Ahsanullah University of Science and Technology

Department of Electrical and Electronic Engineering

Laboratory Manual

for

Electrical and Electronic Engineering Sessional Course

Student Name :
Student ID :

Course no : EEE 2110

Course Title : Programming Language Laboratory

For the students of the

Department of Electrical and Electronic Engineering

2nd Year, 1st Semester

Preface

Welcome to the Programming Language Laboratory (EEE 2110). This is the first laboratory on programming language and is required for all electrical and electronic engineering students. This course introduces the fundamental concepts of structured programming.

This laboratory manual is designed to be used in conjunction with the course Programming Language (EEE 2109). The experiments in this laboratory are based on theory and concepts learned in EEE 2109. The laboratory activities typically illustrate concepts from class lectures by using examples, implementations, and problems that are designed to challenge the student. This laboratory is designed in such a way that the environment provides a student with the opportunity to try various options or approaches and to receive immediate feedback.

This laboratory manual will help the students to learn fundamental knowledge of the basic terminologies used in computer programming. The students will also be able to proficiently transform designs of problem solutions into a standard programming language, use an integrated development environment (IDE) to write, compile, and execute programs involving a small number of source files. In addition to that, they will proficiently use fundamental programming elements including variable declaration, data types and simple data structures (arrays, strings, and structures), decision structures, loop structures, functions/methods, input and output for console and text files. In addition, students will apply debugging and testing techniques to locate and resolve errors and to determine the effectiveness of a program.

In each week, the lectures are followed by a three-hour laboratory. Each laboratory activity in this manual is designed to illustrate and explore concepts from that week's lectures. Thus, the pace of the course is very much driven by the laboratory activities. This laboratory has close to thirty students, with two supervising instructors and a laboratory assistant. In the laboratory, every student has a computer, but sometimes students work together in small groups of two so that students help each other and share the responsibility of teaching and learning. We encourage the students to seek assistance from supervising instructors at any point during the laboratory if they need help.

This manual is designed in such a way that students can easily grasp the fundamental of C++. Students will start with the introduction to the computers for setting up the IDE for C++. Then they will gradually explore variables and constants, operators, expressions, control statements, functions, array, pointer, string, and finally the object-oriented programming using C++. In the end, this laboratory reviews the skills developed in the manual by requiring the student to develop several small programs. In this way, students can be prepared for the final project. Such several programs are presented at the end of each laboratory in the post laboratory problems. Each program focuses on a programming skill that students should now be able to perform on their own. The featured skills are prompting for and extracting input, translating mathematical formulas to C++ code, checking the validity of input according to some stated criteria, writing a function that accepts optional parameters, opening a data file, and processing the data.

The instructor has some flexibility in deciding how to use the laboratories. Much of Laboratory 1 should be review material for many students. If desired, Laboratories 1 and 2 can be combined into a single laboratory. In this way, the instructor can decide whether two Laboratories can be combined into one or one Laboratory can be split into two depending on the students' understanding.

Finally, we thank the users of this laboratory manual. We welcome your comments, suggestions, and ideas for improving this material.

Course Outline

1. **Title:** Programming Language Lab
2. **Code:** EEE 2110
3. **Credit hours:** 1.5
4. **Level:** Year 2, Semester 1
5. **Faculty:** Engineering
6. **Department:** Electrical and Electronic Engineering (EEE)
7. **Programme:** B.Sc. in Electrical and Electronic Engineering
8. **Synopsis from the approved curriculum:** Laboratory experiment based on the theory and the concept learnt in EEE 2109.
9. **Type of course:** Core
10. **Prerequisite(s):** no prerequisite course(s)
11. **Themes to be covered:** Intruduction to the programming language, data types, selection statements, iteration, function, arrays, pointers, strings, files, and classes.
12. **Course outcomes:** By the end of this course, students are expected to
    a) identify the given problem statements to solve systematically using a basic logic approach.
    b) implement different functions for input and output, various data types, basic operators, files and functions.
    c) construct a C++ program correctly from the analyzed problems using structured approach.
    d) implement programming techniques to solve problems in the C++ programming language.
13. **Reference materials:**
    a) **Books/Papers:**
        i. *The C++ Programming Language* by Bjarne Stroustrup.
        ii. *C++ How to Program 10th Edition* by Paul Deitel, Harvey Deitel.
        iii. *Teach Yourself C++* by Herbert Schildt.
        iv. *Schaum's Outline of Programming with C++* by John Hubbard .
    b) **Websites (if any):**
        i. https://www.cplusplus.com/doc/tutorial/
        ii. https://www.programiz.com/cpp-programming
        iii. https://www.geeksforgeeks.org/c-plus-plus/
    c) **Handout (if any):**
        i. none
    d) **Others:**
        i. none

Course Assessment

1. **Online and offline task:** A combination of online and offline assignments may be provided for asessing the performance.
2. **Mode of evaluation**: Individual code implementation and evaluation will be conducted in the form of presentation, quiz and/or viva. Late submissions are to be highly discouraged and will be penalized.
3. **Semester project:** An individual project will be assigned to the student before the end of the semster. Students will be provided a time period for completing and submitting the project. Students will be asked to attend a viva on their respective projects.
   a) **Basic Requirements:** Clear understanding of all the lab experiments of C++ programming.
   b) **Presentation Requirement:** Yes.
   c) **Demonstration Requirement:** Depending on the semester project, demonstration will be conducted by respective lab teachers.
   d) **Reporting Requirement:** A brief report along with the codes must be submitted.
4. **Final quiz:** There will be a final quiz based on all the lab experiments done in this lab. Quiz must not include specific project oriented questions. The quiz may consist of MCQ, True/False, Small code snippet (i.e., possible outputs, any existing bug, etc.).
5. **Percentages of assessment methods:**

Method	**Percentage (%)**
Attendance and Lab Performance	10
Lab Test 1	10
Lab Test 2	10
Lab Test 3	10
Lab Test 4	10
Lab Test 5	10
Semester Project	20
Final Quiz	20

Contents

Laboratory 1

Introduction to the C++

1.1. Objective

This introductory laboratory teaches students the basic skills that they will need to complete future laboratories. These skills include gaining access to the machine, having first contact with the IDE, writing your first C++ program, compiling C++ programs, executing C++ programs.

1.2. Familiarize with the local computer system

You may be working with a wide variety of computer equipment. No matter what computer you used earlier, your first contact with the computer will be through its operating system. Operating system itself is a program that coordinates the activities of the machine and performs tasks as directed by the machine's user (or users). You will communicate with the operating system through a keyboard, mouse, and monitor screen. Your instructor will explain how to establish contact with the operating system and will supply you with any customized information (such as user identification numbers or passwords) that you may need.

Through the machine's operating system you will be able to activate numerous auxiliary programs, some of which are designed to assist in program development. These are the programs you will learn to use in this laboratory session. In some cases these programs may be bundled as an integrated package; in other cases they may appear as individual programs whose services you must explicitly request. Your instructor will describe how your particular system operates.

1.3. The Program Preparation Process

The steps required to develop programs using the C++ language will depend on the computer installation being used. However, some features of the process are common to all systems.

As a first step, the programmer uses a program called an editor to type a C++ language version of the program being developed. This editor may be a stand-alone utility program or a part of an integrated software development package, which is also known as integrated development environment (IDE). Once the program has been typed, it is usually saved as a file in mass storage. This version of the program is known as the source program because it is the initial, or source, version of the program. It is this version to which you will return when alterations to the program are required.

A program in its source form cannot be executed by the computer; it must first be translated into the machine's own low-level language. This translation process is performed by a program known as a translator or compiler. Your instructor will explain the details of how to type, save, translate, and finally execute programs using your particular computer system. In the next section we will discuss how we use the IDE for developing C++ programs for this laboratory.

1.4. Code::Blocks IDE

You will use Code::Blocks as your primary IDE for C++ programming in this laboratory. Code::Blocks is an open-source, cross-platform (Windows, Linux, etc.), and free C/C++ IDE. It supports many compilers, such as GNU GCC (MinGW and Cygwin) and MS Visual C++. It supports interactive debugging (via GNU GDB or MS CDB). Code::Blocks is surprisingly versatile, and in my opinion, much better than the Visual Studio suite. The mother site of Code::Blocks is *www.codeblocks.org*.

1.4.1. Install Code::Blocks

For Windows

1. Make sure your computer is connected to the internet.
2. To download the software from the official source go to the URL *http://www.codeblocks.org/downloads*.
3. Then click **Download the binary release** in that web page.
4. Then select your operating platform (e.g., Windows XP / Vista / 7 / 8.x / 10). Download the installer with GCC Compiler, e.g., mingw-setup.exe. This installer includes the MinGW's GNU GCC compiler and GNU GDB debugger.
5. Now install **Code::Blocks** by running the downloaded installer. Accept the default options by clicking a series of **Next** buttons.

For Ubuntu (and Ubuntu based other Linux distributions)

1. Make sure your computer is connected to the internet.
2. Go to the **software center** of Ubuntu. Then search for **Code::Blocks** in the search bar. From the list, select **Code::Blocks**, and click the install button to install it.
3. Alternatively, you can install **Code::Blocks** via terminal with the help of the following command.

```
sudo apt install codeblocks
```

1.4.2. Using Code::Blocks for the first time

1. Launch the Code::Blocks. After opening the Code::Blocks window, launch the Project Wizard through ***File > New > Project...*** to start a new project. A window will come up. You will find many pre-configured templates here for various types of projects, including the option to create custom templates. From that list select the **Console application**, as this is the most common for our laboratory experiment purposes. Then click **Go**.
2. Click next until you get to the Language Selection window. In this window, you will be asked to choose whether you want to use C or C++. Select **C++** and click **Next**.
3. In the next screen, Code::Blocks will prompt you with where you'd like to save the console application. Give the project a name and select a destination folder. Code::Blocks will generate the remaining entries from these two.
4. Finally, the wizard will ask if this project should use the default compiler (normally GCC) and the two default builds: Debug and Release. All of these settings are fine. You don't need to do anything here. Just accept the defaults by hitting **Finish**.
5. At this stage the project will be generated. The main window will turn gray, but that is not a problem. Only the source file needs to be opened. In the **Projects** tab of the **Management** pane on the left expand the folders and double click on the source file ***main.cpp*** to open it in the editor.
6. This ***main.cpp*** file contains the following standard code. This is a simple basic C++ code, which prints ***Hello World!*** in the output console.

```
#include <iostream>

using namespace std;

int main()
{
        cout << "Hello world!" << endl;
        return 0;
}
```

7. At this point, you will have your main.cpp file, which you can modify if you like. For now, it just says ***Hello World!***, so we can run it as is. Hit **F9**, which will first compile it and then run it.
8. If it compiles successfully, the **Build log** at the bottom of the Code::Blocks window will display the following message.

```
        Process terminated with status 0 (0 minute(s), 0 second(s))
        0 error(s), 0 warning(s) (0 minute(s), 0 second(s))
```

   a) The above message indicates your program is free of errors.
   b) At the same time, you will get the following console output:

```
        Hello world!

        Process returned 0 (0x0)   execution time : 0.002 s
        Press ENTER to continue.
```

9. If you have compiler errors, follow the instruction below.
   a) Click the **Build log** at the bottom of the Code::Blocks window.
   b) To read the first error message, scroll to the top of the error list.
   c) If you double click on an error message, a red mark in the editor will identify the statement corresponding to the error message.
   d) Return to the editor and correct the error. And recompile the program by hitting **F9**.

You now have a running program! To write your own C++ code simply edit the ***main.cpp*** and, then hit **F9** to compile it and run it again.

1.5. Introduction to the C++ programs

In this part, you will write your first C++ program along with some other examples.

Example 01

Below is a simple "Hello world!" program. You can consider it as the basic C++ template.

```
1 // my first program in C++
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6         cout << "Hello world!" << endl;
7         cout << "I'm a C++ program." << endl;
8         return 0;
9 }
```

In C++, the ***#include*** is called a preprocessor directive. Preprocessor directives begin with a # sign. They are processed before compilation. The directive ***#include <iostream>*** tells the preprocessor to include the ***iostream*** header file to support input/output operations.

The ***using namespace std;*** statement declares std as the default namespace used in this program. The names ***cout*** and ***endl***, which is used in this program, belong to the ***std namespace***. These two lines shall be present in all our programs.

The ***main()*** function is the entry point of program execution. ***main()*** is required to return an ***int*** (integer).

The output to the display console is done via ***cout*** and the stream insertion (or put-to) operator <<. A special symbol called ***endl*** (END-of-Line) can be used to produce a newline. Whenever an ***endl*** is printed, there is no visible output, but the cursor advances to the beginning (left-margin) of the next line. A string, which is enclosed by a pair of double quotes, will be printed as it is, including the white spaces and punctuation marks within the double quotes.

The last line of the program is ***return 0;*** which terminates ***the main()*** function and returns a value of 0 to the operating system. Typically, return value of 0 signals normal termination; whereas value of non-zero (usually 1) signals abnormal termination. This line is optional. C++ compiler will implicitly insert a ***return 0;*** to the end of the ***main()*** function.

The console output of the above program will be as follows.

```
Hello world!
I'm a C++ program.
```

Example 02

This is another version of the "Hello world!" program. As you can see in this example, you can print as many items as you wish to ***cout***, by chaining the items with the << operator.

```
1 // prints "Hello world!":
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6         cout << "Hel" << "lo wo" << "rld!" << endl;
7         cout << "I'm" << " a C+" << "+ program." << endl;
8         return 0;
9 }
```

The console output will be as follows.

```
Hello world!
I'm a C++ program.
```

Example 03

The following program will help you to understand insertion of numeric literals into the standard output stream of a C++ program. Integers (such as 1, 2, 3) and floating-point numbers (such as 1.1, 2.2) can be printed with the help of ***cout*** too.

```
1 // prints "Hello world!":
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6         cout << "Hello world!" << endl;
```

```
7         cout << "I'm a C++ program." << endl;
8         cout << "I'll show you todays' date." << endl;
9         cout << "Today is January " << 15 << " " << 2021 << "." << endl;
10        return 0;
11 }
```

The console output will be as follows.

```
Hello world!
I'm a C++ program.
I'll show you todays' date.
Today is January 15 2021.
```

Example 04

The following program will help you to understand insertion of escape characters (***\n*** and ***\t***) in a C++ program. Beside the endl, you can also use ***\n***, which denotes a newline character, to advance the cursor to the next line. Similarly, you could use ***\t***, which denote a tab character, to advance the cursor to the next tab position. ***\n*** and ***\t*** are known as escape sequences.

```
1  // usage of \n and \t in a C++ program
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6         cout << "This is an example of using '\\n'." << endl;
7         cout << "\n*\n**\n***\n****\n*****\n" << endl;
8         cout << "This is an example of using '\\t'." << endl;
9         cout << "\t*\t**\t***\t****\t*****\t" << endl;
10        cout << "This is an example of using both '\\n' and '\\t'." << endl;
11        cout << "\n\t*\n\t**\n\t***\n\t****\n\t*****\n\t" << endl;
12        return 0;
13 }
```

It is strongly recommend that you use ***endl*** to print a newline, instead of ***\n***. This is because line delimiter is system dependent: Windows use ***\r\n***; UNIX/Linux/Mac use ***\n***. The ***endl*** produces system-specific newline. Furthermore, ***endl*** guarantees that the output is flushed; while ***\n*** does not.

Example 05

Write and run the folowing program to print ***W*** as a block letter.

```
1  // prints "W" as a block letter
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6         cout << "#                 #" << endl;
7         cout << " #               #" << endl;
8         cout << "  #             #" << endl;
9         cout << "   #     #     #" << endl;
10        cout << "    #   # #   #" << endl;
11        cout << "     # #   # #" << endl;
12        cout << "      #     #" << endl;
13 }
```

Example 06

Write and run the following program to print your name, student ID, year, semester, section and department name.

```
// prints personal information
#include <iostream>
using namespace std;
int main()
{
        cout << "Name: your_name" << endl;
        cout << "ID: your_student_ID" << endl;
        cout << "Year: your_current_year" << endl;
        cout << "Semester: your_current_semester" << endl;
        cout << "Section: your_section" << endl;
        cout << "Department: your_department" << endl;
        return 0;
}
```

Post-laboratory Problems

1. What is/are the error(s) in the following C++ program? Write down the exact output of the following program after correcting the error.

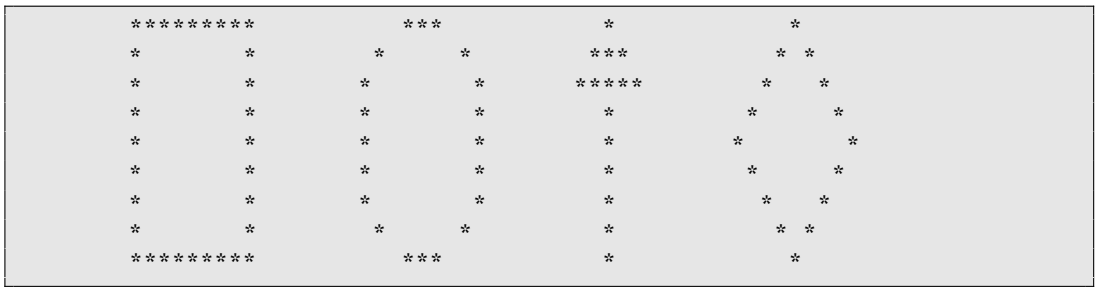
```
#include <iostream>
int main()
        // this program prints
        "Hello, World!":
        cout << "Hello, World!\n"
        return 0;
}
```

2. What is/are the error(s) in the following C++ program? Write down the exact output of the following program after correcting the error.

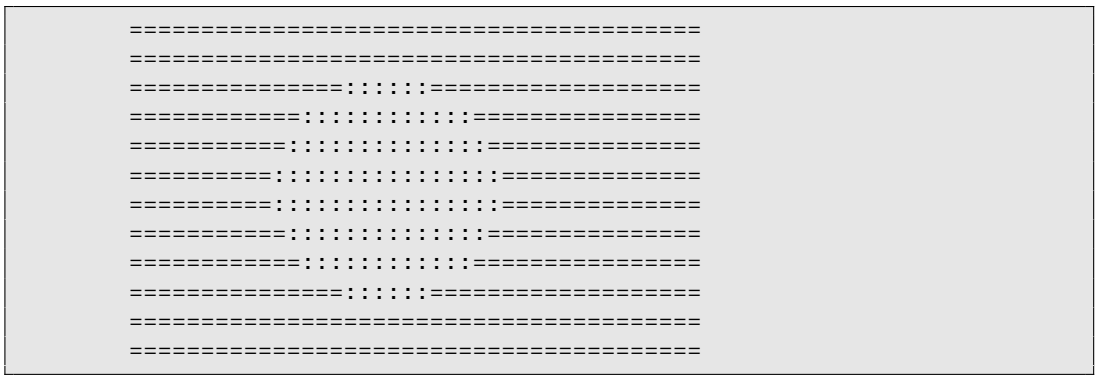
```
include <iostream.h>
int main()
{
        cout << "Patri";
        cout << "ot " << "G";
        cout << "a" << "m" << "e" << "\n";
        cout << "\t Written" << " ";
        cout << "B" << "y" << "\n";
        cout << "\t\t Tom" << " " << "C" ;
        cout << "lancy\n\n\n";
        return 0;
}
```

3. Write a C++ program that displays six of your favourite colours name in -
   a) 1 row (use comma and space to separate them)
   b) 6 rows (use **n** to separate them in multiple rows and use **t** to separate them in each row)
   c) 3 rows (use **n** to separate them in multiple rows and use **t** to separate them in each row)

4. Write and run a C++ program that prints a box, an oval, an arrow and a diamond as below.



5. Write and run a C++ program that prints the above patterns using only one ***cout*** statement.
6. Write and run a C++ program that prints the first letter of your last name as a block letter in a 7×7 grids of asterisks (the star symbol in your keyboard).
7. Write and run a C++ program that prints the flag of Bangladesh as below.



Laboratory 2

Variables, operators and type conversions

2.1. Objective

In this laboratory students are guided through the process of decomposing a problem into steps and then translating those steps into working C++ code. The laboratory also exercises basic C++ programming skills that have been introduced in the lectures. Students practice input and output operations using the ***iostream*** objects ***cin*** and ***cout*** and also translating mathematical formulas into C++ assignment statements. In addition, this laboratory introduce students to the variables that are used in C++ with several simple arithmetic operations as well as mathematic functions.

2.2. The variables

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Consider variables as the *mailboxes,* in which data necessary for program execution can be stored. You can store many types of data in these *mailboxes*. Integers, Floating-point numbers, and Characters are a few of these data types. Whole numbers, including 0 and negative whole numbers, are called ***integers***. Decimal numbers, including negative decimal numbers and all integers, are called ***floating-point*** numbers. A ***character*** type is an integral type whose variables represent characters like the letter ***A*** or the digit ***8***. Standard C++ has 14 different fundamental data types which are shown in the Figure 2.1 .

Example 01

The following example of a C++ program is simply going to set a variable and print it out.

```
1  // prints "n = 44":
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6          int n;
7          n = 44;
8          cout << "n = " << n << endl;
9  }
```

In line 6 of the above code, you see a new statement that looks like ***int n;***. This declares a variable named ***n*** of type ***int***. In C++, ***int*** means integer. As we discussed before, an integer is a data type which can only contain non-floating point values. In other words, only whole numbers.

Then in the line 7 you see ***n = 44;***. This is commonly known as initialization of variable. This line simply assigns the value of ***44*** to ***n***. The ***mailbox*** of the variable now has a letter that says ***44***. The = operator is

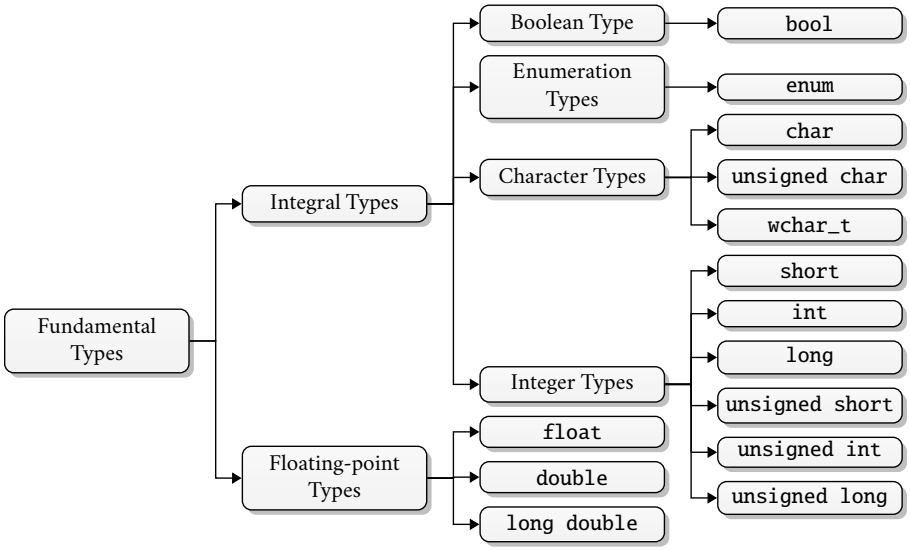


Figure 2.1. Fundamental data types

called the assignment operator. It assigns the value of whatever is on the right to whatever is on the left. However, this newly assigned data can later be used in our program for output in the next line.

The next line indicates - ***cout << "n = " << n << endl;***. This line tells the computer to output the value stored in ***n*** (which is ***44*** by the way) with a text ***n =,*** and then an ***endl*** (end-of-line character). It is done the same way with most variables, simply append an extra << operator for each value you wish to attach. The final output of this program will be as below.

```
        n = 44

        Process returned 0 (0x0)   execution time : 0.002 s
        Press ENTER to continue.
```

2.3. The input operator

In C++, input is almost as simple as output. The input operator >> (also known as get operator or extraction operator) works like the output operator << . The function for input is ***cin***. It is used to accept the input from the standard input device, such as, keyboard. Following example program will clarify the operation of ***cin***.

Example 02

The following example of a C++ program takes different data types as input from the user and print those in the output.

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5         int m, n;
```

```
        cout << "Enter two integers (use SPACE to separate): ";
        cin >> m >> n;
        cout << "m = " << m << ", n = " << n << endl;
        double x, y, z;
        cout<< "Enter three decimal numbers (use SPACE to separate): ";
        cin >> x >> y >> z;
        cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
        char c1, c2, c3, c4;
        cout << "Enter four characters (use SPACE to separate): ";
        cin >> c1 >> c2 >> c3 >> c4;
        cout << "c1 = " << c1 << ", c2 = " << c2 << ", c3 = " << c3;
        cout << ", c4 = " << c4 << endl;
}
```

2.4. Arithmetic operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example, in $a + b$; the $+$ operator is used to add two variables a and b . Similarly there are various other arithmetic operators in C++. The following two examples will show the exact use of different types of arithmetic operators.

Example 03

```
/* Integer Arithmetic
Testing the operators +, -, *, /, and % */
#include <iostream>
using namespace std;
int main()
{
        int m=54, n=20;
        cout << "m = " << m << " and n = " << n << endl;
        cout << "m+n = " << m+n << endl;
        cout << "m-n = " << m-n << endl;
        cout << "m*n = " << m*n << endl;
        cout << "m/n = " << m/n << endl;
        cout << "m%n = " << m%n << endl;
}
```

Example 04

```
/* Floating-Point Arithmetic
Testing the operators +, -, *, /, and % */
#include <iostream>
using namespace std;
int main()
{
        float x=54, y=20;
        cout << "x = " << x << " and y = " << y << endl;
        cout << "x+y = " << x+y << endl;
        cout << "x-y = " << x-y << endl;
```

```
11          cout << "x*y = " << x*y << endl;
12          cout << "x/y = " << x/y << endl;
13  }
```

2.5. Mathematical functions

C++ provides a large number of mathematical functions that can be used directly in the program. These functions are widely used in the mathematical calculations. These functions are included in the ***cmath*** library. The following two examples demonstrate a few familiar mathematical functions.

Example 05

```
1   // Some important trigonometric functions
2   #include <iostream>
3   #include <cmath>
4   using namespace std;
5   int main ()
6   {
7           double PI = 3.14159;
8           cout << "cos(60) = " << cos (60.0*PI/180.0) << endl;
9           cout << "sin(60) = " << sin (60.0*PI/180.0) << endl;
10          cout << "tan(45) = " << tan (45.0*PI/180.0) << endl;
11  }
```

Example 06

```
1   // abs computes the absolute value of a given number.
2   // sqrt used to find the square root of the given number.
3   // pow returns the result by raising base to the given exponent.
4   // exp returns the exponential raised to the given argument.
5   // log returns the natural logarithm of the argument.
6   // log10 returns the base-10 logarithm of the argument.
7
8   #include <iostream>
9   #include <cmath>
10  using namespace std;
11  int main ()
12  {
13          double param, result;
14          param = 10.57;
15          result = abs(param);
16          cout << "Absolute value of 10.57 is " << result << endl;
17          param = -10.57;
18          result = abs(param);
19          cout << "Absolute value of -10.57  is " << result << endl;
20          param = 25;
21          result = sqrt (param);
22          cout<< "Square root of 25 is " << result << endl;
23          cout<< "Value of 2^4 is " << pow(2, 4) << endl;
24          cout<< "Value of e^5 is " << exp(5) << endl;
```

```
        cout<< "Value of ln(10) is " << log(10) << endl;
        cout<< "Value of log(10) is " << log10(10) << endl;
}
```

2.6. Examples on arithmetic operators and mathematical functions

Example 07

Write and run a C++ program that accepts the radius of a circle from the user and compute the area and circumference.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
        float radius, area, circum, PI = 3.14159;
        cout <<" Input the radius(1/2 of diameter) of a circle : ";
        cin >> radius;
        circum = 2*PI*radius;
        area = PI*pow(radius,2);
        cout << " The area of the circle is : " << area << endl;
        cout << " The circumference of the circle is : " << circum << endl;
        cout << endl;
}
```

The console output will be as follows.

```
Input the radius(1/2 of diameter) of a circle : 7.8
The area of the circle is : 191.134
The circumference of the circle is : 49.0088
```

Example 08

Write a program in C++ to enter length in centimeter and convert it into meter and kilometer.

```
#include<iostream>
using namespace std;
int main()
{
        float km, met,cent;
        cout << "Input the distance in centimeter : ";
        cin >> cent;
        met = (cent/100);
        km = (cent/100000);
        cout << "The distance in meter is: " << met << endl;
        cout << "The distance in kilometer is: " << km << endl;
        cout << endl;
}
```

The console output will be as follows.

```
Input the distance in centimeter : 2000
The distance in meter is: 20
The distance in kilometer is: 0.02
```

Example 09

Simple interest is a quick and easy method of calculating the interest charge on a loan. It is determined by multiplying interest rate by the principal by the number of years that elapse between payments. Write a program in C++ to enter principle, time, rate of interest and calculate simple interest.

```
#include<iostream>
using namespace std;
int main()
{
        int p, r, t, i;
        cout << "Input the Principle: ";
        cin >> p;
        cout << "Input the Rate of Interest: ";
        cin >> r;
        cout << "Input the Time: ";
        cin >> t;
        i=(p*r*t)/100;
        cout << "The Simple interest for the amount " << p << " for ";
        cout << t << " years @ " << r << " % is: " << i << endl;
}
```

The console output will be as follows.

```
Input the Principle: 1000
Input the Rate of Interest: 4
Input the Time: 4
The Simple interest for the amount 1000 for 4 years @ 4 % is: 160
```

Example 10

At its closest point to Earth, Mars is approximately 34,000,000 miles away. Assuming there is someone on Mars that you want to talk with, what is the delay between the time a radio signal leaves Earth and the time it arrives on Mars? Radio signals travel at the speed of light, approximately 186,000 miles per second. Thus, to compute the delay, you will need to divide the distance by the speed of light. Now, write and run a C++ program to display the delay in terms of seconds and also in minutes.

```
#include <iostream>
using namespace std;
int main()
{
        double distance, lightspeed, delay, delay_in_min;
        distance = 34000000.0; // 34,000,000 miles
        lightspeed = 186000.0; // 186,000 per second
        delay = distance / lightspeed;
        cout << "Time delay when talking to Mars: " << delay << " seconds.\n";
        delay_in_min = delay / 60.0;
        cout << "This is " << delay_in_min << " minutes.";
}
```

The console output will be as follows.

```
Time delay when talking to Mars: 182.796 seconds.
This is 3.04659 minutes.
```

2.7. Increment and decrement

The values of integral objects can be incremented and decremented with the $++$ and $--$ operators, respectively. Each of these operators has two versions: a ***pre*** version and a ***post*** version. The ***pre*** version performs the operation (either adding 1 or subtracting 1) on the object before the resulting value is used in its surrounding context. On the other hand, the ***post*** version performs the operation after the object's current value has been used. Both the increment and decrement operators can either precede *(prefix)* or follow *(postfix* or *suffix)* the operand. For example, $\boldsymbol{x = x + 1;}$ can be written either as $\boldsymbol{++x;}$ //prefix form or as $\boldsymbol{x++;}$ //postfix form

In the above lines, there is no difference whether the increment is applied as a prefix or a postfix. However, when an increment or decrement is used as part of a larger expression, there is an important difference. When an increment or decrement operator precedes its operand, C++ will perform the operation prior to obtaining the operand value for using by the rest of the expression. If the operator follows its operand, then C++ will obtain the operand value before incrementing or decrementing it. Consider the following lines of code,

$\boldsymbol{x = 10; y = ++x;}$

In this case, y will be set to 11. However, if the code is written as below,

$\boldsymbol{x = 10; y = x++;}$

then y will be set to 10. In both cases, x is still set to 11. The difference is – *when it happens*. There are significant advantages in being able to control when the increment or decrement operation takes place.

Example 11

```
1  // example of prefix and postfix
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6          int m, n;
7          m = 44;
8          n = ++m;
9          cout << "Prefix increment: \t m = " << m << " and n = " << n << endl;
10         m = 44;
11         n = m++;
12         cout << "Postfix increment: \t m = " << m << " and n = " << n << endl;
13         m = 44;
14         n = --m;
15         cout << "Prefix decrement: \t m = " << m << " and n = " << n << endl;
16         m = 44;
17         n = m--;
18         cout << "Postfix decrement: \t m = " << m << " and n = " << n << endl;
19 }
```

The console output will be as follows.

```
Prefix increment:       m = 45 and n = 45
Postfix increment:      m = 45 and n = 44
Prefix decrement:       m = 43 and n = 43
Postfix decrement:      m = 43 and n = 44
```

2.8. Composite assignment operators

The standard assignment operator in C++ is the equals sign (i.e., $=$). In addition to this operator, C++ also includes the following composite assignment operators,

$+=, -=, *=, /=, \%=$

When applied to a variable on the left, each applies the indicated arithmetic operation to it using the value of the expression on the right .

Example 12

```
1  // example of omposite assignment operators
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6          int n = 22;
7          cout << "The value of n = " << n << endl;
8          n += 9;
9          cout << "After n += 9, n = " << n << endl;
10         n -= 5;
11         cout << "After n -= 5, n = " << n << endl;
12         n *= 2;
13         cout << "After n *= 2, n = " << n << endl;
14         n /= 3;
15         cout << "After n /= 3, n = " << n << endl;
16         n %= 7;
17         cout << "After n %= 7, n = " << n << endl;
18 }
```

The console output will be as follows.

```
The value of n = 22
After n += 9, n = 31
After n -= 5, n = 26
After n *= 2, n = 52
After n /= 3, n = 17
After n %= 7, n = 3
```

2.9. Type conversions

C++ allows us to convert data of one type to that of another. This is known as *type conversion*. There are two types of type conversion in C++.

1. Implicit conversion
2. Explicit conversion (also known as Type casting)

2.9.1. Implicit conversion

The type conversion that is automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion. Let look at two examples of implicit type conversion.

Example 13

```
// conversion from int to double
#include <iostream>
using namespace std;
int main()
{
        int num_int = 9;
        double num_double;
        num_double = num_int;
        cout << "num_int = " << num_int << endl;
        cout << "num_double = " << num_double << endl;
        return 0;
}
```

The console output will be as follows.

```
num_int = 9
num_double = 9
```

In the program, we have assigned an ***int*** data to a ***double*** variable. Here, the ***int*** value is automatically converted to ***double*** by the compiler before it is assigned to the ***num_double*** variable. This is an example of implicit type conversion.

Example 14

```
// conversion from double to int
#include <iostream>
using namespace std;
int main()
{
        int num_int;
        double num_double = 9.99;
        num_int = num_double;
        cout << "num_int = " << num_int << endl;
        cout << "num_double = " << num_double << endl;
        return 0;
}
```

The console output will be as follows.

```
num_int = 9
num_double = 9.99
```

In the program, we have assigned a ***double*** data to an ***int*** variable. Here, the ***double*** value is automatically converted to ***int*** by the compiler before it is assigned to the ***num_int*** variable. Since ***int*** cannot have a decimal part, the digits after the decimal point are truncated in the above example.

As we have seen from the above example, conversion from one data type to another is prone to data loss. This happens when data of a larger type is converted to data of a smaller type. This scenario is shown in Figure 2.2.

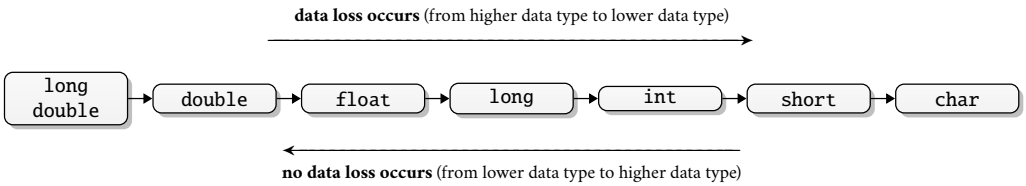


Figure 2.2. Possible data loss during type conversion

2.9.2. Explicit conversion

When the user manually changes data from one type to another, this is known as explicit conversion. In general, if T is a data type and v is a value of different data type, then either of the expressions $(T)v$ or $T(v)$ converts v to type T. This type of conversion is also known as type casting. Following is an example of the type casting.

Example 15

```
1  // type casting
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6          double num_double = 3.56;
7          cout << "num_double = " << num_double << endl;
8          int num_int1 = (int)num_double;
9          cout << "num_int1   = " << num_int1 << endl;
10         int num_int2 = int(num_double);
11         cout << "num_int2   = " << num_int2 << endl;
12         return 0;
13 }
```

The console output will be as follows.

```
num_double = 3.56
num_int1   = 3
num_int2   = 3
```

Example 16

This example promotes a ***char*** to a ***short*** to an ***int*** to a ***float*** to a ***double***:

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5          char c = 'A';
6          cout << "char c = " << c << endl;
```

```
        short k = c;
        cout << "short k = " << k << endl;
        int m = k;
        cout << "int m = " << m << endl;
        long n = m;
        cout << "long n = " << n << endl;
        float x = n;
        cout << "float x = " << x << endl;
        double y = x;
        cout << "double y = " << y << endl;
}
```

2.10. Numeric overflow

On most computers, the ***long int*** type allows 4,294,967,296 different values. That is a lot of value, but it is still finite. Computers are finite, so the range of any type must also be finite. But in mathematics, there are infinitely many integers. Consequently, computers are manifestly prone to error when their numeric values become too large. That kind of error is called numeric overflow.

Example 17

```
// numeric overflow
#include <iostream>
using namespace std;
int main()
{
        int n = 1000;
        cout << "n = " << n << endl;
        n *= 1000;              // multiplies n by 1000
        cout << "n = " << n << endl;
        n *= 1000;              // multiplies n by 1000
        cout << "n = " << n << endl;
        n *= 1000;              // multiplies n by 1000
        cout << "n = " << n << endl;
}
```

The console output will be as follows.

```
n = 1000
n = 1000000
n = 1000000000
n = -727379968
```

This shows that the computer that ran this program cannot multiply 1,000,000,000 by 1000 correctly. Integer overflow *wraps around* to negative integers. Floating-point overflow *sinks* into the abstract notion of infinity.

Post-laboratory Problems

1. Write and run a C++ program for a calculator that takes two decimal numbers from a user. It performs the addition, subtraction, division and multiplication. It shows the results on the output screen.

2. Write and run a C++ program that asks the user to type the width and the length of a rectangle to calculate the area and the perimeter of that rectangle. Then it shows the result results on the output screen.
3. Write and run a C++ program that converts centimeters to inches.
   Hints: *1.0 cm = 0.393 inch.*
4. Write and run a C++ program that converts any temperature from Celsius scale to Fahrenheit scale.
   Hints: *relation between Celsius and Fahrenheit is* $C = \frac{5}{9} \times (F - 32)$*; where F is Fahrenheit and C is Celsius.*
5. Write and run a C++ program to calculate the hypotenuse a right triangle from the given length of two legs.
6. Write and run a C++ program to calculate the distance between two co-ordinates (x_1, y_1) and (x_2, y_2) of a two dimensional graph.
   Hints: *The distance between two points* (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
7. Write and run a C++ program to find the roots of a quadratic equation. The standard form of a quadratic equation is $ax^2 + bx + c = 0$, where a, b, and c are real numbers and $a \neq 0$.
   Hints: `cout << "The equation is: "<< a << "*x*x+"<< b << "*x+"<< c << "=0"<< endl;`

Laboratory 3

Selection statements

3.1. Objective

The objective of this laboratory is to ensure that students have a good understanding of the operation and use of the selection statements, such as ***if, if-else, switch, conditional operators*** and ***compound conditions***. The programs in the previous two laboratories have sequential execution, i.e., each statement in the program executes once, and they are executed in the same order that they are listed. This laboratory shows the students how to use selection statements for more flexible programs.

3.2. The *if* statement

In C++ programming, the ***if*** statement allows the programmer to change the logical order of a program, i.e., it makes the order in which the statements are executed differ from the order in which they are listed in the program. In other words, the ***if*** statement is used to run a block code only when a certain condition is met. For example, assigning grades based on marks obtained by a student.
— if the percentage is above 80, assign grade A+
— if the percentage is above 75, assign grade A
— if the percentage is above 70, assign grade A-
The ***if*** statement uses a Boolean expression to determine whether to execute a statement or to skip it. The syntax of the ***if*** statement is as follows.

```
if (condition)
{
        statement; //body of if statement
}
```

The code inside the curly brackets {} is the body of the ***if*** statement.The ***if*** statement evaluates the condition inside the parentheses (). If the condition evaluates to ***true***, the code inside the body of ***if*** is executed. If the condition evaluates to ***false***, the code inside the body of ***if*** is skipped.

Example 01

The following example of a C++ program prints positive number entered by the user. If the user enters a negative number, it is skipped.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5          int number;
6          cout << "Enter an integer: ";
7          cin >> number;
8          if (number > 0)  // checks if the number is positive
```

```
9         {
10                cout << "You entered a positive integer: " << number << endl;
11        }
12        cout << "This statement is always executed.";
13 }
```

If the user enters 5, the console output will be as follows.

```
Enter an integer: 5
You entered a positive number: 5
This statement is always executed.
```

When the user enters 5, the condition $number > 0$ is evaluated to ***true*** and the statement inside the body of ***if*** is executed. Now if the user enters -5, the console output will be as follows.

```
Enter a number: -5
This statement is always executed.
```

When the user enters -5, the condition $number > 0$ is evaluated to ***false*** and the statement inside the body of ***if*** is executed.

Example 02

The following example of a C++ program finds the minimum of three integers.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5         int n1, n2, n3;
6         cout << "Enter three integers: ";
7         cin >> n1 >> n2 >> n3;
8         int min = n1;  // now min <= n1
9         if (n2 < min)
10        {
11                min = n2;  // now min <= n1 and min <= n2
12        }
13        if (n3 < min)
14        {
15                min = n3;  // now min <= n1, min <= n2, and min <= n3
16        }
17        cout << "Their minimum is " << min << endl;
18 }
```

The console output will be as follows.

```
Enter three integers: 44 11 99
Their minimum is 11
```

Example 03

The following example of a C++ program determines whether the user's guessed number matches the magic number.

```
1  #include <iostream>
2  #include <cstdlib>
```

```
3  using namespace std;
4  int main()
5  {
6          int magic, guess;
7          magic = rand();
8          cout << "Enter your guess: ";
9          cin >> guess;
10         if (guess == magic)
11         {
12                 cout << "Right Guess!" << endl;
13         }
14 }
```

This program uses the ***if*** statement to determine whether the user's guess matches the magic number. If it does, the message is printed on the screen.

3.3. The *if...else* statement

The ***if...else*** statement uses a Boolean expression to determine which one of the two statements to execute. The syntax of the ***if...else*** statement is as follows

```
if (condition)
{
        // block of code if the condition is true
}
else
{
        // block of code if the condition is false
}
```

The ***if...else*** statement evaluates the condition inside the parenthesis. If the condition evaluates ***true***,
— the code inside the body of ***if*** is executed
— the code inside the body of ***else*** is skipped from execution
If the condition evaluates ***false***,
— the code inside the body of ***else*** is executed
— the code inside the body of ***if*** is skipped from execution

Example 04

The following example of a C++ program checks whether an integer is positive or negative. This program considers 0 as a positive number.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5          int number;
6          cout << "Enter an integer: ";
7          cin >> number;
8          if (number >= 0)
9          {
10                 cout << "You entered a positive integer: " << number << endl;
11         }
```

```
12          else
13          {
14                  cout << "You entered a negative integer: " << number << endl;
15          }
16          cout << "This statement is always executed.";
17          return 0;
18  }
```

If the user enters 4, the console output will be as follows.

```
Enter an integer: 4
You entered a positive integer: 4.
This statement is always executed.
```

In the above program, we have the condition $number >= 0$. If we enter the number greater or equal to 0, then the condition evaluates true. Here, we enter 4. So, the condition is ***true***. Hence, the statement inside the body of ***if*** is executed. Now, if the user enters -4, the console output will be as follows.

```
Enter an integer: -4
You entered a negative integer: -4.
This statement is always executed.
```

Here, we enter -4. So, the condition is ***false***. Hence, the statement inside the body of ***else*** is executed.

Example 05

The following example of a C++ program takes the magic number program further. This version uses the ***else*** to print a message when the wrong number is picked.

```
1   #include <iostream>
2   #include <cstdlib>
3   using namespace std;
4   int main()
5   {
6           int magic, guess;
7           magic = rand();
8           cout << "Enter your guess: ";
9           cin >> guess;
10          if (guess == magic)
11          {
12                  cout << "!Right Guess!" << endl;
13          }
14          else
15          {
16                  cout << "Sorry! You guessed wrong." << endl;
17          }
18  }
```

3.4. Nested *if...else* statement

An ***if*** statement uses Boolean expression to determine whether to execute or skip a statement. An ***if...else*** statement uses a Boolean expression to determine which one of the two statements to execute. The statements to be executed or skipped could be simple statements or compound statements. They also can be an ***if...else*** statement. An ***if...else*** statement within another ***if...else*** statement is called a *nested* ***if...else*** *statement*.

Example 06

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Enter an integer: ";
    cin >> num;
    if (num != 0)
    {
        if ((num % 2) == 0)
        {
            cout << "The number is even." << endl;
        }
        else
        {
            cout << "The number is odd." << endl;
        }
    }
    else
        {
        cout << "The number is 0 and it is neither even nor odd." << endl;
        }
    cout << "This statement is always executed." << endl;
}
```

If the user enters 34, the console output will be as follows.

```
Enter an integer: 34
The number is even.
This statement is always executed.
```

If the user enters 35, the console output will be as follows.

```
Enter an integer: 35
The number is odd.
This statement is always executed.
```

If the user enters 0, the console output will be as follows.

```
Enter an integer: 0
The number is 0 and it is neither even nor odd.
This statement is always executed.
```

Example 07

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int magic, guess;
    magic = rand();
```

```
    cout << "Enter your guess: ";
    cin >> guess;
    if (guess == magic)
    {
        cout << "Right Guess!" << endl;
        cout << magic << " is the number." << endl;
    }
    else
    {
        cout << "Sorry! You guessed wrong." << endl;
        if (guess > magic)
        {
            cout << "Your guess is too high" << endl;
        }
        else
        {
            cout << "Your guess is too low." << endl;
        }
    }
    return 0;
}
```

Example 08

The following example of a C++ program finds the minimum of three integers.

```
#include <iostream>
using namespace std;
int main()
{
int n1, n2, n3;
cout << "Enter three integers: ";
cin >> n1 >> n2 >> n3;
if (n1 < n2)
{
    if (n1 < n3)
    {
        cout << "The minimum is " << n1 << endl;
    }
    else
    {
        cout << "The minimum is " << n2 << endl;
    }
}
else
{
    if (n2 < n3)
    {
        cout << "The minimum is " << n2 << endl;
    }
    else
    {
        cout << "The minimum is " << n3 << endl;
    }
```

```
}
return 0;
}
```

3.5. *if...else if...else* statement

The ***if...else*** statement is used to execute a block of code among two alternatives. However, if we need to make a choice between more than two alternatives, we use the ***if...else if...else*** statement. The syntax of the ***if...else if...else*** statement is as follows.

```
if (condition1)
{
         // code block 1
}
else if (condition2)
{
        // code block 2
}
else
{
        // code block 3
}
```

Here,

— If ***condition 1*** evaluates to ***true***, the ***code block 1*** is executed.
— If ***condition 1*** evaluates to ***false***, then ***condition 2*** is evaluated.
— If ***condition 2*** is ***true***, the ***code block 2*** is executed.
— If ***condition 2*** is false, the ***code block 3*** is executed.

There can be more than one ***else if*** statement but only one ***if*** and ***else*** statements.

Example 09

The following example of a C++ program checks whether an integer is positive, negative or zero.

```
#include <iostream>
using namespace std;
int main()
{
    int number;
    cout << "Enter an integer: ";
    cin >> number;
    if (number > 0)
    {
        cout << "You entered a positive integer: " << number << endl;
    }
    else if (number < 0)
    {
        cout << "You entered a negative integer: " << number << endl;
    }
    else
    {
        cout << "You entered 0." << endl;
```

```
19      }
20      cout << "This statement is always executed.";
21      return 0;
22  }
```

If the user enters 1, the console output will be as follows.

```
Enter an integer: 1
You entered a positive integer: 1.
This statement is always executed.
```

If the user enters -2, the console output will be as follows.

```
Enter an integer: -2
You entered a negative integer: -2.
This statement is always executed.
```

If the user enters 0, the console output will be as follows.

```
Enter an integer: 0
You entered 0.
This statement is always executed.
```

In this program, we take a number from the user. We then use the ***if...else if...else*** ladder to check whether the number is positive, negative, or zero. If the number is greater than 0, the code inside the ***if*** block is executed. If the number is less than 0, the code inside the ***else if*** block is executed. Otherwise, the code inside the ***else*** block is executed.

3.6. The *switch* statement

The ***switch*** statement allows programmers to execute a block of code among many alternatives. The syntax of the ***switch*** statement is as follows.

```
switch (expression)
{
        case constant1:
                // code to be executed if
                // expression is equal to constant1
                break;
        case constant2:
                // code to be executed if
                // expression is equal to constant2
                break;
        .
        .
        .
        default:
                // code to be executed if
                // expression doesn't match any constant
}
```

Here, the ***expression*** is evaluated once and compared with the values of each ***case*** label.

— If there is a match, the corresponding code after the matching label is executed. For example, if the value of the variable is equal to ***constant2***, the code after ***case constant2:*** is executed until the ***break*** statement is encountered.
— If there is no match, the code after ***default:*** is executed.

You can do the same thing with the ***if...else if..else*** ladder. However, the syntax of the **switch** statement is cleaner and much easier to read and write.

Example 10

The following example of a C++ program checks whether a number is even or odd using *switch* statement.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Enter any number to check even or odd: ";
    cin >> num;
    switch (num%2)
    {
        case 0:
            cout << "Number is Even";
            break;
        case 1:
            cout << "Number is Odd";
            break;
    }
    return 0;
}
```

Example 11

The following example of a C++ program creates a basic calculator using the *switch* statement.

```
#include <iostream>
using namespace std;
int main()
{
    cout << " Basic Calculator" << endl;
    cout << " ----------------" << endl;
    cout << " [1] Addition " << endl;
    cout << " [2] Subtraction" << endl;
    cout << " [3] Multiplication" << endl;
    cout << " [4] Division" << endl;
    float num1, num2, result;
    int choice;
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;
    cout << "Enter your choice [1/2/3/4]: ";
    cin >> choice;
    switch (choice)
    {
        case 1:
```

```
            result = num1+num2;
            cout << "Summation = "<< result;
            break;
        case 2:
            result = num1-num2;
            cout << "Subtraction = " << result;
            break;
        case 3:
            result = num1*num2;
            cout << "Multiplication = " << result;
            break;
        case 4:
            result = num1/num2;
            cout << "Division = " << result;
            break;
        default :
            cout << "Invalid choice!" << endl;
    }
}
```

3.7. Ternary operator

In C++, the ***ternary operator*** (also known as the conditional operator) can be used to replace ***if...else*** in certain scenarios. A ***ternary operator*** evaluates the test condition and executes a block of code based on the result of the condition. Its syntax is as follows.

```
condition ? expression1 : expression2;
```

Here, ***condition*** is evaluated and
— if ***condition*** is ***true***, then ***expression1*** is executed.
— if ***condition*** is ***false***, then ***expression2*** is executed.
The *ternary operator* takes three operands (condition, expression1 and expression2). Hence, the name *ternary operator*.

Example 12

The following example of a C++ program finds the minimum of two integers using *ternary operator*.

```
#include <iostream>
using namespace std;
int main()
{
    int x, y, result;
    cout << "Enter two integers: ";
    cin >> x >> y;
    result = (x<y) ? x : y;
    cout << result << " is the minimum." << endl;
}
```

Example 13

The following example of a C++ program finds the minimum of three integers using *ternary operator*.

```
#include <iostream>
using namespace std;
int main()
{
    int x, y, z, result;
    cout << "Enter three integers: ";
    cin >> x >> y >> z;
    result = ((x<y) ? (x<z ? x : z) : (y<z ? y : z));
    cout << result << " is the minimum." << endl;
}
```

3.8. Compound conditions

Conditions such as $n\%d$ and $x >= y$ can be combined to form compound conditions. This is done using the logical operators && (and), || (or), and ! (not). They are defined by
— $p\&\&q$; evaluates to ***true*** if and only if both p and q evaluate to ***true***
— $p||q$; evaluates to ***false*** if and only if both p and q evaluate to ***false***
— $!p$; evaluates to ***true*** if and only if p evaluates to ***false***
For example, $(n\%d||x >= y)$ will be ***false*** if and only if n is zero and x is less than y. The definitions of the three logical operators are usually given by the truth tables in the Figure 3.1 below.

p	q	$p\&\&q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p\|\|q$
T	T	T
T	F	T
F	T	T
F	F	F

p	$!p$
T	F
F	T

Figure 3.1. Truth tables for three logical operators

Figure 3.1 shows, for example, that if p is true and q is false, then the expression $p\&\&q$ will be ***false*** and the expression $p||q$ will be ***true***.

Example 13

The following example of a C++ program finds the minimum of three integers using *compound conditions*.

```
#include <iostream>
using namespace std;
int main()
{
    int x, y, z;
    cout << "Enter three integers: ";
    cin >> x >> y >> z;
    if (x <= y && x <= z)
    {
            cout << "Their minimum is " << x <<endl;
    }
```

```
12      if (y <= x && y <= z)
13      {
14              cout << "Their minimum is " << y <<endl;
15      }
16      if (z <= x && z <= y)
17      {
18              cout << "Their minimum is " << z <<endl;
19      }
20  }
```

Post-laboratory Problems

1. Write and run a C++ program that reads the users' age and then prints the following
   a) *You are a child.* if the age is less than 18 years,
   b) *You are an adult.* if the age is in between 18 years and 65 years, and
   c) *You are a senior citizen.* if the age is more than 65 years.
2. Write and run a C++ program that reads the students' mark and then prints the appropriate grade. Your program must follow the grading system of AUST.
3. Write and run a C++ program to check whether a year is a leap-year or not. Leap-year is a special year containing one extra day, i.e., total 366 days in a year. A year is said to be leap-year, if the year is exactly divisible by 4 and not divisible by 100. Year is also a leap-year if it is exactly divisible by 400.
4. Write and run a C++ program that takes input of five integers from the keyboard and then determines and prints the largest and the smallest integers among them. using only the *ternary operator.*
5. Write and run a C++ program to check whether an alphabet is vowel or consonant, using the *switch* statement.
6. Write and run a C++ program to print the number of days in a month, using only the *switch* statement. January, March, May, July, August, October, and December have 31 days. April, June, September, and November have 30 days. February has 28/29 days (due to the leap-year).
7. An AC power supply of V volts is applied to a circuit load with impedance of $Z(\phi)$ with currentI. Write a C++ program to display the real power P, the reactive power R, the apparent power S and the power factor PF of the load. Test the program with a voltage of 120 volts and an impedance of 8 ohms at 30 degrees.
8. Write and run a C++ program to find the roots of a quadratic equation including complex solution. The standard form of a quadratic equation is $ax^2+bx+c=0$, where a, b, and c are real numbers and $a \neq 0$. A quadratic equation can have either one or two distinct real or complex roots depending upon nature of discriminant of the equation. Where discriminant of the quadratic equation is given by $\Delta = b^2-4ac$.

Laboratory 4

Iteration

4.1. Objective

In this laboratory the students explore three C++ iteration constructs — ***while***, ***do...while***, and ***for*** statements. Iteration is the repetition of a statement or block of statements in a program. Iteration statements are also called ***loops*** because of their cyclic nature. The objective of this laboratory is to teach students how to use and write looping constructs. In addition to that, the students will learn about the nested loops as well as common looping problems, such as infinite loops, off-by-one loops, improper initialization of a loop counter, and incorrect termination conditions, etc.

4.2. The *while* loop

The syntax of the ***while*** statement is as follows.

```
while (condition)
{
    // body of the loop
}
```

Here,
— A ***while*** loop evaluates the ***condition***
— If the ***condition*** evaluates to ***true***, the code inside the w***hile*** loop is executed.
— The ***condition*** is evaluated again.
— This process continues until the ***condition*** is ***false***.
— When the ***condition*** evaluates to ***false***, the loop terminates.

Example 01

The following example of a C++ program prints numbers from 1 to 5.

```
#include <iostream>
using namespace std;
int main()
{
    int i = 1;
    while (i <= 5)
    {
        cout << i << " ";
        ++i;
    }
    return 0;
}
```

The console output will be as follows.

```
1 2 3 4 5
```

Example 02

The following example of a C++ program prints sum of positive numbers only. The negative number works as the termination key and is not added to the sum.

```
#include <iostream>
using namespace std;
int main()
{
    int number;
    int sum = 0;
    cout << "Enter a number: ";
    cin >> number;
    while (number >= 0)
    {
        sum += number;
        cout << "Enter a number: ";
        cin >> number;
    }
    cout << endl << "The sum is " << sum << endl;
    return 0;
}
```

The console output will be as follows.

```
Enter a number: 6
Enter a number: 12
Enter a number: 7
Enter a number: 0
Enter a number: -2

The sum is 25
```

In this program, the user is prompted to enter a number, which is stored in the variable ***number***. In order to store the sum of the numbers, we declare a variable ***sum*** and initialize it to the value of 0. The ***while*** loop continues until the user enters a negative number. During each iteration, the number entered by the user is added to the ***sum*** variable. When the user enters a negative number, the loop terminates. Finally, the total sum is displayed.

Example 03

This program computes the sum of reciprocals, i.e., $sum = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$.

```
#include <iostream>
using namespace std;
int main()
{
    int bound, i=0;
    cout << "Enter a positive integer: ";
    cin >> bound;
    double sum=0.0;
```

```
    while (i < bound)
    {
        sum += 1.0/++i;
    }
    cout << "The sum of the first " << i << " reciprocals is " << sum << endl;
    return 0;
}
```

Example 04

The following C++ program generates and prints the *Fibonacci sequence* up to a certain number. In mathematics, the Fibonacci numbers, commonly denoted F_n, form a sequence, called the *Fibonacci sequence*, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is, $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. Thus the beginning of the sequence is $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$.

```
#include <iostream>
using namespace std;
int main()
{
    int t1 = 0, t2 = 1, nextTerm = 0, n;
    cout << "Enter a positive number: ";
    cin >> n;
    cout << "Fibonacci Series: " << t1 << ", " << t2 << ", ";
    nextTerm = t1 + t2;
    while (nextTerm <= n)
    {
        cout << nextTerm << ", ";
        t1 = t2;
        t2 = nextTerm;
        nextTerm = t1 + t2;
    }
    return 0;
}
```

The console output will be as follows.

```
Enter a positive integer: 100
Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
```

4.3. The *do...while* loop

The ***do...while*** loop is a variant of the ***while*** loop with one important difference: the body of ***do...while*** loop is executed once before the condition is checked. The syntax of the ***do...while*** statement is as follows.

```
do
{
    // body of loop;
}
while (condition);
```

Here,

— The body of the loop is executed at first. Then the ***condition*** is evaluated.

— If the ***condition*** evaluates to ***true***, the body of the loop inside the ***do*** statement is executed again.
— The ***condition*** is evaluated once again.
— If the ***condition*** evaluates to ***true***, the body of the loop inside the ***do*** statement is executed again.
— This process continues until the ***condition*** evaluates to ***false***. Then the loop stops.

Example 05

The following example of a C++ program prints numbers from 1 to 5, same as the Example 01. However, this example uses the ***do...while*** loop.

```
#include <iostream>
using namespace std;
int main()
{
    int i = 1;
    do
    {
        cout << i << " ";
        ++i;
    }
    while (i <= 5);
    return 0;
}
```

The console output will be as follows.

```
1 2 3 4 5
```

Example 06

The following example of a C++ program takes a number from the user and then prints it in reverse orders.

```
#include <iostream>
using namespace std;
int main()
{
    int num, digit;
    cout << "Enter your number: ";
    cin >> num;
    cout << "The number in reverse order is: ";
    do
    {
        digit = num%10;
        cout << digit;
        num /= 10;
    }
    while (num!=0);
}
```

The console output will be as follows.

```
Enter your number: 10249
The number in reverse order is: 94201
```

Example 07

The following example of a C++ program takes a number from the user and then calculates the sum of all of its' digits.

```
#include <iostream>
using namespace std;
int main()
{
    int num, digit, actual_num, sum=0;
    cout << "Enter your number: ";
    cin >> num;
    actual_num = num;
    do
    {
        digit = num%10;
        num /= 10;
        sum+=digit;
    }
    while (num!=0);
    cout << "Sum of the all digits"
}
```

The console output will be as follows.

```
Enter your number: 10249
The number in reverse order is: 94201
```

4.4. The *for* loop

The syntax of the ***for*** statement is as follows.

```
for (initialization; condition; update)
{
    // body of-loop
}
```

Here,
— ***initialization*** - initializes variables and is executed only once.
— ***condition*** - if ***true***, the body of ***for*** loop is executed. Andif ***false***, the for loop is terminated.
— ***update*** - updates the value of initialized variables and again checks the condition.

Example 08

The following example of a C++ program prints numbers from 1 to 5, same as the Example 01 and Example 05. However, this example uses the ***for*** loop.

```
#include <iostream>
using namespace std;
int main()
{
    for (int i = 1; i <= 5; ++i)
    {
        cout << i << " ";
```

```
    }
    return 0;
}
```

The console output will be as follows.

```
1 2 3 4 5
```

Example 09

The following example of a C++ program finds the sum of first n natural numbers.

```
#include <iostream>
using namespace std;
int main()
{
    int num, sum = 0;
    cout << "Enter a positive integer: ";
    cin >> num;
    for (int count = 1; count <= num; ++count)
    {
        sum += count;
    }
    cout << "Sum = " << sum << endl;
    return 0;
}
```

The console output will be as follows.

```
Enter a positive integer: 10
Sum = 55
```

Example 10

The following example of a C++ program finds the factorial of a number.

```
#include <iostream>
using namespace std;
int main()
{
    int num, factorial=1;
    cout << "Input a number to find the factorial: ";
    cin >> num;
    for (int a=1; a <= num; a++)
    {
        factorial = factorial*a;
    }
    cout << "The factorial of " << num << " is " << factorial << endl;
}
```

The console output will be as follows.

```
Input a number to find the factorial: 5
The factorial of 5 is 120
```

Example 11

The following example of a C++ program calculates the sum of the series $x + x^2 + x^3 + x^4 + \cdots + x^n$.

```
#include <iostream>
using namespace std;
int main()
{
    int i, n, x, t, sum=0;
    cout << "Enter the value of x: ";
    cin >> x;
    cout << "Enter the number of terms: ";
    cin >> n;
    t = x;
    for (i=1; i<=n; i++)
    {
        sum+=t;
        t*=x;
    }
    for (i=1; i<n; i++)
    {
        cout << x << "^" << i<< " + ";
    }
    cout << x << "^" << n << " = " << sum;
    return 0;
}
```

The console output will be as follows.

```
Enter the value of x: 2
Enter the number of terms: 5
2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 62
```

Example 12

The following example of a C++ program calculates the sum of the series $1 - 2^3 + 3^3 - 4^3 + \cdots$.

```
#include <iostream>
using namespace std;
int main()
{
    int n, j, temp, sign=1, sum=0;
    cout << "Enter the number of terms: ";
    cin >> n;
    for (int i=1; i<=n; i++)
    {
        j = i*i*i;
        temp = sign*j;
        sum += temp;
        sign = (-1)*sign;
    }
    cout << "Sum = " << sum;
    return 0;
}
```

The console output will be as follows.

```
Enter the number of terms: 3
Sum = 20
```

4.5. Nested loop

A loop within another loop is called a nested loop. That is, the inner loop is nested inside the outer loop. Nested loops are useful when for each pass through the outer loop, you need to repeat some action on the data in the outer loop.

Example 13

The following example of a C++ program displays 7 days of 3 weeks

```
#include <iostream>
using namespace std;
int main()
{
    int weeks = 3, days_in_week = 7;
    for (int i = 1; i <= weeks; ++i)
    {
        cout << "Week: " << i << endl;
        for (int j = 1; j <= days_in_week; ++j)
        {
            cout << "\t Day:" << j << endl;
        }
    }
    return 0;
}
```

The console output will be as follows.

```
Week: 1
        Day:1
        Day:2
        Day:3
        Day:4
        Day:5
        Day:6
        Day:7
Week: 2
        Day:1
        Day:2
        Day:3
        Day:4
        Day:5
        Day:6
        Day:7
Week: 3
        Day:1
        Day:2
        Day:3
```

```
Day:4
Day:5
Day:6
Day:7
```

Example 14

The following example of a C++ program displays a pattern with 5 rows and 3 columns.

```
#include <iostream>
using namespace std;
int main()
{
    int rows = 5, columns = 3;
    for (int i = 1; i <= rows; ++i)
    {
        for (int j = 1; j <= columns; ++j)
        {
            cout << "* \t";
        }
        cout << endl;
    }
    return 0;
}
```

The console output will be as follows.

```
* * *
* * *
* * *
* * *
* * *
```

Example 15

The following example of a C++ program displays a pattern of half pyramid with numerics.

```
#include <iostream>
using namespace std;
int main()
{
    int i=0, j=0;
    for (i=5; i>=1; i--)
    {
        for (j=1; j<=i; j++)
        {
            cout << j << "\t" ;
        }
        cout << endl;
    }
    return 0;
}
```

The console output will be as follows.

```
1       2       3       4       5
1       2       3       4
1       2       3
1       2
1
```

Example 16

The following example of a C++ program displays a pattern of inverted full pyramid.

```
#include <iostream>
using namespace std;
int main()
{
    int rows;
    cout << "Enter number of rows: ";
    cin >> rows;
    for (int i = rows; i >= 1; --i)
    {
        for (int space = 0; space < rows-i; ++space)
        {
            cout << "  ";
        }
        for(int j = i; j <= 2*i-1; ++j)
        {
            cout << "* ";
        }
        for (int j = 0; j < i-1; ++j)
        {
            cout << "* ";
        }
        cout << endl;
    }
}
```

The console output will be as follows.

```
Enter number of rows: 4
* * * * * * *
  * * * * *
    * * *
      *
```

4.6. Infinite loop

If the condition in a loop is always ***true***, it runs forever (until memory is full). For example,

```
// infinite for loop
for (int i = 1; i > 0; i++)
{
    // block of code
}
```

```
// infinite while loop
while (true)
{
    // body of the loop
}
```

```
// infinite do...while loop
int count = 1;
do
{
    // body of loop
}
while(count == 1);
```

In the above programs, the ***condition*** is always ***true*** which will then run the code for infinite times. That means, a loop becomes infinite loop if a ***condition*** never becomes ***false***.

Example 17

The following example of a C++ program shows the example of infinite loop.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      for( ; ; )
6      {
7          cout <<"This loop will run forever." << endl;
8      }
9      return 0;
10 }
```

In the above program, since none of the three expressions that form the ***for*** loop are required, you can make an infinite loop by leaving the conditional expression empty. When the conditional expression is absent, it is assumed to be ***true***. You can terminate an infinite loop by pressing $Ctrl + C$ keys.

Post-laboratory Problems

1. Write and run a C++ program to test a number if it is prime number or not. Use the *while* loop.
2. Write and run a C++ program to find the $sum = 1 + 3^2 + 5^2 + ... + n^2$. Use the *while* loop.
3. Write and run a C++ program to find the Fibonacci sequence using the *do...while* loop.
4. Write and run a C++ program to find the factorial of a given number using the *do...while* loop.
5. Write and run a C++ program to find the sum of the even numbers using the *do..while* loop.
6. Write and run a C++ program to calculate sum of numbers until a negative number is entered. Use the *for* loop.
7. Write and run a C++ program to find the sum and average of all the digits of a given number using the *for* loop.
8. Write and run a C++ program to find the $sum = 1 - 3^2 + 5^2 - ....$ using the *for* loop.
9. Write and run a C++ program to find the $sum = 1 + 2^2 + 4^4 + ...$ using the *for* loop.

Laboratory 5

Function

5.1. Objective

In this laboratory, students begin an in-depth exploration of function invocation. The focus of this laboratory is the parameter passing mechanisms. With the help of the function, students explore value and reference parameter-passing mechanisms. The objective of the laboratory is to introduce the students to the function, both in-built and user defined. In addition to that, students will have a strong understanding of parameter passing mechanisms.

5.2. Function

Most computer programs that solve real-world problems are much larger than the programs presented in the first laboratory. Experience has shown that the best way to develop and maintain a large program is to construct it from smaller modules, each of which is more manageable than the original program. Besides dividing a complex problem into smaller modules makes any program easy to understand and reusable. Each of these modules is a small block of code that performs a specific task. These modules in C++ are called functions.

There are two types of function:

1. **Standard library functions:** these functions are redefined in C++
2. **User-defined function:** these functions are created by users

5.3. C++ library functions

Library functions are the built-in functions in C++ programming. Programmers can use library functions by invoking the functions directly; they don't need to write the functions themselves. Some common library functions in C++ are *sqrt()*, *abs()*, *isdigit()*, etc. In order to use library functions, we usually need to include the header file in which these library functions are defined. For instance, in order to use mathematical functions such as *sqrt()* and *abs()*, we need to include the header file *cmath*.

Example 01

The following example of a C++ program finds the square root of a number.

```
#include <iostream>
using namespace std;
int main()
{
    double number, squareRoot;
    number = 25.0;
    squareRoot = sqrt(number);
    cout << "Square root of " << number << " = " << squareRoot;
```

```
9       return 0;
10  }
```

The console output will be as follows.

```
Square root of 25 = 5
```

In this program, the ***sqrt()*** library function is used to calculate the square root of a number. The function declaration of ***sqrt()*** is defined in the *cmath* header file. That's why we need to use the code ***#include <cmath>*** to use the ***sqrt()*** function.

5.4. C++ user-defined function

C++ allows the programmer to define their own function. A user-defined function groups code to perform a specific task and that group of code is given a name (identifier). When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

5.4.1. C++ function declaration

The syntax to declare a function is as follows.

```
return_type function_name (parameter1, parameter2,...)
{
    // function body
}
```

Here is an example of a function declaration.

```
int max (int x, int y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

5.4.2. Calling a function

In the above program, we have declared a function named ***max()*** before the ***main()*** function. To use the ***max()*** function, we need to call it. Here's how we can call the above ***max()*** function.

```
int max (int x, int y)
{
    if (x < y)
        return y;
    else
        return x;
}
int main()
{
    int m, n;
    cout << "Enter two integers: ";
```

```
    cin >> m >> n;
    cout << max(m,n) << " the maximum." << endl;
}
```

5.4.3. Function prototype

In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype.

Example 01

The following example of a C++ program shows function prototyping.

```
1  #include <iostream>
2  using namespace std;
3
4  int max (int x, int y);
5
6  int main()
7  {
8      int m, n;
9      cout << "Enter two integers: ";
10     cin >> m >> n;
11     cout << max(m,n) << " is the maximum." << endl;
12 }
13
14 int max (int x, int y)
15 {
16     if (x < y)
17         return y;
18     else
19         return x;
20 }
```

The console output will be as follows.

```
Enter two integers: 12 79
79 is the maximum.
```

In the above code, the function prototype is in the line 3, i.e., ***int max (int x, int y);***

This provides the compiler with information about the function name and its parameters. That's why we can use the code to call a function before the function has been defined.

Example 02

The following example of C++ program prints all the factorial values for the numbers 1 to 6.

```
1  #include <iostream>
2  using namespace std;
3
4  long fact (int);
5
6  int main()
7  {
8      for (int i=1; i<6; i++)
```

```
9       {
10          cout << " " << fact(i);
11      }
12      cout << endl;
13  }
14
15  long fact (int n)
16  {
17      if (n < 0)
18          return 0;
19      int f = 1;
20      while (n > 1)
21      {
22          f *= n--;
23      }
24      return f;
25  }
```

5.5. The *void* function

A function needs not to return a value. In other programming languages, such a function is called a procedure or a subroutine. In C++, such a function is identified simply by placing the keyword ***void*** where the function's return type would be. A type specifies a set of values. For example, the type ***short*** specifies the set of integers from $-32,768$ to $32,768$. The ***void*** type specifies the empty set. Consequently, no variable can be declared with ***void*** type. A ***void*** function is simply one that returns no value.

Example 03

The following example of C++ program takes the numeric values for day, month, and year, and then prints in a formal fashion.

```
1   #include <iostream>
2   using namespace std;
3
4   void print_date (int, int, int);
5
6   int main()
7   {
8       int month, day, year;
9       do
10      {
11          cout << "Enter the date: ";
12          cin >> day;
13          cout << "Enter the month: ";
14          cin >> month;
15          cout << "Enter the year: ";
16          cin >> year;
17          print_date (month, day, year);
18      }
19      while (month > 0);
20  }
21
```

```
22 void print_date (int m, int d, int y)
23 {
24     if (m < 1 || m > 12 || d < 1 || d > 31 || y < 0)
25     {
26         cout << "Error: parameter out of range." << endl;
27         return;
28     }
29     cout << "The date is: ";
30     switch (m)
31     {
32         case 1: cout << "January "; break;
33         case 2: cout << "February "; break;
34         case 3: cout << "March "; break;
35         case 4: cout << "April "; break;
36         case 5: cout << "May "; break;
37         case 6: cout << "June "; break;
38         case 7: cout << "July "; break;
39         case 8: cout << "August "; break;
40         case 9: cout << "September "; break;
41         case 10: cout << "October "; break;
42         case 11: cout << "November "; break;
43         case 12: cout << "December "; break;
44     }
45     cout << d << ", " << y << endl;
46 }
```

The console output will be as follows.

```
Enter the date: 15
Enter the month: 3
Enter the year: 2021
The date is: March 15, 2021
```

5.6. Boolean functions

In some situations it is helpful to use a function to evaluate a condition, typically within an if statement or a while statement. Such functions are called boolean functions. These functions can return ***bool*** values just like any other type, which is often convenient for hiding complicated tests inside functions.

Example 04

The following example of C++ program prints all the prime numbers up to 80.

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 bool is_prime(int);
6
7 int main()
8     {
9     for (int n=0; n < 80; n++)
```

```
    {
        if (is_prime(n))
            cout << n << " ";
    }
    cout << endl;
}

bool is_prime(int n)    // returns true if n is prime, false otherwise
{
    if (n < 2)
        return false;       // 0 and 1 are not primes
    if (n < 4)
        return true;        // 2 and 3 are the first primes
    if (n%2 == 0)
        return false;       // 2 is the only even prime
    for (int d=3; d <= sqrt(n); d += 2)
    {
        if (n%d == 0)
            return false;   // n has a nontrivial divisor
    }
    return true;            // n has no nontrivial divisors
}
```

The console output will be as follows.

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
```

Example 05

The following example of C++ program checks whether the given year is a leap-year.

```
#include <iostream>
using namespace std;

bool is_leap_year (int);

int main()
{
    int year;
    do
    {
        cout << "Enter a year: ";
        cin >> year;
        if (is_leap_year (year))
            cout << year << " is a leap year." << endl;
        else
            cout << year << " is not a leap year." << endl;
    } while (year > 1);
}

bool is_leap_year (int y)
{
    return (y%4==0 && y%100!=0 || y%400==0);
}
```

The console output will be as follows.

```
Enter a year: 2021
2021 is not a leap year.
Enter a year: 2020
2020 is a leap year.
Enter a year: 1
1 is not a leap year.
```

5.7. Passing arguments by value and by reference

In many programming languages, there are two ways to pass arguments — *pass-by-value* and *pass-by-reference*. When arguments are *passed-by-value*, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect an original variable's value in the caller. When an argument is *passed-by-reference*, the caller allows the called function to modify the original variable's value.

Pass-by-value should be used whenever the called function does not need to modify the value of the caller's original variable. This prevents the accidental side effects (variable modifications) that so greatly hinder the development of correct and reliable software sys- tems. *Pass-by-reference* should be used only with trusted called functions that need to modify the original variable.

Example 06

This example shows the difference between passing-by-value and passing-by-reference.

```
1  #include <iostream>
2  using namespace std;
3
4  void dummy (int, int&);
5
6  int main()
7  {
8  int a = 22, b = 44;
9  cout << "a = " << a << ", b = " << b << endl;
10 dummy(a, b);
11 cout << "a = " << a << ", b = " << b << endl;
12 dummy(2*a-3, b);
13 cout << "a = " << a << ", b = " << b << endl;
14 }
15
16 void dummy (int x, int& y)
17 {
18 x = 88;
19 y = 99;
20 }
```

The console output will be as follows.

```
a = 22, b = 44
a = 22, b = 99
a = 22, b = 99
```

The call ***dummy(a, b)*** passes a by value to ***x*** and it passes ***b*** by reference to ***y***. Therefore, ***x*** is a local variable that is assigned ***a***'s value of ***22***, while ***y*** is an alias for the variable ***b*** whose value is ***33***. The function

assigns ***88*** to ***x***, but that has no effect on ***a***. But when it assigns ***99*** to ***y***, it is really assigning ***99*** to ***b***, because ***y*** is an alias for ***b***. Therefore, when the function terminates, ***a*** still has its original value ***22***, while ***b*** has the new value ***99***. The argument ***a*** is read-only, while the argument ***b*** is read-write.

5.8. Function overloading

In C++, two functions can have the same name if the number and/or type of arguments passed is different. These functions having the same name but different arguments are known as *overloaded functions*. As long as they have different parameter type lists, the compiler will regard them as different functions. To be distinguished, the parameter lists must either contain a different number of parameters, or there must be at least one position in their parameter lists where the types are different.

Example 07

This example shows the function overloading in C++.

```
#include <iostream>
using namespace std;

int max (int, int);
int max (int, int, int);
float max (float, float);
float max (float, float, float);

int main()
{   int p, q, r;
    float x, y, z;
    cout << "Enter three integers: ";
    cin >> p >> q >> r;
    cout << "Enter three floating point numbers: ";
    cin >> x >> y >> z;
    cout << "Maximum of the first two integers is: " << max(p, q) << endl;
    cout << "Maximum of the three integers is: " << max(p, q, r) << endl;
    cout << "Maximum of the first two floating numbers is: " << max(x, y) << endl;
    cout << "Maximum of the three floating numbers is: " << max(x, y, z) << endl;
}

int max (int x, int y)
{
    return (x > y ? x : y);
}

int max (int x, int y, int z)
{
    int m = (x > y ? x : y);
    return (z > m ? z : m);
}

float max (float x, float y)
{
    return (x > y ? x : y);
}

float max (float x, float y, float z)
{
    float m = (x > y ? x : y);
    return (z > m ? z : m);
}
```

The console output will be as follows.

```
Enter three integers: 12 345 7890
Enter three floating point numbers: 1.23 45.76 897.235
Maximum of the first two integers is: 345
Maximum of the three integers is: 7890
Maximum of the first two floating numbers is: 45.76
Maximum of the three floating numbers is: 897.235
```

5.9. Default arguments (parameters)

In C++ programming, we can provide default values for function parameters. If a function with default arguments is called without passing arguments, then the default parameters are used. However, if arguments are passed while calling the function, the default arguments are ignored.

Example 08

The following ***dummy()*** function evaluates the third degree polynomial $a_0 + a_1x + a_2x^2 + a_3x^3$. The actual evaluation is done using *Horner's Algorithm*, grouping the calculations as $a_0 + (a_1 + (a_2 + a_3x)x)x$ for greater efficiency.

```
1  #include <iostream>
2  using namespace std;
3
4  double dummy (double, double, double=0, double=0, double=0);
5
6  int main()
7  {
8      double x, a0, a1, a2, a3;
9      cout << "Enter the value of x: " ;
10     cin >> x;
11     cout << "Enter the value of a0, a1, a2, and a3: ";
12     cin >> a0 >> a1 >> a2 >> a3;
13     cout << "Considering x and a0, the result is: ";
14     cout << dummy (x, a0) << endl;
15     cout << "Considering x, a0, and a1, the result is: ";
16     cout << dummy (x, a0, a1) << endl;
17     cout << "Considering x, a0, a1, and a2, the result is: ";
18     cout << dummy (x, a0, a1, a2) << endl;
19     cout << "Considering x, a0, a1, a2, and a3, the result is: "
20     cout << dummy (x, a0, a1, a2, a3) << endl;
21 }
22
23 double dummy (double x, double a0, double a1, double a2, double a3)
24 {
25     return (a0+(a1+(a2+a3*x)*x)*x);
26 }
```

The console output will be as follows.

```
Enter the value of x: 2.0003
Enter the value of a0, a1, a2, and a3: 7 6 5 4
Considering x and a0, the result is: 7
```

```
Considering x, a0, and a1, the result is: 19.0018
Considering x, a0, a1, and a2, the result is: 39.0078
Considering x, a0, a1, a2, and a3, the result is: 71.0222
```

Post-laboratory Problems

1. Write and run a C++ program to test the following ***average()*** function that returns the average of four numbers.

```
float average (float x1, float x2, float x3, float x4);
```

2. Write and run a C++ program to test the following ***min()*** function that returns the smallest of four given integers.

```
int min (int, int, int, int);
```

3. Write and run a C++ program to test the following ***is_square()*** function that determines whether the given integer is a square number.

```
int is_square (int n);
```

Hints: *The first ten square numbers are 0, 1, 4, 9, 16, 25, 36, 49, 64, and 81.*

4. Write and run a C++ program to test the following ***compute_sphere()*** function that returns the volume ***v*** and the surface area ***s*** of a sphere with given radius ***r***.

```
void compute_sphere (float& v, float& s, float r);
```

5. Write and run a C++ program to find the solution for a polynomial of a given maximum degree of 5, i.e., $c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$ for arbitrary values of the coefficients c_5, c_4, c_3, c_2, c_1, and c_0. The formula for evaluating the polynomial at a given value of x is $result = ((((c_5 * x + c_4) * x + c_3) * x + c_2) * x + c_1) * x + c_0$. The program must invoke at least two functions to
   a) prompt for and read the values of the coefficients.
   b) calculate the result of the polynomial
6. Write and run a C++ program to test the following ***power()*** function that returns ***x*** raised to the power ***n***, where ***n*** can be any integer.

```
double power (double x, int p);
```

Hints: *Use the algorithm that would compute x^{20} by multiplying 1 by x for 20 times.*

7. Write and run a C++ program to find the sum of the following series invoking function.
   a) $sum = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$
   b) $sum = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots$

Laboratory 6

Arrays

6.1. Objective

This laboratory develops the ability to use and manipulate arrays. This laboratory also introduces the activity of searching a list for a key value by examining, modifying, and running programs that use sort as well as search techniques. The student performs an experiment that measures the efficiency of these search techniques.

6.2. Arrays

In C++, an array is a variable that can store multiple values of the same type. For example, suppose a class has 27 students, and we need to store the grades of all of them. Instead of creating 27 separate variables, we can simply create an array that can hold a maximum of 27 elements of same type data.

6.2.1. Array declaration in C++

The syntax of declaring an array is as follows.

```
data_type array_name[array_s ize];
```

Arrays occupy space in memory. You specify the type of each element and the number of elements each array requires so that the computer may reserve the appropriate amount of memory. The following definition reserves 10 elements for integer array ***x***.

```
int x[10];
```

Here,
— ***int*** - type of element to be stored
— ***x*** - name of the array
— ***10*** - size of the array
In C++, the size and type of arrays cannot be changed after its declaration.

6.2.2. Access elements in array

In C++, each element in an array is associated with a number. The number is known as an *array index*. We can access elements of an array by using those indices. To refer to a particular location or element in the array, we specify the array's *name* and the *array index* of the particular element in the array as follows.

```
// syntax to access array elements
array_name[index];
```

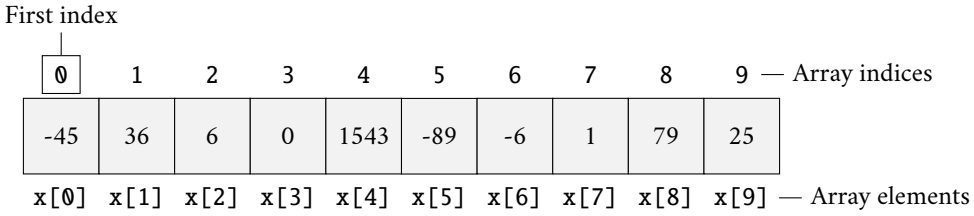


Figure 6.1. Elements of an array in C++

For example, consider the array ***x*** we have seen above. Figure 6.1 shows the integer array called ***x***, containing 10 elements. Any one of these elements may be referred to by giving the array's name followed by the position number (or index) of the particular element in square brackets. The position number (or index) within square brackets is called a *subscript*. A *subscript* must be an integer or an integer expression. The first element of every array is in the zeroth index. An array name, like other variable names, can contain only letters, digits and underscores and cannot begin with a digit. Here, the array's name is ***x***. Its' 10 elements are referred to as $x[0], x[1], x[2], ..., x[8]$ and $x[9]$. The value stored in $x[0]$ is -45, the value of $x[1]$ is 36, $x[2]$ is 6, $x[7]$ is 1 and $x[9]$ is 25 .

6.2.3. Array initialization

In C++, it is possible to initialize an array during declaration. For example,

```
int x[10] = {-45, 36, 6, 0, 1543, -89, -6, 1, 79, 25};
```

Following is another method to initialize array during declaration.

```
int x[] = {-45, 36, 6, 0, 1543, -89, -6, 1, 79, 25};
```

Here, we have not mentioned the size of the array. In such cases, the compiler automatically computes the size.

Example 01

The following example of a C++ program displays array elements.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int numbers[5] = {7, 5, 6, 12, 35};
6      cout << "The numbers are: ";
7      for (int i = 0; i < 5; ++i)
8      {
9          cout << numbers[i] << "  ";
10     }
11     return 0;
12 }
```

The console output will be as follows.

```
The numbers are: 7  5  6  12  35
```

Here, we have used a ***for*** loop to iterate from $i = 0$ to $i = 4$. In each iteration, we have printed ***numbers[i]***.

Example 02

The following example of a C++ program takes inputs from user and store them in an array. Then it prints the stored values.

```
#include <iostream>
using namespace std;
int main()
{
    int numbers[5];
    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; ++i)
    {
        cin >> numbers[i];
    }
    cout << "The numbers are: ";
    for (int n = 0; n < 5; ++n)
    {
        cout << numbers[n] << " ";
    }
    return 0;
}
```

The console output will be as follows.

```
Enter 5 numbers: 12 67 -9 0 5
The numbers are: 12 67 -9 0 5
```

Example 02

The following example of a C++ program displays array elements.

```
#include <iostream>
using namespace std;
int main()
{
    int numbers[5] = {7, 5, 6, 12, 35};
    cout << "The numbers are: ";
    for (int i = 0; i < 5; ++i)
    {
        cout << numbers[i] << "  ";
    }
    return 0;
}
```

The console output will be as follows.

```
The numbers are: 7  5  6  12  35
```

Example 03

The following example of a C++ program displays largest element of an array.

```
#include <iostream>
using namespace std;
int main()
{
    int i, n;
    float arr[100];
    cout << "Enter total number of elements(1 to 100): ";
    cin >> n;
    cout << endl;
    for (i = 0; i < n; ++i)
    {
        cout << "Enter Number " << i + 1 << " : ";
        cin >> arr[i];
    }
    for (i = 1;i < n; ++i)
    {
        if (arr[0] < arr[i]) arr[0] = arr[i];
    }
    cout << "Largest element = " << arr[0];
    return 0;
    }
```

The console output will be as follows.

```
Enter total number of elements(1 to 100): 5

Enter Number 1 : 17
Enter Number 2 : 29
Enter Number 3 : 42
Enter Number 4 : 36
Enter Number 5 : 66
Largest element = 66
```

6.3. Passing array to function

In C++, we can pass arrays as an argument to a function. And, also we can return arrays from a function. The syntax for passing an array to a function is as follows.

```
returnType functionName (dataType arrayName[arraySize])
{
    // function body
}
```

For an example, if we pass an ***int*** type array named ***marks*** to the function ***total()***, and the size of the array is ***5***, then we can express it as follows.

```
int total (int marks[5])
{
    // function body
}
```

Example 04

The following example of a C++ program display marks of 5 students.

```
#include <iostream>
using namespace std;

void display (int m[5]);

int main()
{
    int marks[5] = {88, 76, 90, 61, 69};
    display(marks);
    return 0;
}

void display (int m[5])
{
    cout << "Displaying marks: " << endl;
    for (int i = 0; i < 5; ++i)
    {
        cout << "Student " << i + 1 << ": " << m[i] << endl;
    }
}
```

The console output will be as follows.

```
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69
```

6.3.1. Linear search

The simplest type of searching process is the linear search. In linear search algorithm, we compare targeted element with each element of the array. If the element is found then its position is displayed.

Example 05

The following example of a C++ program searches any element or number in an array.

```
#include <iostream>
using namespace std;

int input[100];

void read (int count);
void search (int num, int count);

int main()
{
    int count, num;
    cout << "Enter the number of elements in array: ";
```

```
13      cin >> count;
14      cout << "Enter " << count << " numbers: ";
15      read (count);
16      cout << "Enter a number to search in the array: ";
17      cin >> num;
18      search (num, count);
19      return 0;
20  }
21
22  void read (int count)
23  {
24      for (int i = 0; i < count; i++)
25      {
26          cin >> input[i];
27      }
28  }
29
30  void search (int num, int count)
31  {
32      int i;
33      for (i = 0; i < count; i++)
34      {
35          if (input[i] == num)
36          {
37              cout << "Element is found at index " << i << "." << endl;
38              break;
39          }
40      }
41      if (i == count)
42      {
43          cout  << "Element is not present in the array." << endl;
44      }
45  }
```

The console output will be as follows.

```
Enter the number of elements in array: 5
Enter 5 numbers: 1 14 9 87 64
Enter a number to serach in Array: 9
Element found at index 2.
```

6.3.2. Bubble sort

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. Following is an elaborated example of the *bubble sort*. In this example, *bubble sort* is used to arrange five numbers in ascending order.

First Pass:

(**5** **1** 4 2 8) → (**1** **5** 4 2 8), here, the algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5** **4** 2 8) → (1 **4** **5** 2 8), swaps since 5 > 4

(1 4 **5** **2** 8) → (1 4 **2** **5** 8), swaps since 5 > 2

(1 4 2 **5** **8**) → (1 4 2 **5** **8**), since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1** **4** 2 5 8) → (**1** **4** 2 5 8)
(1 **4** **2** 5 8) → (1 **2** **4** 5 8), swaps since 4 > 2
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)
(1 2 4 **5** **8**) → (1 2 4 **5** **8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:
(**1** **2** 4 5 8) → (**1** **2** 4 5 8)
(1 **2** **4** 5 8) → (1 **2** **4** 5 8)
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)
(1 2 4 **5** **8**) → (1 2 4 **5** **8**)

Example 06

The following example of a C++ program utilizes the *bubble sort* algorithm.

```
#include <iostream>
using namespace std;

void bubble_sort (int array[], int size);
void print_array (int array[], int size);

int main()
{
    int data[] = {-2, 45, 0, 11, -9};
    int size = sizeof(data) / sizeof(data[0]);
    bubble_sort (data, size);
    cout << "Sorted Array in Ascending Order: " << endl;
    print_array (data, size);
}

void bubble_sort (int array[], int size)
{
    for (int step=0; step<size-1; ++step)
    {
        for (int i=0; i<size-step-1; ++i)
        {
            if (array[i] > array[i + 1])
            {
                int temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
            }
        }
    }
}

void print_array (int array[], int size)
{
    for (int i=0; i<size; ++i)
    {
        cout << "  " << array[i];
```

```
37      }
38      cout << endl;
39      }
```

The console output will be as follows.

```
Sorted Array in Ascending Order:
  -9   -2   0   11   45
```

6.4. Multidimensional arrays

The arrays we have used previously have all been one-dimensional. This means that they are linear, i.e., sequential. But the element type of an array can be almost any type, including an array type. An array of arrays is called a ***multidimensional array***. A one-dimensional array of one-dimensional arrays is called a two-dimensional array, a one-dimensional array of two-dimensional arrays is called a three-dimensional array, and so on. Data in multidimensional arrays are stored in tabular form (in row major order).

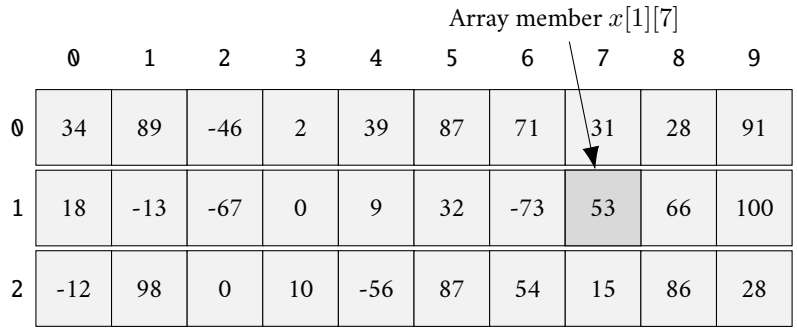


Figure 6.2. Elements of a 2-dimensional array

Figure 6.2 illustrates a 2-dimensional array, ***x***. The array contains three rows and 10 columns, so it is said to be a *3-by-10 array*. In general, an array with *m rows* and *n columns* is called an *m-by-n array*. Every element in array ***x*** is identified in Figure 6.2 by an element name of the form $\boldsymbol{x}[\boldsymbol{i}][\boldsymbol{j}]$; where, ***x*** is the name of the array, and ***i*** and ***j*** are the subscripts (rows and columns respectively in this case) that uniquely identify each element in ***x***.

6.4.1. Declaration of multidimensional arrays

The syntax of declaring an array is as follows.

```
data_type array_name[size1][size2]....[sizeN];
```

Here,
— **data_type:** Type of data to be stored in the array.
— **array_name:** Name of the array
— **size1, size2, ... , sizeN:** Sizes of the dimensions
According to the above syntax, we can declare the two dimensional array of Figure 6.2 as follows.

```
int x[3][10];
```

And thus, we can declare a three dimensional array as below.

```
int x[3][10][25];
```

6.4.2. Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example, the array ***int x[3][10]*** can store total $(3*10) = 30$ elements. Similarly array ***int x[3][10][25]*** can store total $(3 * 10 * 25) = 750$ elements.

6.4.3. Initialization of multidimensional arrays

We can initialize a two-dimensional array as follows.

```
int  test[2][3] = { {2, 4, 5}, {9, 0, 19}};
```

The above array has 2 rows and 3 columns, which is why we have 2 rows of elements with 3 elements each. Same way we can initialize a three-dimensional array.

```
int test[2][3][4] = {
                        { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },
                        { {13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9} }
                    };
```

Notice the dimensions of the above three-dimensional array. The first dimension has the value ***2***. So, the two elements comprising the first dimension are,

```
Element-1 is {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}} and
Element-2 is {{13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9}}
```

The second dimension has the value ***3***. Notice that each of the elements of the first dimension has three elements each,

```
{3, 4, 2, 3}, {0, -3, 9, 11} and {23, 12, 23, 2} for Element-1.
{13, 4, 56, 3}, {5, 9, 3, 5} and {5, 1, 4, 9} for Element-2.
```

Finally, the third dimension has the value ***4***. And there are four ***int*** numbers inside each of the elements of the second dimension,

```
{3, 4, 2, 3}
{0, -3, 9, 11}
... ... ... ...
... ... ... ...
```

Example 07

The following example of a C++ program shows how a two-dimensional array can be processed

```
1 #include <iostream>
2 using namespace std;
3
4 void read (int a[][5]);
5 void print (int a[][5]);
```

```
6
7  int main()
8  {
9      int a[3][5];
10     read(a);
11     print(a);
12 }
13
14 void read (int a[][5])
15 {
16     cout << "Enter 15 integers (5 per row): " << endl;
17     for (int i=0; i<3; i++)
18     {
19         cout << "Row " << i << ": ";
20         for (int j=0; j<5; j++)
21         {
22             cin >> a[i][j];
23         }
24     }
25 }
26
27 void print (int a[][5])
28 {
29     cout << "You have entered the following integers: " << endl;
30     for (int i=0; i<3; i++)
31     {
32         for (int j=0; j<5; j++)
33         {
34             cout << " " << a[i][j];
35         }
36         cout << endl;
37     }
38 }
```

The console output will be as follows.

```
Enter 15 integers (5 per row):
Row 0: 9 7 5 3 1
Row 1: 2 4 6 8 0
Row 2: 1 2 8 9 5
You have entered the following integers:
    9 7 5 3 1
    2 4 6 8 0
    1 2 8 9 5
```

Example 08

The following example of a C++ program finds transpose of a matrix.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int a[10][10], transpose[10][10], row, column, i, j;
```

```
6       cout << "Enter rows and columns of matrix: ";
7       cin >> row >> column;
8       cout << "Enter elements of matrix: " << endl;
9       for (int i = 0; i < row; ++i)
10      {
11          for (int j = 0; j < column; ++j)
12          {
13              cout << "Enter element a" << i + 1 << j + 1 << ": ";
14              cin >> a[i][j];
15          }
16      }
17      cout << endl << "Entered Matrix: " << endl;
18      for (int i = 0; i < row; ++i)
19      {
20          for (int j = 0; j < column; ++j)
21          {
22          cout << " " << a[i][j];
23          if (j == column - 1)
24              cout << endl;
25          }
26      }
27      for (int i = 0; i < row; ++i)
28      {
29          for (int j = 0; j < column; ++j)
30          {
31              transpose[j][i] = a[i][j];
32          }
33      }
34      cout << endl << "Transpose of Matrix: " << endl;
35      for (int i = 0; i < column; ++i)
36      {
37          for (int j = 0; j < row; ++j)
38          {
39              cout << " " << transpose[i][j];
40              if (j == row - 1)
41              cout << endl;
42          }
43      }
44  }
```

The console output will be as follows.

```
Enter rows and columns of matrix: 3 4
Enter elements of matrix:
Enter element a11: 1
Enter element a12: 3
Enter element a13: 4
Enter element a14: 2
Enter element a21: 5
Enter element a22: 7
Enter element a23: 9
Enter element a24: 6
Enter element a31: 8
Enter element a32: 0
```

```
Enter element a33: 3
Enter element a34: 1

Entered Matrix:
 1 3 4 2
 5 7 9 6
 8 0 3 1

Transpose of Matrix:
 1 5 8
 3 7 0
 4 9 3
 2 6 1
```

Example 09

The following example of a C++ program finds the number of zeros in a three-dimensional array.

```
#include <iostream>
using namespace std;

int num_zeros (int a[][4][3], int n1, int n2, int n3);

int main()
{
    int a[2][4][3] = {
                        { {5, 0, 2}, {0, 0, 9}, {4, 1, 0}, {7, 7, 7} },
                        { {3, 0, 0}, {8, 5, 0}, {0, 0, 0}, {2, 0, 9} }
                    };
    cout << "This array has " << num_zeros(a,2,4,3) <<  " zeros." << endl;
}

int num_zeros (int a[][4][3], int n1, int n2, int n3)
{
    int count = 0;
    for (int i = 0; i < n1; i++)
    {
        for (int j = 0; j < n2; j++)
        {
            for (int k = 0; k < n3; k++)
            {
                if (a[i][j][k] == 0)
                    ++count;
            }
        }
    }
    return count;
}
```

The console output will be as follows.

```
This array has 11 zeros.
```

Post-laboratory Problems

1. Write and run a C++ program to test the following function that returns the minimum value among the first ***n*** elements.

```
float min (float a[], int n);
```

2. Write and run a C++ program to find the largest three elements in an array.
3. Write and run a C++ program to find the most occurring element in an array of integers.
4. Write and run a C++ program to separate even and odd numbers of an array of integers. Put all the even numbers first, and then all the odd numbers.
5. Write and run a C++ program to find the first repeating element in an array of integers.
6. Write and run a C++ program to sort the elements of an array from the maximum to the minimum value.
7. Write and run a C++ program that will multiply two matrices of any columns and row, if it passes the test for the conditions of matrix multiplication.
8. Write and run a C++ program to test the following function

```
double stdev (double x[], int n);
```

Hints: *The function returns the standard deviation of a data set of* ***N*** *numbers, i.e.,* $x_0, x_1, \ldots, x_{N-1}$ *defined by the following formula*

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N-1} (x_i - \overline{x})^2}{N-1}}$$

Laboratory 7

Pointers

7.1. Objective

This laboratory focuses on the memory concept of variables, pointers and how to use variable identifiers and pointers to refer to the variable. Students will learn the pointer variable declarations and initialization, direct and indirect referencing a variable using the pointer operators using the de-reference $(*)$ and address $(\&)$ operators. In addition, students will develop knowledge on passing and returning pointer from functions as well as use of pointers with functions.

7.2. Pointers

In C++, pointers are variables that store the memory addresses of other variables. In other words, a pointer is an object that contains a memory address. Very often this address is the location of another object (e.g., a variable). For example, if ***x*** contains the address of ***y***, then ***x*** is said to ***point to y***.

7.2.1. Address-of operator $(\&)$

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign $(\&)$, known as *address-of operator*. For example,

```
khulna = &dhaka;
```

This would assign the address of variable ***dhaka*** to ***khulna***. Here, by preceding the name of the variable ***dhaka*** with the *address-of operator* $(\&)$, the address of ***dhaka*** is assigned to ***khulna***, instead of the content.

The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that ***dhaka*** is placed during runtime in the memory address ***20022***. In this case, consider the following code fragment,

```
dhaka = 19;
khulna = &dhaka;
sylhet = dhaka;
```

The values contained in each variable after the execution of the above code fragment are shown in the Figure 7.1.

— In the first statement, we have assigned the value ***19*** to ***dhaka*** (a variable whose address in memory we assumed to be ***20022***).
— The second statement assigns ***khulna*** the address of ***dhaka***, which we have assumed to be ***20022***.
— Finally, the third statement, assigns the value contained in ***dhaka*** to ***sylhet***. This is a standard assignment operation, as already done many times in earlier laboratories.

The main difference between the second and third statements is the *address-of operator* $(\&)$.

Thus we can say that, variable that stores the address of another variable (e.g., ***khulna*** in the above case) is called a *pointer* in C++.

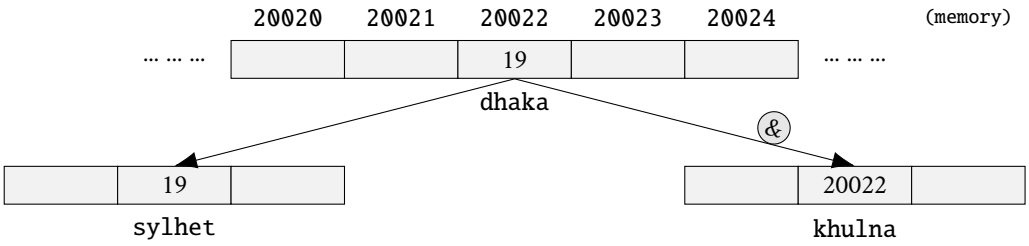


Figure 7.1. The address-of operator (&)

7.2.2. Dereference operator $(*)$

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* $(*)$. The operator itself can be read as *value pointed to by*.

To elaborate the idea of the deference operator, we extend the previous example with the following statement,

```
bbaria = *khulna;
```

The above statement assigns the value ***19*** to ***bbaria***, since ***khulna*** is ***20022***, and the value pointed to by ***20022*** (following the example of the address-of operator) is ***19***. The whole process is shown in the Figure 7.2. It is important to understand that ***khulna*** refers to the value ***20022***, while ****khulna*** (with an asterisk * preceding the identifer) refers to the value stored at address ***20022***, which in this case is ***19***.

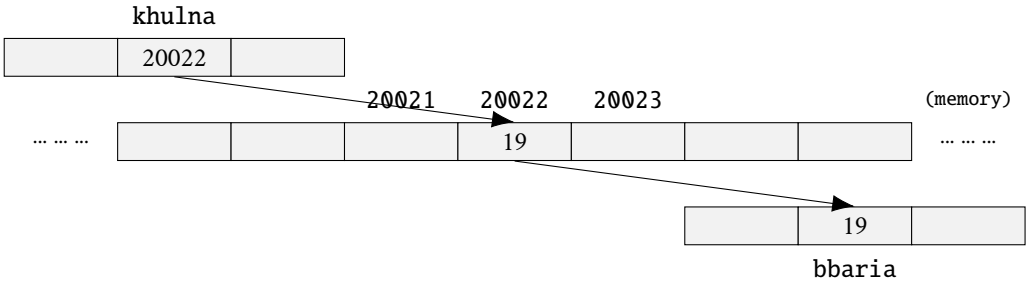


Figure 7.2. The dereference operator

7.2.3. Pointer declaration

The syntax of declaring a pointer is as follows.

```
data_type *pointer_name;
```

Here, ***data_type*** is the type of data pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to. And ***pointer_name*** is the name of the pointer variable.

Example 01

The following example of a C++ program demonstrates the C++ pointers.

```
#include <iostream>
using namespace std;
int main()
{
    int var = 5, *point_var;
    point_var = &var;
    cout << "var = " << var << endl;
    cout << "&var = " << &var << endl;
    cout << "pointVar = " << point_var << endl;
    cout << "*pointVar = " << *point_var << endl;
    return 0;
}
```

The console output will be as follows.

```
var = 5
&var = 0x7ffced7ef74c
pointVar = 0x7ffced7ef74c
*pointVar = 5
```

Example 02

The following example of a C++ program demonstrates to change the value pointed by pointers.

```
#include <iostream>
using namespace std;
int main()
{
    int var = 5, *point_var;
    point_var = &var;
    cout << "var = " << var << endl;
    cout << "*point_var = " << *point_var << endl << endl;
    cout << "Changing value of var to 7:" << endl;
    var = 7;
    cout << "var = " << var << endl;
    cout << "*point_var = " << *point_var << endl << endl;
    cout << "Changing value of *point_var to 16:" << endl;
    *point_var = 16;
    cout << "var = " << var << endl;
    cout << "*point_var = " << *point_var << endl;
    return 0;
}
```

The console output will be as follows.

```
var = 5
*point_var = 5

Changing value of var to 7:
var = 7
*point_var = 7
```

```
Changing value of *point_var to 16:
var = 16
*point_var = 16
```

7.3. Pointers and arrays

The concept of arrays is related to that of pointers. Pointers are variables that hold addresses of other variables. Not only can a pointer store the address of a single variable, it can also store the address of cells of an array. Consider the following example.

```
int *ptr;
int arr[5];
ptr = arr;
```

Here, ***ptr*** is a pointer variable while ***arr*** is an ***int*** array. The code $ptr = arr$; stores the address of the first element of the array in variable ***ptr***. Notice that we have used arr instead of $\&arr[0]$. This is because both are the same. So, the code below is the same as the code above.

```
int *ptr;
int arr[5];
ptr = &arr[0];
```

The addresses for the rest of the array elements are given by $\&arr[1]$, $\&arr[2]$, $\&arr[3]$, and $\&arr[4]$.

Example 03

The following example of a C++ program demonstrates the relation between arrays and pointers.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      int numbers[5];
6      int *p;
7      p = numbers;
8      *p = 10;
9      p++;
10     *p = 20;
11     p = &numbers[2];
12     *p = 30;
13     p = numbers+3;
14     *p = 40;
15     p = numbers;
16     *(p+4) = 50;
17     cout << "The numbers are: ";
18     for (int n=0; n<5; n++)
19     {
20         cout << numbers[n] << " ";
21     }
22     return 0;
23 }
```

The console output will be as follows.

```
The numbers are: 10 20 30 40 50
```

Example 04

The following example of a C++ program shows an array used as pointer.

```
#include <iostream>
using namespace std;
int main()
{
    float arr[5];
    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; ++i)
    {
        cin >> *(arr + i) ;
    }
    cout << "Displaying data: " << endl;
    for (int i = 0; i < 5; ++i)
    {
        cout << *(arr + i) << endl ;
    }
    return 0;
}
```

The console output will be as follows.

```
Enter 5 numbers: 12 23 34 45 56
Displaying data:
12
23
34
45
56
```

7.3.1. Dynamic arrays

A dynamic array is quite similar to a regular array, but its size is modifiable during program runtime. DynamArray elements occupy a contiguous block of memory. Once an array has been created, its size cannot be changed. However, a dynamic array is different. A dynamic array can expand its size even after it has been filled. The syntax of declaring a dynamic array is as follows.

```
pointer_variable = new data_type;
```

Here, ***pointer_variable*** is the name of the pointer variable, ***new*** is a keyword to create a dynamic array, and ***data_type*** must be a valid C++ data type. The keyword then returns a pointer to the first item. After creating the dynamic array, we can delete it using the ***delete*** keyword.

Example 05

The following example of a C++ program demonstrates the creation as well as deletion of a dynamic array.

```
#include <iostream>
using namespace std;
int main()
{
    int x, n;
    cout << "How many numbers will you type?" << endl;
    cin >> n;
    int *arr = new int(n);
    cout << "Enter " << n << " numbers:" << endl;
    for (x = 0; x < n; x++)
    {
        cin >> arr[x];
    }
    cout << "You typed: ";
    for (x = 0; x < n; x++)
    {
        cout << arr[x] << " ";
    }
    cout << endl;
    delete [] arr;
    return 0;
}
```

The console output will be as follows.

```
How many numbers will you type? 5
Enter 5 numbers:
11 22 33 44 55
You typed: 11 22 33 44 55
```

7.4. Pointers to functions

Like an array name, a function name is actually a constant pointer. You can think of its' value as the address of the code that implements the function. A pointer to a function is simply a pointer whose value is the address of the function name. Since that name is itself a pointer, a pointer to a function is just a pointer to a constant pointer.

The value of function pointers is that they help to define functions of functions. This is done by passing a function pointer as a parameter to another function.

Example 06

The following example of a C++ program demonstrates pointers to functions.

```
#include <iostream>
using namespace std;

int addition (int a, int b);
int subtraction (int a, int b);
int operation (int x, int y, int (*functocall)(int,int));

int main()
{
```

```
    int m,n;
    int (*minus)(int,int) = subtraction;
    m = operation (6, 7, addition);
    n = operation (30, m, minus);
    cout << "The result is: " << n;
    return 0;
}

int addition (int a, int b)
{
    return (a+b);
}

int subtraction (int a, int b)
{
    return (a-b);
}

int operation (int x, int y, int (*functocall)(int,int))
{
    int g;
    g = (*functocall)(x,y);
    return (g);
}
```

The console output will be as follows.

```
The result is: 17
```

Example 07

The following example of a C++ program demonstrates the passing by reference using pointers.

```
#include <iostream>
using namespace std;

void swap(int*, int*);

int main()
{
    int a = 10, b = 20;
    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    swap(&a, &b);
    cout << endl << "After swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}

void swap (int* n1, int* n2)
{
    int temp;
```

```
22      temp = *n1;
23      *n1 = *n2;
24      *n2 = temp;
25  }
```

The console output will be as follows.

```
Before swapping
a = 10
b = 20

After swapping
a = 20
b = 10
```

Post-laboratory Problems

1. Write and run a C++ program to test the following function that returns the sum of the floats pointed to by the first ***n*** pointers in the array ***p***.

```
float sum(float* p[], int n);
```

2. Write and run a C++ program that use pointers to swap four integer values.
3. Modify the program of Example 05 so that, it takes and prints values using the following two functions respectively.

```
void get(double *&a, int& n);
void print(double *a, int n);
```

4. Following is a function for the (indirect) Bubble Sort. Here, on each iteration of the inner loop, if the floats of adjacent pointers are out of order, then the pointers are swapped.

```
void sort(float* p[], int n)
{
    float* temp;
    for (int i = 1; i < n; i++)
        for (int j = 0; j < n-i; j++)
            if (*p[j] > *p[j+1])
            {
                temp = p[j];
                p[j] = p[j+1];
                p[j+1] = temp;
            }
}
```

Write and run a C++ program to test the above function.

Laboratory 8

Strings

8.1. Objective

This laboratory focuses on the concept of C-strings as well as the Standard C++ strings. Students will learn to apply the useful functions defined in the header files for C-strings and Standard C++ strings. In addition, students will develop knowledge on applying the concept of strings.

8.2. C-strings

A C-string (also called a character string) is a sequence of contiguous characters in memory terminated by the ***NULL*** character '**\0**'. For example:

```
char c[] = "AUST EEE";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character '**\0**' at the end by default.

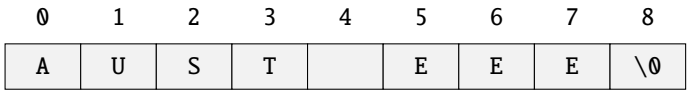


Figure 8.1. Memory Diagram

C-strings are accessed by variables of type ***char**** (pointer to char). For an example, if ***s*** has type ***char****, then, ***cout << s << endl;*** will print the characters stored in the memory beginning at the address ***s*** and ending with the first occurrence of ***NULL*** charater.

8.2.1. Initialization of C-strings

Example 01

The following example of a C++ program demonstrates the basic idea of the ***NULL*** character in strings.

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     char str[]="AUST";
6     for (int i = 0; i < 5; i++)
7     {
8         cout << "s[" << i << "] = " << str[i] << endl;
9     }
```

```
10 }
```

The console output will be as follows.

```
s[0] = A
s[1] = U
s[2] = S
s[3] = T
s[4] =
```

Example 02

The following example of a C++ program demonstrates the ways of initialization of C-string with string literal.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char a[]= "Dhaka" ;
6      cout << "The default string is: " << a << endl;
7      int max_length = 6; char b[max_length];
8      cout << "Enter a string not more than " << max_length << " characters: ";
9      cin.getline(b,max_length);
10     cout << "You entered: " << b << endl;
11 }
```

The console output will be as follows.

```
The default string is: Dhaka
Enter a string not more than 7 characters: Bangladesh
You entered: Bangla
```

In line 5 of the above code, the C-string is initialize with a string literal. However, in line 9, we use the ***cin.getline( )*** function with two parameters. The call ***cin.getline(str, n)*** reads up to ***n*** characters into ***str*** and ignores the rest. That is why, we got the output ***Bangla*** instead of ***Bangladesh***.

8.2.2. Standard C-string Functions

Example 03

The following example of a C++ program demonstrates the ***strlen( )*** function.

The ***strlen( )*** function takes a string as an argument and returns its length (i.e., number of characters in the string that precede the first occurrence of the ***NULL*** character). The returned value is of type ***size_t*** (the unsigned integer type).

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      char a[20] = "AUSTEEE";
6      char b[20] = {'A','U','S','T','E','E','E','\0'};
7      cout << "Length of string a: " << strlen(a) << endl;
8      cout << "Length of string b: " << strlen(b) << endl;
```

```
    char buffer[80];
    cout << "Enter string: ";
    cin >> buffer;
    cout << "Length of the entered string: " << strlen(buffer) << endl;
}
```

The console output will be as follows.

```
Length of string a: 7
Length of string b: 7
Enter string: Bangladesh
Length of the entered string: 10
```

Example 04

The following example of a C++ program demonstrates the ***strcpy( )*** function.

The ***strcpy( )*** function copies the string pointed by source (including the ***NULL*** character) to the destination. The ***strcpy( )*** function also returns the copied string.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char S1[] = "ABCDEFGH";
    char S2[] = "xyz";
    cout << "Before strcpy(S1, S2)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl << endl;
    strcpy(S1, S2);
    cout << "After strcpy(S1, S2)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl;
}
```

The console output will be as follows.

```
Before strcpy(S1, S2)
Content of S1: ABCDEFGH
Length of S1: 8
Content of S2: xyz
Length of S2: 3

After strcpy(S1, S2)
Content of S1: xyz
Length of S1: 3
Content of S2: xyz
Length of S2: 3
```

The effect of ***strcpy(S1, S2)*** can be visualized as shown in Figure 8.2. Since ***S2*** has a length of 3, ***strcpy(S1, S2)*** copies 4 bytes (including the ***NULL*** character), overwriting the first 4 characters of ***S1***. This changes the length of S1 to 3.

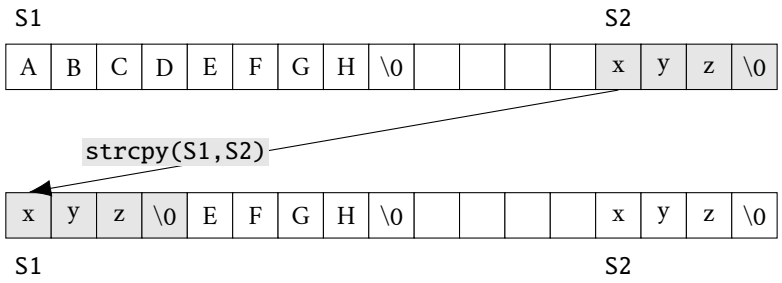


Figure 8.2. Illustration of Example 04

Example 05

The following example of a C++ program demonstrates the ***strncpy( )*** function.
The ***strncpy( )*** function copies first ***n*** characters from the string pointed by source to the destination.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char S1[] = "ABCDEFGH";
    char S2[] = "xyz";
    int n = 2;
    cout << "Before strncpy(S1, S2, n)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl << endl;
    strncpy(S1, S2, n);
    cout << "After strncpy(S1, S2, n)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl;
}
```

The console output will be as follows.

```
Before strncpy(S1, S2, n)
Content of S1: ABCDEFGH
Length of S1: 8
Content of S2: xyz
Length of S2: 3

After strncpy(S1, S2, n)
Content of S1: xyCDEFGH
```

```
Length of S1: 8
Content of S2: xyz
Length of S2: 3
```

The effect of ***strncpy(S1, S2, n)*** can be visualized as shown in Figure 8.3. Since ***n*** is 2 and ***S2*** has a length of 3, ***strncpy(S1, S2, n)*** copies 2 bytes (excluding the ***NULL*** character), overwriting the first 2 characters of ***S1***. This has no effect upon the length of ***S1*** which is 8.

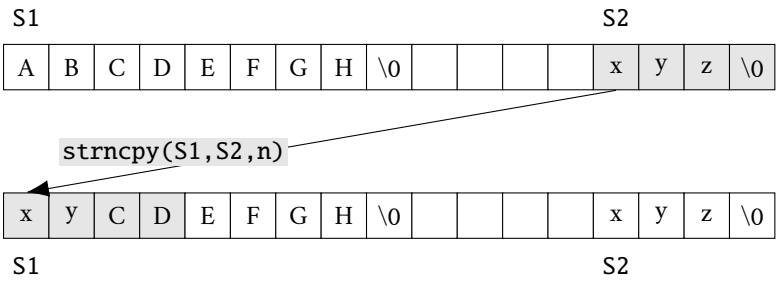


Figure 8.3. Illustration of Example 05

Example 06

The following example of a C++ program demonstrates the ***strcat( )*** function.

The ***strcat( )*** function contcatenates (joins) two strings. This function concatenates the destination string and the source string, and the result is stored in the destination string.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char S1[] = "ABCDEFGH";
    char S2[] = "xyz";
    cout << "Before strcat(S1, S2)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl << endl;
    strcat(S1, S2);
    cout << "After strcat(S1, S2)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl;
}
```

The console output will be as follows.

```
Before strcat(S1, S2)
Content of S1: ABCDEFGH
Length of S1: 8
Content of S2: xyz
```

```
Length of S2: 3

After strcat(S1, S2)
Content of S1: ABCDEFGHxyz
Length of S1: 11
Content of S2: xyz
Length of S2: 3
```

The effect of ***strcat(S1, S2)*** can be visualized as shown in Figure 8.4. Since ***S2*** has length 3, ***strcat(S1, S2)*** copies 4 bytes (including the ***NULL*** character), overwriting the ***NULL*** characters of S1 and its following 3 bytes. The length of ***S1*** is increased to 11.

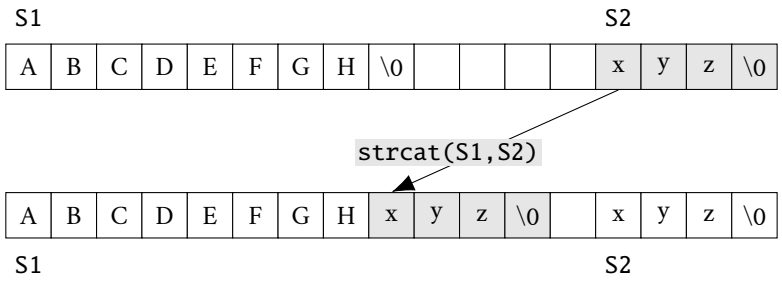


Figure 8.4. Illustration of Example 06

Example 07

The following example of a C++ program demonstrates the ***strncat( )*** function.

This function appends not more than ***n*** characters from the string pointed to by source to the end of the string pointed to by destination plus a terminating ***NULL*** character. The initial character of the source string overwrites the ***NULL*** character present at the end of destination string.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char S1[] = "ABCDEFGH";
    char S2[] = "xyz";
    int n = 2;
    cout << "Before strncat(S1, S2, n)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl << endl;
    strncat(S1, S2, n);
    cout << "After strncat(S1, S2, n)" << endl;
    cout << "Content of S1: " << S1 << endl;
    cout << "Length of S1: " << strlen(S1) << endl;
    cout << "Content of S2: " << S2 << endl;
    cout << "Length of S2: " << strlen(S2) << endl;
}
```

The console output will be as follows.

```
Before strncat(S1, S2, n)
Content of S1: ABCDEFGH
Length of S1: 8
Content of S2: xyz
Length of S2: 3

After strncat(S1, S2, n)
Content of S1: ABCDEFGHxy
Length of S1: 10
Content of S2: xyz
Length of S2: 3
```

The effect of ***strcat(S1, S2)*** can be visualized as shown in Figure 8.5. Since, ***n*** is 2 and ***S2*** has length 3, ***strncat(S1, S2, n)*** copies 2 bytes overwriting the ***NULL*** character of ***S1*** and the byte that follows it. Then it puts the NULL character in the next byte to complete the C-string ***S1***. This increases its length to 10.

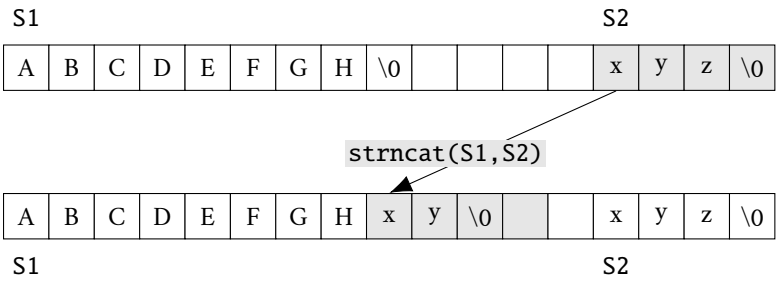


Figure 8.5. Illustration of Example 07

Example 08

The following example of a C++ program demonstrates the ***strcmp( )*** function.

The function takes two parameters, e.g., ***S1*** and ***S2***, where both of them are string. This function compares ***S1*** and ***S2*** character by character. If the strings are equal, the function returns 0. If the first non-matching character in ***S1*** is greater (in ASCII) than that of ***S2***, the function returns a positive integer. If the first non-matching character in ***S1*** is lower (in ASCII) than that of ***S2***, the function returns a negetaive integer.

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4  int main()
5  {
6      char S1[] = "AUST", S2[] = "aust", S3[] = "AUST";
7      int result;
8      result = strcmp(S1, S2);
9      cout << "strcmp(S1, S2) = " << result << endl;
10     result = strcmp(S2, S3);
11     cout << "strcmp(S2, S3) = " << result << endl;
12     result = strcmp(S3, S1);
```

```
13      cout << "strcmp(S3, S1) = " << result << endl;
14  }
```

The console output will be as follows.

```
strcmp(S1, S2) = -32
strcmp(S2, S3) = 32
strcmp(S3, S1) = 0
```

In the program, strings ***S1*** and ***S2*** are not equal. Hence, the result is a non-zero integer. Hoever, strings ***S1*** and ***S3*** are equal. Hence, the result is 0.

Example 09

The following example of a C++ program demonstrates the ***strtok()*** function.

This function splits a string by some delimiter. Splitting a string is a very common task. For example, we have a comma separated list of items from a file and we want individual items in an array.

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4  int main()
5  {
6      char str[] = " Barishal, Chittagong, Dhaka, Khulna, Rajshahi, \
7                     Rangpur, Mymensingh, Sylhet";
8      char* token = strtok(str, ",");
9      while (token != NULL)
10     {
11         cout << token << endl;
12         token = strtok(NULL, ",");
13     }
14 }
```

The console output will be as follows.

```
 Barishal
 Chittagong
 Dhaka
 Khulna
 Rajshahi
 Rangpur
 Mymensingh
 Sylhet
```

Example 10

The following example of a C++ program demonstrates a user defined function to copy one string into another string without using the predefined ***strcpy( )*** function.

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4  char* stringcopy (char* str1, char* str2)
5  {
6      char* p = str1;
```

```
7       for( ; *str2; p++, str2++)
8       {
9           *p=*str2;
10      }
11      *p= NULL;
12      return str1;
13  }
14  int main()
15  {
16      char S1[] = "ABCDEFGH", S2[] = "xyz" ;
17      cout << "Before stringcopy(S1, S2)" << endl;
18      cout << "Content of S1: " << S1 << endl;
19      cout << "Length of S1: " << strlen(S1) << endl;
20      cout << "Content of S2: " << S2 << endl;
21      cout << "Length of S2: " << strlen(S2) << endl << endl;
22      stringcopy(S1, S2);
23      cout << "After stringcopy(S1, S2)" << endl;
24      cout << "Content of S1: " << S1 << endl;
25      cout << "Length of S1: " << strlen(S1) << endl;
26      cout << "Content of S2: " << S2 << endl;
27      cout << "Length of S2: " << strlen(S2) << endl;
28  }
```

The console output will be as follows.

```
Before stringcopy(S1, S2)
Content of S1: ABCDEFGH
Length of S1: 8
Content of S2: xyz
Length of S2: 3

After stringcopy(S1, S2)
Content of S1: xyz
Length of S1: 3
Content of S2: xyz
Length of S2: 3
```

8.3. Standard C++ Strings

The classic C-strings described in the previous section are an important part of C++. They provide a very efficient means for fast data processing. However, as with ordinary arrays, the efficiency of C-strings comes at a price - the risk of run-time errors, resulting primarily from their dependency upon the use of the NULL character as a string terminator. Standard C++ strings provide a safe alternative to C-strings. By encapsulating the length of the string with the string itself, there is no direct reliance on string terminators.

Standard C++ defines its string type in the ***<string>*** header file. Objects of type string can be declared and initialized in several ways:

```
string s1;                  // s1 contains 0 characters
string s2 = "Chittagong";   // s2 contains 8 characters
string s3(60, '*');      // s3 contains 60 asterisks
string s4 = s3;          // s4 contains 60 asterisks
string s5(s2, 4, 2);        // s5 is the 2-character string "ta"
```

Example 11

The following example of a C++ program counts the number of characters in a string.

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
    string small, large;
    small = "I am a student.";
    large = "I study in the EEE department of AUST.";
    cout << "The small string has " << small.length()
            << " characters." << endl;
    cout << "The large string has " << large.length()
            << " characters." << endl;
}
```

The console output will be as follows.

```
The small string has 15 characters.
The large string has 38 characters.
```

Example 12

The following example of a C++ program transforms the lowercase characters into the uppercase characters. This example also demonstrates to access the individual characters in a string.

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
    string s;
    cout << "Enter an string with lowercase characters: ";
    getline (cin, s);
    for (int i = 0; s[i] != '\0'; i++)
    {
        if(s[i] >= 'a' && s[i] <= 'z')
            s[i] = s[i] - ('a'-'A');
    }
    cout<< "After transforming into uppercase characters: "<< s << endl;
}
```

The console output will be as follows.

```
Enter an string with lowercase characters: Hello World.
After transforming into uppercase characters: HELLO WORLD.
```

Example 13

The following example of a C++ program compares two strings.

```
#include <iostream>
#include <string>
```

```
using namespace std;
int main( )
{
    string my_name = "Kishor";
    string user_name;
    while (true)
    {
        cout << "Enter your name (or 'quit' to exit): ";
        getline (cin, user_name);
        if (user_name == "Musa")
        {
            cout << "Hi, Musa!  Welcome back!" << endl;
        }
        else if (user_name == "quit")
        {
            cout << endl;
            break;
        }
        else if (user_name != my_name)
        {
            cout << "Hello, " << user_name << "!" << endl;
        }
        else
        {
            cout << "Oh, its you, " << my_name << "!" << endl;
        }
    }
}
```

The console output will be as follows.

```
Enter your name (or 'quit' to exit): Kishor
Oh, its you, Kishor!
Enter your name (or 'quit' to exit): Musa
Hi, Musa!  Welcome back!
Enter your name (or 'quit' to exit): Robin
Hello, Robin!
Enter your name (or 'quit' to exit): quit
```

To compare two strings for equality the '==' and '**!=**' operators are used in the above example. You can use '<', '<=', '>', and '>=' to compare strings as well. These operators compare strings lexicographically, character by character and are case-sensitive. The following comparisons all evaluate to true: ***"A" < "B"***, ***"App" < "Apple"***, ***"help" > "hello"***, ***"Apple" < "apple"***. The last one might be a bit confusing, but the ASCII value for ***A*** is 65, and comes before ***a***, whose ASCII value is 97. So ***"Apple"*** comes before ***"apple"*** (or, for that matter, any other word that starts with a lower-case letter).

Example 15

The following example of a C++ program to append two or more strings.

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
```

```
    string firstname, middlename, lastname, fullname;
    cout << "Enter your first name: ";
    getline(cin, firstname);
    cout << "Enter your middle name: ";
    getline(cin, middlename);
    cout << "Enter your last name: ";
    getline(cin, lastname);
    fullname = firstname + " " + middlename + " " + lastname;
    cout << "Your full name: " << fullname << endl;
    fullname += ", BSc in EEE.";
    cout << "Your full name and qualification: " << fullname << endl;
}
```

The console output will be as follows.

```
Enter your first name: Kazi
Enter your middle name: Nazrul
Enter your last name: Islam
Your full name: Kazi Nazrul Islam
Your full name and qualification: Kazi Nazrul Islam, BSc in EEE.
```

Example 16

The following example of a C++ program to to search one or more characters within a string.

The string member function ***find*** is used to search within a string for a particular string or character. A sample usage such as ***str.find(key)*** searches the receiver string ***str*** for the ***key***. The parameter ***key*** can either be a string or a character. There is an optional second integer argument to find which allows you to specify the starting position; when this argument is not given, 0 is assumed. Thus, ***str.find(key, n)*** starts at position ***n*** within ***str*** and will attempt to find ***key*** from that point on.

```
#include <iostream>
#include <string>
using namespace std;
int main( )
{
string sentence = "Yes, we went to class after we left the dorm.";
int first_we = sentence.find("we");
int second_we = sentence.find("we", first_we + 1);
int third_we = sentence.find("we", second_we + 1);
int c_pos = sentence.find('c');
int d_pos = sentence.find('d');
cout << "Position of first we: " << first_we << endl;
cout << "Position of second we: " << second_we << endl;
cout << "Position of third we: " << third_we << endl;
cout << "Position of c: " << c_pos << endl;
cout << "Position of d: " << d_pos << endl;
}
```

The console output will be as follows.

```
Position of first we: 5
Position of second we: 8
Position of third we: 28
Position of c: 16
```

```
Position of d: 40
```

Post-laboratory Problems

1. Write and run a C++ program to test the ***strncmp( )*** function.
2. Write and run a C++ program that does exactly the same task of ***strncpy( )*** without using the default ***strncpy( )*** function.
3. Write and run a C++ program that does exactly the same task of ***strcat( )*** without using the default function ***strcat( )***.
4. Write and run a C++ program that counts the vowels of a given string.
5. Write and run a C++ program that reads one line of text and then prints it with all its blanks removed.
6. Write and run a C++ program that reads one line of text and then prints the line in reverse order. Following is an example of the console output.

```
Input: Today is Friday.
Output: .yadirF si yadoT
```

Laboratory 9

Structures and Classes

9.1. Objective

This laboratory focuses on the concept of the Object Oriented Programming (OOP). Students will learn to apply the OOP concept by introducing Structures and Classes. In addition, students will develop ability to define as well as apply Structure and Class for solving problems.

9.2. Structure

Structure is a collection of variables of different data types under a single name. For example, a banking app wants to store some information about one person, such as, name, account number and account balance. As a programmer you can easily create different variables name, i.e., name, number, balance to store these information separately. However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person, i.e., name1, number1, balane1, name2, number2, balance2, name3, number3, balance3, etc. It can be easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it is going to be a daunting task. A better approach will be to have a collection of all related information under a single name ***Person***, and use it for every person. Now, the code looks much cleaner, readable and efficient as well. This collection of all related information under a single name ***Person*** is a structure.

9.2.1. Declaring and defining a structure in C++

The ***struct*** keyword defines a structure type followed by an identifier (name of the structure). Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example,

```
struct Bank
{
    string name;
    int number;
    double balance;
};
```

Here a structure ***Bank*** is defined which has three members – ***name***, ***number*** and ***balance***. When a structure is created, no memory is allocated. The structure definition is only the blueprint for the creating of variables. You can imagine it as a data type. When you define an integer as ***int n;*** the ***int*** specifies that, variable ***n*** can hold integer element only. Similarly, structure definition only specifies that, what property a structure variable holds when it is defined.

Once you declare a structure ***Bank*** as above, you can define a structure object as ***Bank account;***. Thus, an object ***account*** is defined which is of type structure ***Bank***. When object is defined, only then the required memory is allocated by the compiler.

9.2.2. Accessing members of a structure

The members of structure object is accessed using a dot **(.)** operator. Suppose, you want to access ***balance*** of the object ***account*** and assign a value of ***50.5*** to it. You can perform this task by using following code below.

```
Bank account;
account.balance = 50.5;
```

Example 01

The following example of a C++ program demonstrates the basic idea of the structure.

```
#include <iostream>
#include<string>
using namespace std;
struct Bank
{
    string name;
    int number;
    double balance;
};
int main()
{
    Bank account;
    cout << "Enter Full name: ";
    getline(cin, account.name);
    cout << "Enter account number: ";
    cin >> account.number;
    cout << "Enter account balance: ";
    cin >> account.balance;
    cout << endl << "Displaying Information." << endl;
    cout << "Full name: " << account.name << endl;
    cout << "Account number: " << account.number << endl;
    cout << "Account balance: " << account.balance << " BDT";
    return 0;
}
```

The console output will be as follows.

```
Enter Full name: Musa Aman
Enter account number: 01020304
Enter account balance: 50.5

Displaying Information.
Full name: Musa Aman
Account number: 1020304
Account balance: 50.5 BDT
```

Here a structure ***Bank*** is defined which has three members – ***name***, ***number*** and ***balance***. Inside the ***main( )*** function, a structure object ***account*** is defined. Then, the user is asked to enter information and data entered by user is displayed.

9.2.3. Structure to function *(Passing by value)*

Here, you can pass structure object as an argument to a function similar to passing a variable.

Example 02

The following example of a C++ program demonstrates to pass any structure to a function by the *Passing by Value* method.

```
1  #include <iostream>
2  #include<string>
3  using namespace std;
4  struct Bank
5  {
6      string name;
7      int number;
8      double balance;
9  };
10 Bank getData(Bank);
11 void displayData(Bank);
12 int main()
13 {
14     Bank account;
15     account = get_data(account);
16     display_data(account);
17     return 0;
18 }
19 Bank get_data(Bank account)
20 {
21     cout << "Enter Full name: ";
22     getline(cin, account.name);
23     cout << "Enter account number: ";
24     cin >> account.number;
25     cout << "Enter account balance: ";
26     cin >> account.balance;
27     return account;
28 }
29 void display_data(Bank account)
30 {
31     cout << endl << "Displaying Information." << endl;
32     cout << "Full name: " << account.name << endl;
33     cout << "Account number: " << account.number << endl;
34     cout << "Account balance: " << account.balance << " BDT";
35 }
```

The console output will be as follows.

```
Enter Full name: Musa Aman
Enter account number: 01020304
Enter account balance: 50.5

Displaying Information.
Full name: Musa Aman
Account number: 1020304
Account balance: 50.5 BDT
```

In the line 15, the structure object ***account*** is passed to ***get_data( )*** function which takes input from user which is then returned to ***main( )*** function. *(The value of all members of a structure variable can be assigned*

to another structure using assignment operator (=), if both structure variables are of same type. You don't need to manually assign each members.)

Then, the object ***account*** is to passed to a function using ***display_data(account)***; in line 16. The return type of ***display_data( )*** is void and a single argument of type structure is passed. Then the members of structure ***Bank*** is displayed from this function.

9.2.4. Structure to function *(Passing by reference)*

In passing by reference, the address of a structure object is passed to a function. In this, if we change the object which is inside the function, the original structure object which is used for calling the function changes. This was not the case in calling by value.

Example 03

The following example of a C++ program elaborates the passing by reference method.

```
1  #include <iostream>
2  #include<string>
3  using namespace std;
4  struct Bank
5  {
6      string name;
7      int number;
8      double balance;
9  };
10 void display_data(Bank *acc);
11 int main()
12 {
13     Bank account;
14     cout << "Enter name: ";
15     cin >> account.name;
16     cout << "Enter account number: ";
17     cin >> account.number;
18     cout << "Enter account balance: ";
19     cin >> account.balance;
20     display_data(&account);
21     return 0;
22 }
23 void display_data(Bank *acc)
24 {
25     cout << endl << "Displaying Information. " << endl;
26     cout << "Name: " << acc->name << endl;
27     cout << "Account number: " << acc->number << endl;
28     cout << "Account balance: " << acc->balance << endl;
29 }
```

The console output will be as follows.

```
Enter name: Kishor
Enter account number: 010809
Enter account balance: 12000

Displaying Information.
Name: Kishor
```

```
Account number: 10809
Account balance: 12000
```

This case is similar to the previous one (*passing by value)*, the only difference is that this time, we are passing the address of the structure object to the function. While declaring the function, we passed the pointer of the copy ***acc*** of the structure object ***account*** in its parameter. In the function, we accessed the members of the pointer using '->' sign.

9.2.5. Array of structures

You can also make an array of structures. Since structures are types, they can also be used as the type of arrays to construct tables or databases of them.

Example 04

The following example of a C++ program creates a structure and use object with array of size 10 to store information of 10 persons. Using ***for*** loop, the program takes the information of 10 persons from the user and displays it on the screen.

```
#include <iostream>
#include<string>
using namespace std;
struct Bank
{
    string name;
    int number;
    double balance;
};
int main()
{
    Bank account[10];
    for(int i=0; i<10; i++)
    {
        cout << "Enter name: ";
        cin >> account[i].name;
        cout << "Enter account number: ";
        cin >> account[i].number;
        cout << "Enter account balance: ";
        cin >> account[i].balance;
    }

    cout << endl << "Displaying Information." << endl << endl;
    for(int i=0; i<10; i++)
    {
        cout << "Name: " << account[i].name << endl;
        cout << "Account number: " << account[i].number << endl;
        cout << "Account balance: " << account[i].balance << " BDT" << endl;
    }
    return 0;
}
```

The console output will be as follows.

```
Enter name: Kishor
```

```
Enter account number: 001
Enter account balance: 11000.50
Enter name: Musa
Enter account number: 002
Enter account balance: 22000.75
Enter name: Robin
Enter account number: 003
Enter account balance: 33000.25
                .
                .
                .
                .
                .
                .
                .

Displaying Information.

Name:: Kishor
Account number: 001
Account balance: 11000.50 BDT
Name:: Musa
Account number: 002
Account balance: 22000.75 BDT
Name:: Robin
Account number: 003
Account balance: 33000.25 BDT
                .
                .
                .
                .
                .
                .
                .
```

9.3. Class

Classes are an expanded concept of data structures. Like data structures, they can contain data members, but they can also contain functions as members. An object is an instantiation of a class. For a clear understanding, in terms of variables, a class would be the type, and an object would be the variable.

In other words, A class is a blueprint for the object. You can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

9.3.1. Declaring and defining a class in C++

A class is defined in C++ using keyword ***class*** followed by the name of the class. The body of the class is defined inside the curly brackets and terminated by a semicolon at the end. For example,

```
class Rectangle
{
```

```
        int length;
        int breadth;
    public:
        void set_length(int L);
        void set_breadth(int B);
        int get_area();
};
```

We have defined our own class named ***Rectangle***. Here, ***class*** is a keyword which means that ***Rectangle*** is a class. Inside the ***Rectangle*** class, we declared two variables and three functions. These variables and functions belong to the class ***Rectangle*** since these are declared inside the class and thus are called members of the class. There are two types of members in a class – data members (e.g., ***length*** and ***breadth***) and member functions, e.g., ***set_length( )***, ***set_breadth( )*** and ***get_area( )***.

We declared the member functions of the class as ***public***. Here, ***public*** is an access modifier which allows the members of a class to be accessed directly from outside the class. The access modifiers decide how the members of a class can be accessed. Like ***public***, there are two more modifiers – ***private*** and ***protected***. In the above code, the two data members of the class are declared as ***private.*** When we declare any class member as ***public***, that variable becomes available everywhere in the program, even outside the function in which it was declared. The member declared as ***private*** can only be accessed inside the class in which it is declared. Thus, the object of the class cannot directly access its members. By default, all the members of a class are ***private***.

9.3.2. Accessing members of a class

The members of class object is accessed using a dot **(.)** operator. Suppose, you want to access the memeber ***set_length( )*** of the object ***rect*** and pass a value of *7* to it. You can perform this task by using following code below.

```
Rectangle rect;
rect.set_length(7);
```

Example 05

The following example of a C++ program demonstrates the basic idea of the class.

```
1  #include <iostream>
2  #include<string>
3  using namespace std;
4  class Rectangle
5  {
6          int length;
7          int breadth;
8      public:
9          void set_length(int L);
10         void set_breadth(int B);
11         int get_area();
12 };
13 void Rectangle::set_length(int L)
14 {
15     length = L;
16 }
17 void Rectangle::set_breadth(int B)
18 {
```

```
19      breadth = B;
20  }
21  int Rectangle::get_area()
22  {
23  return length * breadth;
24  }
25  int main()
26  {
27      Rectangle rect;
28      rect.set_length(7);
29      rect.set_breadth(4);
30      int area = rect.get_area();
31      cout << "Area of the rectangle : " << area << endl;
32      return 0;
33  }
```

The console output will be as follows.

```
Area of the rectangle: 28
```

While defining the member functions, we have written ***Rectangle::*** before the function name. This is to tell the compiler that the function belongs to the class ***Rectangle***.

In line 28, ***rect.set_length(7);*** statement calls the function ***set_length( )*** with the parameter value *7*. To call any function, we use dot **(.)** after the object and then call that function. Since ***rect*** is an object of the ***Rectangle*** class, therefore, ***rect.set_length( )*** calls the function ***set_length( )*** of ***Rectangle*** class for ***rect***. This sets the value of length as 7 for ***rect***.

Similarly, in line 29, ***rect.set_breadth(4)*** calls the function ***set_breadth( )*** and sets the value of breadth as ***4***.

In line 30, ***rect*** calls the function ***get_area()*** which returns ***length * breadth*** which is ***28*** (since the value of length is 7 and that of breadth is 4). This value then gets assigned to the variable ***area***.

9.3.3. Constructor

What would happen in the previous example if we called the member function ***get_area( )*** before having called ***set_length( )*** and ***set_breadth( )***? An undetermined result, since the members ***length*** and ***breadth*** had never been assigned a value. In order to avoid that, a class can include a special function called its constructor, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.

In other words, a constructor is a special type of member function that is called automatically when an object is created. In C++, a constructor has the same name as that of the class and it does not have a return type.

Example 06

The following example of a C++ program demonstrates the basic idea of constructor.

```
1  #include <iostream>
2  #include<string>
3  using namespace std;
4  class Rectangle
5  {
6          int length;
7          int breadth;
```

```
    public:
        Rectangle(int, int);
        int get_area();
};
Rectangle::Rectangle(int L, int B)
{
    length = L;
    breadth = B;
}
int Rectangle::get_area()
{
return length * breadth;
}
int main()
{
    Rectangle rect(7, 4);
    int area = rect.get_area();
    cout << "Area of the rectangle : " << area << endl;
    return 0;
}
```

The console output will be as follows.

```
Area of the rectangle: 28
```

In this example when you created the object ***rect*** of class ***Rectangle***, the constructor ***Rectangle( )*** automatically got called and initialized the data members for the object ***rect***. It initialized the ***length*** and ***breadth*** of ***rect*** to 7 and 4 respectively.

When the constructor was called, ***length*** and ***breadth*** were created and then in the body of the constructor, these member variables were assigned values.

9.3.4. Overloading constructors

Like any other function, a constructor can also be overloaded with diferent versions taking diferent parameters: with a diferent number of parameters and/or parameters of diferent types. The compiler will automatically call the one whose parameters match the arguments.

Example 06

The following example of a C++ program demonstrates the basic idea of overloading constructors.

```
#include <iostream>
#include<string>
using namespace std;
class Rectangle
{
        int length;
        int breadth;
    public:
        Rectangle();
        Rectangle(int, int);
        int get_area();
};
Rectangle::Rectangle()
```

```
{
length = 10;
breadth = 3;
}
Rectangle::Rectangle(int L, int B)
{
    length = L;
    breadth = B;
}
int Rectangle::get_area()
{
return length * breadth;
}
int main()
{
    Rectangle rect_A(7, 4), rect_B(2, 10), rect_C;
    int area;
    area = rect_A.get_area();
    cout << "Area of the rectangle A: " << area << endl;
    area = rect_B.get_area();
    cout << "Area of the rectangle B: " << area << endl;
    area = rect_C.get_area();
    cout << "Area of the rectangle C: " << area << endl;
    return 0;
}
```

The console output will be as follows.

```
Area of the rectangle A: 28
Area of the rectangle B: 20
Area of the rectangle C: 30
```

In the above example, three objects of class Rectangle are constructed: ***rect_A***, ***rect_B*** and ***rect_C***. Here, ***rect_A***, and ***rect_B*** both are constructed with two arguments. But this example also introduces a special kind constructor – the "default constructor". The default constructor is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments. In the example above, the default constructor is called for ***rect_C***.

Post-laboratory Problems

1. Write and run a C++ program that converts a number entered in Roman numerals to decimal. Your program should consist of a class, say, ***roman_type***. An object of type ***roman_type*** should do the following.
   a) Store the number as a Roman numeral.
   b) Convert and store the number into decimal form.
   c) Print the number as a Roman numeral or decimal number as requested by the user. The decimal values of the Roman numerals are:

```
M   1000
D   500
C   100
L   50
X   10
```

```
V   5
I   1
```

d) Test your program using the following Roman numerals:
   i. MCXIV
   ii. CCCLIX
   iii. MDCLXVI.

2. Write and run a C++ program to design and implement a class ***day_type*** that implements the day of the week in a program. The class ***day_type*** should store the day, such as, Sun for Sunday. The program should be able to perform the following operations on an object of type ***day_type.***
   a) Set the day.
   b) Print the day.
   c) Return the day.
   d) Return the next day.
   e) Return the previous day.
   f) Calculate and return the day by adding certain days to the current day. For example, if the current day is Monday and we add 4 days, the day to be returned is Friday. Similarly, if today is Tuesday and we add 13 days, the day to be returned is Monday.
   g) Add the appropriate constructors.
3. Write and run a C++ program to implement a ***Time*** class. Each object of this class will represent a specific time of day, storing the hours, minutes, and seconds as integers. Include a constructor, access functions, a function ***advance(int h, int m, int s)*** to advance the current time of an existing object, a function ***reset(int h, int m, int s)*** to reset the current time of an existing object, and a ***print( )*** function.

Laboratory 10

File I/O and Vector

10.1. Objective

This laboratory discusses the file processing in C++. Students will learn to open and close files as well as to read and write on files. Apart from that, this laboratory also focuses on the standard C++ vectors. Students will be familiarize with passing the vector to function as well as some necessary member functions which makes adding and deleting elements from the vector easier.

10.2. File I/O

When a program runs, the data is in the memory but when it ends or the computer shuts down, it gets lost. To keep data permanently, we need to write it in a file. For this purpose, we will use ***fstream***, which is another C++ standard library like ***iostream*** and is used to read and write on files. Following are the data types used for file handling from the ***fstream*** library.

Table 10.1. Data types used for file handling from the ***fstream*** library

Data type	Description
ofstream	It is used to create files and write on files.
ifstream	It is used to read from files.
fstream	It can perform the function of both ***ofstream*** and ***ifstream*** which means it can create files, write on files, and read from files.

10.2.1. Creating and opening a file

We need to tell the computer the purpose of opening the file, e.g., to write on the file, to read from the file, etc. Following are the different modes in which we can open a file.

Table 10.2. Modes for opening the file

Mode	Description
ios::app	opens a text file for appending. (appending means to add text at the end).
ios::ate	opens a file for output and move the read/write control to the end of the file.
ios::in	opens a text file for reading.
ios::out	opens a text file for writing.
ios::trunc	truncates the content before opening a file, if file exists.

Example 01

The following example of a C++ program demonstrates how to open a file, then write and read that file and, then close that file.

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char text[2000];
    fstream file;
    file.open ("example.txt", ios::out | ios::in );
    cout << "Write text to be written on file:" << endl;
    cin.getline(text, sizeof(text));
    file << text << endl; // Writing on file
    file >> text;       // Reading from file
    cout << endl << "The written text is:" << endl;
    cout << text << endl;
    file.close();
    return 0;
}
```

The console output will be as follows.

```
Write text to be written on file:
Ahsanullah University of Science and Technology

The written text is:
Ahsanullah University of Science and Technology
```

We have opened the file ***example.txt*** for both reading and writing purposes. Therefore, you must create the ***example.txt*** file in your working directory. We can also open the file for both reading and writing purposes. You should remember that, C++ automatically close and release all the allocated memory. But a programmer should always close all the opened files. We use << and >> to write and read from the file *'example.txt'* respectively. Now open the file ***example.txt*** and check the content.

Example 02

The following example of a C++ program demonstrates how to open a text file for appending. That means, this example adds new text at the end of existing text of the ***example.txt*** file in your working directory.

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char text[2000];
    fstream file;
    file.open ("example.txt", ios::app);
    cout << "Write the text to be added on the file:" << endl;
    cin.getline(text, sizeof(text));
    file << text << endl;
    file.close();
    return 0;
}
```

The console output will be as follows.

```
Write the text to be added on the file::
Dhaka, Bangladesh
```

Now check the ***example.txt*** file for the change you have just made.

Example 03

The following example of a C++ program demonstrates reads words from the external file named ***input.txt***, capitalizes them, and then writes them to the external file named ***output.txt***.

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    ifstream infile("input.txt");
    ofstream outfile("output.txt");
    string word;
    char c;
    while (infile >> word)
    {
        if (word[0] >= 'a' && word[0] <= 'z')
        {
            word[0] += 'A' - 'a';
        }
        outfile << word;
        infile.get(c);
        outfile.put(c);
    }
    return 0;
}
```

Now check the ***output.txt*** file for the change you have just made.

10.3. Vector

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators (similar to pointer). In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

10.3.1. Declaration of vector

The declaration syntax of vector is the same as that of array, with the difference that you do not need to specify the array length along with the data type as shown below.

```
vector<datatype> array_name;
```

You need to include the ***<vector>*** header in our program. Now, look at the declaration of a vector named ***marks*** of type ***int*** to store the marks of students.

```
vector<int> marks;
```

10.3.2. Initialization of vector

The initialization of an vector is also the same as that of array. We initialize an vector by the following way.

```
vector<int> marks = {50, 45, 47, 65, 80};
```

We can also assign values to the vector after declaration as shown below.

```
vector<int> marks;
marks = {50, 45, 47, 65, 80};
```

In the above declarations, we stored the marks of 5 students in a vector named ***marks***. Since we did not declare the array length, so the length of ***marks*** became equal to the number of values it was initialized with. Now, we may change the number of students, i.e. either store the marks of more students or remove the marks of some students.

10.3.3. Functions associated with the vector

Table 10.3. Iterator related functions associated with the vector

Function	Description
begin()	Returns an iterator pointing to the first element in the vector
end()	Returns an iterator pointing to the theoretical element that follows the last element in the vector
rbegin()	Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
rend()	Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
cbegin()	Returns a constant iterator pointing to the first element in the vector.
cend()	Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
crbegin()	Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
crend()	Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

Example 04

The following example of a C++ program demonstrates the above iterators associated with the vector.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6      vector<int> v1;
7      for (int i = 1; i <= 5; i++)
8      {
```

```
        v1.push_back(i);
    }
    cout << "Output of begin and end: \t";
    for (auto i = v1.begin(); i != v1.end(); ++i)
    {
        cout << *i << " ";
    }
    cout << endl << "Output of cbegin and cend: \t";
    for (auto i = v1.cbegin(); i != v1.cend(); ++i)
    {
        cout << *i << " ";
    }
    cout << endl << "Output of rbegin and rend: \t";
    for (auto ir = v1.rbegin(); ir != v1.rend(); ++ir)
    {
        cout << *ir << " ";
    }
    cout << endl << "Output of crbegin and crend: \t";
    for (auto ir = v1.crbegin(); ir != v1.crend(); ++ir)
    {
        cout << *ir << " ";
    }
    return 0;
}
```

The console output will be as follows.

```
Output of begin and end:        1 2 3 4 5
Output of cbegin and cend:      1 2 3 4 5
Output of rbegin and rend:      5 4 3 2 1
Output of crbegin and crend:    5 4 3 2 1
```

Table 10.4. Capacity related functions associated with the vector

Function	Description
size()	Returns the number of elements in the vector.
max_size()	Returns the maximum number of elements that the vector can hold.
capacity()	Returns the size of the storage space currently allocated to the vector expressed as number of elements.
resize(n)	Resizes the container so that it contains 'n' elements.
empty()	Returns whether the container is empty.
shrink_to_fit()	Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
reserve()	Requests that the vector capacity be at least enough to contain n elements.

Example 05

The following example of a C++ program demonstrates the above capacity function associated with the vector.

```
#include <iostream>
#include <vector>
using namespace std;
```

```
int main()
{
    vector<int> v1;
    for (int i = 1; i <= 5; i++)
    {
        v1.push_back(i);
    }
    cout << "Size : " << v1.size();
    cout << endl << "Capacity : " << v1.capacity();
    cout << endl << "Max_Size : " << v1.max_size();
    v1.resize(4);
    cout << endl << "Size : " << v1.size();
    if (v1.empty() == false)
    {
        cout << endl << "Vector is not empty";
    }
    else
    {
        cout << endl << "Vector is empty";
    }
    v1.shrink_to_fit();
    cout << endl << "Vector elements are: ";
    for (auto it = v1.begin(); it != v1.end(); it++)
    {
        cout << *it << " ";
    }
    return 0;
}
```

The console output will be as follows.

```
Size : 5
Capacity : 8
Max_Size : 2305843009213693951
Size : 4
Vector is not empty
Vector elements are: 1 2 3 4
```

Table 10.5. Element access related functions associated with the vector

Function	Description
reference operator [v]	Returns a reference to the element at position ***v*** in the vector.
at(V)	Returns a reference to the element at position ***v*** in the vector.
front()	Returns a reference to the first element in the vector.
back()	Returns a reference to the last element in the vector.
data()	Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

Example 06

The following example of a C++ program demonstrates the above element access function associated with the vector.

```
#include<iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v1;
    for (int i = 1; i <= 10; i++)
    {
        v1.push_back(i*10);
    }
    cout << endl << "Reference operator [v] : v1[2] = " << v1[2];
    cout << endl << "at : v1.at(4) = " << v1.at(4);
    cout << endl << "front() : v1.front() = " << v1.front();
    cout << endl << "back() : v1.back() = " << v1.back();
    int* pos = v1.data();
    cout << endl << "The first element is " << *pos;
    return 0;
}
```

The console output will be as follows.

```
Reference operator [v] : v1[2] = 30
at : v1.at(4) = 50
front() : v1.front() = 10
back() : v1.back() = 100
The first element is 10
```

Table 10.6. Modifier related functions associated with the vector

Function	Description
assign()	It assigns new value to the vector elements by replacing old ones
push_back()	It push the elements into a vector from the back
pop_back()	It is used to pop or remove elements from a vector from the back.
insert()	It inserts new elements before the element at the specified position
erase()	It is used to remove elements from a container from the specified position or range.
swap()	It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
clear()	It is used to remove all the elements of the vector container
emplace()	It extends the container by inserting new element at position
emplace_back()	It is used to insert a new element into the vector container, the new element is added to the end of the vector

Example 07

The following example of a C++ program explains the above modifier function associated with vector.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
```

```
    vector<int> v;
    v.assign(5, 10);
    cout << "The vector elements are: ";
    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i] << " ";
    }
    v.push_back(15);
    int n = v.size();
    cout << endl << "The last element is: " << v[n - 1];
    cout << endl << "The vector elements are: ";
    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i] << " ";
    }
    v.pop_back();
    cout << endl << "The vector elements are: ";
    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i] << " ";
    }
    v.insert(v.begin(), 5);
    cout << endl << "The first element is: " << v[0];
    v.erase(v.begin());
    cout << endl << "The first element is: " << v[0];
    v.emplace(v.begin(), 15);
    cout << endl << "The first element is: " << v[0];
    v.emplace_back(20); n = v.size();
    cout << endl << "The last element is: " << v[n - 1];
    v.clear();
    cout << endl << "Vector size after erase(): " << v.size();
    return 0;
}
```

The console output will be as follows.

```
The vector elements are: 10 10 10 10 10
The last element is: 15
The vector elements are: 10 10 10 10 10 15
The vector elements are: 10 10 10 10 10
The first element is: 5
The first element is: 10
The first element is: 15
The last element is: 20
Vector size after erase(): 0
```

Example 08

The following example of a C++ program demonstrate the sorting an unsorted vector.

```
#include <iostream>
#include<vector>
#include <algorithm>
using namespace std;
```

```
int main()
{
    vector<int> v = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
    cout << "Unsorted: ";
    for (auto x : v)
    {
        cout << x << " ";
    }
    sort(v.begin(), v.end());
    cout << endl << "Sorted: ";
    for (auto x : v)
    {
        cout << x << " ";
    }
    sort(v.begin(), v.end(), greater<int>());
    cout << endl << "Sorted Descending: ";
    for (auto x : v)
    {
        cout << x << " ";
    }
    return 0;
}
```

The console output will be as follows.

```
Unsorted: 1 5 8 9 6 7 3 4 2 0
Sorted: 0 1 2 3 4 5 6 7 8 9
Sorted Descending: 9 8 7 6 5 4 3 2 1 0
```

Example 09

The following example of a C++ program demonstrate the process of reading a text file and storing the data from that file in vector.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
using namespace std;
struct Weather
{
    int a,b,e;
    double c,d,f;
};
int main()
{
    vector<Weather> data_weather;
    Weather w;
    string line;
    ifstream myfile ("weather.txt");
    if (!myfile.is_open())
    {
        cerr << "unable to open file";
        return 0;
```

```
    }
    getline(myfile, line);
    cout << line << endl;
    while (getline(myfile, line))
    {
        istringstream buffer(line);
        buffer >> w.a >> w.b >> w.c >> w.d >> w.e >> w.f ;
        data_weather.push_back(w);
        cout << line << endl;
    }
    myfile.close();
    cout << endl << endl <<"========== READING VECTOR ========"
        << endl << endl;
    for(auto w1:data_weather)
    {
        cout << w1.a <<"\t" << w1.b <<"\t"<< w1.c<<"\t" << w1.d
            <<"\t" << w1.e <<"\t"<< w1.f <<"\n";
    }
    return 0;
}
```

The console output will be as follows.

```
a       b   c       d       e   f
2004    9   20.5    8.8     0   37.4
2005    10  13.6    4.2     5   77.8
2006    11  11.8    4.7     3   45.5
2007    12  7.7     0.1     17  65.1
2008    11  7.3     0.8     14  74.6
2009    9   6.5     0.1     13  3.3


========== READING VECTOR ========
2004    9       20.5    8.8     0       37.4
2005    10      13.6    4.2     5       77.8
2006    11      11.8    4.7     3       45.5
2007    12      7.7     0.1     17      65.1
2008    11      7.3     0.8     14      74.6
2009    9       6.5     0.1     13      3.3
```

In the above example, ***weather.txt*** is a text file that contains rows of data with a title row. Each data row contain six data delaminated by space as the following.

```
a       b   c       d       e   f
2004    9   20.5    8.8     0   37.4
2005    10  13.6    4.2     5   77.8
2006    11  11.8    4.7     3   45.5
2007    12  7.7     0.1     17  65.1
2008    11  7.3     0.8     14  74.6
2009    9   6.5     0.1     13  3.3
```

Since, the data contains six fields it is better to use a structure named ***Weather*** that contains six variables. Out of these variables three are integer and other three are double. In the main program a vector named ***data_weather*** of type ***Weather*** is used which contains all the 6 data with their corresponding sub fileds.

myfile of ***ifstream*** type object is used to read the data from ***weather.txt*** file sequentially. ***getline(myfile, line)*** is a function which reads the line from ***myfile*** and copy the text to ***line*** and send the iterator(pointer) to the starting of next line. In the ***while*** loop this process continues until ***eof()***. When ***eof()*** is reached ***getline(myfile, line)*** generates false or zero and ***while*** loop terminates.

buffer is of ***istringstream*** type which takes entire ***line*** and read each word delaminated by space. All the data fields of single row is extracted and copied to ***Weather*** typed variable ***w***. Then ***w*** is ***push_back*** (i.e., copied) to the vector named ***data_weather***.

Post-laboratory Problems

1. Write and run a C++ program that counts and prints the number of lines, words, and letter frequencies from an input text file.
2. Write and run a C++ program that reads full names, one per line from a text file and then prints them in the standard Bangladeshi passport format (e.g., Last Name, First Name, Middle Name).
3. Write and run a C++ program to manage a shopping list. Each shopping list item is represented by a string. Your design requires a ***print*** function that prints out the contents of the shopping list. Using a vector to hold the shopping list items, write a ***print*** function to print out the contents of a vector of strings. Test your print function with a main program that does the following:
   a) Create an empty vector. Print it.
   b) Append the items, "eggs," "milk," "sugar," "chocolate," and "flour" to the list. Print it.
   c) Remove the last element from the vector. Print it.
   d) Append the item, "coffee" to the vector. Print it.
   e) Write a loop that searches for the item, "sugar" and replace it with "honey." Print the vector.
   f) Write a loop that searches for the item, "milk," and then remove it from the vector. (You are permitted to reorder the vector to accomplish the removal, if you want.) Print the vector.
   g) Search for the item, "milk" and find a way to remove it without changing the order of the rest of the vector. (This is not necessarily an efficient operation.) Print the vector one more time.