STEMLoyola

Practical Guideline 1

Programming and Data Structures in C++

Version 1.0

Last Update: November 2019

Copyright

This guideline is part of our practical guidelines series that includes:

- Practical Guideline 1: Programming and Data Structures in C++ (2019)
- Practical Guideline 2: Algorithms and Problem Solving in C++ (2020)

Connect with STEM Loyola:

- Main website: https://www.stemloyola.org
- Programming Challenges: https://challenges.stemloyola.org
- Email: stemloyola@gmail.com
- Twitter: @stemloyolatz (https://twitter.com/STEMLoyolaTZ)
- Instagram: @stemloyolatz (https://www.instagram.com/stemloyolatz)
- GitHub: @stemloyolatz (https://github.com/stemloyolatz)

Contents

Copyright ii
Contents iii
Dedication v
Acknowledgement vi
List of Examples vii
List of Figures viii
List of Tables viii
1 Introduction 1
1.1 Guideline Objective 1
1.2 What is Programming? 1
1.3 Programming Languages 1
1.4 Integrated Development Environment (IDE) 2
2 C++ Basics 3
2.1 Background 3
2.2 Program Entry Point 4
2.3 Header Files 4
2.4 Comments 5
2.5 Source Code Structure 6
3 Basic Data Types (Primitive Data Structures) 8
3.1 Whole Numbers (Integers) 9
3.2 Decimal Numbers (Real Numbers) 9
3.3 Single Characters 10
3.4 String (Words and Text) 11
3.5 Automatic Data Type (C++11 and up) 12
4 Basic Operators 13
4.1 Maths Operations 13
4.2 Comparison 13
5 Program Control 14
5.1 If…Else 14
5.2 Switch 17
6 Repetition 18
6.1 While Loop 18

6.2 Do…while Loop....21
6.3 For Loop....21
6.4 Range-Based For Loop (C++11 and up)....23
6.5 Nested Loops....24
6.6 Break and Continue....26
7 Data Structures....27
7.1 Linear Data Structures....27
7.1.1 Static Array....27
7.1.2 Dynamic Array (a.k.a Vector)....29
7.1.3 Stack....32
7.1.4 Queue....36
7.2 Non-Linear Data Structures....39
7.2.1 Map....39
7.2.2 Set....43
7.2.3 Priority Queue....46
8 Data Streams....48
8.1 Standard Input and Output Streams....48
8.2 Redirecting the Standard Input and Output....49
8.3 User Defined Input and Output Streams....50
8.4 String Streams....53
9 Guarding Against User Input....55
9.1 Anticipating Exceptional Cases....55
9.2 Handling Invalid User Input....56
9.3 Handling Long String Input....59
9.4 Ignoring User Input....59

This guideline is dedicated to all men and women who strive to live for others.

Acknowledgement

This programming guideline is a living document that receives regular updates. STEM Loyola is grateful to the following individuals for their efforts and various contributions in compiling and reviewing this guideline. In alphabetical order of first names, STEM Loyola thanks:

Content Writers:

1. Francis Sowani (Loyola Form Six Class of 2010, B.Sc. Computer Engineering)
2. Lidunda Moyo (Loyola Form Four Class of 2007, B.Sc. Electrical & Computer Engineering)
3. Victor Sowani (Loyola Form Six Class of 2015, B. Sc. Computer Science & Engineering)

Reviewers:

1. Alvin Mrema (Loyola Form Six Class of 2018)
2. Berwin Sengo (Loyola Form Six Class of 2010, B.Sc. Computer Science)
3. Nathaniel Mwaipopo (Loyola Form Six Class of 2019)
4. Vedastus Watosha (Loyola Form Six Class of 2019)

List of Examples

Example 1: Binary Code 1
Example 2: Shortest C++ Code 4
Example 3: Hello World! 4
Example 4: Hello World with STD Namespace 5
Example 5: Comments 5
Example 6: Badly Structured Source Code 6
Example 7: Badly Commented Source Code 7
Example 8: Well Structured and Commented Source Code 7
Example 9: Storing Whole Numbers 9
Example 10: Storing Decimal Numbers 10
Example 11: Storing Single Characters 10
Example 12: Storing Words/Text 11
Example 13: Automatic Data Type 12
Example 14: Basic If Statement 14
Example 15: Compound If Statement 15
Example 16: Shorthand If...Else (Ternary Operator) 16
Example 17: Ternary Operator in Action 16
Example 18: Switch...Case in Action 17
Example 19: While Loop 18
Example 20: Non-Executed Body of While Loop 19
Example 21: Infinite While Loop 19
Example 22: Non-Zero as True and Zero as False 20
Example 23: Do...While 21
Example 24: For Loop 22
Example 25: Infinite For Loop 22
Example 26: Initializing Multiple Variables in For Loop 23
Example 27: Range-based For Loop (C++11 and up) 23
Example 28: Generating a Diamond Shape 24
Example 29: Generating Multiplication Table 25
Example 30: Continue Next Iteration 26
Example 31: Break a Loop 26
Example 32: Basic Array 27
Example 33: Initialize an Array 28
Example 34: Accessing Array Elements (C++11 and up) 28
Example 35: Basic Dynamic Array 29
Example 36: Using iterators in Vectors 30
Example 37: Accessing Elements in a Vector (C++11 and up) 31
Example 38: Basic Stack Operations 33
Example 39: Stack Use Case (Balanced Symbols Problem) 35
Example 40: Basic Queue Operations 37
Example 41: Translation Using a Map 41
Example 42: Translation Using Map (C++11 and up) 42
Example 43: Basic Set Operations 43
Example 44: Set Operations (C++11 and up) 44
Example 45: Set Intersection (C++11 and up) 45

Example 46: Priority Queue in Action 46
Example 47: Default Standard Input and Output Streams 49
Example 48: Redirected Standard Input and Output Streams 50
Example 49: User Defined Input and Output Streams 52
Example 50: String Stream 53
Example 51: Data Conversion Using String Streams 54
Example 52: Division Problem 55
Example 53: Wrong User Input 56
Example 54: Guarding Against Wrong User Input 57
Example 55: Guarding and Inspecting Wrong User Input 58
Example 56: Reading String Containing Space (Problem) 59
Example 57: Reading String Containing Space (Solution) 59
Example 58: Ignoring User Input 60
Example 59: Ignoring User Input Until a Character 60

List of Figures

Figure 1: Selecting C++ version in CodeBlocks 3
Figure 2: Stack of Plates 32
Figure 3: Queue of People 36
Figure 4: Default Standard Input and Output Streams 48
Figure 5: Redefined Standard Input and Output Streams 49
Figure 6: User Defined Input and Output Streams 51
Figure 7: String Stream 53

List of Tables

Table 1: Common C++ Header Files 4
Table 2: Basic Data Types in C++ 8
Table 3: Supported Basic Operations 13
Table 4: Comparison/Logical Operators 13
Table 5: Common Vector Operations 31
Table 6: Common Stack Operations 33
Table 7: Common Queue Operations 37
Table 8: Common Set Operations 44
Table 9: Common Priority Queue Operations 46

1 Introduction

1.1 Guideline Objective

This document is a concise guideline to help you understand key concepts in programming. Priority has been given to those concepts that are critical to helping you gain vital skills and become competitive in STEM Loyola Programming Challenges and related programming contests.

1.2 What is Programming?

This guideline is about *programming*. Put simply, programming is an act of instructing computers to carry out tasks or actions. Programming is often referred to as *coding*. A person who writes the instructions is called a *programmer* (also known as a *coder* or a *software developer*). A complete set of instructions that a computer can execute is called a *program* (also known as *code*, *application*, or *app*). When a computer performs the tasks/actions contained in a program, we call this *executing* or *running* the program.

1.3 Programming Languages

Computers can understand only one language called *machine language*. English has 26 letters in its alphabet, but machine language has only two letters: one (1) and zero (0). In programming, we call these two letters as *binary digits* or *bits* in short. When instructions are written in machine language, the resulting code is called *machine code*. Below is an example of machine code for the words "Hello World".

```
01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111
01110010 01101100 01100100
```

Example 1: Binary Code

A language like this that one can use to instruct computers is called a *programming language*. Machine language is an example of a low-level programming language. It is called low-level because the instructions need little or no translation before a computer can understand. *Assembly language* is another kind of low-level programming language. Assembly code needs to be translated to machine code before computers can execute. As you may have guessed, writing instructions in machine language is not ideal for us.

There are programming languages that are closer to human languages like English. These are called *high-level programming languages*. Examples include C, C++, Python, Java, JavaScript, PHP, and Visual Basic. In this guideline, we will only use C++.

When a program is written using a high-level programming language, it has to be translated to a machine language before it can be executed by a computer. This process is called *compiling* or *interpreting*. Programs that can perform the translations are called *compilers* or *interpreters*

respectively. Interpreting involves translating and executing one instruction at a time. On the other hand, compiling involves translating all the instructions before a single one can be executed. Programming languages that support interpreting are like PHP, JavaScript, and Python. Those that support compiling are like C, C++, Java and Visual Basic.

1.4 Integrated Development Environment (IDE)

An Integrated Development Environment (IDE) is an application that contains all the necessary tools required to write and test code effectively. Simply, it is the application that you will need to create, test, and run your programs.

For C++, we recommend CodeBlocks. You can download CodeBlocks from the official website at https://www.codeblocks.org/downloads/26. A walkthrough tutorial is available from the official STEM Loyola YouTube channel at https://youtu.be/AQOOqgn6IpQ. The video will walk you through downloading, installing, and setting up CodeBlocks in Windows.

Visit https://challenges.stemloyola.org/article/recommended-ides for a comprehensive list of IDEs we recommend for other programming languages like Python, Java, and JavaScript.

2 C++ Basics

2.1 Background

C++ is a programming language developed in 1980 by Bjarne Stroustrup at the Bell Telephone Laboratories in the United States. C++ is an enhanced form of another programming language called C. C++ is an object-oriented programming (OOP) language, which follows OOP concepts like inheritance, encapsulation, abstraction, and polymorphism.

Like most other programming languages, C++ comes in many versions. Newer versions bring improvements as well as remove or add new features to the language. C++ has the following standard versions:

- C++98 is the first edition (1998)
- C++03 is the second edition (2003)
- C++11 is the third edition (2011)
- C++14 is the fourth edition (2014)
- C++17 is the fifth edition (2017)

Currently, *C++ Standards Committee* is responsible for C++ releases and has currently fixed a three-year release cycle. As of 2019, the next version (C++20) is currently under preparations.

Important:

- Please confirm the C++ version that will be used for your NECTA practical exams ahead of time. It is probably the second edition (C++03).
- When you install CodeBlocks, by default it selects the second edition (2003) of C++. If you use features introduced in C++11 and above, you will have to explicitly select the desired C++ version. From `Settings` section, go to `Compiler`... and check the box corresponding to the desired C++ version as shown in the figure below for C++11.

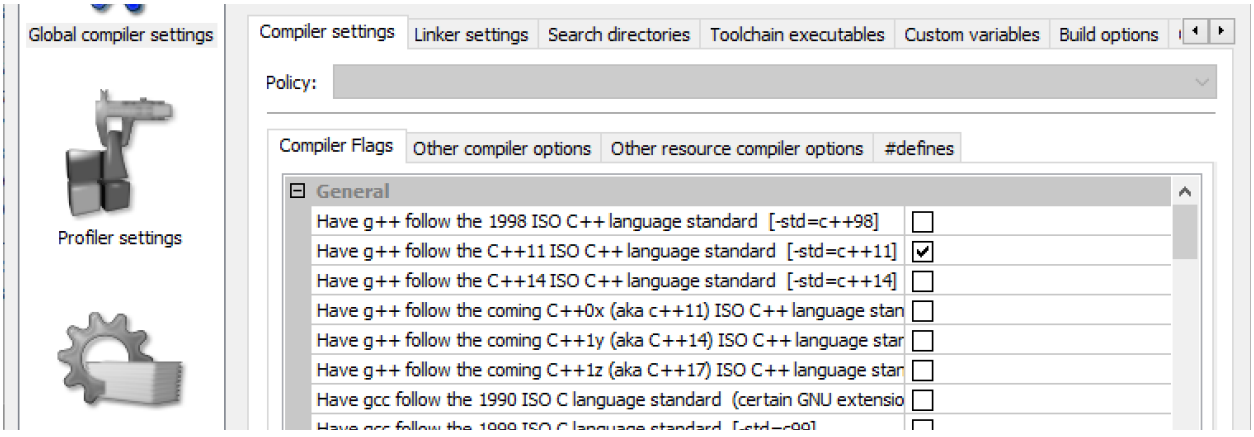

Figure 1: Selecting C++ version in CodeBlocks

2.2 Program Entry Point

In writing a C++ program, the first instruction to be executed will be the first instruction inside the `main` function. The following is the simplest way to define the main function, and also the smallest C++ code you can write. The code does not contain any instructions and hence, does nothing.

```
main(){ }
```

Example 2: Shortest C++ Code

2.3 Header Files

C++ includes many built-in instructions that we can re-use to write our own instructions. These built-in instructions are collected in special files called *header files*. The following table describes some common header files in C++.

Table 1: Common C++ Header Files

Header File	Description
`iostream`	Support for data input from the standard input (keyboard) and data output to standard output (console screen)
`fstream`	Support for reading and writing data to and from files
`iomanip`	Support for formatting data during data input/output e.g. setting how many decimal places should be displayed
`string`	Support for manipulating text
`cmath`	Support for mathematics operations
`limits`	Support for defined limits like what is the largest supported integer value
`algorithm`	Support for common algorithms like sorting

In C++, you can inform a compiler that you want to use a particular header file using the `include` keyword. The following program displays "Hello World!" to the console screen using the `cout` keyword. `endl` keyword ensures that the words are written on their own line.

```
#include<iostream>   // cout, endl

int main() {
    std::cout << "Hello World!" << std::endl;

    return 0;
}
```

Example 3: Hello World!

Inside a header file, it is common to group similar instructions. In C++, we call such groups as *namespaces*. `cout` and `endl` belong to a group/namespace called *std* (i.e. standard namespace).

We can avoid having to write the group/namespace over and over again by specifying to the compiler the default namespace to look into. The above program can be re-written as follows.

```
#include<iostream>  // cout, endl

using namespace std;

int main() {
    cout << "Hello World!" << endl;

    return 0;
}
```

Example 4: Hello World with STD Namespace

2.4 Comments

Comments are part of your program that you want a compiler to ignore. It is a good practice to add comments throughout your program code to describe what different parts are doing. This is useful when you work with others or for your own future reference. You can specify a comment using either `//` or `/*  */`. Below is the Example 4 re-written with comments.

```
/*
 Author: STEM Loyola
   Year: 2019
*/

#include<iostream>  // cout, endl

using namespace std;

int main() {
    // Greet the world
    cout << "Hello World!" << endl;

    return 0;
}
```

Example 5: Comments

Note:

- `//` can only be used for a single line comment
- `/*  */` can be used for both single line and multiple line comments

2.5 Source Code Structure

You may have noticed that in the examples provided so far, each instruction is written in its own line. Sometimes an empty line is included between instructions. Also, lines inside the `main()` begin with space. First of all, the action of adding space at the beginning of some lines is called *indentation*. It is used to mark a group of instructions belonging to the same block. This is not needed by the compiler, but it helps other programmers or yourself to read/understand the source code quicker.

C++ compiler ignores any space between different instructions. Actually, different instructions can be written on the same line or a single instruction can span different lines. Also, C++ permits empty statements (i.e. statements that just have the semicolon) like *Line 6* in Example 6. Multiple statements can be grouped using curly braces, { }. Usually, this is done to group statements executed under structures like functions, *if…else*, and loops. However, unnecessary grouping can lead to hard-to-read source code (like in lines 10 and 15 in Example 6). Below is a badly structured C++ code but it is valid nonetheless.

```
1.  #include<iostream       >    // cout, endl
2.
3.  using      namespace
4.
5.  std;
6.  ;
7.  int
8.  main(
9.
10. )      {  {{{cout
11. << "Please don't write your code like this!"
12.         <<
13.     endl
14. ;}}
15. }cout << "Please don't!" << endl;;;  {{return 0;}} }
```

Example 6: Badly Structured Source Code

Making matters worse. Comments can be added at almost any part of the code. When we say, “at almost any part”, we really mean *at almost any part*! Below is a poorly commented code, but again, it is still a valid C++ code.

```cpp
#include /* one  */ <iostream>  // cout, endl

using /* two */ namespace std;

int /* three */ main(/* four */) {
    cout << /* five */ "Please don't write your code like this!" << endl;
    cout << "Please don't!" << endl;
    ; /* these are two useless empty statements */ ;
    return /* six */ 0;
}
```

Example 7: Badly Commented Source Code

Compare the above two examples with the one below which showcases a well written and commented source code. Hence, it is not enough to write valid code, but you should strive to write well-structured and appropriately commented code, that is not only easier to understand but also beautiful to look at.

```cpp
#include <iostream>  // cout, endl

using namespace std;

int main() {
    // Display a message
    cout << "Please write your code like this!" << endl;
    cout << "Please!" << endl;

    return 0;
}
```

Example 8: Well Structured and Commented Source Code

3 Basic Data Types (Primitive Data Structures)

Programs process different kinds of data such as whole numbers, decimal numbers, text, images, sounds, and videos. Before programs can process such data, the data needs to be stored in memory using structures called *variables*. Variables contain an address of the location in memory where a particular data is located.

In a program, we can define a variable by stating two things:

i. *name*: the label that we will use inside our program to refer to the variable
ii. *data type*: the type of data that a variable will store. This information is used by a compiler to determine how much space in memory should be set aside for the variable.

The following table summarizes the basic data types in C++.

Table 2: Basic Data Types in C++

Category	Data Type	C++ Keyword	Description
Whole numbers	Small integers	`short int`	Integers between -32,767 and 32,767
	Normal integers	`int`	Integers between -2,147,483,648 to 2,147,483,647
	Large integers	`long long int`	Integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807
Decimal Numbers	Single precision decimal numbers	`float`	Can accommodate seven digits. Its range is approximately $1.5\ x10^{-45}$ to $3.4\ x10^{38}$
	Double precision decimal numbers	`double`	Can accommodate 15 to 16 digits, with a range of approximately $5.0\ x10^{-345}$ to $1.7\ x10^{308}$
Characters	ASCII characters	`char`	Individual letters, numbers, and symbols
Text	String	`string`	Text such as words, sentences, paragraphs, etc.
Logical values	Boolean	`bool`	Stores the two logical values: `true` or `false`
Nothing	*No data type*	`void`	Is used in some cases such as functions to indicate that no values will be passed into or returned from the function

The above data types can be used to store data as follows:

3.1 Whole Numbers (Integers)

```cpp
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    // Request and display the user's age
    cout << "Enter your age: ";

    int age;
    cin >> age;

    cout << "You are " << age << " years old!" << endl;

    return 0;
}
```

```
Enter your age: 19
You are 19 years old!
```

Example 9: Storing Whole Numbers

3.2 Decimal Numbers (Real Numbers)

```cpp
#include<iostream>  // cout, endl
#include<iomanip>   // setprecision

using namespace std;

int main() {
    // Displaying the value of PI in single precision
    // (Reliable to 7 decimal places)
    float pi_s = 3.14159265358979323846264338327 9;
    cout << "SINGLE PRECISION:" << endl;
    cout << "Default decimal places: Pi = " << pi_s << endl;
    cout << "20 decimal places: Pi = " << setprecision(20) << pi_s << endl;

    // Displaying the value of PI in double precision
    // (Reliable to 15 or 16 decimal places)
    double pi_d = 3.14159265358979323846264338327 9;
    cout << "DOUBLE PRECISION:" << endl;
    cout << "20 decimal places: Pi = " << setprecision(20) << pi_d << endl;

    return 0;
}
```

```
SINGLE PRECISION:
Default decimal places: Pi = 3.14159
20 decimal places: Pi = 3.1415927410125732422
DOUBLE PRECISION:
20 decimal places: Pi = 3.141592653589793116
```

Example 10: Storing Decimal Numbers

When the output of the above program (second box) is inspected, it can be seen that using `float`, the value of Pi is correctly captured to about seven decimal places. When using `double`, the value is correctly captured to about 15 or 16 decimal places. This is critical to remember when dealing with real numbers.

3.3 Single Characters

```
#include<iostream>  // cout, endl

using namespace std;

int main() {
    char physics_grade = 'A';
    char maths_grade = 'B';

    cout << "Physics: " << physics_grade << endl;
    cout << "Maths: " << maths_grade << endl;

    return 0;
}
```

```
Physics: A
Maths: B
```

Example 11: Storing Single Characters

3.4 String (Words and Text)

```cpp
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    // Request and display user's name
    cout << "Enter your first name: ";

    // Input user's name (only the first word/name is captured)
    string name;
    cin >> name;

    cout << "Nice to meet you " << name << "!" << endl;

    // Clear the rest of user input
    cin.ignore();

    // Request and display user's school motto
    cout << "What is your school motto? ";

    // Input the entire line (capture everything user types before
    // pressing <Enter>
    string motto;
    getline(cin, motto);

    cout << "So your school motto is \"" << motto << "\"" << endl;

    return 0;
}
```

```
Enter your first name: Aisha
Nice to meet you Aisha!
What is your school motto? Men and Women for Others!
So your school motto is "Men and Women for Others!"
```

Example 12: Storing Words/Text

Note: Please refer to section *9.3 Handling Long String Input* for more examples on capturing *string* input from a user. Section *9.4 Ignoring User Input* elaborates more on ignoring user input.

3.5 Automatic Data Type (C++11 and up)

Since the 2011 version of C++ (C++11), you can let the compiler predict the data type of a variable using *auto* keyword. When using *auto*, the variable must be initialized because the compiler will infer the data type from the data that is being stored in the variable. Refer to Example 13.

Note: You must have C++11 or later selected for this example. Refer to *Figure 1: Selecting C++ version in CodeBlocks* for instructions on how to specify C++ version in CodeBlocks.

```cpp
#include<iostream>  // cout, endl

using namespace std;

int main() {
    auto name = "Alice";
    auto age = 17;

    cout << "Next year, " << name << " will be " << age + 1 << "!" << endl;

    return 0;
}
```

```
Next year, Alice will be 18!
```

Example 13: Automatic Data Type

4 Basic Operators

C++ supports most of the basic mathematical and comparison operations on data.

4.1 Maths Operations

C++ support the basic mathematics operations using the following operators:

Table 3: Supported Basic Operations

Operation	C++ Operator	Example
Assignment	=	`int age = 18;`
Brackets	( )	`int results = ( 1 + 4 );`
Division	/	`float pi = 22.0 / 7;`
Multiplication	*	`int product = 6 * 82;`
Addition	+	`int sum = 5 + 9 + 18 + 19;`
Subtraction	-	`int difference = 875 - 8;`
Increment by one	++	`int x = 2;` `x++;  // Now the value of x is 3`
Decrement by one	--	`int x = 2;` `x--;  // Now the value of x is 1`

Modulus	%	`// y contains the remainder of`
		`// dividing 17 by 5`
		`int y = 17 % 5;`

4.2 Comparison

Table 4: Comparison/Logical Operators

Comparison	C++ Operator
Equal to	==
Not equal to	!=
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=
And	&&
Or	| |
Not	!

5 Program Control

In life, we make decisions and take different actions depending on the decision we make. For instance, at some point during O-Level, students decide to either take Science, Commercial, or Arts streams. The subjects that students will study in Form 3 and 4 depend on this decision.

Likewise, in programming, we need to be able to take a different course of action depending on user's actions or available data. For instance, allowing a user to open a file or not depending on if the user has permissions to view the file or not. A number of program control structures are available in C++ to support us to achieve this.

5.1 If…Else

This is the simplest decision-making structure. In its simplest, it takes a general form of:

```
if ( condition ) body
```

Where statements inside *body* will be executed if the *condition* is true (or non-zero). For instance, in Example 14, although the greeting "Hello!" will always be displayed, the phrase "You are ready for school!" will only be displayed if *age* is more than 5 years.

```
#include<iostream>   // cout, endl

using namespace std;

int main() {
    int age = 10;

    cout << "Hello!" << endl;

    if (age > 5) cout << "You are ready for school!" << endl;

    return 0;
}
```

```
Hello!
You are ready for school!
```

Example 14: Basic If Statement

Usually, a decision involves multiple conditions and a course of action for each condition. A compound *if* statement has the following general form:

```
if ( condition-1 ) body-1
else if ( condition-2 ) body-2
.
.
else if ( condition-n ) body-n
else default-body
```

Each *if* or *else if* will have its own *condition* and a *body* of statements that will be executed if that condition is true (or non-zero). If none of the conditions are evaluated to be true, then the *default-body* defined in *else* will be evaluated. In a compound *if* statement, only one *body* will be executed. Either of the first *condition* to be evaluated as true or of the *else* part. If the compound *if* statements contains more than one *conditions* that are true, only the first one will be executed.

Note: if the *body* contains more than one statement, they must be surrounded by curly braces, { }. If the *body* contains only a single statement, the curly braces are optional and a matter of personal preference or agreed rules (if you are coding with others as part of a project).

```
1.  #include<iostream>  // cout, cin, endl
2.
3.  using namespace std;
4.
5.  int main() {
6.      cout << "Enter your exam score: ";
7.      int score;
8.      cin >> score;
9.
10.     if (score > 100 || score < 0) {
11.         cout << "Invalid score!" << endl;
12.     } else if (score > 80) {
13.         cout << "That's a grade of A" << endl;
14.     } else if (score > 60) {
15.         cout << "That's a grade of B" << endl;
16.     } else if (score > 40) {
17.         cout << "That's a grade of C" << endl;
18.     } else if (score > 20) {
19.         cout << "That's a grade of D" << endl;
20.     } else {
21.         cout << "That's a grade of F" << endl;
22.     }
23.
24.     return 0;
25. }
```

```
Enter your score: 78
That's a grade of B
```

Example 15: Compound If Statement

In Example 15, we can understand the *if* statements as follows.

- *Line 10*: checks if the score is negative or more than 100. This is an invalid score.
- *Line 12*: at this point, the score will be a number from 0 to 100 (otherwise it would have been captured by *Line 10*). *Line 12* checks if the score is more than 80 and evaluates this as grade of A. There is no need to check if the score is less than 100 (`score > 80 && score <= 100`) because *Line 10* already eliminated all the scores above 100 as invalid.
- *Line 14*: checks if the score is more than 60 and evaluates this as a grade of B. There is no need to also check if the score is less than 80 (`score > 60 && score <= 80`) because if was more than 80, *Line 12* would have captured it and *Line 14* would not be executed.

- The same principle applies to lines 16 and 18.
- *Line 20*: captures the remaining possible situation if all above conditions are false: the score is between 0 and 20. This is evaluated as grade of F.

Most of the time, compound *if* statements contain two cases: it is either this way or that way. This is very common that it has a one-line shorthand structure for it called a *conditional/ternary operator* with the following general form:

```
condition ? body-if-true : body-if-false ;
```

This can be seen in action in Example 16 below.

```
#include<iostream>  // cout

using namespace std;

int main() {
    int age = 13;

    age >= 18 ? cout << "Can drive!\n" : cout << "Cannot drive!\n";

    return 0;
}
```

```
Cannot drive!
```

Example 16: Shorthand If...Else (Ternary Operator)

The ternary operator is especially useful when we need to assign a value that is based on another condition. Consider a case where you want to manually find the magnitude of a number. For instance, the magnitude of -5 is 5 and the magnitude of 5 is 5. We can accomplish this as shown in Example 17.

```
#include<iostream>  // cout, endl

using namespace std;

int main() {
    int num = -777;
    int magnitude = (num > 0) ? num : -num;

    cout << "Magnitude: " << magnitude << endl;

    return 0;
}
```

```
Magnitude: 777
```

Example 17: Ternary Operator in Action

5.2 Switch

Switch statement simplifies building long compound *if* statements that compare a variable to several integral values. *Switch* takes the following general form:

```
switch (variable) {
    case value-1: body-1; break;
    case value-2: body-2; break;
    .
    .
    case value-n: body-n; break;
    default: default-body;
}
```

A common application of *switch* is during the evaluation of user input.

```
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    cout << "Please select the next action:" << endl
         << "-----------------------------" << endl
         << "1. Quit the program." << endl
         << "2. Resume the program." << endl
         << "3. Restart the program." << endl
         << "Enter your choice: ";

    int choice;
    cin >> choice;

    switch (choice) {
        case 1: cout << "Quitting..." << endl; break;
        case 2: cout << "Resuming..." << endl; break;
        case 3: cout << "Restarting..." << endl; break;
        default: cout << "Invalid choice!" << endl; break;
    }

    return 0;
}
```

```
Please select the next action:
-----------------------------
1. Quit the program.
2. Resume the program.
3. Restart the program.
Enter your choice: 3
Restarting...
```

Example 18: Switch...Case in Action

Note: If *break* is omitted, all the statements following a matching case will be executed until a *break* is found or the end of *switch* is reached.

6 Repetition

Loops are used when a similar action needs to be performed repeatedly for a given number of times or until a certain condition is met. There are two forms of loops in C++:

i. *For* loop — usually used when repetition is for a given number of times
ii. *While* loop — usually used when repetition should continue until a condition is met

However, any *for* loop can be re-written using a *while* loop and vice versa.

6.1 While Loop

A *while* loop has the following general form:

```
while ( condition ) body
```

If the value of the *condition* is true (or non-zero), the *body* will be executed. Then, the *condition* will be tested again, if it is still true (or non-zero), the *body* will be executed again. This will continue until the *condition* is false.

For instance, the following loop will print all positive numbers less than ten.

```
#include<iostream>  // cout

using namespace std;

int main() {
    int num = 1;

    while (num < 10) {
        cout << num << " ";
        num++;
    }

    return 0;
}
```

```
1 2 3 4 5 6 7 8 9
```

Example 19: While Loop

Note:

i. The *body* can be a single statement or more. If the body has more than one statement, the statements must be enclosed between the curly braces as shown in Example 19.
ii. In Example 19, the *body* is executed nine times before the condition becomes false (when *num=10* since the expression "*10 < 10*" is false).
iii. It is possible that the *body* of loop will never be executed. This happens when the condition can never be true. Like in the following example, a number cannot be less than itself.

```cpp
#include<iostream>  // cout

using namespace std;

int main() {
    int num = 1;

    while (num < num) {
        cout << num << " ";
        num++;
    }

    return 0;
}
```

Example 20: Non-Executed Body of While Loop

iv. Also, if the condition cannot become false, the *body* will be executed repeatedly without end (infinite loop).

```cpp
#include<iostream>  // cout, endl

using namespace std;

int main() {

    while (true) {
        cout << "Run this at your own risk!" << endl;
    }

    return 0;
}
```

Example 21: Infinite While Loop

v. Also, below is an example that demonstrates that the condition will be evaluated as true if a non-zero number is passed and as false if a zero is passed.

```cpp
#include<iostream> // cout

using namespace std;

int main() {
     int num = 10;

     while (num) {
          cout << num << " ";
          num--;
     }

    return 0;
}
```

```
10 9 8 7 6 5 4 3 2 1
```

Example 22: Non-Zero as True and Zero as False

6.2 Do…while Loop

In C++, the *while* loop can be written to begin with the body. This can be useful in such cases when the body must be executed at least once. In Example 23, "Hello buddy!!" greeting will be displayed at least once. The exact number of greetings that will be displayed will depend on user's input.

```
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    cout << "Enter 'Y' to stop or any other letter to repeat" << endl;

    char response;

    do {
        cout << "Hello buddy!!" << endl;
        cin >> response;
    } while (response != 'Y' && response != 'y');

    return 0;
}
```

Example 23: Do...While

6.3 For Loop

A *for* loop has the following general form:

```
for (initialization; condition; increase) body
```

Similar to the *while* loop, in a *for* loop, the *body* will be executed as long as the *condition* is true. In addition, a *for* loop contains two additional segments:

i. *Initialization*: this statement is executed only once, at the beginning of the loop. Usually, this defines and initializes a counter variable.
ii. *Increase*: this statement is executed between *iterations* of the loop. Usually, this increments or decrements the counter that tracks the loop execution.

Note: An *iteration* is a single execution/repetition of a loop. For instance, in Example 24, the loop repeats five times. Hence, we say, the loop had five iterations. In iteration 1, it displayed A. In iteration 2, it displayed B. This continues until iteration 5 where it displays E. After this, the value of "c" becomes 'F' and the expression "F < F" becomes false. Hence, the loop terminates.

```cpp
#include<iostream>  // cout

using namespace std;

int main() {

    for (char c = 'A'; c < 'F'; c++) {
        cout << c << " ";
    }

    return 0;
}
```

```
A B C D E
```

Example 24: For Loop

In a *for* loop, any of the parts (*initialization*, *condition*, or *increment*) can be omitted. An easy way to create an infinite loop is to omit all the three parts as shown below.

```cpp
#include<iostream>  // cout, endl

using namespace std;

int main() {

    for ( ; ; ) {
        cout << "This is an infinite loop!" << endl;
    }

    return 0;
}
```

Example 25: Infinite For Loop

Also, more than one variable can be initialized or incremented in a for loop. If all the variables to be initialized are of the same data type, they can also be defined in the *initialization* part. If they are of different data types, they must be defined outside the *for* loop. In Example 26, we count from A to E and display the letters with the first five odd numbers. Since *num* and *ch* are of different data types, they have been defined outside the *for* loop.

```cpp
#include<iostream>  // cout

using namespace std;

int main() {
    int num;
    char ch;

    for (num = 1, ch = 'A'; ch < 'F'; num += 2, ch++) {
        cout << ch << num << "\t";
    }

    return 0;
}
```

```
A1      B3      C5      D7      E9
```

Example 26: Initializing Multiple Variables in For Loop

6.4 Range-Based For Loop (C++11 and up)

Since C++11, C++ includes a shorthand syntax for a *for* loop that is especially useful in accessing individual values in a list/array.

Note: You must have C++11 or later selected for this example. Refer to *Figure 1: Selecting C++ version in CodeBlocks* for instructions on how to specify C++ version in CodeBlocks.

```cpp
#include<iostream>  // cout, endl

using namespace std;

int main() {
    string studentsList[] = { "Anne", "Bakari", "Caren", "Daudi" };

    for (string name : studentsList) {
        cout << name << endl;
    }

    return 0;
}
```

```
Anne
Bakari
Caren
Daudi
```

Example 27: Range-based For Loop (C++11 and up)

Also, in the shorthand version, you can let a compiler automatically select the data type using the *auto* keyword. The header of the *for* loop in Example 27 can be also written as follows.

```cpp
    for (auto name : studentsList) {
```

6.5 Nested Loops

The body of one loop can contain other loops. This is called *nesting* the loops. Loops can be nested to solve various problems.

Example 28 generates a diamond shape using asterisk (*) character, space, and nested *for* loops.

```
#include<iostream>  // cout, endl

using namespace std;

int main() {
    const int SIZE = 5;

    // Display the top triangle
    for (int row = 0; row < SIZE; row++) {
        cout << endl;  // Start a new row

        // Add space before the asterisks
        for (int col = SIZE-row-1; col > 0; col--) cout << " ";

        // Display the asterisks
        for (int col = 0; col < 2*row+1; col++) cout << "*";
    }

    // Display the bottom triangle
    for (int row = 0; row < SIZE; row++) {
        cout << endl;  // Start a new row

        // Add space before the asterisks
        for (int col = 0; col < row+1; col++) cout << " ";

        // Display the asterisks
        for (int col = 2*(SIZE-row)-3; col > 0; col--) cout << "*";
    }

    return 0;
}
```

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

Example 28: Generating a Diamond Shape

Example 29, generates a multiplication table using nested loops.

```cpp
#include<iostream>  // cout, endl
#include<iomanip>   // setw

using namespace std;

int main() {
    const int SIZE = 10;
    const int SPACE = 4;

    for (int row = 1; row <= SIZE; row++) {
        cout << endl;

        for (int col = 1; col <= SIZE; col++) {
            cout << setw(SPACE) << row * col << " ";
        }
    }

    return 0;
}
```

```
   1    2    3    4    5    6    7    8    9   10
   2    4    6    8   10   12   14   16   18   20
   3    6    9   12   15   18   21   24   27   30
   4    8   12   16   20   24   28   32   36   40
   5   10   15   20   25   30   35   40   45   50
   6   12   18   24   30   36   42   48   54   60
   7   14   21   28   35   42   49   56   63   70
   8   16   24   32   40   48   56   64   72   80
   9   18   27   36   45   54   63   72   81   90
  10   20   30   40   50   60   70   80   90  100
```

Example 29: Generating Multiplication Table

6.6 Break and Continue

Break and *continue* are used within loops to control execution of the body of the loop. *Continue* stops the execution of the current iteration and goes to the next iteration if the *condition* of the loop is still true. *Break* stops the execution of the current iteration and exits the loop regardless of the *condition* of the loop.

In Example 30, for all even values of *num*, the execution of the iteration is stopped before the number can be printed in Line 10.

```
1.  #include<iostream>  // cout
2.
3.  using namespace std;
4.
5.  int main() {
6.
7.      for (int x = 1; x <= 10; x++) {
8.          if (x % 2 == 0) continue;  // Stop iteration if number is even
9.
10.         cout << x << " ";
11.     }
12.
13.     return 0;
14. }
```

```
1 3 5 7 9
```

Example 30: Continue Next Iteration

In Example 31, when *num* is 7, *break* is executed, and the loop execution stops. As a result, only 1 through 6 gets to be displayed.

```
1.  #include<iostream>  // cout
2.
3.  using namespace std;
4.
5.  int main() {
6.
7.      for (int num = 1; num <= 10; num++) {
8.          if (num == 7) break;  // Stop the entire loop if number is seven
9.
10.         cout << num << " ";
11.     }
12.
13.     return 0;
14. }
```

```
1 2 3 4 5 6
```

Example 31: Break a Loop

7 Data Structures

To become a good programmer, one needs to master the use of both: *data structures* and *algorithms*. A data structure is a particular way to store and organize data in a computer memory that enables effective and efficient processing of the data. In this context, an algorithm refers to a well-defined procedure that enables a computer to solve a particular problem. This guideline will focus on data structures and some examples of their application.

Please refer to *Practical Guideline 2: Algorithms and Problem Solving in C++* for an in-depth coverage of algorithms.

7.1 Linear Data Structures

A data structure is classified as linear if its elements are organized in a sequence.

7.1.1 Static Array

A static array is a series of elements of the same data type in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. A static array can be declared by specifying the data type of its elements, its name, and its maximum size.

```
type name [size];
```

In Example 32, an array that can hold 3 strings is defined and populated.

```
#include<iostream>  // cout, endl

using namespace std;

int main() {

    string subjects [3];

    subjects[0] = "Economics";
    subjects[1] = "Geography";
    subjects[2] = "Mathematics";

    for (int k = 0; k < 3; k++) cout << subjects[k] << endl;

    return 0;
}
```

```
Economics
Geography
Mathematics
```

Example 32: Basic Array

Note:

i. Each element in an array can be referenced using a key starting from zero. The first element is at 0, the second at 1, and so on.
ii. The size of an array cannot be changed once it has been defined. You need to think ahead and define an array with the maximum possible size you need.

Like a normal variable, an array can be initialized during its definition as shown in Example 33.

```
#include<iostream>  // cout, endl

using namespace std;

int main() {

    string subjects [] = { "Economics", "Geography", "Mathematics" };

    for (int k = 0; k < 3; k++) cout << subjects[k] << endl;

    return 0;
}
```

Example 33: Initialize an Array

Also, since C++11, a range-based *for* loop can be used to simplify accessing an array element as shown in Example 34.

```
#include<iostream>  // cout, endl

using namespace std;

int main() {

    string subjects [] = { "Economics", "Geography", "Mathematics" };

    for (auto subject : subjects) cout << subject << endl;

    return 0;
}
```

Example 34: Accessing Array Elements (C++11 and up)

7.1.2 Dynamic Array (a.k.a Vector)

A dynamic array (vector) is an array whose size can be changed after it has been defined. A vector definition takes the following general form:

```
vector<type> name;
```

Example 32 can be re-written using a vector as follows:

```
#include<iostream>  // cout, endl
#include<vector>    // vector

using namespace std;

int main() {

    vector<string> subjects;

    subjects.push_back("Economics");
    subjects.push_back("Geography");
    subjects.push_back("Mathematics");

    for (int k = 0; k < subjects.size(); k++) cout << subjects[k] << endl;

    return 0;
}
```

Example 35: Basic Dynamic Array

In Example 35, *size* stores the current number of elements that have been stored in the vector. Elements can be accessed using their positions (starting at 0) like in static arrays.

Also, elements in a vector can be accessed using an iterator as shown below.

```
1.  #include<iostream>  // cout, endl
2.  #include<vector>    // vector
3.
4.  using namespace std;
5.
6.  int main() {
7.
8.      vector<string> subjects;
9.
10.     subjects.push_back("Economics");
11.     subjects.push_back("Geography");
12.     subjects.push_back("Mathematics");
13.
14.     vector<string>::iterator it;
15.     for (it = subjects.begin(); it != subjects.end(); it++) {
16.         cout << *it << endl;
17.     }
18.
19.     return 0;
20. }
```

Example 36: Using iterators in Vectors

Note:

i. `subjects.begin()` is a pointer to the first element in the vector (the beginning)

ii. `subjects.end()` is a pointer to the location **after** the last element in the vector. It does not point to the last element but to the location after the last element.

iii. The iterator (`it`) is a pointer and stores the address of where data is stored in memory. To retrieve the actual data (*Line 16*), we need to instruct the compiler by adding an asterisk (*) before the iterator (`it`). This is called *de-referencing* a pointer.

Also, since C++11, a vector can be initialized similar to a static array and a range-based *for* loop can be used to access elements in a vector. This can simply Example 35 to the following.

```
#include<iostream>  // cout, endl
#include<vector>    // vector

using namespace std;

int main() {

    vector<string> subjects = { "Economics", "Geography", "Mathematics" };

    for (auto subject : subjects) cout << subject << endl;

    return 0;
}
```

Example 37: Accessing Elements in a Vector (C++11 and up)

Vector includes a handful of other common operations that have been summarized in Table 5.

Table 5: Common Vector Operations

Operation	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
`empty()`	Checks whether the vector is empty. Returns *true* if the vector is empty, *false* otherwise
`size()`	Returns the current number of elements in the vector
`capacity()`	Returns the size of the storage space currently allocated to the vector expressed as number of elements based on the memory allocated to the vector
`resize(n)`	Resizes the vector so that it contains “n” elements. If the current size of the vector is greater than *n* then the extra elements at the back are removed from the vector. If the current size is smaller than *n* then extra elements are inserted at the back of the vector
`push_back(x)`	Adds a new element “x” at the end of the vector, after its current last element. This effectively increases the container size by one
`clear()`	Removes all elements from the vector (which are destroyed), leaving the vector with a size of 0
`swap(v)`	Exchanges the contents of one vector with another vector “v” of the same type. Sizes may differ.

7.1.3 Stack

A stack is a container that operates with LIFO (Last In First Out) principle. In a stack, the last element to be added will be the first element that can be retrieved. It is similar to a stack of plates (Figure 2), new plates are placed on top and the last plate to be added will be the first to be removed.

Figure 2: Stack of Plates

In C++, a definition of a stack takes the following general form:

```
stack<type> name;
```

In Example 38, four numbers are added into a numbers stack. Then one by one, the numbers are removed from the stack and displayed. It can be seen that the order of the displayed numbers is reversed from the original.

```
#include<iostream>  // cout, endl
#include<stack>     // stack

using namespace std;

int main() {
    stack<int> numbers;

    // Add four numbers
    numbers.push(10);
    numbers.push(20);
    numbers.push(30);
    numbers.push(40);

    while (numbers.empty() == false) {
        cout << numbers.top() << endl;  // Display the top element
        numbers.pop();  // Remove the top element
    }

    return 0;
}
```

```
40
30
20
10
```

Example 38: Basic Stack Operations

Standard stack includes a handful of other common operations that have been summarized in Table 6.

Table 6: Common Stack Operations

Operation	Description
`empty()`	Checks whether the stack is empty. Returns *true* if the stack is empty, *false* otherwise
`size()`	Returns the current number of elements in the stack
`push(x)`	Adds a new element "x" at the top of the stack
`pop()`	Deletes the top most element of the stack
`top()`	Returns a reference to the top most element of the stack
`clear()`	Removes all elements from the stack (which are destroyed), leaving the stack with a size of 0

Stack Use Case: Balanced Symbols Problem

Understanding the Problem

Balanced symbols problem is a common problem in programing that involves balancing and nesting different kinds of opening and closing symbols. To simplify the problem, we will consider only three symbols:

- square brackets: [and]
- curly braces: { and }
- parentheses: (and)

The challenge is to determine whether a given expression such as "[{ ( ) }] ( )" is balanced or not. For instance, the following expressions are all balanced:

- {{([][])}()}
- ([{}]{([])})[{()}]()

On the other hand, the following expressions are not balanced:

- (((])
- ([{}]{[)})[[()}]

Consider a balanced expression "[{ ( ) }]", suppose we read the expression from left to right, we can observe that:

- The opening square bracket "[" is the first to be read and its closing square bracket "]" will be last to be read
- The opening curly brace "{" is the second to be read and its closing curly brace "}" will be the second last to be read
- The opening parentheses "(" is the last opening symbol to be read and its closing parentheses ")" will be the first closing symbol to be read

Clearly, this is last-in-first-out (LIFO) structure and we can use a stack to solve this problem.

Formulating an Algorithm

Beginning with an empty stack, we can process the expression from left to right. When we encounter an opening symbol , "[", "{" or "(", we push it onto the stack, since its corresponding closing symbol will appear later if the expression is balanced. When we encounter a closing symbol, we check if it matches the symbol on top of the stack e.g. "}" matches "{". If they match, remove the opening symbol from the top of the stack (i.e. pop the stack) and continue processing the expression. If they do not match, the expression is not balanced. As long as it is possible to pop the stack to match every closing symbol, the expression remains balanced. At the end of the expression, when we have processed all the symbols, if the expression is balanced, the stack should be empty.

Implementing the Algorithm

We can implement the above algorithm as follows.

```cpp
#include<iostream>  // cout, endl
#include<stack>     // stack

using namespace std;

int main() {
    const string EXPRESSION = "([{}]{([])})[{()}]()";

    stack<char> symbols;
    bool isBalanced = true;

    // Process the expression from left to right
    for (int k = 0; k < EXPRESSION.size(); k++) {
        char ch = EXPRESSION[k];

        // Push an opening symbol onto the stack
        if (ch == '{' || ch == '[' || ch == '(') symbols.push(ch);

        // Process a closing symbol
        else if (ch == '}' || ch == ']' || ch == ')') {
            // Check if the closing symbol matches what is on top of the stack
            if ((ch == '}' && symbols.top() == '{') ||
                (ch == ']' && symbols.top() == '[') ||
                (ch == ')' && symbols.top() == '(')) {
                // Remove the top opening symbol since they match
                symbols.pop();
            } else {
                // Mismatching symbols found
                isBalanced = false;
                break;  // No need to continue since a mismatch is already found
            }
        }

        // Unknown symbol found
        else {
            cout << "Unknown symbol '" << ch << "' in the expression!" << endl;
            isBalanced = false;
            break;
        }
    }

    // A balanced stack leaves an empty stack at the end of the day
    if (symbols.size() > 0) isBalanced = false;

    // Display results
    if (isBalanced) cout << "\"" << EXPRESSION << "\" is balanced!" << endl;
    else cout << "\"" << EXPRESSION << "\" is NOT balanced!" << endl;

    return 0;
}
```

```
"([{}]{([])})[{()}]()" is balanced!
```

Example 39: Stack Use Case (Balanced Symbols Problem)

7.1.4 Queue

A queue is a container that operates with FIFO (First In First Out) principle. In a queue, the first element to be added will be the first element that can be retrieved. It is similar to a queue of people (Figure 3), the first person to arrive will be the first one to be served and an additional person joins at the back of the queue.

Figure 3: Queue of People

In C++, a definition of a queue takes the following general form:

```
queue<type> name;
```

In Example 40, four numbers are added into a queue. Then one by one, the numbers are retrieved from the queue and displayed. It can be seen that the order of the displayed numbers is the same as the original.

```cpp
#include<iostream>  // cout, endl
#include<queue>     // queue

using namespace std;

int main() {
    queue<int> numbers;

    // Add four numbers
    numbers.push(10);
    numbers.push(20);
    numbers.push(30);
    numbers.push(40);

    while (numbers.empty() == false) {
        cout << numbers.front() << endl;  // Display the front element
        numbers.pop();  // Remove the front element
    }

    return 0;
}
```

```
10
20
30
40
```

Example 40: Basic Queue Operations

Standard queue includes a handful of other common operations that have been summarized in Table 7.

Table 7: Common Queue Operations

Operation	Description
`empty()`	Checks whether the queue is empty. Returns *true* if the queue is empty, *false* otherwise
`size()`	Returns the current number of elements in the queue
`push(x)`	Adds a new element "x" at the back of the queue
`pop()`	Deletes the front most element of the queue
`front()`	Returns a reference to the front most element of the queue
`clear()`	Removes all elements from the queue (which are destroyed), leaving the queue with a size of 0

Queue Applications

Queues have extensively been used to solve real-world problems that involve some aspect of someone/something waiting in line, such as:

i. Key press sequence in a keyboard
ii. Ticketing systems in public service stations where who comes first will be served first
iii. ATM booth line
iv. Printing jobs sent to a printer
v. Job scheduling in an operating system

7.2 Non-Linear Data Structures

A non-linear data structure organizes its elements in a non-sequential manner. For some types of data, like a set of *key* and *value* pairs, organizing the data in non-sequential manner results in quicker operations like insertion, deletion, and retrieval.

7.2.1 Map

A map is an associative container that stores elements formed by a combination of a *key* and its corresponding *value*. For instance, a map of a student can look like:

"name" : "Alice Bob"
"gender" : "Female"
"form" : "Five"
"stream" : "ECA"
"total exams": "Five"
"joined" : "2019"

In the above map, the keys are *name*, *gender*, *form*, *stream*, *total exams*, and *joined*. Each key must be unique and associated with a value. The values can be the same between different keys (e.g. between *form* and *total exams*). All keys should have the same data type, and all values should have the same data type. The data type of keys and of values can be different.

In C++, a definition of a map takes the following general form:

```
map<keys-type, values-type> name;
```

In Example 41, the maps contain a translation of digits between English and Kiswahili as well as between English words and numbers. The maps can be used to translate a phone number given in English to Kiswahili and its corresponding digits.

```cpp
#include<iostream>  // cout, endl
#include<map>       // map

using namespace std;

int main() {
    // Build an English to Kiswahili mapping of digits
    map<string, string> engToKisw;
    engToKisw["zero"]  = "sifuri";
    engToKisw["one"]   = "moja";
    engToKisw["two"]   = "mbili";
    engToKisw["three"] = "tatu";
    engToKisw["four"]  = "nne";
    engToKisw["five"]  = "tano";
    engToKisw["six"]   = "sita";
    engToKisw["seven"] = "saba";
    engToKisw["eight"] = "nane";
    engToKisw["nine"]  = "tisa";

    // Build an English words to numbers mapping of digits
    map<string, char> engToNum;
    engToNum["zero"]  = '0';
    engToNum["one"]   = '1';
    engToNum["two"]   = '2';
    engToNum["three"] = '3';
    engToNum["four"]  = '4';
    engToNum["five"]  = '5';
    engToNum["six"]   = '6';
    engToNum["seven"] = '7';
    engToNum["eight"] = '8';
    engToNum["nine"]  = '9';

    // Test the conversions
    string digits[] = {
        "zero", "seven", "five", "four", "two",
        "four", "zero", "nine", "seven", "six"
    };
    string trans_eng = "";
    string trans_kisw = "";
    string trans_num = "";

    for (int k = 0; k < sizeof(digits)/sizeof(string); k++) {
        trans_eng += " " + digits[k];
        trans_kisw += " " + engToKisw[digits[k]];
        trans_num += engToNum[digits[k]];
    }

    cout << "English: " << trans_eng << endl;
    cout << "Kiswahili: " << trans_kisw << endl;
    cout << "Number: " << trans_num << endl;

    return 0;
}
```

```
English:  zero seven five four two four zero nine seven six
Kiswahili:  sifuri saba tano nne mbili nne sifuri tisa saba sita
Number: 0754240976
```

Example 41: Translation Using a Map

Since C++11, the map can easily be initialized during its definition. Example 41 can be re-written as follows:

```cpp
#include<iostream>  // cout, endl
#include<map>       // map
#include<vector>    // vector

using namespace std;

int main() {
    // Build an English to Kiswahili mapping of digits
    map<string, string> engToKisw = {
        {"zero", "sifuri"}, {"one", "moja"}, {"two", "mbili"},{"three", "tatu"},

        {"four", "nne"}, {"five", "tano"}, {"six", "sita"}, {"seven", "saba"},
        {"eight", "nane"}, {"nine", "tisa"}
    };

    // Build an English words to numbers mapping of digits
    map<string, char> engToNum = {
        {"zero", '0'}, {"one", '1'}, {"two", '2'}, {"three", '3'},{"four", '4'},

        {"five", '5'}, {"six", '6'}, {"seven", '7'},{"eight", '8'},{"nine", '9'}

    };

    // Test the conversions
    vector<string> digits = {
        "zero", "seven", "five", "four", "two",
        "four", "zero", "nine", "seven", "six"
    };
    string trans_eng = "";
    string trans_kisw = "";
    string trans_num = "";

    for (int k = 0; k < digits.size(); k++) {
        trans_eng += " " + digits[k];
        trans_kisw += " " + engToKisw[digits[k]];
        trans_num += engToNum[digits[k]];
    }

    cout << "English: " << trans_eng << endl;
    cout << "Kiswahili: " << trans_kisw << endl;
    cout << "Number: " << trans_num << endl;

    return 0;
}
```

```
English:  zero seven five four two four zero nine seven six
Kiswahili:  sifuri saba tano nne mbili nne sifuri tisa saba sita
Number: 0754240976
```

Example 42: Translation Using Map (C++11 and up)

7.2.2 Set

A set is a container that stores *unique* elements following a specific order. All values in a set must have the same data type. The values in a set cannot be modified, but they can be inserted or deleted.

In C++, a definition of a set takes the following general form:

```
set<type> name;
```

In Example 43, seven combinations are added into a set. Then one by one, the combinations are retrieved from the set and displayed. It can be seen that this results in a unique and sorted list of combinations.

```
#include<iostream>  // cout, endl
#include<set>       // set

using namespace std;

int main() {
    set<string> combinations;

    combinations.insert("EGM");
    combinations.insert("PCM");
    combinations.insert("PCB");
    combinations.insert("ECA");
    combinations.insert("HGL");
    combinations.insert("EGM");
    combinations.insert("HGL");

    cout << "Unique sorted combinations: ";

    set<string>::iterator it;
    for (it = combinations.begin(); it != combinations.end(); it++) {
        cout << *it << " ";
    }

    return 0;
}
```

```
Unique sorted combinations: ECA EGM HGL PCB PCM
```

Example 43: Basic Set Operations

Since C++11, set initialization is supported and Example 43 can be written as shown in Example 44.

```cpp
#include<iostream>  // cout, endl
#include<set>       // set

using namespace std;

int main() {
    set<string> combinations = {
        "EGM", "PCM", "PCB", "ECA", "HGL", "EGM", "HGL"
    };

    cout << "Unique sorted combinations: ";

    set<string>::iterator it;
    for (it = combinations.begin(); it != combinations.end(); it++) {
        cout << *it << " ";
    }

    return 0;
}
```

```
Unique sorted combinations: ECA EGM HGL PCB PCM
```

Example 44: Set Operations (C++11 and up)

Standard set includes a handful of other common operations that have been summarized in Table 8.

Table 8: Common Set Operations

Operation	Description
`empty()`	Checks whether the set is empty. Returns *true* if the set is empty, *false* otherwise
`size()`	Returns the current number of elements in the set
`insert(x)`	Adds a new element "x" into the set (if it does not already exist) and ensures elements in the set are sorted
`erase(x)`	Removes the element with a value of "x" from the set
`find(x)`	Returns a reference (iterator) to the element with a value of "x"
`clear()`	Removes all elements from the set (which are destroyed), leaving the set with a size of 0

Set Use Case: Intersection of Two Lists

A common problem in programming is to determine the common elements between two lists. For instance, given a list of all students taking Advanced Mathematics and a list of all students taking Chemistry, can you generate a list of all students taking both Chemistry and Advanced Mathematics?

In C++, this can easily be accomplished using *set*. Other containers can be used as well (like arrays and vectors) but they will have to be sorted first. Sets are ideal because their elements are already sorted.

```cpp
#include<iostream>  // cout, endl
#include<algorithm> // set_intersection
#include<set>       // set, inserter

using namespace std;

int main() {
    // Set of students taking Chemistry
    set<string> chem = {
        "Aidan", "Flora", "Victoria", "Caleb", "Nick", "Diana", "John"
    };

    // Set of students taking Advanced Mathematics
    set<string> math = {
        "Jasmin", "Victoria", "Nick", "Aidan", "Latifa", "Rashidi", "Ali"
    };

    // Generate a set of students taking both subjects
    set<string> both;
    set_intersection(chem.begin(), chem.end(), math.begin(), math.end(),
                     inserter(both, both.begin()));

    // Display the list of students taking both subjects
    cout << "Students taking both subjects: ";

    set<string>::iterator it;
    for (it = both.begin(); it != both.end(); it++) {
        cout << *it << " ";
    }

    return 0;
}
```

```
Students taking both subjects: Aidan Nick Victoria
```

Example 45: Set Intersection (C++11 and up)

Note: Sometimes it is critical to allow duplicates in a set. In C++, this can be achieved using a *multiset*. *Multiset* operations are similar to that of a *set*. A *multiset* can be defined as:

```
multiset<type> name;
```

7.2.3 Priority Queue

A priority queue is a container that organizes its elements in such a way that its first element is always the greatest of all the elements it contains.

In C++, a definition of a priority queue takes the following general form:

```
queue<type> name;
```

In Example 46, three numbers are added into a priority queue. At this point, 30 will be the top number. Then, 40 is added into the priority queue, making it the new top number. Then, the top number is removed from the priority queue, making 30 the new top number.

```
#include<iostream>  // cout, endl
#include<queue>     // priority_queue

using namespace std;

int main() {
    priority_queue<int> numbers;

    numbers.push(10);
    numbers.push(30);
    numbers.push(20);

    cout << "Front number: " << numbers.top() << endl;

    numbers.push(40);

    cout << "Front after adding 40: " << numbers.top() << endl;

    numbers.pop();

    cout << "Front after removing the top number: " << numbers.top() << endl;

    return 0;
}
```

```
Front number: 30
Front after adding 40: 40
Front after removing the top number: 30
```

Example 46: Priority Queue in Action

Standard queue includes a handful of other common operations that have been summarized in Table 9.

Table 9: Common Priority Queue Operations

Operation	Description
`empty()`	Checks whether the priority queue is empty. Returns *true* if the priority queue is empty, *false* otherwise

`size()`	Returns the current number of elements in the priority queue
`push(x)`	Adds a new element “x” into the priority queue. If this is the largest element, it will also become the top element
`pop()`	Deletes the largest value from the priority queue
`top()`	Returns a reference to the largest element in the priority queue

Priority Queue Applications

Priority queues are extensively used in the implementation of algorithms, such as:

i. Dijkstra’s shortest path algorithm
ii. Prim’s algorithm
iii. A* search algorithm
iv. Huffman codes for data compression
v. Load balancing and interrupt handling in operating systems

8 Data Streams

8.1 Standard Input and Output Streams

So far, we have been creating programs that take input from the keyboard and display output on the console screen. We have used the inbuilt **cin** and **cout** provided by the **iostream** standard library to read user input from the keyboard and write output on the console screen respectively (Figure 4).

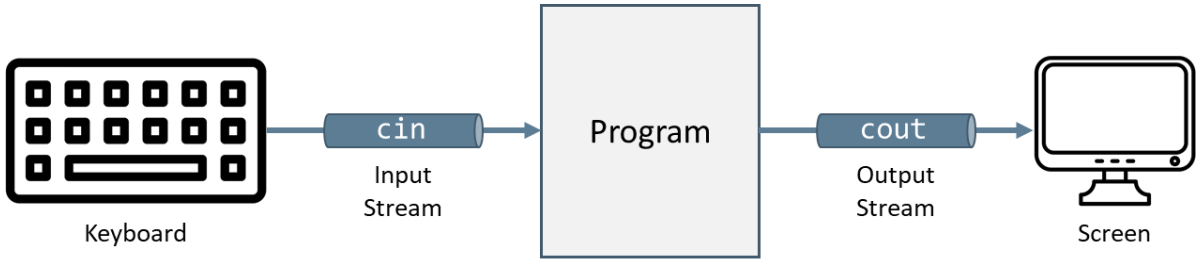

Figure 4: Default Standard Input and Output Streams

cin and **cout** are examples of *streams*. A stream provides a data transfer connection between our program and a data object like a file, keyboard, or a console screen. If a stream allows data to flow from a data object (keyboard, file) into our program, we call it an *input stream* (e.g. *cin*). On the other hand, if a stream allows data to flow from our program into a data object (console screen, file), we call it an *output stream* (e.g. *cout*).

cin is an in-built input stream that by default is hooked to the keyboard. Hence, we can use **cin** in its default configuration to input data from a keyboard into our program.

cout is an in-built output stream that by default is hooked to the console screen. Hence, we can use **cout** in its default configuration to output data from our program to the console screen.

In Example 47 below, you can notice that ">>" and "<<" operators point in the direction of data flow. For instance, in *Line 6*, data (`"Enter your name: "`) flows from our program to the **cout** stream (and eventually to the console screen). In *Line 8*, data (whatever the user will input as the name) flows from the **cin** stream into our program and gets stored inside **name**. The same data stored in **name** will be part of the data that flows out of our program to the **cout** stream in *Line 10*.

```
1. #include<iostream>  // cout, cin, endl
2.
3. using namespace std;
4.
5. int main() {
6.     cout << "Enter your name: ";
7.     string name;
8.     cin >> name;
9.
10.    cout << "Thank you " << name << "!" << endl;
11.
12.    return 0;
13.}
```

```
Enter your name: Alice
Thank you Alice!
```

Example 47: Default Standard Input and Output Streams

8.2 Redirecting the Standard Input and Output

As demonstrated in *Section 8.1*, by default **cin** is hooked to the keyboard and **cout** is hooked to the console screen. C++ includes the ability to redefine what **cin** and **cout** are hooked to. Usually, **cin** and **cout** are redefined to point to text files.

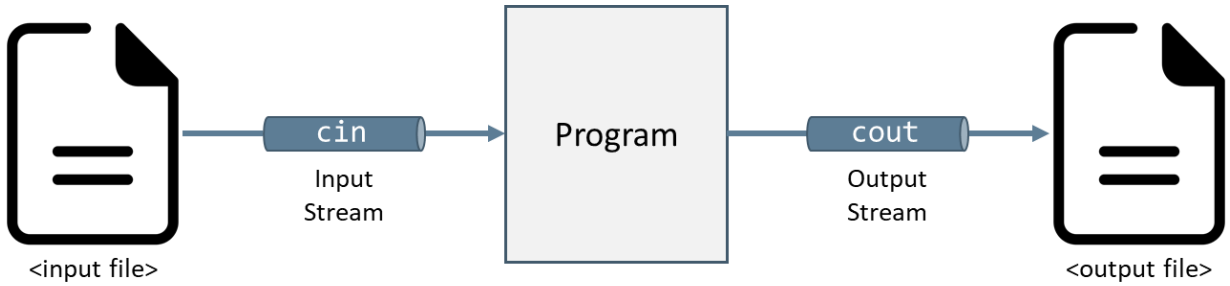

Figure 5: Redefined Standard Input and Output Streams

Before executing Example 48, create a text file called "`data.txt`" in the same location where the program source file is located. Add your name, save, and close the file. If the program is now compiled and executed, nothing will appear on the console screen. This is because all the output data will be saved in a file called "`output.txt`".

Notice that the program (*Lines 14 to 19*) did not change after we redirected the standard input and standard output streams. Also, you can decide to just redirect a single standard stream (**cin** only or **cout** only). For instance, if you comment out *Line 11*, the name will still be fetched from the "`data.txt`" file but the output will be displayed on the console screen.

```
Alice
```
Input File

```cpp
#include<iostream>  // cout, cin, endl
#include<cstdio>    // freopen, stdin, stdout

using namespace std;

int main() {
    // Redefine the standard input to point to our own file
    freopen("data.txt", "r", stdin);

    // Redefine the standard output to point to our own file
    freopen("output.txt", "w", stdout);

    //cout << "Enter your name: ";
    string name;
    cin >> name;

    cout << "Thank you " << name << "!" << endl;

    return 0;
}
```

```
Thank you Alice!
```
Output File

Example 48: Redirected Standard Input and Output Streams

8.3 User Defined Input and Output Streams

What if we want to get some of the data from the keyboard and some of the data from a file at the same time. What if we want to save some output into a file and display some output on the console screen?

We can leave **cin** and **cout** in their default configurations (hooked to the keyboard and console screen respectively) and define our own input and output streams that point to our desired files. This can enable us to interact with files as well as the keyboard and console screen at the same time (Figure 6).

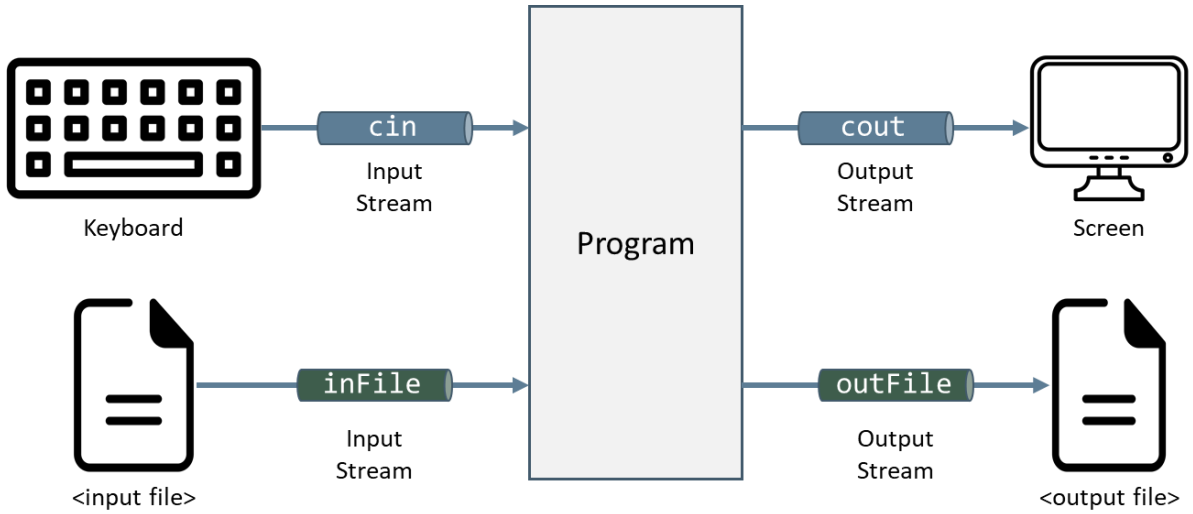

Figure 6: User Defined Input and Output Streams

In Example 49, *Line 9* defines our custom input stream called "`inFile`" and bind it to "`country.txt`" file. This file should exist before compiling and executing the example. Hence, create the text file and add your country, save, and close the file.

Line 12 defines our custom output stream called "`outFile`" and bind it to "`results.txt`". This file will be created when the program is executed. If a file with the same filename exists, its data will be overwritten.

`inFile` and `outFile` are the variables that reference the streams we have created. You can assign any names to the custom streams like any other variable.

Lines 21 & 28 show how the streams can be used. As we can see, `inFile` and `outFile` can be used similar to `cin` and `cout` respectively.

Lines 31 & 32 close the two streams we have created. It is the best practice to close the resources we have created once we are done using them.

Once the program is executed, we can see that it fetches some data from the file and requests some from the user. Also, it displays some output on the console screen and save some data in a results file.

Tanzania

Input File

```
1. #include<iostream>  // cout, cin, endl
2. #include<cstdio>    // freopen, stdin, stdout
3. #include<fstream>   // ifstream, ofstream
4.
5. using namespace std;
6.
7. int main() {
8.     // Define an input file stream and bind it to our own file
9.     ifstream inFile("country.txt");
10.
11.    // Define an output file stream and bind it to our own file
12.    ofstream outFile("results.txt");
13.
14.    // Request for a user's name
15.    cout << "Enter your name: ";
16.    string name;
17.    cin >> name;
18.
19.    // Extract the country from the file
20.    string country;
21.    if (inFile.is_open()) { // Check if the file was opened successfully
22.        inFile >> country;
23.    } else {
24.        country = "N/A";   // Set a default country value
25.    }
26.
27.    // Display results to the console screen
28.    cout << "Thank you " << name << " from " << country << "!"<< endl;
29.    cout << "This message is also saved in \"results.txt\"" << endl;
30.
31.    // Save the same results into a file
32.    outFile << "Thank you " << name << " from " << country << "!"<< endl;
33.
34.    // Close the streams we defined
35.    inFile.close();
36.    outFile.close();
37.
38.    return 0;
39.}
```

Console Screen

```
Enter your name: Alice
Thank you Alice from Tanzania!
This message is also saved in "results.txt"
```

Output File

```
Thank you Alice from Tanzania!
```

Example 49: User Defined Input and Output Streams

8.4 String Streams

A string stream provides a stream that we can put data into as if it were an output stream like **cout** and we can read from it as if it were an input stream like **cin**.

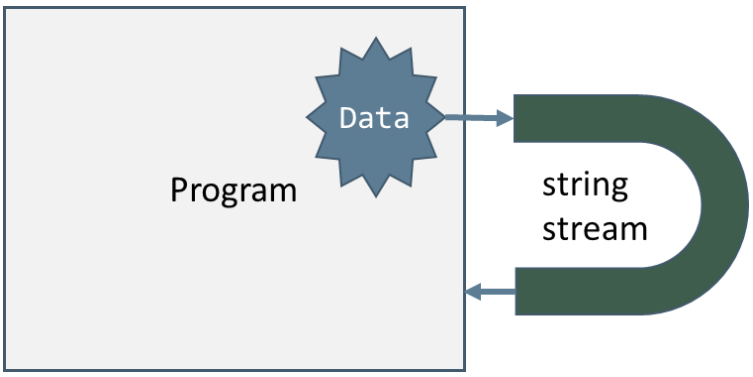

Figure 7: String Stream

From Figure 7, we can see that we can add data into a string stream and read from the front of a string stream. This provides a strong method to parse user input or manipulate data between different data types as shown in the following examples.

```
1. #include<iostream>  // cout, cin, endl
2. #include<sstream>    // stringstream
3.
4. using namespace std;
5.
6. int main() {
7.     // Create a string stream from string data
8.     stringstream ss("Alice 19");
9.
10.    // Read data from the string stream
11.    string name;
12.    ss >> name;
13.
14.    int age;
15.    ss >> age;
16.
17.    cout << name << " is " << age << " years old!" << endl;
18.
19.    return 0;
20.}
```

```
Alice is 19 years old!
```

Example 50: String Stream

In Example 50, a string stream "`ss`" is created with initial data "`Alice 19`" in *Line 8*. The data contained in "`ss`" can be read just as we have been reading data from **cin** as shown in *Lines 12 & 15*.

A common use of a string stream is to convert between different data types. In Example 51, we try to convert two strings (`cleanData` and `dirtData`) to integers. An attempt is made to convert each string (*Lines 16 & 27*), then the status of the string stream is checked (*Lines 18 & 29*) to determine if the conversion was successful or not. As shown in Section 9.2, this technique can be used to guard a program from crashing when a user supplies an invalid input.

```
1. #include<iostream>  // cout, cin, endl
2. #include<sstream>    // stringstream
3.
4. using namespace std;
5.
6. int main() {
7.     string cleanData = "538";
8.     string dirtData = "WB78";
9.
10.    stringstream ss; // Create an empty string stream
11.
12.    int num;
13.
14.    // Attempt to convert clean data to a number
15.    ss << cleanData;   // Add clean string data into the stream
16.    ss >> num;   // Try to read data from the stream as an integer
17.
18.    if (ss.fail()) { // Check if stream was read successfully
19.        cout << "Error converting " << cleanData << " to integer!" << endl;
20.    } else {
21.        cout << num << " was read successfully!" << endl;
22.    }
23.
24.    // Attempt to convert dirt data to a number
25.    ss.str("");  // Clear the contents of the stream (if any)
26.    ss.clear();  // Reset any errors in the stream
27.
28.    ss << dirtData;   // Add dirt string data into the stream
29.    ss >> num;   // Try to read data from the stream as an integer
30.
31.    if (ss.fail()) { // Check if stream was read successfully
32.        cout << "Error converting \"" << dirtData << "\" to integer!" << endl;
33.    } else {
34.        cout << num << " was read successfully!" << endl;
35.    }
36.
37.    return 0;
38.}
```

```
538 was read successfully!
Error converting "WB78" to integer!
```

Example 51: Data Conversion Using String Streams

Note: In Example 51, at any time, we can get all the contents currently in the stream using:

```
string allStreamData = ss.str();
```

9 Guarding Against User Input

Murphy's law states that "*Things will go wrong in any given situation, if you give them a chance*". Hard-drives can crash, power can go off, user can enter the wrong input, and so on. All such factors can prevent programs we write from running in the way we intend them to run. Some of those factors are always beyond the ability of a programmer to address (like power going off). However, we can prevent our programs from crashing by anticipating what can go wrong when we write our code, especially in handling data. We can minimize what can go wrong when our program is executed.

9.1 Anticipating Exceptional Cases

Consider the code in Example 52. The program asks the user to enter two integers and computes the quotient when the first is divided by the second. We expect the user to enter numbers like 24 and 6 (first case). However, a user can enter zero as the second number (second case). This is problematic because dividing an integer by zero is undefined in Mathematics and results into an undefined behavior in C++. Like in the second case of output, entering 24 and 0 will crash the program.

```
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    // Request two integers from the user
    cout << "Enter two integers: ";
    int a, b;
    cin >> a >> b;

    cout << a << " divide by " << b << " is " << a / b << endl;

    return 0;
}
```

```
Enter two integers: 24 6
24 divide by 6 is 4
```

```
Enter two numbers: 24 0
```

Example 52: Division Problem

Hence, during programming, one of critical task is to think about such exceptional cases and write our code in such a way that we protect the program from crashing when such extreme cases are encountered. From Example 52, we can simply check if the second number is zero before dividing the two numbers. If the user entered zero as the second number, we can alert the user that the second number cannot be zero or decide another course of action.

9.2 Handling Invalid User Input

We all know such people. The program asks for their age, they enter their name. ~~Sometimes~~ Most of the time, users make mistakes. It is our duty as programmers to help users achieve their goals despite the mistakes they make. We have all used the "undo" button.

Consider the following program (Example 53). Suppose a user accidentally types "B21" instead of "21", the program will provide a wrong feedback to the user (Garbage In, Garbage Out). How can we protect the user against such a mistake?

```
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    cout << "Enter your age: ";
    int age;
    cin >> age;

    if (age > 18) cout << "You can vote!" << endl;
    else cout << "You can start voting after " << 18 - age << " years!\n";

    return 0;
}
```

```
Enter your age: B21
You can start voting after 18 years!
```

Example 53: Wrong User Input

One solution is to check if the process of reading the input was successful. This can be done by inspecting the input stream (*cin*) for a failure. Example 53 can be updated as follows.

```cpp
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    cout << "Enter your age: ";
    int age;
    cin >> age;

    if (cin.fail()) {  // Check if reading an integer was successful
        cout << "Invalid input!" << endl;
    } else {
        if (age > 18) cout << "You can vote!" << endl;
        else cout << "You can start voting after " << 18 - age << " years!\n";
    }

    return 0;
}
```

```
Enter your age: B21
Invalid input!
```

```
Enter your age: 21
You can vote!
```

Example 54: Guarding Against Wrong User Input

Another solution is to always receive user input as *string*. Then inspect the input to see if it is what was expected and act accordingly after that. Example 53 can be re-written as shown in Example 55.

```cpp
#include<iostream>  // cout, cin, endl, atoi, isdigit

using namespace std;

int main() {
    // Grab user input as text
    cout << "Enter your age: ";
    string input;
    cin >> input;

    // Inspect if the input is valid (in our case: contains only digits)
    bool isValidInput = true;
    for (int k = 0; k < input.size(); k++) {
        if (isdigit(input[k]) == false) {
            isValidInput = false;
            break;
        }
    }

    // Take right action depending on input status
    if (isValidInput) {
        int age = atoi(input.c_str());  // Convert input to a number

        if (age > 18) cout << "You can vote!" << endl;
        else cout << "You can start voting after " << 18 - age << " years!\n";
    } else {
        cout << "\"" << input << "\" is invalid. "
             << "Age should be a positive number!\n";
    }

    return 0;
}
```

```
Enter your age: 21B
"21B" is invalid. Age should be a positive number!
```

```
Enter your age: 21
You can vote!
```

```
Enter your age: 7
You can start voting after 11 years!
```

Example 55: Guarding and Inspecting Wrong User Input

9.3 Handling Long String Input

The program in Example 56 asks the user to enter his/her full name (first name and last name). If we read the full name as we usually do, only the first name will be captured.

```
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    cout << "Enter your full name: ";
    string fullName;
    cin >> fullName;

    cout << "Welcome " << fullName << "!" << endl;

    return 0;
}
```

```
Enter your full name: Alice Bakara Charles
Welcome Alice!
```

Example 56: Reading String Containing Space (Problem)

This can easily be overcome by using *getline()* function as follows.

```
#include<iostream>  // cout, cin, endl, getline

using namespace std;

int main() {
    cout << "Enter your full name: ";
    string fullName;
    getline(cin, fullName);

    cout << "Welcome " << fullName << "!" << endl;

    return 0;
}
```

```
Enter your full name: Alice Baraka Charles
Welcome Alice Baraka Charles!
```

Example 57: Reading String Containing Space (Solution)

9.4 Ignoring User Input

Sometimes the user input or data source contains more data than we may need. Suppose the user always enters a phone number containing the Tanzanian country code (e.g. +255756123456), but our program needs the number without the country code (e.g. 0756123456). One solution is to ignore the segment of the phone that we do not need when we are capturing the phone number into our program as shown below.

```cpp
#include<iostream>  // cout, endl

using namespace std;

int main() {
    cout << "Enter your phone number (+255XXXXXXXXX): ";

    string number;
    cin.ignore(4);  // Ignore the first four characters
    cin >> number;  // Grab the rest of the number
    number = "0" + number;   // Add the leading zero

    cout << "Your number is: " << number << endl;

    return 0;
}
```

```
Enter your phone number (+255XXXXXXXXX): +255787123456
Your number is: 0787123456
```

Example 58: Ignoring User Input

Also, `ignore()` can specify a terminating character. For instance, Example 59 asks to skip the next 10 characters or until a hyphen (-) is found in the user input, whatever that comes first.

```cpp
#include<iostream>  // cout, cin, endl

using namespace std;

int main() {
    cout << "Enter your name: ";

    string name;
    cin.ignore(10, '-');  // Ignore the first 10 characters or until '-' is read
    cin >> name;

    cout << name << endl;

    return 0;
}
```

```
Enter your name: Alice-Bob-Charles
Bob-Charles
```

Example 59: Ignoring User Input Until a Character

**** The End of Practical Guideline 1 ****