To begin the Cygwin installation, simply double-click the downloaded `setup-x86_64.exe` file. You will then see the following dialog, displaying a small introduction of the Cygwin setup program. Simply click the **Next** button to continue:

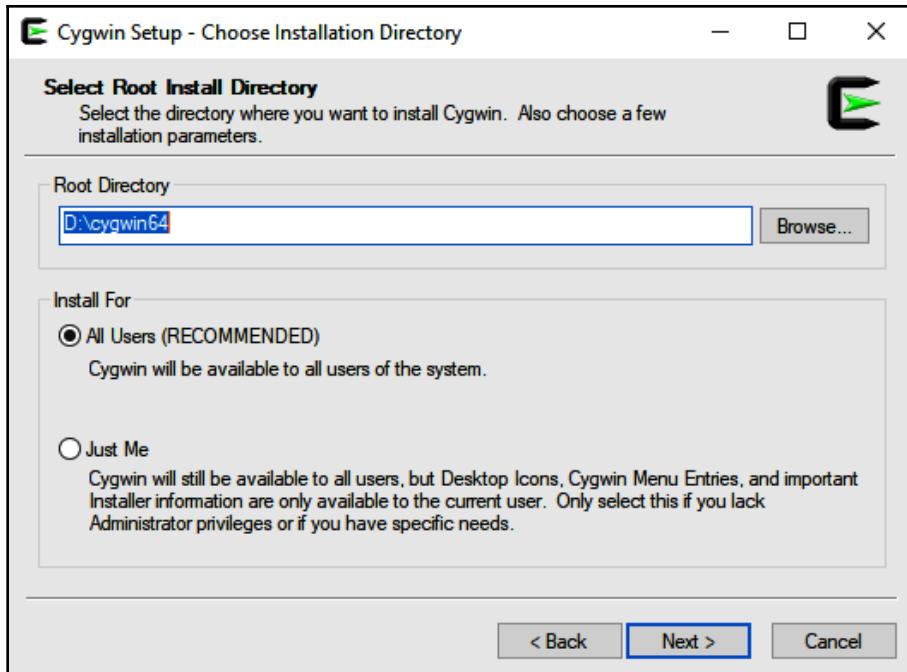


2. You will then see a dialog asking you to choose the type of installation you want from the following three options:
 - **Install from Internet:** This option will download the Cygwin files from the net and will install them. The downloaded files will be saved on the local hard disk for future use.
 - **Download Without Installing:** This option will download the entire Cygwin program, but will not install it (but you can install it from the downloaded files anytime in the future).
 - **Installation from Local Directory:** This option installs Cygwin from the setup files that were downloaded earlier.

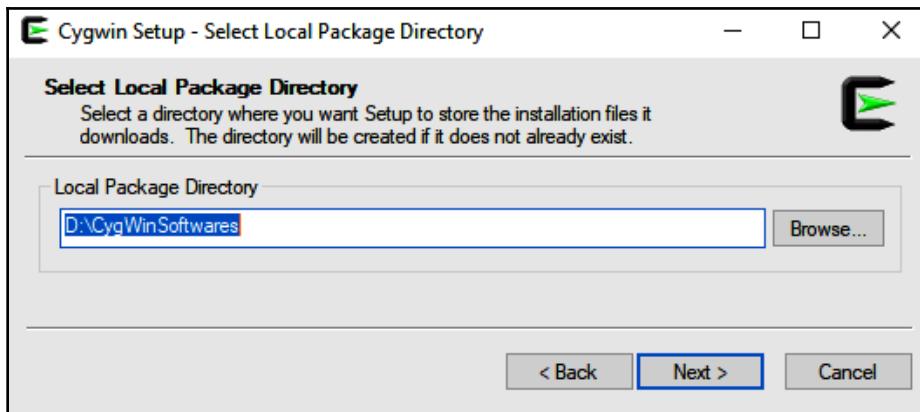
Because we want to download and install files from the internet, let's select the first option, **Install from Internet**, and click the **Next** button, as shown in the following screenshot:



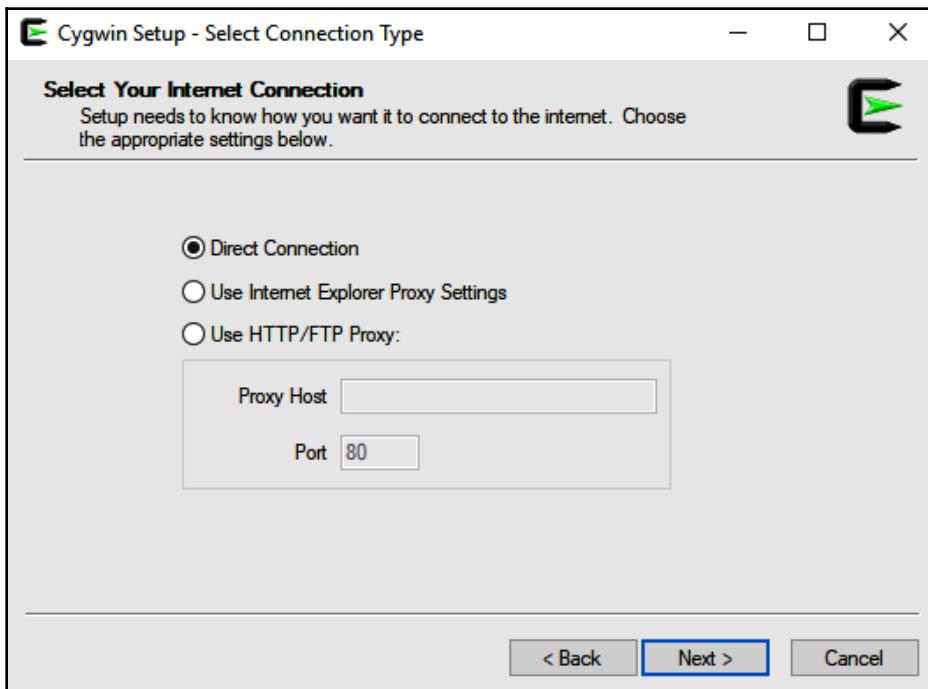
3. You will be prompted to specify the drive and directory that you want to install Cygwin in. I want to install Cygwin in the D: drive, in the folder named `cygwin64` (as shown in the following screenshot). You can specify any drive or folder you desire. You also need to specify whether you want Cygwin to be used only by you or by all of the users of this system. Because we want Cygwin to be used by all of the users of the system, we will choose the first option, selecting **All Users** (which is the default choice), and click the **Next** button, as shown in the following screenshot:



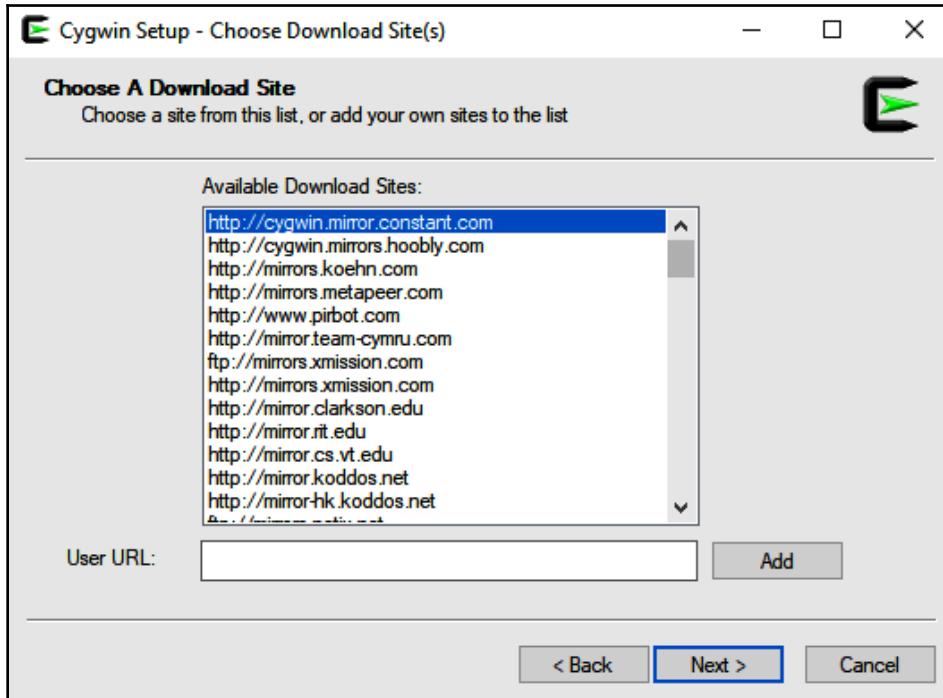
4. You will be prompted to specify the drive and directory in which you want to store the installation files (as shown in the following screenshot). After specifying the desired drive and directory, click the **Next** button to continue:



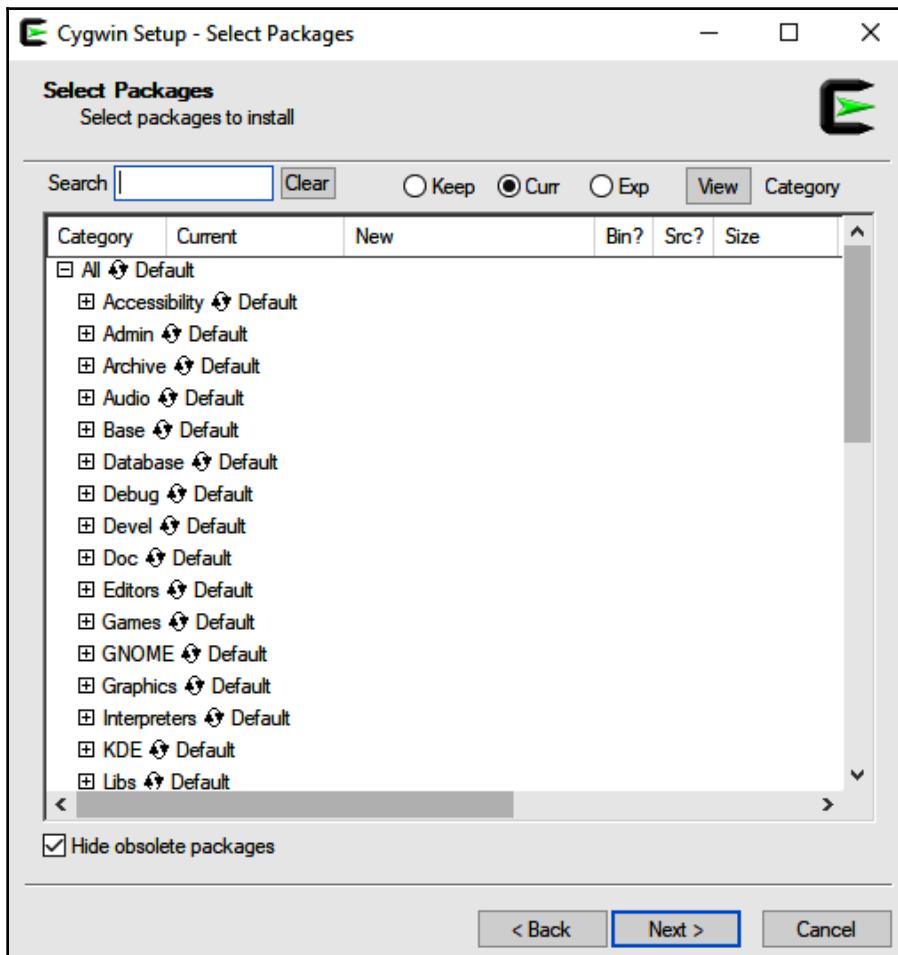
5. You will be asked to specify how you are connected to the internet—that is, whether you are directly connected to the internet or are connected via a proxy server. If you are connected through a proxy, you need to specify the details of the proxy host and its port number. Because I am directly connected to the internet, I will select the **Direct Connection** option (as shown in the following screenshot) and then click the **Next** button:



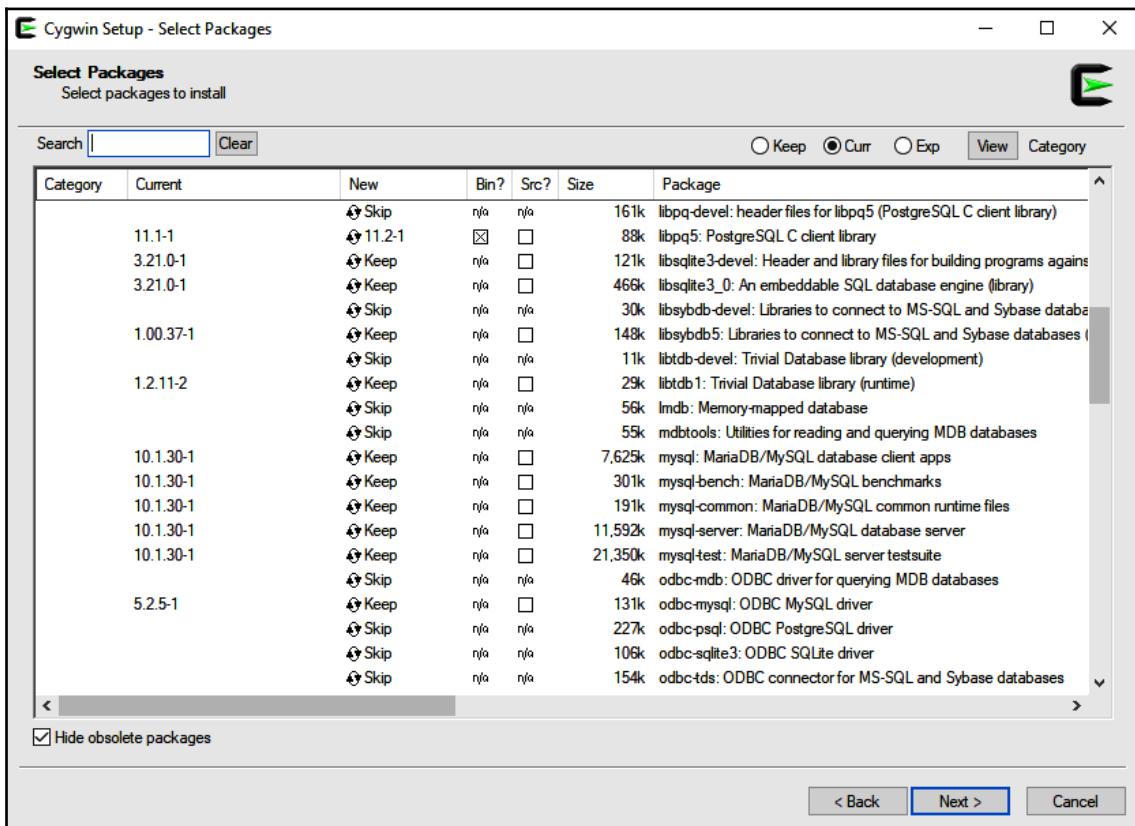
6. You will be shown a list of sites that you can download and install Cygwin from, as shown in the following screenshot. Select any option from the list and click the **Next** button:



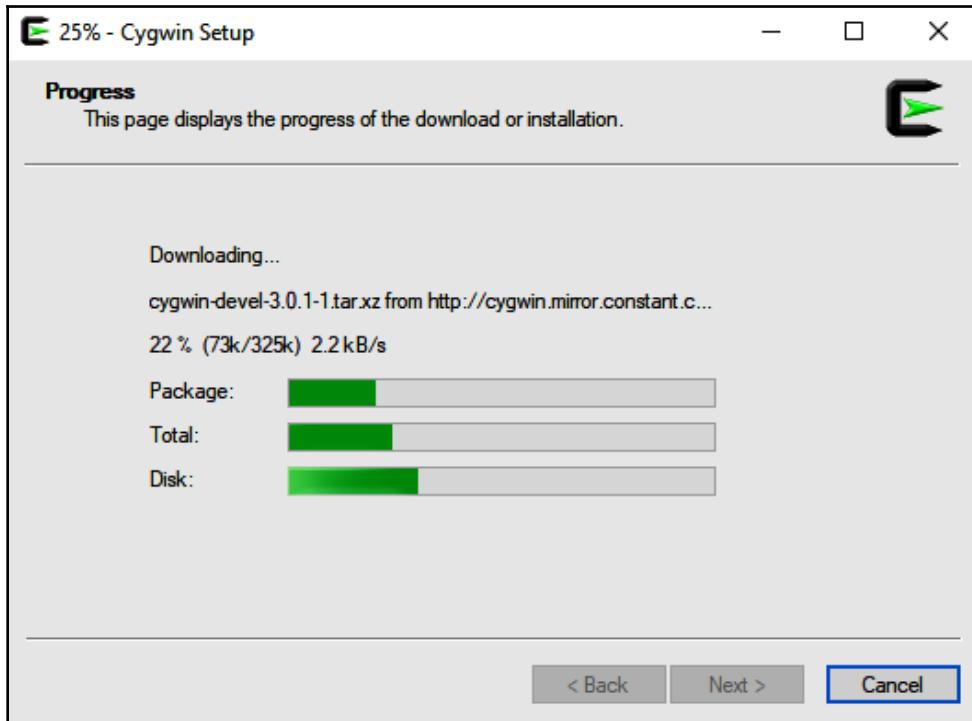
7. You will be shown a list of categories of packages that you want to install (as shown in the following screenshot). Besides the packages that are selected by default, we need to install some files from the **Database** package category too. We will be requiring **Database** packages because we will be accessing MySQL databases and tables using C programs in this book. You can click the + symbol in the **Database** category node to expand it:



8. You will get the list of the packages that are available to download under the **Database** category node. Select the packages that begin with the prefix **mysql**, such as **mysql client apps**, **mysql bench**, **mysql common**, **mysql server**, as shown in the following screenshot. After selecting the required packages, click the **Next** button to download and install Cygwin:



9. The Cygwin files will then be downloaded from the net and installed on your machine. Once Cygwin is successfully installed, you will get a confirmation dialog. Click the **Finish** button to close the dialog box:



Now you are ready to use Cygwin for implementing the recipes in this book.

Appendix C

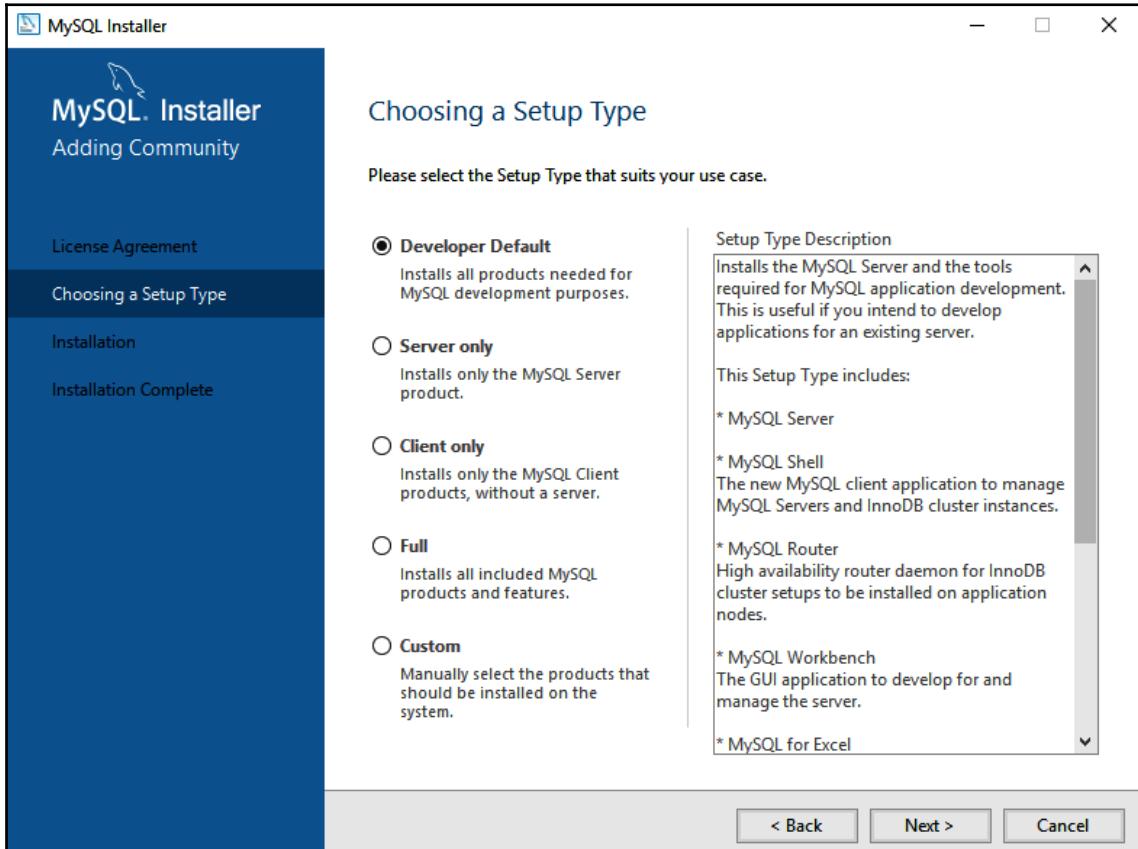
In this section of the book, we will quickly review how to install MySQL Server. This is required to implement the recipes in Chapter 8, *Using MySQL Database*.

Installing MySQL Server

You need to download the MySQL Community Server from, <https://dev.mysql.com/downloads/>. The latest version of MySQL Server that is available at the time of writing is 8.0.15. The installer file that will be downloaded is named `mysql-installer-community-8.0.15.0.msi`.

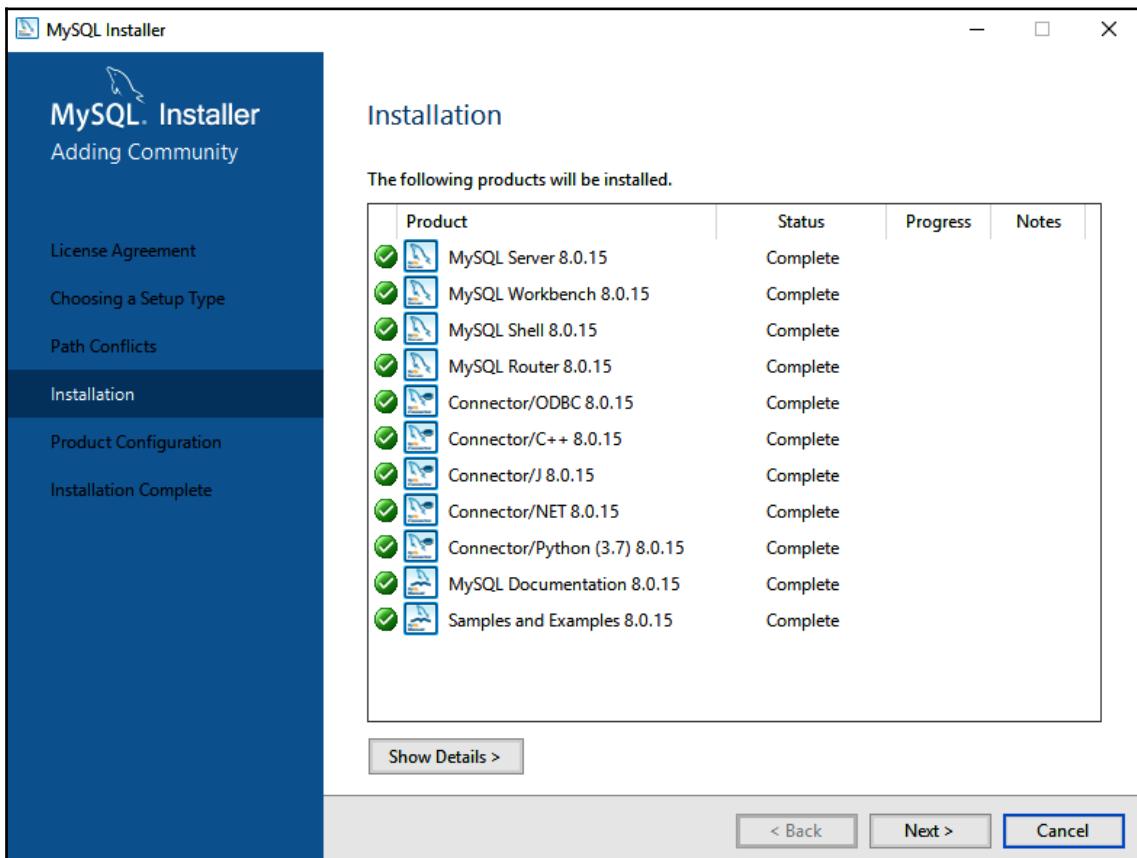
1. Simply double-click the file to initiate the installation. The first screen that will appear is a License Agreement. Go through the different terms and conditions that are mentioned. If you agree with the terms, click the, **I accept the license terms** checkbox and then click the **Next** button.
2. The next dialog will prompt you to choose the setup type. The following options will be displayed:
 - **Developer Default:** It will install MySQL Server and other tools that enable developing applications too.
 - **Server only:** It will install only MySQL Server.
 - **Client only:** It will install MySQL applications and connectors, enabling access to the MySQL database tables from the client machine.
 - **Full:** It will install all the available MySQL products.
 - **Custom:** It will prompt you to select the MySQL products that you want to install.

Because we want to develop applications that access the MySQL database tables, we will select the **Developer Default** option and click the **Next** button, as shown in the following screenshot:



3. You will be prompted to specify the directory where the MySQL Server files need to be installed. Also, you will be asked to specify the data directory. The dialog box also shows the directories by default. You can go ahead with the default directories for installing MySQL Server and the data files, then click the **Next** button to continue.

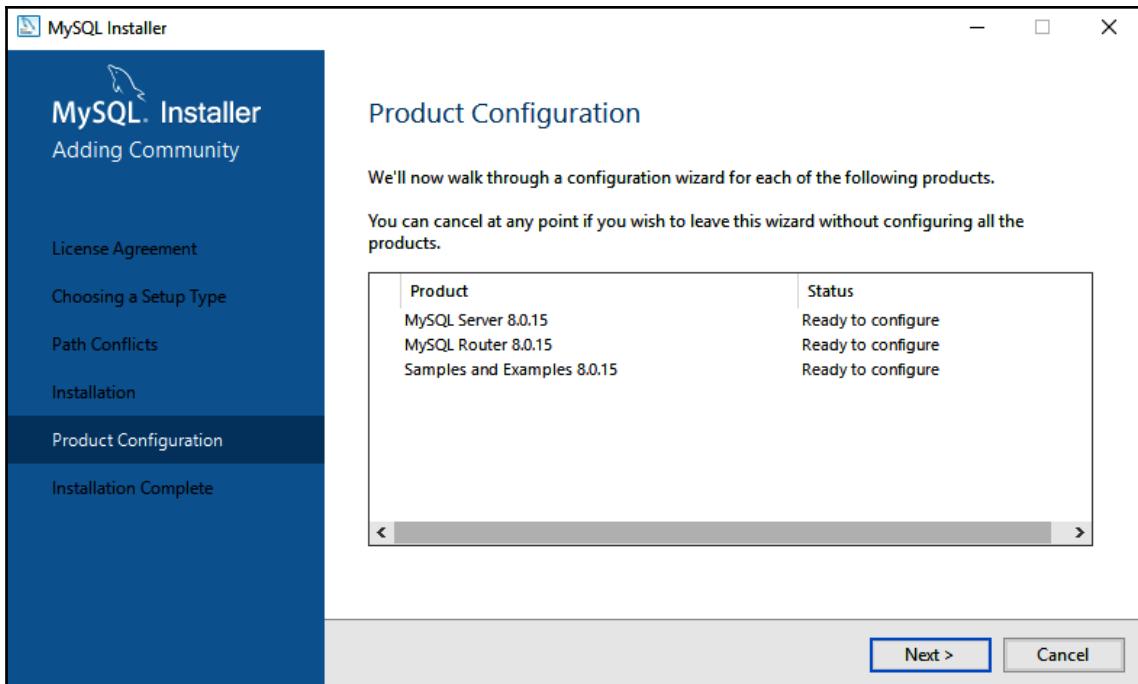
4. Simply accept the default values in the dialog boxes and click the **Next** button to continue with the installation procedure. You will be shown a dialog box indicating the products that will be installed on your machine. Click the **Execute** button to download and install the shown products.
5. When the products are downloaded and installed, you will be shown a dialog box showing the list of products that are installed on your machine, as shown in the following screenshot. Click the **Next** button to move further:



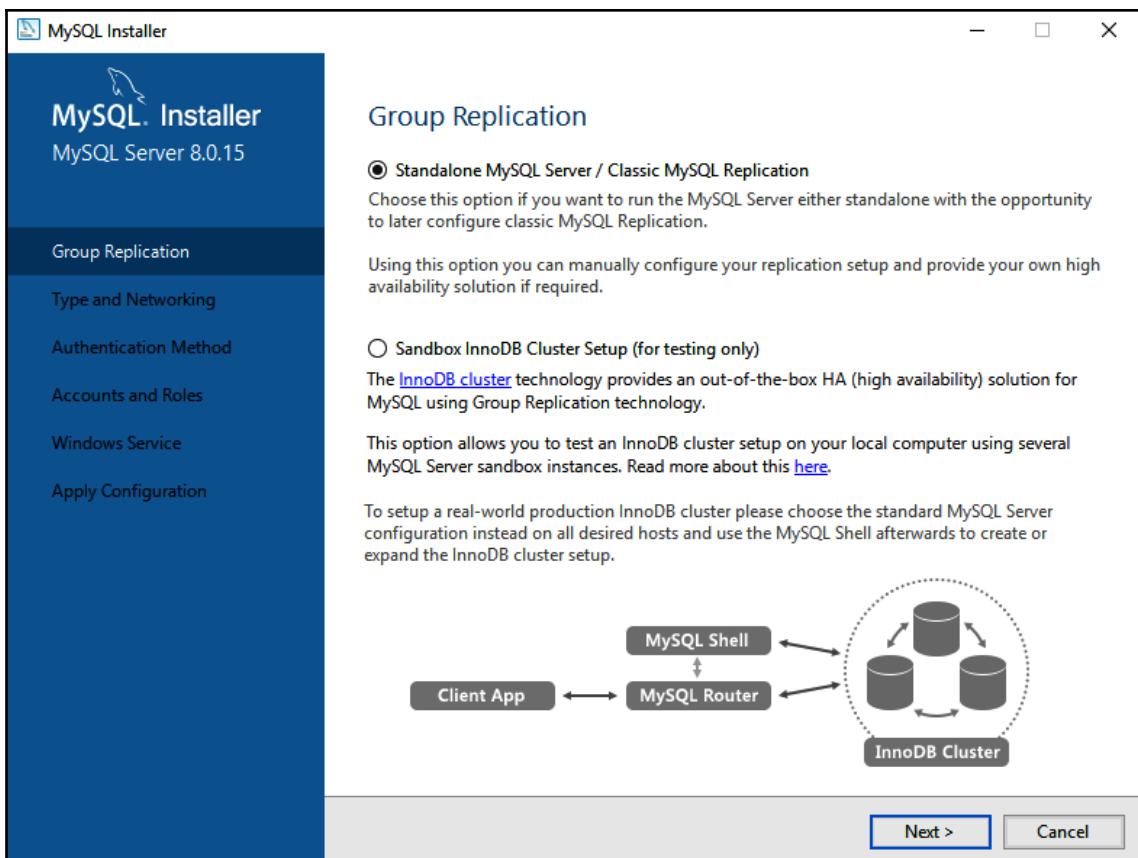
6. The next dialog will show you the following things that need to be configured on your machine:

- MySQL Server
- MySQL Router
- Samples and Examples

Click the **Next** button to configure these three items, as shown in the following screenshot:



7. You will be asked whether you want to configure MySQL Server as a standalone server or as a **Sandbox InnoDB Cluster setup**. In a Sandbox InnoDB Cluster setup, you can easily configure and manage at least three MySQL Server instances in the format of InnoDB clusters. Each of the MySQL Servers has the capability to replicate data within InnoDB clusters. The cluster is smart enough to reconfigure automatically in case a failure occurs in any server instance. But because we want a single standalone server instance, select the **Standalone MySQL Server** option and click the **Next** button to continue, as shown in the following screenshot:



8. You will be prompted to select the server configuration type. You need to select the configuration type that suits your needs and the available resources. The following three options will be displayed:

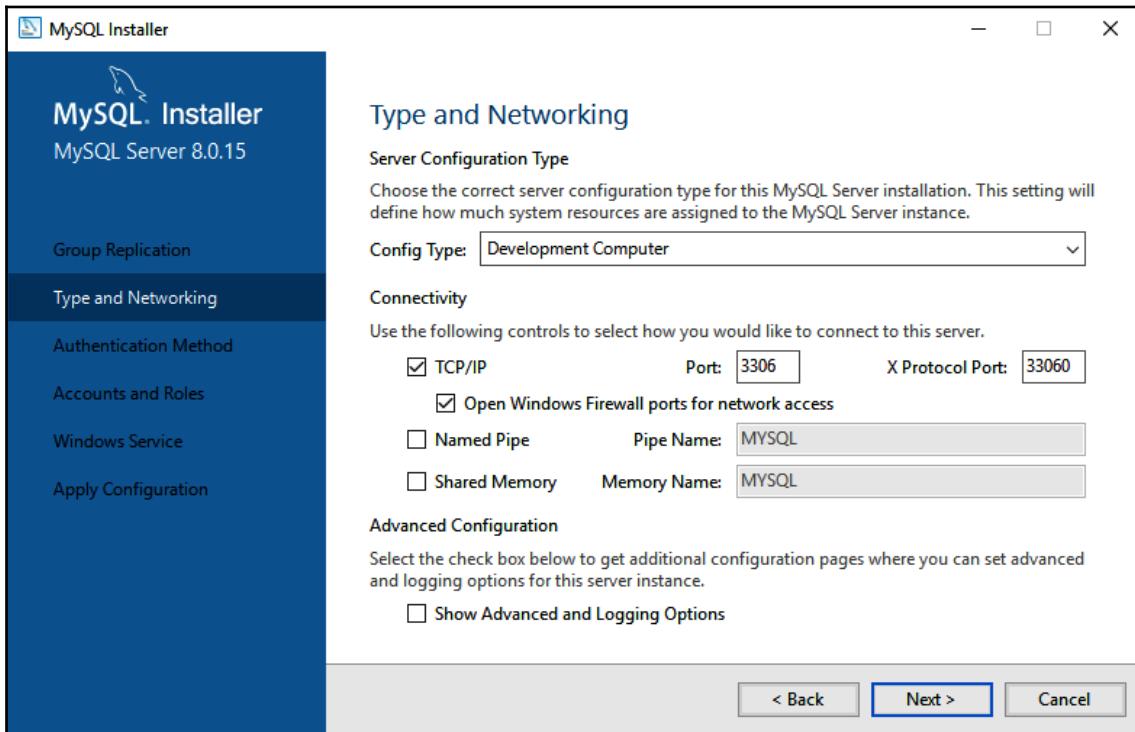
- **Development Computer:** In this configuration type, your machine can run MySQL Server along with other servers and different development frameworks. The machine will be used to develop applications that can access the MySQL database and tables. The MySQL Server will be configured to use the smallest amount of memory.
- **Server:** In this configuration type, besides MySQL Server, other servers might also be run, including web servers. MySQL Server will be configured to use a medium amount of memory.
- **Dedicated:** In this setup type, the machine will be dedicated to run only MySQL Server. Because no other applications or servers will be running on this machine, MySQL Server will be configured to use the majority of the available memory.

Because we will be using this machine for development, that is, we will be running MySQL Server along with other servers and applications, we will choose the default **Config Type, Development Computer**.

9. The dialog box also asks you how you want your applications to connect to MySQL Server. By default, TCP/IP networking is selected, along with port 3306. Also, a checkbox is selected by default that opens the Windows firewall ports for network access. Besides this, you will be provided with two more options, **Named Pipe** and **Shared Memory** to connect with MySQL Server.

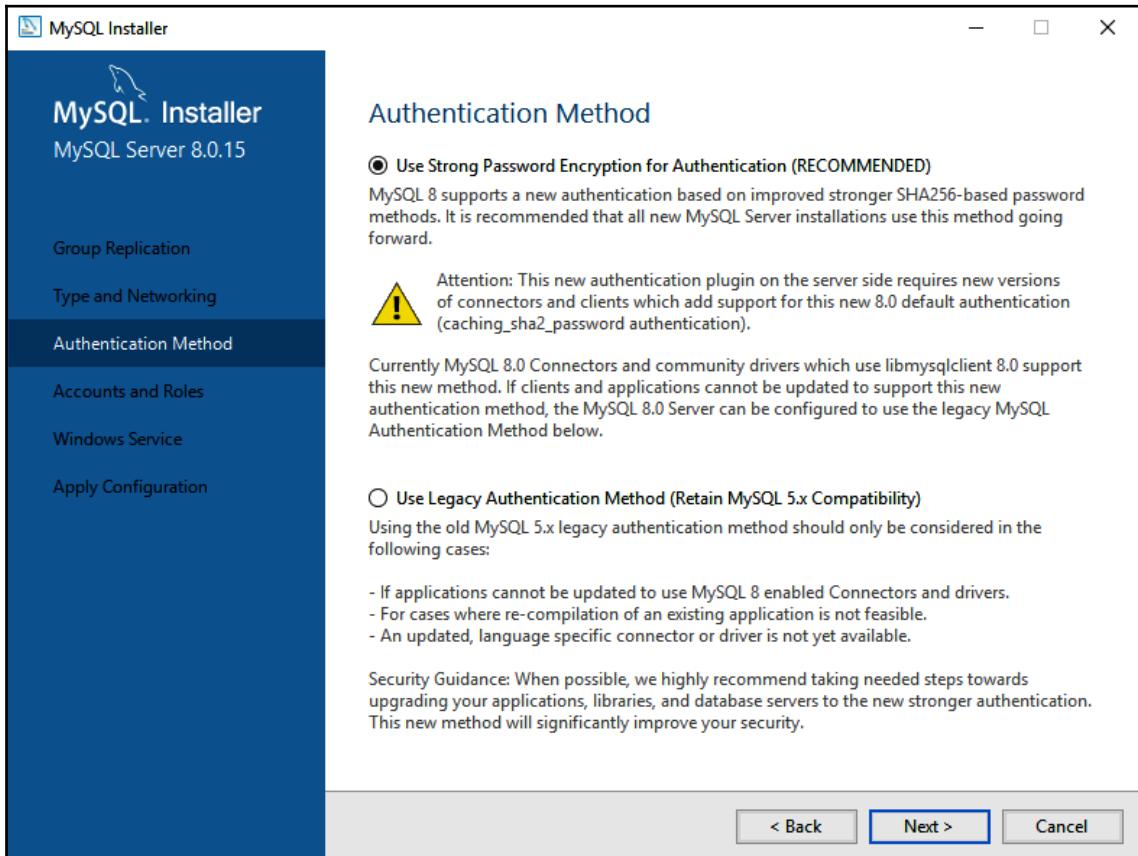
If you want to connect with MySQL Server through a named pipe, you need to enable and define the **Pipe Name**. Similarly, if you want to connect with MySQL Server using the **Shared Memory**, you need to enable and then define the **Memory Name**.

We will keep the default connectivity options TCP/IP, along with port 3306. Click the **Next** button to continue, as shown in the following screenshot:

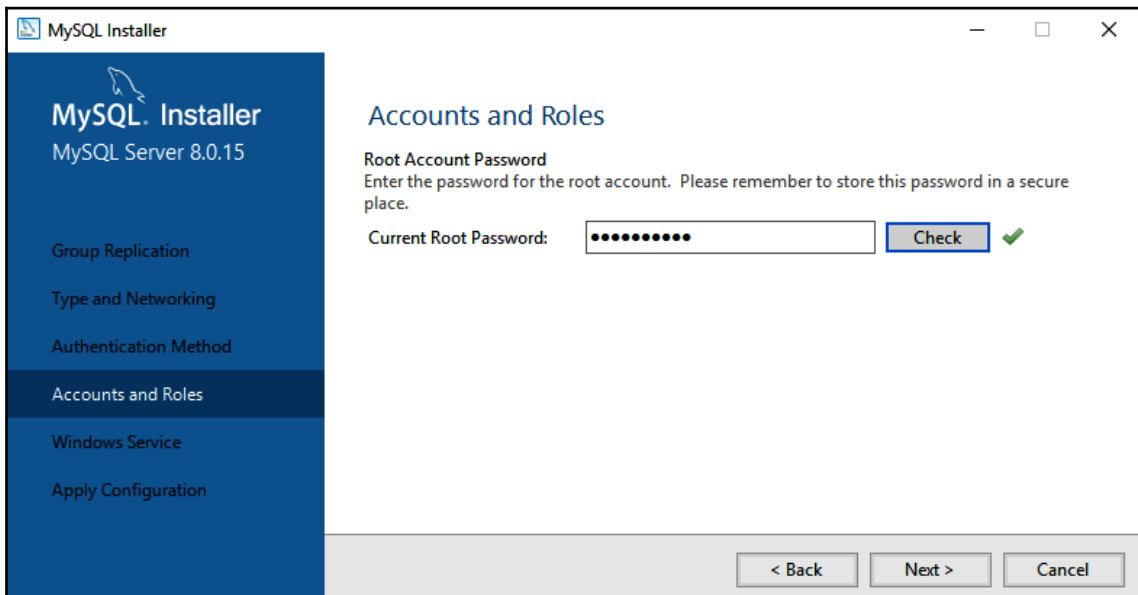


10. The next dialog prompts you to select the authentication method you want to use to connect with the MySQL Server. The following two options will be displayed:
 - **Use Strong Password Encryption for Authentication (RECOMMENDED)**
 - **Use Legacy Authentication Method (Retain MySQL 5.x Compatibility)**

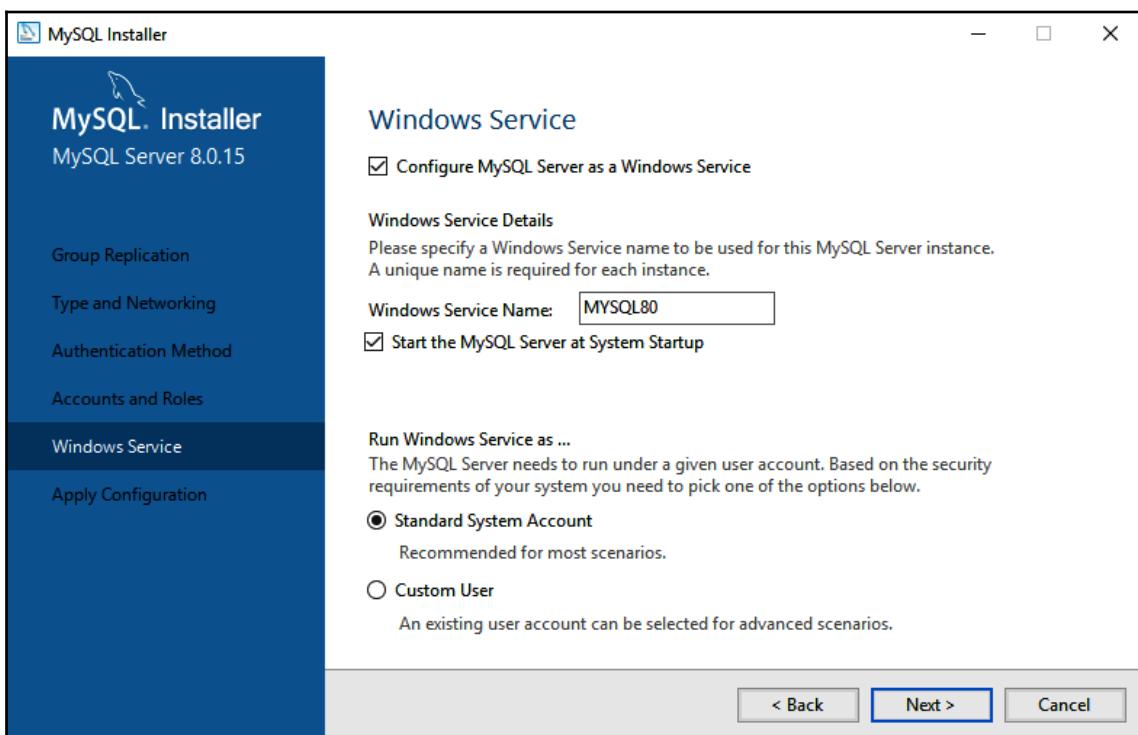
Choose the **Use Legacy Authentication Method** option for any application that cannot be updated to use the latest MySQL 8.0 connectors and drivers. Hence, keep the default first option, **Use Strong Password Encryption for Authentication**, selected and click the **Next** button to continue, as shown in the following screenshot:



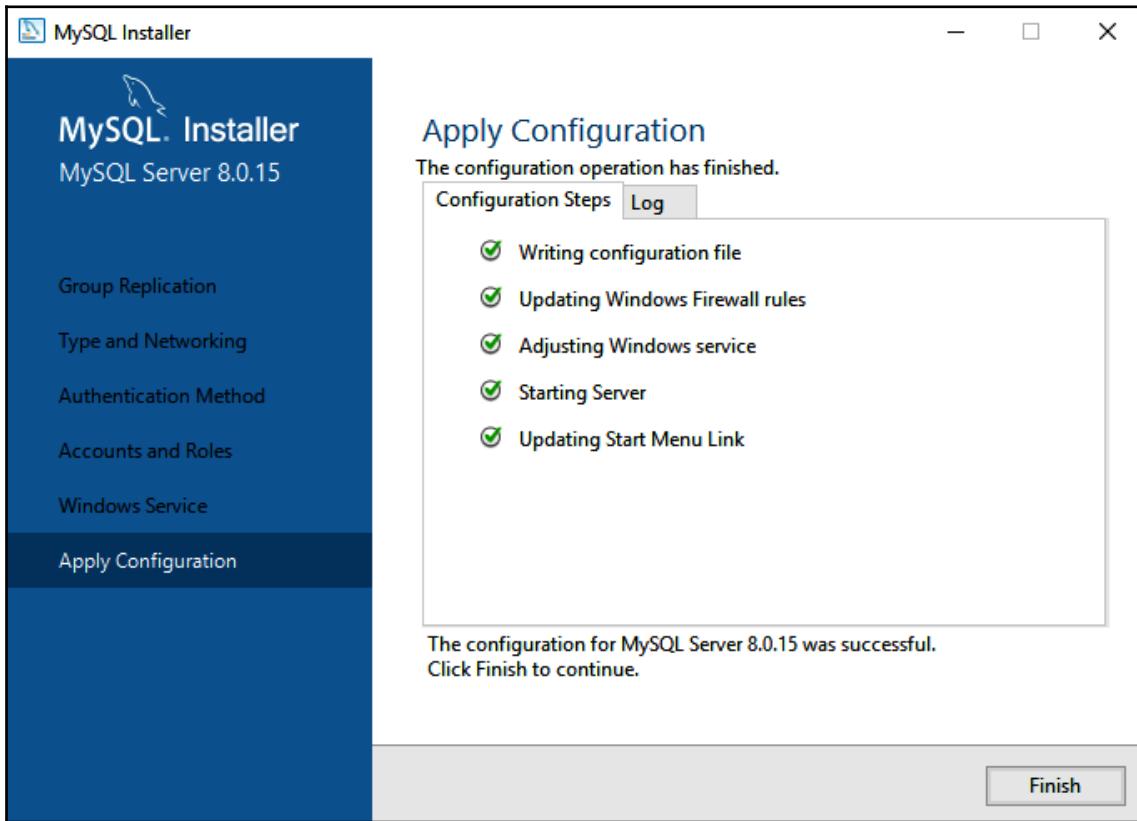
11. In the next dialog, you will be prompted to enter a **Root Account Password**. Enter a strong password comprising lower and upper case letters, numbers, and symbols. After entering the **Current Root Password**, click the **Check** button on its right to confirm whether the entered password is strong or not. If you get a tick on clicking the **Check** button, that means the entered password is perfectly fine. Also, please remember the password as you will be required to enter it in the future:



12. In the next dialog, you will be asked to **Configure MySQL Server as a Windows Service**. If MySQL Server is configured as a Windows service, it will automatically start and stop when Windows starts and stops. By default, a **Windows Service Name** will be displayed based on the MySQL version that you are using. Also, a checkbox, **Start the MySQL Server at System Startup**, will be selected by default. We will stick to the default values. Also, you will be asked whether you want to run the Windows service under a **Standard System Account** or under a **Custom User**. **Standard System Account** will be found auto selected by default. Because we want MySQL Server to run as a Windows service under a **Standard System Account**, we will keep the default options selected and click the **Next** button to move on, as shown in the following screenshot:



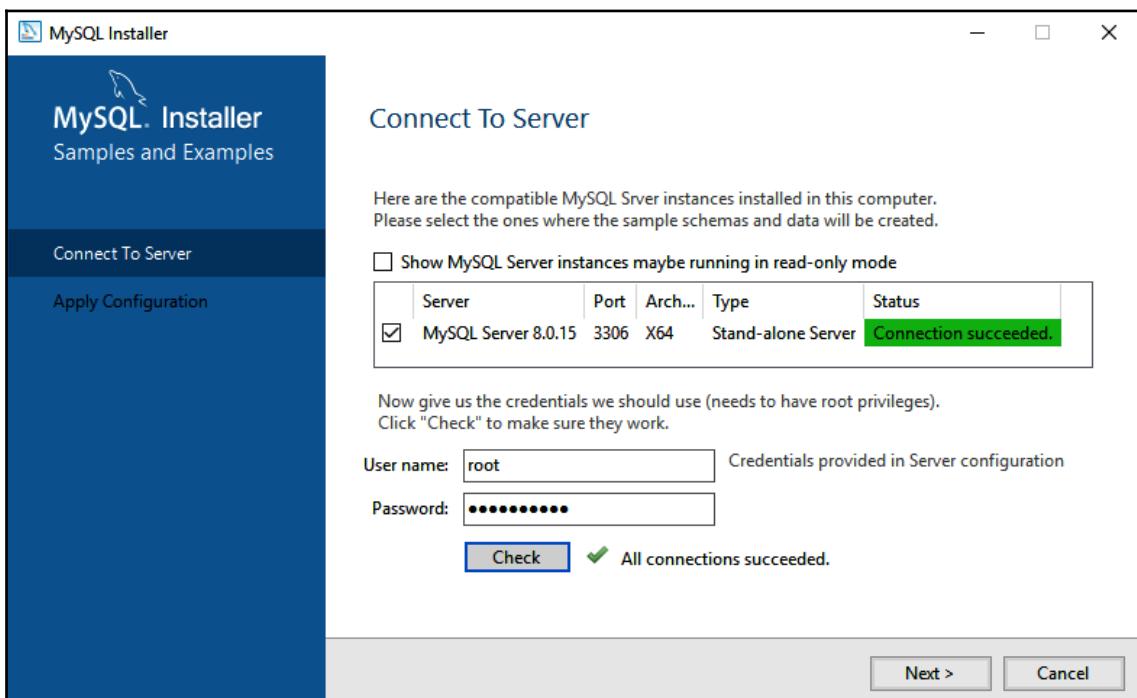
13. The next screen will ask whether to apply the configuration settings that have been chosen by you. Click the **Execute** button at the bottom of the dialog to apply the configuration settings selected by us. On application of the configuration steps, you will get the following dialog box informing us that MySQL Server has been configured successfully. Click the **Finish** button to continue:



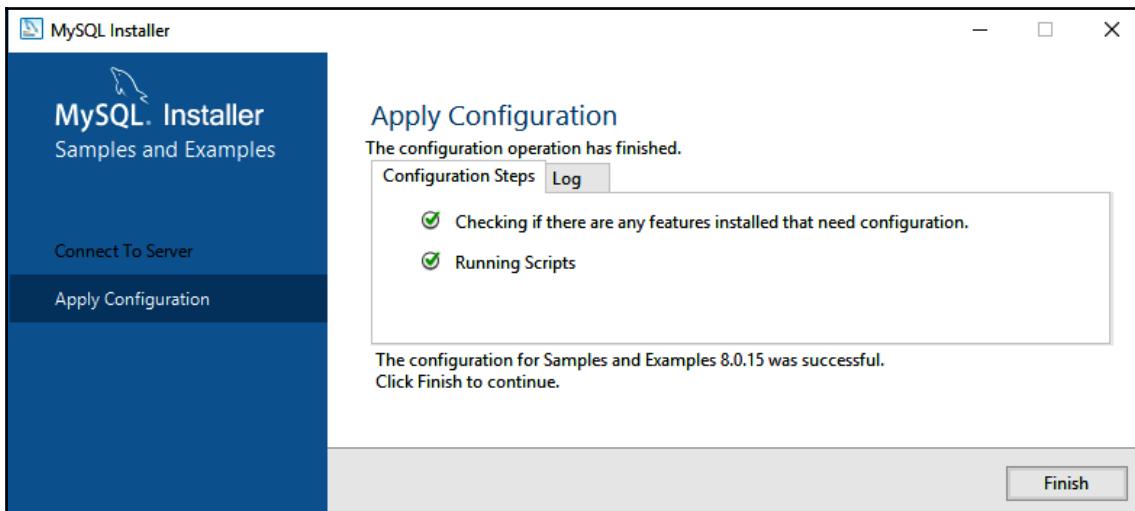
14. You will be informed that now MySQL Router needs to be configured, and you will be asked to select the **Next** button to proceed with the configuration of MySQL Router. Click the **Next** button to display the MySQL Router configuration dialog. The router is used to route the heavy database traffic to the running MySQL Servers so that resources are used efficiently and the database tables are accessed quickly and efficiently.
15. Because we are using a standalone installation of MySQL Server, we do not need to configure MySQL Router, so, click the **Finish** button to close the router configuration dialog.

16. The next dialog will ask whether you want to configure samples and examples for MySQL Server. The sample schema can be directly used in our applications, and examples help us understand MySQL better. So, click the **Next** button to configure samples and examples. You get a dialog box that prompts you to select the server on which the sample schemas and data needs to be created (see the following screenshot). Because we have installed a standalone MySQL server, only one MySQL server instance will be displayed, and that too will be selected by default.

At the bottom of the dialog, you will be prompted to enter the root password to confirm authentication before creating the samples and examples. After entering the root password, click on the **Check** button to confirm authentication. If you get a tick with a message saying **All connections succeeded**, that means you have entered the root password correctly and you can go ahead and click the **Next** button to begin the procedure of creating of samples and examples, as shown in the following screenshot:

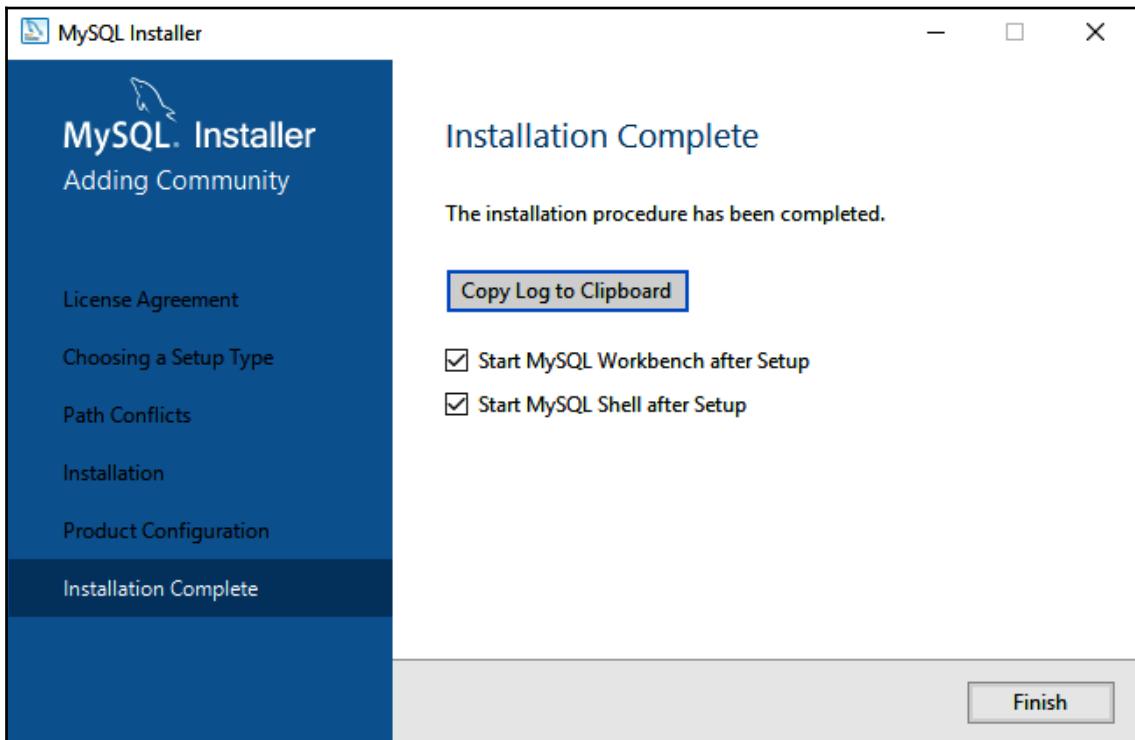


17. The next dialog will prompt you to click the **Execute** button to run the scripts that create sample schemas and related data. Click the **Execute** button to begin the procedure. When the script is executed and the samples and examples are successfully created and configured on the running MySQL server, you get the dialog box, as shown in the following screenshot. Click the **Finish** button to move on:



18. The next dialog box confirms that the task of configuring MySQL Server, MySQL Router, and the samples and examples is complete. So, click the **Next** button to move on.

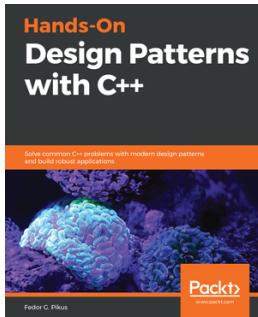
19. The final dialog box confirms that the installation of MySQL Server and its related products is complete (see the following screenshot). The two checkboxes, **Start MySQL Workbench after Setup** and **Start MySQL Shell after Setup**, will be auto selected by default. You can keep the two checkboxes selected if you want to start them once the setup is complete, or you can uncheck either of the checkboxes if you wish to invoke them later when required. Click on the **Finish** button to finish the installation:



Now, you are ready to use MySQL to implement the recipes in Chapter 8, *Using MySQL Database*.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

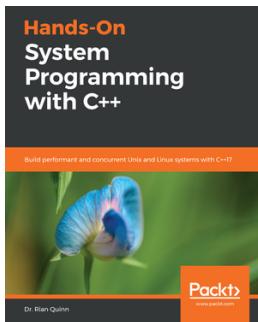


Hands-On Design Patterns with C++

Fedor G. Pikus

ISBN: 9781788832564

- Recognize the most common design patterns used in C++
- Understand how to use C++ generic programming to solve common design problems
- Explore the most powerful C++ idioms, their strengths, and drawbacks
- Rediscover how to use popular C++ idioms with generic programming
- Understand the impact of design patterns on the program's performance



Hands-On System Programming with C++

Dr. Rian Quinn

ISBN: 9781789137880

- Understand the benefits of using C++ for system programming
- Program Linux/Unix systems using C++
- Discover the advantages of Resource Acquisition Is Initialization (RAII)
- Program both console and file input and output
- Uncover the POSIX socket APIs and understand how to program them
- Explore advanced system programming topics, such as C++ allocators
- Use POSIX and C++ threads to program concurrent systems
- Grasp how C++ can be used to create performant system applications

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

actual arguments 70
Armstrong number
 finding 72, 75, 78, 79
array
 common elements, finding 22, 24, 25, 26
 element, inserting 8, 10, 11, 12, 14
 largest value finding, pointer used 113, 114,
 116, 117
 maximum values, returning 79, 81, 83, 84
 minimum values, returning 79, 81, 83, 84
 unique elements, finding 31, 34
 versus sets 27, 29, 30
asynchronous communication 200

B

binary number
 converting, into hexadecimal number 88, 89, 92,
 93, 94, 95
built-in tables
 displaying, in default mysql database 250, 251,
 253

C

C
 used, for deleting data from database 272, 276,
 278
character
 converting, into uppercase 146, 147, 150, 151
 displaying, in string 57, 60
client-server communication
 socket programming, used 211
client-server model
 about 212
accept() 215
bind() 215

connect() 216
htonl() 214
listen() 215
memset() 214
recv() 216
send() 216
socket() 213
struct sockaddr_in structure 212
client
 data, sending 217, 218
common elements
 finding, in array 22, 24, 25, 26
consonants
 counting, in sentence 63, 65
Cygserver
 executing 229

D

data
 deleting, from database using C 272, 276, 278
 reading, from server 219, 220
 sending, to client 217, 218
database
 desired information, searching 258, 261, 262,
 263
 table information, updating 264, 268, 269, 270,
 271
deadlock
 about 188
 avoiding 194, 196, 199
 creating 188, 189, 190, 192, 193
desired information
 searching, in database 258, 261, 262, 263

F

fclose() function 144
fgets() function 144

f
file
 encrypting 166, 167, 169, 170, 171
First in First out (FIFO)
 about 201
 data, reading 210, 211
 data, writing 208, 209
 used, for communicating processes 208
fopen() function 143
fputs() function 144
fseek() function 145
ftell() function 145
functions
 about 70
 fseek() function 145
 ftell() function 145
 rewind() function 146
 used, in file handling 143
 used, in random files 144
 used, in sequential file handling 143

G

greatest common divisor (GCD)
 finding, recursion used 84, 86, 87

H

hexadecimal number
 binary number, converting into 88, 89, 92, 93,
 94, 95

I

information
 updating, in database table 264, 268, 269, 270,
 271
Internetwork Packet Exchange (IPX) 213
interprocess communication (IPC)
 about 200
 ftok() 230
 functions, message queues used 230
 functions, shared memory used 230
 msgget() 232
 msgrcv() 233
 msgsnd() 234
 shmat() 231
 shmctl() 232
 shmdt() 232

shmget() 230

L

Last In First Out (LIFO) 72

M

matrices
 multiplying 15, 16, 17, 19, 21, 22
matrix
 finding, whether sparse 35, 37, 39
 transpose finding, pointer used 129, 131, 133,
 134, 135
memory leak 107
message
 passed, from process to another process using
 message queue 230
 reading, from message queue 237, 238
 writing, into message queue 235
multiple tasks
 performing, with multiple threads 179, 180, 181,
 182

multithreading 174

mutex

 used, to share data between threads 182, 183,
 184, 185, 186, 187

mutual exclusion 174

MySQL database

 information, storing 254, 255, 257, 258

MySQL

 database, creating 247, 248, 249
 functions 244
 mysql_close() 247
 mysql_fetch_row() 246
 mysql_free_result() 247
 mysql_init() 245
 mysql_num_fields() 247
 mysql_query() 246
 mysql_real_connect() 245
 mysql_use_result() 246
 table, creating 249
 tables, creating 247

P

palindrome number

 about 96

finding 95, 99, 100, 101
pointer
 about 103, 104, 107
 used, finding transpose of matrix 129, 131, 133, 134, 135
 used, for accessing structure 136, 137, 140, 141
 used, for finding largest value in array 113, 114, 116, 117
 used, for reversing string 107, 108, 110, 112, 113
processes
 about 173
 communicating, FIFO used 208
 communicating, pipes used 201
 communicating, shared memory used 239
 connecting 201
 creating 201
 fork() 203
 mkfifo() 202
 perror() 203
 pipe() 201
 read() 202
 reading, from pipe 203, 205, 206, 207
 write() 202
 writing, from pipe 203, 205
 writing, into pipe 206, 207
program counter (PC) 173

R

random file
 content, displaying in reverse order 151, 153, 154, 155, 156
 functions, used 144
 handling 142
recursion
 greatest common divisor (GCD), finding 84
 used, for finding greatest common divisor (GCD) 84, 86, 87
Remote Procedure Call (RPC) 200
rewind() function 146

S

semaphore 175
sequential file handling

about 142
fclose() function 144
fgets() function 144
fopen() function 143
fputs() function 144
functions, used 143
sets
 versus array 27, 29, 30
shared memory
 message, reading 241
 message, writing 239
 used, for communicating processes 239
single thread
 used, for performing task 176, 178
singly linked list
 creating 122, 123, 124
 data 118
 iteration 125, 126, 127, 128, 129
 pointer 118
 sorting 117, 118, 120, 122, 124, 125
socket programming
 used, for communicating client-server 211
sorted array
 merging, into single array 42, 45, 47
stack 71, 72
string
 character, displaying 57, 60
 palindrome 48, 50, 52
 repetitive character 52, 55, 57
 reversing, pointer used 107, 108, 110, 112, 113
structure
 accessing, pointer used 136, 137, 140, 141
synchronous communication 200

T

task
 performing, with single thread 176, 178
text file
 reading 146, 147, 150, 151
thread 173
thread identifier 174, 178
thread pool 174
Transmission Control Protocol (TCP) 214

U

UDP socket
 bzero() 221
Cygserver, executing 229
INADDR_ANY 222
message, awaiting from client 223, 225
message, sending from client 223, 225
message, sending to server 226, 228
recvfrom() 223
reply, receiving from server 226, 228
sendto() 222
used, for communicating processes 221
used, for server-client communication 221

unique elements

 finding, in array 31, 34

User Datagram Protocol (UDP) 214

V

vowels

 converting, sentence into uppercase 66, 69

 counting, in file 156, 157, 159, 160

 counting, in sentence 63, 65

W

word

 replacing, in file 161, 162, 164, 165, 166